

Java Source and Bytecode Formalizations in Isabelle: Bali

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

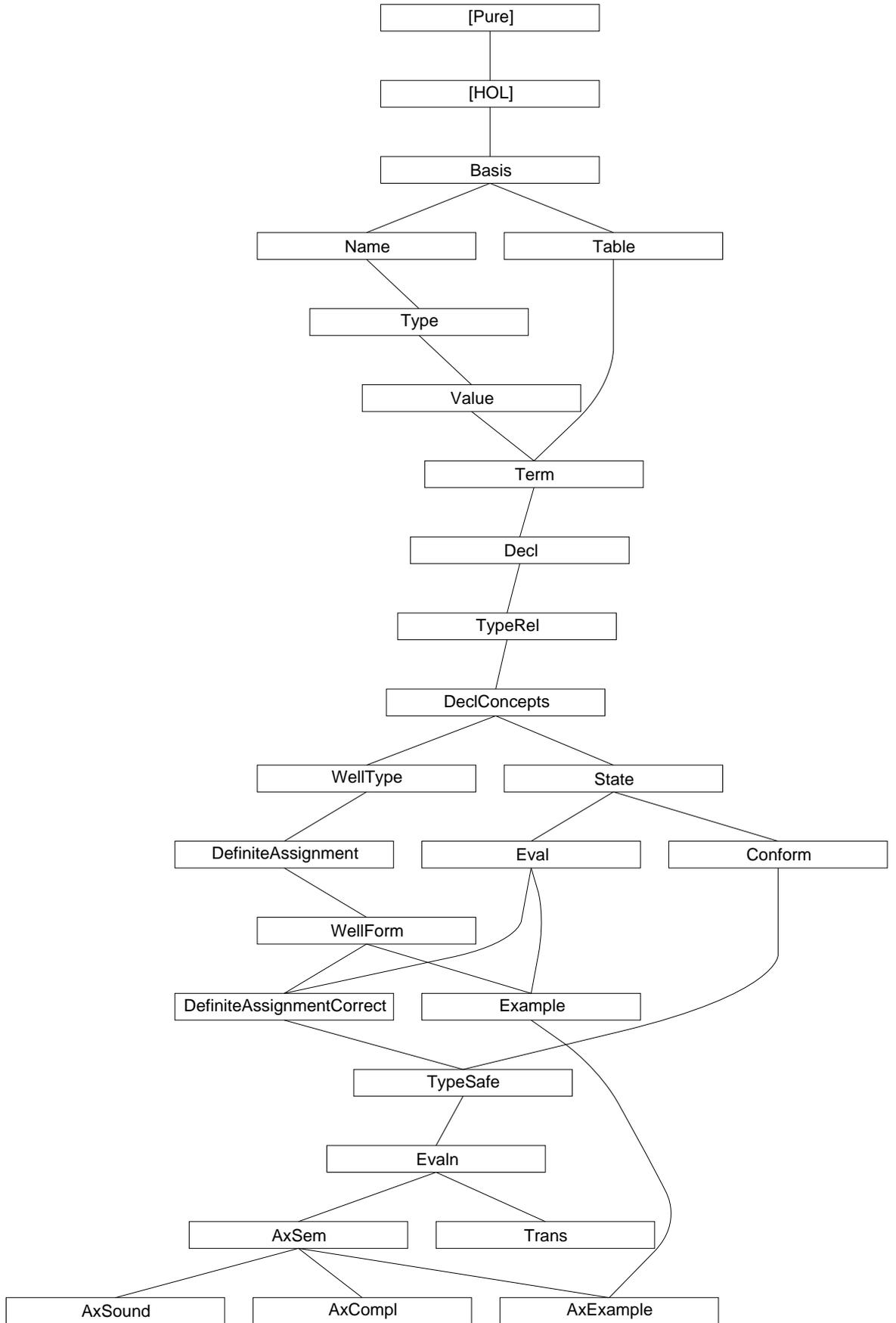
November 22, 2007

Contents

1	Overview	7
2	Basis	11
1	Definitions extending HOL as logical basis of Bali	12
3	Table	17
2	Abstract tables and their implementation as lists	18
4	Name	25
3	Java names	26
5	Value	29
4	Java values	30
6	Type	31
5	Java types	32
7	Term	33
6	Java expressions and statements	34
8	Decl	43
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
9	TypeRel	57
15	The relations between Java types	58
10	DeclConcepts	67
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	89
21	methd	90
22	accmethd	91
23	dynmethd	91

24	dynlookup	93
25	fields	93
26	accfield	94
27	is methd	95
11	WellType	97
28	Well-typedness of Java programs	98
12	DefiniteAssignment	109
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
13	WellForm	125
34	Well-formedness of Java programs	126
35	accessibility concerns	143
14	State	147
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
15	Eval	163
42	Operational evaluation (big-step) semantics of Java expressions and statements	164
16	Example	181
43	Example Bali program	182
17	Conform	199
44	Conformance notions for the type soundness proof for Java	200
18	DefiniteAssignmentCorrect	209
45	Correctness of Definite Assignment	210
19	TypeSafe	219
46	The type soundness proof for Java	220
47	accessibility	228
48	Ideas for the future	233
20	Evaln	235
49	Operational evaluation (big-step) semantics of Java expressions and statements	236
21	Trans	243
22	AxSem	249
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.

AxCompl 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

declare [[*unify-search-bound* = 40, *unify-trace-bound* = 40]]

misc

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

declare *split-if-asm* [*split*] *option.split* [*split*] *option.split-asm* [*split*]
 <ML>

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

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

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

lemma *subset-insertD*:

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

syntax

$3 :: nat \quad (3)$

$4 :: nat \quad (4)$

translations

$3 == Suc\ 2$

$4 == Suc\ 3$

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

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

lemma *trancl-rtrancl-trancl*:

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

lemma *rtrancl-into-trancl3*:

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

lemma *rtrancl-into-rtrancl2*:

$\llbracket (a, b) \in r; (b, c) \in r^{\wedge*} \rrbracket \implies (a, c) \in r^{\wedge*}$
 <proof>

lemma *triangle-lemma*:

$$\llbracket \bigwedge a b c. \llbracket (a,b) \in r; (a,c) \in r \rrbracket \implies b=c; (a,x) \in r^*; (a,y) \in r^* \rrbracket$$

$$\implies (x,y) \in r^* \vee (y,x) \in r^*$$
 <proof>

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

$$\llbracket (a,b) \in r^*; a = b \implies P; (a,b) \in r^+ \implies P \rrbracket \implies P$$
 <proof>

theorems *converse-rtrancl-induct*
 = *converse-rtrancl-induct* [*consumes 1, case-names Id Step*]

theorems *converse-trancl-induct*
 = *converse-trancl-induct* [*consumes 1, case-names Single Step*]

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

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

lemma *list-all2-trans*: $\forall a b c. P1 a b \longrightarrow P2 b c \longrightarrow P3 a c \implies$
 $\forall xs2 xs3. \text{list-all2 } P1 xs1 xs2 \longrightarrow \text{list-all2 } P2 xs2 xs3 \longrightarrow \text{list-all2 } P3 xs1 xs3$
 <proof>

pairs

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

lemma *fst-splitE* [*elim!*]:
 $\llbracket \text{fst } s' = x'; !!x s. \llbracket s' = (x,s); x = x' \rrbracket \implies Q \rrbracket \implies Q$
 <proof>

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

quantifiers

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*]:
 $(EX w v. P w v \ \& \ Q v) = (EX v. (EX w. P w v) \ \& \ Q v)$

$\langle \text{proof} \rangle$

lemma *ex-miniscope2* [*simp*]:

$$(EX\ v.\ P\ v \ \&\ Q \ \&\ R\ v) = (Q \ \&\ (EX\ v.\ P\ v \ \&\ R\ v))$$

$\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

sums

hide *const In0 In1*

syntax

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

translations

$$\text{fun-sum} == \text{CONST sum-case}$$

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

$$\text{In1l } e == \text{In1 } (\text{Inl } e)$$

$$\text{In1r } c == \text{In1 } (\text{Inr } c)$$

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

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

translations

$$\text{the-In1l} == \text{the-Inl} \circ \text{the-In1}$$

$$\text{the-In1r} == \text{the-Inr} \circ \text{the-In1}$$

$\langle \text{ML} \rangle$

translations

option <= (*type*) *Datatype.option*

list <= (*type*) *List.list*

sum3 <= (*type*) *Basis.sum3*

quantifiers for option type**syntax**

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

syntax (symbols)

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

translations

$! x:A: P \quad == \ ! x:o2s\ A.\ P$
 $? x:A: P \quad == \ ? x:o2s\ A.\ P$

Special map update

Deemed too special for theory Map.

constdefs

$chg\text{-}map :: ('b \Rightarrow 'b) \Rightarrow 'a \Rightarrow ('a \rightsquigarrow \Rightarrow 'b) \Rightarrow ('a \rightsquigarrow \Rightarrow 'b)$
 $chg\text{-}map\ f\ a\ m == \ case\ m\ a\ of\ None \Rightarrow m \mid Some\ b \Rightarrow m(a|\text{-}\>f\ b)$

lemma $chg\text{-}map\text{-}new[simp]: m\ a = None \implies chg\text{-}map\ f\ a\ m = m$
 $\langle proof \rangle$

lemma $chg\text{-}map\text{-}upd[simp]: m\ a = Some\ b \implies chg\text{-}map\ f\ a\ m = m(a|\text{-}\>f\ b)$
 $\langle proof \rangle$

lemma $chg\text{-}map\text{-}other\ [simp]: a \neq b \implies chg\text{-}map\ f\ a\ m\ b = m\ b$
 $\langle proof \rangle$

unique association lists**constdefs**

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

lemma $uniqueD\ [rule\text{-}format\ (no\text{-}asm)]:$
 $unique\ l \text{-}\longrightarrow (\!x\ y.\ (x,y):set\ l \text{-}\longrightarrow (\!x'\ y'.\ (x',y'):set\ l \text{-}\longrightarrow x=x' \text{-}\longrightarrow y=y'))$
 $\langle proof \rangle$

lemma $unique\text{-}Nil\ [simp]: unique\ []$
 $\langle proof \rangle$

lemma $unique\text{-}Cons\ [simp]: unique\ ((x,y)\#l) = (unique\ l \ \&\ (\!y.\ (x,y) \rightsquigarrow \text{-:}\ set\ l))$
 $\langle proof \rangle$

lemmas $unique\text{-}ConsI = conjI\ [THEN\ unique\text{-}Cons\ [THEN\ iffD2],\ standard]$

lemma $unique\text{-}single\ [simp]: \!\!p.\ unique\ [p]$
 $\langle proof \rangle$

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

lemma *unique-append* [rule-format (no-asm)]: *unique l' ==> unique l -->*
(!(x,y):set l. !(x',y'):set l'. x' ~ = x) --> unique (l @ l')
 <proof>

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

lemma *map-of-SomeI* [rule-format (no-asm)]: *unique l --> (k, x) : set l --> map-of l k = Some x*
 <proof>

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*
 <proof>

lemma *lsplit2* [simp]: *lsplit P (x#xs) y z = P x xs y z*
 <proof>

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

map of / table of

syntax

$\text{table-of} \quad :: (\text{'a} \times \text{'b}) \text{ list} \Rightarrow (\text{'a}, \text{'b}) \text{ table}$ — concrete table

translations

$\text{table-of} == \text{map-of}$

$(\text{type})\text{'a} \rightarrow \text{'b} \quad <= (\text{type})\text{'a} \Rightarrow \text{'b} \text{ Datatype.option}$

$(\text{type})(\text{'a}, \text{'b}) \text{ table} <= (\text{type})\text{'a} \rightarrow \text{'b}$

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

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

<proof>

Conditional Override

constdefs

cond-override::

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

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

cond-override cond old new \equiv

$\lambda k.$

$(\text{case new } k \text{ of}$

$\text{None} \quad \Rightarrow \text{old } k$

$| \text{Some new-val} \Rightarrow (\text{case old } k \text{ of}$

$\text{None} \quad \Rightarrow \text{Some new-val}$

$| \text{Some old-val} \Rightarrow (\text{if cond new-val old-val}$

then Some new-val

$\text{else Some old-val}))$

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*:

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

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)]:

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

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 (*table-of* t k = 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 $C = \text{snd } ek \wedge \text{table-of } t \text{ (fst } ek) = \text{Some } x$
 ⟨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 *unique* xs \longrightarrow *table-of* xs k = 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 $\forall k. \forall x \in t \text{ k}. \forall y \in s \text{ k}. R \text{ x y}$
hiding-entails-def: *t hiding s entails* R \equiv $\forall k. \forall x \in t \text{ k}. \forall y \in s \text{ k}. R \text{ x y}$
cond-hiding-entails-def: *t hiding s under C entails* R
 \equiv $\forall k. \forall x \in t \text{ k}. \forall y \in s \text{ k}. C \text{ x y} \longrightarrow R \text{ x y}$

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

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

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

lemma *atleast-free-SucI*:
 $\llbracket h \ a = \text{None}; !obj. \ \text{atleast-free} \ (h(a|-\>obj)) \ n \rrbracket \implies \text{atleast-free} \ h \ (\text{Suc} \ n)$
 ⟨proof⟩

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

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

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

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

Chapter 4

Name

3 Java names

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

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

typedecl *pname* — package name

typedecl *mname* — method name

typedecl *vname* — variable or field name

typedecl *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*

| *NegArrSize* | *IndOutBound* | *ArrStore*

lemma *xn-cases*:

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

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

xn = *NegArrSize* \vee *xn* = *IndOutBound* \vee *xn* = *ArrStore*

\langle proof \rangle

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

= *Object'*

| *SXcpt'* *xname*

| *TName tnam*

record *qtname* = — 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 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 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 *pid*::*pname*,*tid*::*tname* \rfloor)

(*type*) '*a qtname-scheme* <= (*type*) (\lfloor *pid*::*pname*,*tid*::*tname*,...::*a* \rfloor)

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

consts

Object :: *qtname*

SXcpt :: *xname* \Rightarrow *qtname*

defs

Object-def: *Object* \equiv (\lfloor *pid* = *java-lang*, *tid* = *Object* \rfloor)

SXcpt-def: *SXcpt* \equiv $\lambda x.$ (\lfloor *pid* = *java-lang*, *tid* = *SXcpt*' *x* \rfloor)

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

\langle proof \rangle

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

\langle proof \rangle

end

Chapter 5

Value

4 Java values

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

typedecl *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 qname* — interface type
 | *ClassT qname* — 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 :: *qname* \Rightarrow *ty*
Class :: *qname* \Rightarrow *ty*
Array :: *ty* \Rightarrow *ty* (-.[*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 *qname*
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 programmss

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'(-) - Else -* [80,79,79]70)
- | *Loop label expr stmt* ($\cdot - \text{While}'(-) - [99,80,79]70$)
- | *Jmp jump* — break, continue, return
- | *Throw expr*
- | *TryC stmt qname vname stmt* (*Try - Catch'(- -)* - [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* (*- Finally -* [79,79]70)
- | *FinA abopt stmt* — Save abrupton 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 semantic's 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* × *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* ⇒ *expr* (!!)
- LAss* :: *vname* ⇒ *expr* ⇒ *stmt* ($\cdot - [90,85]85$)
- Return* :: *expr* ⇒ *stmt*
- StatRef* :: *ref-ty* ⇒ *expr*

translations

```

this      == Acc (LVar This)
!!v       == Acc (LVar (ENAME (VName v)))
v::=e     == Expr (Ass (LVar (ENAME (VName v))) e)
Return e  == Expr (Ass (LVar (ENAME 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*: $\langle e::\text{expr} \rangle = \text{In1}1 e$
 $\langle \text{proof} \rangle$

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]*: $\langle t::\text{expr list} \rangle \neq \langle w::\text{stmt} \rangle$

<proof>

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

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

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

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

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

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

lemma *term-cases*:

$\langle \bigwedge v. P \langle v \rangle_v; \bigwedge e. P \langle e \rangle_e; \bigwedge c. P \langle c \rangle_s; \bigwedge l. P \langle l \rangle_l \rangle$
 $\implies P t$
<proof>

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* $:: \text{binop} \Rightarrow \text{val} \Rightarrow \text{bool}$
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
 $\langle \text{proof} \rangle$

lemma *need-second-arg-CondOr* [*simp*]: *need-second-arg CondOr* (*Bool* b) = $(\neg b)$
 $\langle \text{proof} \rangle$

lemma *need-second-arg-strict* [*simp*]:
 $\llbracket \text{binop} \neq \text{CondAnd}; \text{binop} \neq \text{CondOr} \rrbracket \Longrightarrow \text{need-second-arg binop } b$
 $\langle \text{proof} \rangle$
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.sun.com/bugreport/details/4485402> and <http://developer.java.sun.com/bugreport/details/4493343>)

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* | *Package* | *Protected* | *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* *<proof>*

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*
<proof>

instance *acc-modi:: linorder*
<proof>

lemma *acc-modi-top* [*simp*]: $Public \leq a \implies a = Public$
 ⟨*proof*⟩

lemma *acc-modi-top1* [*simp, intro!*]: $a \leq Public$
 ⟨*proof*⟩

lemma *acc-modi-le-Public*:
 $a \leq Public \implies a=Private \vee a = Package \vee a=Protected \vee a=Public$
 ⟨*proof*⟩

lemma *acc-modi-bottom*: $a \leq Private \implies a = Private$
 ⟨*proof*⟩

lemma *acc-modi-Private-le*:
 $Private \leq a \implies a=Private \vee a = Package \vee a=Protected \vee a=Public$
 ⟨*proof*⟩

lemma *acc-modi-Package-le*:
 $Package \leq a \implies a = Package \vee a=Protected \vee a=Public$
 ⟨*proof*⟩

lemma *acc-modi-le-Package*:
 $a \leq Package \implies a=Private \vee a = Package$
 ⟨*proof*⟩

lemma *acc-modi-Protected-le*:
 $Protected \leq a \implies a=Protected \vee a=Public$
 ⟨*proof*⟩

lemma *acc-modi-le-Protected*:
 $a \leq Protected \implies a=Private \vee a = Package \vee a = Protected$
 ⟨*proof*⟩

lemmas *acc-modi-le-Dests = acc-modi-top* *acc-modi-le-Public*
 acc-modi-Private-le *acc-modi-bottom*
 acc-modi-Package-le *acc-modi-le-Package*
 acc-modi-Protected-le *acc-modi-le-Protected*

lemma *acc-modi-Package-le-cases*
 [*consumes 1, case-names Package Protected Public*]:
 $Package \leq m \implies (m = Package \implies P m) \implies (m=Protected \implies P m) \implies$
 $(m=Public \implies P m) \implies P m$
 ⟨*proof*⟩

Static Modifier

types *stat-modi* = *bool*

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

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

translations

```
decl <= (type) (|access::acc-modi|)
decl <= (type) (|access::acc-modi,...::'a'|)
```

10 Member (field or method)

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

translations

```
member <= (type) (|access::acc-modi,static::bool|)
member <= (type) (|access::acc-modi,static::bool,...::'a'|)
```

11 Field

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

translations

```
field <= (type) (|access::acc-modi, static::bool, type::ty|)
field <= (type) (|access::acc-modi, static::bool, type::ty,...::'a'|)
```

types

```
fdecl
= vname × field
```

translations

```
fdecl <= (type) vname × field
```

12 Method

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

```
record mbody =
  lcls:: (vname × ty) list
  stmt:: stmt
```

```
record methd = mhead +
  mbody::mbody
```

```
types mdecl = sig × methd
```

translations

```
mhead <= (type) (|access::acc-modi, static::bool,
  pars::vname list, resT::ty|)
mhead <= (type) (|access::acc-modi, static::bool,
  pars::vname list, resT::ty,...::'a'|)
mbody <= (type) (|lcls::(vname × ty) list,stmt::stmt|)
mbody <= (type) (|lcls::(vname × ty) list,stmt::stmt,...::'a'|)
methd <= (type) (|access::acc-modi, static::bool,
  pars::vname list, resT::ty,mbody::mbody|)
methd <= (type) (|access::acc-modi, static::bool,
```

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

$$mhead::\text{methd} \Rightarrow mhead$$

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

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

$$\text{memberid} :: 'a::\text{has-memberid} \Rightarrow \text{memberid}$$

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

defs (overloaded)

memberdecl-memberid-def:

$$\text{memberid } m \equiv (\text{case } m \text{ of}$$

$$\quad \text{fdecl } (vn, f) \Rightarrow \text{fid } vn$$

$$\quad | \text{mdecl } (sig, m) \Rightarrow \text{mid } sig)$$

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

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

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

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

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

defs (overloaded)

pair-memberid-def:

$memberid\ p \equiv memberid\ (snd\ p)$

lemma *memberid-pair-simp*[*simp*]: $memberid\ (c,m) = memberid\ m$

<proof>

lemma *memberid-pair-simp1*: $memberid\ p = memberid\ (snd\ p)$

<proof>

constdefs *is-field* :: *qname* × *memberdecl* ⇒ *bool*

is-field *m* ≡ ∃ *declC* *f*. *m*=(*declC*,*fdecl* *f*)

lemma *is-fieldD*: *is-field* *m* ⇒ ∃ *declC* *f*. *m*=(*declC*,*fdecl* *f*)

<proof>

lemma *is-fieldI*: *is-field* (*C*,*fdecl* *f*)

<proof>

constdefs *is-method* :: *qname* × *memberdecl* ⇒ *bool*

is-method *membr* ≡ ∃ *declC* *m*. *membr*=(*declC*,*mdecl* *m*)

lemma *is-methodD*: *is-method* *membr* ⇒ ∃ *declC* *m*. *membr*=(*declC*,*mdecl* *m*)

<proof>

lemma *is-methodI*: *is-method* (*C*,*mdecl* *m*)

<proof>

13 Interface

record *ibody* = *decl* + — interface body

imethods :: (*sig* × *mhead*) *list* — method heads

record *iface* = *ibody* + — interface

isuperIfs:: *qname list* — superinterface list

types

idecl — interface declaration, cf. 9.1

= *qname* × *iface*

translations

ibody <= (*type*) (|*access*::*acc-modi*,*imethods*::(*sig* × *mhead*) *list*)

ibody <= (*type*) (|*access*::*acc-modi*,*imethods*::(*sig* × *mhead*) *list*,...:'*a*)

iface <= (*type*) (|*access*::*acc-modi*,*imethods*::(*sig* × *mhead*) *list*,
isuperIfs::*qname list*)

iface <= (*type*) (|*access*::*acc-modi*,*imethods*::(*sig* × *mhead*) *list*,
isuperIfs::*qname list*,...:'*a*)

idecl <= (*type*) *qname* × *iface*

constdefs

ibody :: *iface* ⇒ *ibody*

ibody *i* ≡ (|*access*=*access* *i*,*imethods*=*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*, (*access=Public*, *cfields=[]*, *methods=Object-mdecls*,
init=Skip, *super=arbitrary*, *superIfs=[]*))
SXcptC-def: SXcptC xn \equiv (*SXcpt xn*, (*access=Public*, *cfields=[]*, *methods=SXcpt-mdecls*,
init=Skip,
super=if xn = Throwable then Object
else SXcpt Throwable,
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 NullPointer, *SXcptC OutOfMemory*, *SXcptC ClassCast*,
SXcptC NegArrSize, *SXcptC IndOutBound*, *SXcptC ArrStore*]

programs

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

translations

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

syntax

iface :: *prog* \Rightarrow (*qname*, *iface*) *table*
class :: *prog* \Rightarrow (*qname*, *class*) *table*
is-iface :: *prog* \Rightarrow *qname* \Rightarrow *bool*
is-class :: *prog* \Rightarrow *qname* \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*: $is-type\ G\ (Iface\ I) \implies is-iface\ G\ I$
 ⟨proof⟩

lemma *type-is-class*: $is-type\ G\ (Class\ C) \implies is-class\ G\ C$
 ⟨proof⟩

subinterface and subclass relation, in anticipation of TypeRel.thy

consts

$subint1\ ::\ prog \Rightarrow (qname \times qname)\ set$ — direct subinterface
 $subcls1\ ::\ prog \Rightarrow (qname \times qname)\ set$ — direct subclass

defs

$subint1-def: subint1\ G \equiv \{(I,J). \exists i \in iface\ G\ I: J \in set\ (isuperIfs\ i)\}$
 $subcls1-def: subcls1\ G \equiv \{(C,D). C \neq Object \wedge (\exists c \in class\ G\ C: super\ c = D)\}$

syntax

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

syntax (*xsymbols*)

$-subcls1\ ::\ prog \Rightarrow [qname, qname] \Rightarrow bool$ (+-<-<_{C1}- [71,71,71] 70)
 $-subclseq::\ prog \Rightarrow [qname, qname] \Rightarrow bool$ (+-<-<_{C} - [71,71,71] 70)
 $-subcls\ ::\ prog \Rightarrow [qname, qname] \Rightarrow bool$ (+-<-<_{C} - [71,71,71] 70)

translations

$G \vdash C \prec_{C1} D \iff (C,D) \in subcls1\ G$
 $G \vdash C \preceq_C D \iff (C,D) \in (subcls1\ G)^*$
 $G \vdash C \prec_C D \iff (C,D) \in (subcls1\ G)^+$

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*:

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

lemma *subcls1-def2*:

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

lemma *subcls-is-class*:

$\llbracket G \vdash C \prec_C D \rrbracket \implies \exists\ c. class\ G\ C = Some\ c$
 ⟨proof⟩

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

⟨proof⟩

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

⟨proof⟩

well-structured programs

constdefs

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

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

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

lemma *ws-progI*:

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

lemma *ws-prog-ideclD*:

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

lemma *ws-prog-cdeclD*:

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

well-foundedness

lemma *finite-is-iface*: *finite* {*I*. *is-iface* *G* *I*}
 ⟨*proof*⟩

lemma *finite-is-class*: *finite* {*C*. *is-class* *G* *C*}
 ⟨*proof*⟩

lemma *finite-subint1*: *finite* (*subint1* *G*)
 ⟨*proof*⟩

lemma *finite-subcls1*: *finite* (*subcls1* *G*)
 ⟨*proof*⟩

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

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

lemmas *subint1-irrefl-lemma2* = *subint1-irrefl-lemma1* [THEN *irrefl-tranclI*']

lemmas *subcls1-irrefl-lemma2* = *subcls1-irrefl-lemma1* [THEN *irrefl-tranclI*']

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

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

lemmas *subint1-acyclic* = *subint1-irrefl-lemma2* [THEN *acyclicI*, *standard*]

lemmas *subcls1-acyclic* = *subcls1-irrefl-lemma2* [THEN *acyclicI*, *standard*]

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

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

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

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

$\langle \text{proof} \rangle$

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

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

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

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

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

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* \Rightarrow (*qname* \times *qname*) *set* — direct implementation

syntax

-subint1 :: *prog* \Rightarrow [*qname*, *qname*] \Rightarrow *bool* (*-|--<:I1-* [71,71,71] 70)

-subint :: *prog* \Rightarrow [*qname*, *qname*] \Rightarrow *bool* (*-|--<=:I-* [71,71,71] 70)

@implmt1 :: *prog* \Rightarrow [*qname*, *qname*] \Rightarrow *bool* (*-|--~>1-* [71,71,71] 70)

syntax (*xsymbols*)

-subint1 :: *prog* \Rightarrow [*qname*, *qname*] \Rightarrow *bool* (*-|--<I1-* [71,71,71] 70)

-subint :: *prog* \Rightarrow [*qname*, *qname*] \Rightarrow *bool* (*-|--<I-* [71,71,71] 70)

-implmt1 :: *prog* \Rightarrow [*qname*, *qname*] \Rightarrow *bool* (*-|--~>1-* [71,71,71] 70)

translations

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

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

$G \vdash C \rightsquigarrow 1 I == (C, I) \in \text{implmt1 } G$

subclass and subinterface relations

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

lemma *subcls-direct1*:

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

$\langle proof \rangle$

lemma *subcls1I1*:

$\llbracket class\ G\ C = Some\ c; C \neq Object; D = super\ c \rrbracket \implies G \vdash C \prec_{C_1} D$
 $\langle proof \rangle$

lemma *subcls-direct2*:

$\llbracket class\ G\ C = Some\ c; C \neq Object; D = super\ c \rrbracket \implies G \vdash C \prec_C D$
 $\langle proof \rangle$

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

$\langle proof \rangle$

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

$\langle proof \rangle$

lemma *SXcpt-subcls-Throwable-lemma*:

$\llbracket class\ G\ (SXcpt\ xn) = Some\ xc;$
 $\quad super\ xc = (if\ xn = Throwable\ then\ Object\ else\ SXcpt\ Throwable) \rrbracket$
 $\implies G \vdash SXcpt\ xn \preceq_C SXcpt\ Throwable$
 $\langle proof \rangle$

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

$\langle proof \rangle$

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

$\langle proof \rangle$

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

$\langle proof \rangle$

lemma *subcls-ObjectI1* [intro!]:

$\llbracket C \neq Object; is-class\ G\ C; ws-prog\ G \rrbracket \implies G \vdash C \prec_C Object$
 $\langle proof \rangle$

lemma *subcls-is-class*: $(C, D) \in (subcls1\ G)^+ \implies is-class\ G\ C$

$\langle proof \rangle$

lemma *subcls-is-class2* [rule-format (no-asm)]:

$G \vdash C \preceq_C D \implies is-class\ G\ D \longrightarrow is-class\ G\ C$
 $\langle proof \rangle$

lemma *single-inheritance*:

$\llbracket G \vdash A \prec_{C_1} B; G \vdash A \prec_{C_1} C \rrbracket \implies B = C$
 $\langle proof \rangle$

lemma *subcls-compareable*:

$$\begin{aligned} & \llbracket G \vdash A \preceq_C X; G \vdash A \preceq_C Y \\ & \rrbracket \implies G \vdash X \preceq_C Y \vee G \vdash Y \preceq_C X \\ & \langle \text{proof} \rangle \end{aligned}$$

lemma *subcls1-irrefl*: $\llbracket G \vdash C \prec_{C_1} D; \text{ws-prog } G \rrbracket$

$$\implies C \neq D$$

$\langle \text{proof} \rangle$

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*]:

$$\begin{aligned} & \llbracket G \vdash C \preceq_C D; C = D \implies P; G \vdash C \prec_C D \implies P \rrbracket \implies P \\ & \langle \text{proof} \rangle \end{aligned}$$

lemma *subclseq-acyclic*:

$$\begin{aligned} & \llbracket G \vdash C \preceq_C D; G \vdash D \preceq_C C; \text{ws-prog } G \rrbracket \implies C = D \\ & \langle \text{proof} \rangle \end{aligned}$$

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*:

$$\begin{aligned} & \llbracket G \vdash C \preceq_C D; \text{ws-prog } G \rrbracket \\ & \implies \neg G \vdash D \prec_C C \\ & \langle \text{proof} \rangle \end{aligned}$$

lemma *invert-subcls*:

$$\begin{aligned} & \llbracket G \vdash C \prec_C D; \text{ws-prog } G \rrbracket \\ & \implies \neg G \vdash D \preceq_C C \\ & \langle \text{proof} \rangle \end{aligned}$$

lemma *subcls-superD*:

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

lemma *subclseq-superD*:

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

implementation relation

defs

— direct implementation, cf. 8.1.3

implmt1-def: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*: $G \vdash C \rightsquigarrow 1I \implies C \neq \text{Object} \wedge (\exists c \in \text{class } G \ C: I \in \text{set } (\text{superIfs } c))$
 ⟨proof⟩

inductive — implementation, cf. 8.1.4

implmt :: *prog* \Rightarrow *qname* \Rightarrow *qname* \Rightarrow *bool* (+~>- [71,71,71] 70)

for *G* :: *prog*

where

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*: $\llbracket G \vdash C \prec_{C_1} D; G \vdash D \rightsquigarrow J \rrbracket \implies G \vdash C \rightsquigarrow J$

lemma *implmtD*: $G \vdash C \rightsquigarrow J \implies (\exists I. G \vdash C \rightsquigarrow 1I \wedge G \vdash I \preceq I \ J) \vee (\exists D. G \vdash C \prec_{C_1} D \wedge G \vdash D \rightsquigarrow J)$
 ⟨proof⟩

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

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

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

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

widening relation

inductive

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

widen :: *prog* \Rightarrow *ty* \Rightarrow *ty* \Rightarrow *bool* (+~>- [71,71,71] 70)

for *G* :: *prog*

where

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*: $G \vdash \text{NT} \preceq \text{RefT } R$
 | *arr-obj*: $G \vdash T.\llbracket \rrbracket \preceq \text{Class } \text{Object}$
 | *array*: $G \vdash \text{RefT } S \preceq \text{RefT } T \implies G \vdash \text{RefT } S.\llbracket \rrbracket \preceq \text{RefT } T.\llbracket \rrbracket$

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

declare *widen.intros* [*simp*]

lemma *widen-PrimT*: $G \vdash \text{PrimT } x \preceq T \implies (\exists y. T = \text{PrimT } y)$
 ⟨proof⟩

lemma *widen-PrimT2*: $G \vdash S \preceq_{\text{PrimT}} x \implies \exists y. S = \text{PrimT } y$
 ⟨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*: $G \vdash \text{PrimT } x \preceq T \implies T = \text{PrimT } x$
 ⟨proof⟩

lemma *widen-PrimT2-strong*: $G \vdash S \preceq_{\text{PrimT}} x \implies S = \text{PrimT } x$
 ⟨proof⟩

Specialized versions for booleans also would work for real Java

lemma *widen-Boolean*: $G \vdash \text{PrimT } \text{Boolean} \preceq T \implies T = \text{PrimT } \text{Boolean}$
 ⟨proof⟩

lemma *widen-Boolean2*: $G \vdash S \preceq_{\text{PrimT}} \text{Boolean} \implies S = \text{PrimT } \text{Boolean}$
 ⟨proof⟩

lemma *widen-RefT*: $G \vdash \text{RefT } R \preceq T \implies \exists t. T = \text{RefT } t$
 ⟨proof⟩

lemma *widen-RefT2*: $G \vdash S \preceq_{\text{RefT}} R \implies \exists t. S = \text{RefT } t$
 ⟨proof⟩

lemma *widen-Iface*: $G \vdash \text{Iface } I \preceq T \implies T = \text{Class } \text{Object} \vee (\exists J. T = \text{Iface } J)$
 ⟨proof⟩

lemma *widen-Iface2*: $G \vdash S \preceq_{\text{Iface}} J \implies S = \text{NT} \vee (\exists I. S = \text{Iface } I) \vee (\exists D. S = \text{Class } D)$
 ⟨proof⟩

lemma *widen-Iface-Iface*: $G \vdash \text{Iface } I \preceq_{\text{Iface}} J \implies G \vdash I \preceq I J$
 ⟨proof⟩

lemma *widen-Iface-Iface-eq [simp]*: $G \vdash \text{Iface } I \preceq_{\text{Iface}} J = G \vdash I \preceq I J$
 ⟨proof⟩

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

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

lemma *widen-Class-Class*: $G \vdash \text{Class } C \preceq_{\text{Class}} \text{Class } cm \implies G \vdash C \preceq_C cm$

$\langle \text{proof} \rangle$

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.\[] \preceq T \implies T = \text{Class Object} \vee (\exists T'. T = T'.\[] \wedge G \vdash S \preceq T')$
 $\langle \text{proof} \rangle$

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

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

lemma *widen-ArrayRefT*:
 $G \vdash \text{RefT } t.\[] \preceq T \implies T = \text{Class Object} \vee (\exists s. T = \text{RefT } s.\[] \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.\[] \preceq \text{RefT } T'.\[] = G \vdash \text{RefT } T \preceq \text{RefT } T'$
 $\langle \text{proof} \rangle$

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

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

lemma *widen-trans-lemma* [rule-format (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*: $\llbracket G \vdash S \preceq U; G \vdash U \preceq T; \text{ws-prog } G \rrbracket \implies G \vdash S \preceq T$
 ⟨proof⟩

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

lemmas *subint-antisym* =
subint1-acyclic [*THEN acyclic-impl-antisym-rtrancl, standard*]

lemmas *subcls-antisym* =
subcls1-acyclic [*THEN acyclic-impl-antisym-rtrancl, standard*]

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

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

constdefs

widens :: *prog* \Rightarrow [*ty list, ty list*] \Rightarrow *bool* (+-[\preceq]- [71,71,71] 70)
 $G \vdash Ts [\preceq] Ts' \equiv \text{list-all2 } (\lambda T T'. G \vdash T \preceq T') Ts Ts'$

lemma *widens-Nil* [*simp*]: $G \vdash [] [\preceq] []$
 ⟨proof⟩

lemma *widens-Cons* [*simp*]: $G \vdash (S \# Ss) [\preceq] (T \# Ts) = (G \vdash S \preceq T \wedge G \vdash Ss [\preceq] Ts)$
 ⟨proof⟩

narrowing relation

inductive — narrowing reference conversion, cf. 5.1.5

narrow :: *prog* \Rightarrow *ty* \Rightarrow *ty* \Rightarrow *bool* (+->- [71,71,71] 70)

for *G* :: *prog*

where

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*: $G \vdash \text{Class Object} \succ T. []$
 | *int-cls*: $G \vdash \text{Iface } I \succ \text{Class } C$
 | *subint*: *imethds* *G I hidings imethds* *G J 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*: $G \vdash \text{RefT } R \succ T \implies \exists t. T = \text{RefT } t$
 ⟨proof⟩

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

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

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

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: $G \vdash \text{PrimT } pt \succ T \implies T = \text{PrimT } pt$
 ⟨proof⟩

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

Specialized versions for booleans also would work for real Java

lemma narrow-Boolean: $G \vdash \text{PrimT } \text{Boolean} \succ T \implies T = \text{PrimT } \text{Boolean}$
 ⟨proof⟩

lemma narrow-Boolean2: $G \vdash S \succ \text{PrimT } \text{Boolean} \implies S = \text{PrimT } \text{Boolean}$
 ⟨proof⟩

casting relation

inductive — casting conversion, cf. 5.5

$\text{cast} :: \text{prog} \Rightarrow \text{ty} \Rightarrow \text{ty} \Rightarrow \text{bool} \text{ (-+-}\preceq\text{? - [71,71,71] 70)}$

for $G :: \text{prog}$

where

$\text{widen}: G \vdash S \preceq T \implies G \vdash S \preceq\text{? } T$

| $\text{narrow}: G \vdash S \succ T \implies G \vdash S \preceq\text{? } T$

lemma cast-RefT: $G \vdash \text{RefT } R \preceq\text{? } T \implies \exists t. T = \text{RefT } t$
 ⟨proof⟩

lemma cast-RefT2: $G \vdash S \preceq\text{? } \text{RefT } R \implies \exists t. S = \text{RefT } t$
 ⟨proof⟩

lemma cast-PrimT: $G \vdash \text{PrimT } pt \preceq\text{? } T \implies \exists t. T = \text{PrimT } t$
 ⟨proof⟩

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

lemma cast-Boolean:

assumes *bool-cast*: $G \vdash \text{Prim}T \text{ Boolean} \preceq? T$
shows $T = \text{Prim}T \text{ Boolean}$
(*proof*)

lemma *cast-Boolean2*:
assumes *bool-cast*: $G \vdash S \preceq? \text{Prim}T \text{ Boolean}$
shows $S = \text{Prim}T \text{ Boolean}$
(*proof*)

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 $G P T \implies is-type\ G\ T \wedge G \vdash T\ accessible-in\ P$
 ⟨proof⟩

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

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 ⟨proof⟩

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

lemma *acc-modi-accmodi-simp*[simp]: *accmodi* ($a::acc-modi$) = a
 ⟨proof⟩

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

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

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

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

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

lemma *pair-acc-modi-simp*[simp]: *accmodi* (x,a) = (*accmodi* a)
 ⟨proof⟩

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

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]:
accmodi (fdecl m) = accmodi m
 ⟨proof⟩

lemma *memberdecl-mdecl-acc-modi-simp*[simp]:
accmodi (mdecl m) = 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 ⇒ qname

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

defs (overloaded)
qname-declclass-def: *declclass (q::qname) ≡ q*

lemma *qname-declclass-simp*[simp]: *declclass (q::qname) = q*
 ⟨proof⟩

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

defs (overloaded)
pair-declclass-def: *declclass p ≡ declclass (fst p)*

lemma *pair-declclass-simp*[simp]: *declclass (c,x) = 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 ⇒ bool

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

defs (overloaded)
decl-is-static-def:
is-static (m::('a::has-static) decl-scheme) ≡ is-static (Decl.decl.more m)

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

defs (overloaded)
static-field-type-is-static-def:
is-static (m::('b::type) member-ext-type) ≡ static-sel m

lemma *member-is-static-simp*: *is-static (m::'a member-scheme) = static m*
 ⟨proof⟩

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

defs (overloaded)
pair-is-static-def: *is-static p ≡ is-static (snd p)*

lemma *pair-is-static-simp* [simp]: *is-static (x,s) = is-static s*

$\langle \text{proof} \rangle$

lemma *pair-is-static-simp1*: $is\text{-static } p = is\text{-static } (snd\ p)$

$\langle \text{proof} \rangle$

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

defs (overloaded)

memberdecl-is-static-def:

$is\text{-static } m \equiv (case\ m\ of$
 $\quad fdecl\ f \Rightarrow is\text{-static } f$
 $\quad | mdecl\ m \Rightarrow is\text{-static } m)$

lemma *memberdecl-is-static-fdecl-simp*[*simp*]:

$is\text{-static } (fdecl\ f) = is\text{-static } f$

$\langle \text{proof} \rangle$

lemma *memberdecl-is-static-mdecl-simp*[*simp*]:

$is\text{-static } (mdecl\ m) = is\text{-static } m$

$\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

constdefs — some mnemonic selectors for various pairs

decliface:: $(qname \times ('a::type)\ decl\text{-scheme}) \Rightarrow qname$
decliface $\equiv fst$ — get the interface component

mbr:: $(qname \times memberdecl) \Rightarrow memberdecl$
mbr $\equiv snd$ — get the memberdecl component

mthd:: $('b \times 'a) \Rightarrow 'a$
— also used for mdecl, mhead
mthd $\equiv snd$ — get the method component

fld:: $('b \times ('a::type)\ decl\text{-scheme}) \Rightarrow ('a::type)\ decl\text{-scheme}$
— also used for $((vname \times qname) \times field)$
fld $\equiv snd$ — get the field component

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

fname:: $(vname \times 'a) \Rightarrow vname$ — also used for fdecl
fname $\equiv fst$

declclassf:: $(vname \times qname) \Rightarrow qname$
declclassf $\equiv snd$

lemma *decliface-simp*[*simp*]: $decliface\ (I, m) = I$

$\langle \text{proof} \rangle$

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

lemma *access-mbr-simp* [simp]: $(accmodi (mbr m)) = accmodi m$
 ⟨proof⟩

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

lemma *fld-simp*[simp]: $fld (C,f) = f$
 ⟨proof⟩

lemma *accmodi-simp*[simp]: $accmodi (C,m) = access m$
 ⟨proof⟩

lemma *access-mthd-simp* [simp]: $(access (mthd m)) = accmodi m$
 ⟨proof⟩

lemma *access-fld-simp* [simp]: $(access (fld f)) = accmodi f$
 ⟨proof⟩

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

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

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

lemma *ext-field-simp* [simp]: $(declclass f, fld f) = f$
 ⟨proof⟩

lemma *ext-method-simp* [simp]: $(declclass m, mthd m) = m$
 ⟨proof⟩

lemma *ext-mbr-simp* [simp]: $(declclass m, mbr m) = m$
 ⟨proof⟩

lemma *fname-simp*[simp]: $fname (n,c) = n$
 ⟨proof⟩

lemma *declclassf-simp*[simp]: $declclassf (n,c) = c$
 ⟨proof⟩

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

$\langle proof \rangle$

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

$\langle proof \rangle$

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

$\langle proof \rangle$

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

$\langle proof \rangle$

lemma $accomdi-methdMembr-simp[simp]$: $accomdi (methdMembr m) = accomdi m$

$\langle proof \rangle$

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

$\langle proof \rangle$

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

$\langle proof \rangle$

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

$\langle proof \rangle$

lemma $accomdi-method-simp[simp]$: $accomdi (method sig m) = accomdi m$

$\langle proof \rangle$

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

$\langle proof \rangle$

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) = *fld* 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 ⇒ ty

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

defs (overloaded)
decl-resTy-def:
resTy (m::('a::has-resTy) decl-scheme) ≡ 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)
 ≡ *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))
 ≡ *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: $resTy\ p \equiv resTy\ (snd\ p)$

lemma *pair-resTy-simp*[simp]: $resTy\ (x,m) = resTy\ m$
 ⟨proof⟩

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

lemma *resTy-mthd* [simp]: $resTy\ (mthd\ m) = resTy\ m$
 ⟨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::

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

$G \vdash membr$ *inheritable-in* *pack*

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

syntax

Method-inheritable-in::

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

translations

$G \vdash Method\ m$ *inheritable-in* $p \equiv G \vdash methdMembr\ m$ *inheritable-in* p

syntax

Methd-inheritable-in::

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

translations

$G \vdash Methd\ s\ m$ *inheritable-in* $p \equiv G \vdash (method\ s\ m)$ *inheritable-in* p

declared-in/undeclared-in

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

cdeclaredmethd $G\ C$

$\equiv (case\ class\ G\ C\ of$
 None $\Rightarrow \lambda\ sig.\ None$

| *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 λ *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

$G \vdash$ *Method m declared-in C* \equiv $G \vdash$ *mdecl (methd m) 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

$G \vdash$ *Methd s m declared-in C* \equiv $G \vdash$ *mdecl (s,methd m) declared-in C*

lemma declared-in-classD:

$G \vdash m$ *declared-in C* \Longrightarrow *is-class G C*

<proof>

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**inductive**

members :: *prog* \Rightarrow (*qname* \times *memberdecl*) \Rightarrow *qname* \Rightarrow *bool*

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

for *G* :: *prog*

where

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

| *Inherited*: $\llbracket G \vdash m$ *inheritable-in (pid C)*; $G \vdash$ *memberid m undeclared-in C*;
 $G \vdash C <_{C_1} S$; $G \vdash$ (*Class S*) *accessible-in (pid C)*; $G \vdash m$ *member-of S*
 $\rrbracket \Longrightarrow G \vdash m$ *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:: $prog \Rightarrow (qname \times mdecl) \Rightarrow qname \Rightarrow bool$
 $(- \vdash Method - member'-of - [61,61,61] 60)$

translations

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

syntax

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

translations

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

syntax

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

translations

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

constdefs

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

$G \vdash C\ inherits\ m$

$\equiv G \vdash m\ inheritable-in\ (pid\ C) \wedge G \vdash memberid\ m\ undeclared-in\ C \wedge$
 $(\exists S. G \vdash C \prec_{C1} S \wedge G \vdash (Class\ S)\ accessible-in\ (pid\ C) \wedge G \vdash m\ member-of\ S)$

lemma *inherits-member*: $G \vdash C\ inherits\ m \implies G \vdash m\ member-of\ C$

<proof>

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

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

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

translations

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

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

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

lemma *member-inD*: $G \vdash m \text{ member-in } C$
 $\implies \exists \text{ prov}C. G \vdash C \preceq_C \text{ prov}C \wedge G \vdash m \text{ member-of } \text{prov}C$
 ⟨proof⟩

lemma *member-inI*: $\llbracket G \vdash m \text{ member-of } \text{prov}C; G \vdash C \preceq_C \text{ prov}C \rrbracket \implies G \vdash m \text{ member-in } C$
 ⟨proof⟩

lemma *member-of-to-member-in*: $G \vdash m \text{ member-of } C \implies G \vdash m \text{ member-in } C$
 ⟨proof⟩

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)

inductive

stat-overridesR :: $\text{prog} \Rightarrow (\text{qname} \times \text{mdecl}) \Rightarrow (\text{qname} \times \text{mdecl}) \Rightarrow \text{bool}$
 (- ⊢ - overrides_S - [61,61,61] 60)
 for G :: prog
 where

Direct: $\llbracket \neg \text{ is-static } \text{new}; \text{msig } \text{new} = \text{msig } \text{old};$
 $G \vdash \text{Method } \text{new} \text{ declared-in } (\text{declclass } \text{new});$
 $G \vdash \text{Method } \text{old} \text{ declared-in } (\text{declclass } \text{old});$
 $G \vdash \text{Method } \text{old} \text{ inheritable-in } \text{pid} (\text{declclass } \text{new});$
 $G \vdash (\text{declclass } \text{new}) \prec_{C_1} \text{superNew};$
 $G \vdash \text{Method } \text{old} \text{ member-of } \text{superNew}$
 $\rrbracket \implies G \vdash \text{new} \text{ overrides}_S \text{old}$

| *Indirect*: $\llbracket G \vdash \text{new} \text{ overrides}_S \text{inter}; G \vdash \text{inter} \text{ overrides}_S \text{old} \rrbracket$
 $\implies G \vdash \text{new} \text{ overrides}_S \text{old}$

Dynamic overriding (used during the typecheck of the compiler)

inductive

overridesR :: $\text{prog} \Rightarrow (\text{qname} \times \text{mdecl}) \Rightarrow (\text{qname} \times \text{mdecl}) \Rightarrow \text{bool}$
 (- ⊢ - overrides - [61,61,61] 60)
 for G :: prog
 where

Direct: $\llbracket \neg \text{ is-static } \text{new}; \neg \text{ is-static } \text{old}; \text{accommodi } \text{new} \neq \text{Private};$
 $\text{msig } \text{new} = \text{msig } \text{old};$
 $G \vdash (\text{declclass } \text{new}) \prec_C (\text{declclass } \text{old});$
 $G \vdash \text{Method } \text{new} \text{ declared-in } (\text{declclass } \text{new});$
 $G \vdash \text{Method } \text{old} \text{ declared-in } (\text{declclass } \text{old});$
 $G \vdash \text{Method } \text{old} \text{ inheritable-in } \text{pid} (\text{declclass } \text{new});$
 $G \vdash \text{resTy } \text{new} \preceq \text{resTy } \text{old}$
 $\rrbracket \implies G \vdash \text{new} \text{ overrides } \text{old}$

| *Indirect*: $\llbracket G \vdash \text{new} \text{ overrides } \text{inter}; G \vdash \text{inter} \text{ overrides } \text{old} \rrbracket$
 $\implies G \vdash \text{new} \text{ overrides } \text{old}$

syntax

sig-stat-overrides ::
 $\text{prog} \Rightarrow \text{sig} \Rightarrow (\text{qname} \times \text{methd}) \Rightarrow (\text{qname} \times \text{methd}) \Rightarrow \text{bool}$

$$(-, \vdash - \text{overrides}_S - [61, 61, 61, 61] \ 60)$$
translations

$$G, s \vdash \text{new overrides}_S \text{ old} \rightarrow G \vdash (\text{qmdecl } s \text{ new}) \text{ overrides}_S (\text{qmdecl } s \text{ old})$$
syntax

$$\text{sig-overrides}:: \text{prog} \Rightarrow \text{sig} \Rightarrow (\text{qname} \times \text{methd}) \Rightarrow (\text{qname} \times \text{methd}) \Rightarrow \text{bool}$$

$$(-, \vdash - \text{overrides} - [61, 61, 61, 61] \ 60)$$
translations

$$G, s \vdash \text{new overrides} \text{ old} \rightarrow G \vdash (\text{qmdecl } s \text{ new}) \text{ overrides} (\text{qmdecl } s \text{ old})$$
Hiding**constdefs** *hides*::
$$\text{prog} \Rightarrow (\text{qname} \times \text{mdecl}) \Rightarrow (\text{qname} \times \text{mdecl}) \Rightarrow \text{bool}$$

$$(\vdash - \text{hides} - [61, 61, 61] \ 60)$$

$$G \vdash \text{new hides} \text{ old}$$

$$\equiv \text{is-static new} \wedge \text{msig new} = \text{msig old} \wedge$$

$$G \vdash (\text{declclass new}) \prec_C (\text{declclass old}) \wedge$$

$$G \vdash \text{Method new declared-in} (\text{declclass new}) \wedge$$

$$G \vdash \text{Method old declared-in} (\text{declclass old}) \wedge$$

$$G \vdash \text{Method old inheritable-in pid} (\text{declclass new})$$
syntax

$$\text{sig-hides}:: \text{prog} \Rightarrow \text{sig} \Rightarrow (\text{qname} \times \text{mdecl}) \Rightarrow (\text{qname} \times \text{mdecl}) \Rightarrow \text{bool}$$

$$(-, \vdash - \text{hides} - [61, 61, 61, 61] \ 60)$$
translations

$$G, s \vdash \text{new hides} \text{ old} \rightarrow G \vdash (\text{qmdecl } s \text{ new}) \text{ hides} (\text{qmdecl } s \text{ old})$$
lemma *hidesI*:
$$\llbracket \text{is-static new}; \text{msig new} = \text{msig old};$$

$$G \vdash (\text{declclass new}) \prec_C (\text{declclass old});$$

$$G \vdash \text{Method new declared-in} (\text{declclass new});$$

$$G \vdash \text{Method old declared-in} (\text{declclass old});$$

$$G \vdash \text{Method old inheritable-in pid} (\text{declclass new})$$

$$\rrbracket \implies G \vdash \text{new hides} \text{ old}$$

<proof>

lemma *hidesD*:
$$\llbracket G \vdash \text{new hides} \text{ old} \rrbracket \implies$$

$$\text{declclass new} \neq \text{Object} \wedge \text{is-static new} \wedge \text{msig new} = \text{msig old} \wedge$$

$$G \vdash (\text{declclass new}) \prec_C (\text{declclass old}) \wedge$$

$$G \vdash \text{Method new declared-in} (\text{declclass new}) \wedge$$

$$G \vdash \text{Method old declared-in} (\text{declclass old})$$

<proof>

lemma *overrides-commonD*:
$$\llbracket G \vdash \text{new overrides} \text{ old} \rrbracket \implies$$

$$\text{declclass new} \neq \text{Object} \wedge \neg \text{is-static new} \wedge \neg \text{is-static old} \wedge$$

$$\text{accmodi new} \neq \text{Private} \wedge$$

$$\text{msig new} = \text{msig old} \wedge$$

$$G \vdash (\text{declclass new}) \prec_C (\text{declclass old}) \wedge$$

$$G \vdash \text{Method new declared-in} (\text{declclass new}) \wedge$$

$$G \vdash \text{Method old declared-in} (\text{declclass old})$$

<proof>

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::

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

$G \vdash \text{membr in class permits-acc-from accclass}$

$\equiv (\text{case } (\text{accmodi membr}) \text{ of}$
 $\quad \text{Private} \Rightarrow (\text{declclass membr} = \text{aclass})$
 $\quad | \text{Package} \Rightarrow (\text{pid } (\text{declclass membr}) = \text{pid accclass})$
 $\quad | \text{Protected} \Rightarrow (\text{pid } (\text{declclass membr}) = \text{pid accclass})$
 $\quad \vee$
 $\quad (G \vdash \text{aclass} \prec_C \text{declclass membr}$
 $\quad \wedge (G \vdash \text{class} \preceq_C \text{aclass} \vee \text{is-static membr}))$
 $\quad | \text{Public} \Rightarrow \text{True})$

The subcondition of the *Protected* case: $G \vdash \text{aclass} \prec_C \text{declclass membr}$ could also be relaxed to: $G \vdash \text{aclass} \preceq_C \text{declclass membr}$ 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.

inductive

accessible-fromR :: $\text{prog} \Rightarrow \text{qname} \Rightarrow (\text{qname} \times \text{memberdecl}) \Rightarrow \text{qname} \Rightarrow \text{bool}$
and *accessible-from* :: $\text{prog} \Rightarrow (\text{qname} \times \text{memberdecl}) \Rightarrow \text{qname} \Rightarrow \text{qname} \Rightarrow \text{bool}$
 $(- \vdash - \text{of } - \text{accessible}'\text{-from } - [61,61,61,61] 60)$
and *method-accessible-from* :: $\text{prog} \Rightarrow (\text{qname} \times \text{mdecl}) \Rightarrow \text{qname} \Rightarrow \text{qname} \Rightarrow \text{bool}$
 $(- \vdash \text{Method } - \text{of } - \text{accessible}'\text{-from } - [61,61,61,61] 60)$
for G :: prog **and** aclass :: qname

where

$G \vdash \text{membr of } cls \text{ accessible-from } accclass \equiv \text{accessible-fromR } G \text{ } accclass \text{ membr } cls$

| $G \vdash \text{Method } m \text{ of } cls \text{ accessible-from } accclass \equiv \text{accessible-fromR } G \text{ } accclass \text{ (methdMembr } m) \text{ } cls$

| *Immediate*: $\llbracket G \vdash \text{membr member-of class};$
 $G \vdash (\text{Class } class) \text{ accessible-in } (pid \text{ } accclass);$
 $G \vdash \text{membr in class permits-acc-from } accclass$
 $\rrbracket \implies G \vdash \text{membr of class accessible-from } accclass$

| *Overriding*: $\llbracket G \vdash \text{membr member-of class};$
 $G \vdash (\text{Class } class) \text{ accessible-in } (pid \text{ } accclass);$
 $\text{membr}=(C, mdecl \text{ } new);$
 $G \vdash (C, new) \text{ overrides } old;$
 $G \vdash \text{class } \prec_C \text{ } supr;$
 $G \vdash \text{Method } old \text{ of } supr \text{ accessible-from } accclass$
 $\rrbracket \implies G \vdash \text{membr of class accessible-from } accclass$

syntax

method-accessible-from::

$prog \Rightarrow sig \Rightarrow (qname \times \text{method}) \Rightarrow qname \Rightarrow qname \Rightarrow bool$
 $(- \vdash \text{Method } - \text{ of } - \text{ accessible'-from } - [61,61,61,61,61] \ 60)$

translations

$G \vdash \text{Method } s \ m \text{ of } cls \text{ accessible-from } accclass$
 $\Rightarrow G \vdash (\text{method } s \ m) \text{ of } cls \text{ accessible-from } accclass$

syntax

field-accessible-from::

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

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

dyn-accessible-fromR :: $prog \Rightarrow qname \Rightarrow (qname \times \text{memberdecl}) \Rightarrow qname \Rightarrow bool$
and *dyn-accessible-from'* :: $prog \Rightarrow (qname \times \text{memberdecl}) \Rightarrow qname \Rightarrow qname \Rightarrow bool$
 $(- \vdash - \text{ in } - \text{ dyn'-accessible'-from } - [61,61,61,61] \ 60)$
and *method-dyn-accessible-from* :: $prog \Rightarrow (qname \times \text{mdecl}) \Rightarrow qname \Rightarrow qname \Rightarrow bool$
 $(- \vdash \text{Method } - \text{ in } - \text{ dyn'-accessible'-from } - [61,61,61,61] \ 60)$
for $G :: prog$ **and** $accclass :: qname$

where

$G \vdash \text{membr in } C \text{ dyn-accessible-from } accC \equiv \text{dyn-accessible-fromR } G \text{ } accC \text{ membr } C$

| $G \vdash \text{Method } m \text{ in } C \text{ dyn-accessible-from } accC \equiv \text{dyn-accessible-fromR } G \text{ } accC \text{ (methdMembr } m) \text{ } C$

| *Immediate*: $\llbracket G \vdash \text{membr member-in class};$
 $G \vdash \text{membr in class permits-acc-from } accclass$
 $\rrbracket \implies G \vdash \text{membr in class dyn-accessible-from } accclass$

| *Overriding*: $\llbracket G \vdash \text{membr member-in class};$
 $\text{membr}=(C, mdecl \text{ } new);$
 $G \vdash (C, new) \text{ overrides } old;$
 $G \vdash \text{class } \prec_C \text{ } supr;$
 $G \vdash \text{Method } old \text{ in } supr \text{ dyn-accessible-from } accclass$
 $\rrbracket \implies G \vdash \text{membr in class dyn-accessible-from } accclass$

syntax*methd-dyn-accessible-from*::
$$\text{prog} \Rightarrow \text{sig} \Rightarrow (\text{qname} \times \text{methd}) \Rightarrow \text{qname} \Rightarrow \text{qname} \Rightarrow \text{bool}$$

$$(- \vdash \text{Methd} \text{ - - in - dyn'-accessible'-from - [61,61,61,61,61] 60)$$
translations $G \vdash \text{Methd } s \ m \ \text{in } C \ \text{dyn-accessible-from } \text{acc}C$ $\Leftrightarrow G \vdash (\text{method } s \ m) \ \text{in } C \ \text{dyn-accessible-from } \text{acc}C$ **syntax***field-dyn-accessible-from*::
$$\text{prog} \Rightarrow \text{vname} \Rightarrow (\text{qname} \times \text{field}) \Rightarrow \text{qname} \Rightarrow \text{qname} \Rightarrow \text{bool}$$

$$(- \vdash \text{Field} \text{ - - in - dyn'-accessible'-from - [61,61,61,61,61] 60)$$
translations $G \vdash \text{Field } \text{fn } f \ \text{in } \text{dyn}C \ \text{dyn-accessible-from } \text{acc}C$ $\Leftrightarrow G \vdash (\text{fieldm } \text{fn } f) \ \text{in } \text{dyn}C \ \text{dyn-accessible-from } \text{acc}C$ **lemma** *accessible-from-commonD*: $G \vdash m \ \text{of } C \ \text{accessible-from } S$ $\Rightarrow G \vdash m \ \text{member-of } C \wedge G \vdash (\text{Class } C) \ \text{accessible-in } (\text{pid } S)$ $\langle \text{proof} \rangle$ **lemma** *unique-declaration*: $\llbracket G \vdash m \ \text{declared-in } C; G \vdash n \ \text{declared-in } C; \text{memberid } m = \text{memberid } n \rrbracket$ $\Rightarrow m = n$ $\langle \text{proof} \rangle$ **lemma** *declared-not-undeclared*: $G \vdash m \ \text{declared-in } C \Rightarrow \neg G \vdash \text{memberid } m \ \text{undeclared-in } C$ $\langle \text{proof} \rangle$ **lemma** *undeclared-not-declared*: $G \vdash \text{memberid } m \ \text{undeclared-in } C \Rightarrow \neg G \vdash m \ \text{declared-in } C$ $\langle \text{proof} \rangle$ **lemma** *not-undeclared-declared*:
$$\neg G \vdash \text{memb-}id \ \text{undeclared-in } C \Rightarrow (\exists m. G \vdash m \ \text{declared-in } C \wedge$$

$$\text{memb-}id = \text{memberid } m)$$
 $\langle \text{proof} \rangle$ **lemma** *unique-declared-in*: $\llbracket G \vdash m \ \text{declared-in } C; G \vdash n \ \text{declared-in } C; \text{memberid } m = \text{memberid } n \rrbracket$ $\Rightarrow m = n$ $\langle \text{proof} \rangle$ **lemma** *unique-member-of*:**assumes** n : $G \vdash n \ \text{member-of } C$ **and** m : $G \vdash m \ \text{member-of } C$ **and***eqid*: $\text{memberid } n = \text{memberid } m$ **shows** $n=m$ $\langle \text{proof} \rangle$

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*:
 $G \vdash m \text{ member-of } C \implies G \vdash C \preceq_C \text{ declclass } m$
 ⟨proof⟩

lemma *member-in-class-relation*:
 $G \vdash m \text{ member-in } C \implies G \vdash C \preceq_C \text{ declclass } m$
 ⟨proof⟩

lemma *stat-override-declclasses-relation*:
 $\llbracket G \vdash (\text{declclass } \text{new}) \prec_{C_1} \text{superNew}; G \vdash \text{Method } \text{old} \text{ member-of } \text{superNew} \rrbracket$
 $\implies G \vdash (\text{declclass } \text{new}) \prec_C (\text{declclass } \text{old})$
 ⟨proof⟩

lemma *stat-overrides-commonD*:
 $\llbracket G \vdash \text{new overrides}_S \text{old} \rrbracket \implies$
 $\text{declclass } \text{new} \neq \text{Object} \wedge \neg \text{is-static } \text{new} \wedge \text{msig } \text{new} = \text{msig } \text{old} \wedge$
 $G \vdash (\text{declclass } \text{new}) \prec_C (\text{declclass } \text{old}) \wedge$
 $G \vdash \text{Method } \text{new} \text{ declared-in } (\text{declclass } \text{new}) \wedge$
 $G \vdash \text{Method } \text{old} \text{ declared-in } (\text{declclass } \text{old})$
 ⟨proof⟩

lemma *member-of-Package*:
 $\llbracket G \vdash m \text{ member-of } C; \text{accomodi } m = \text{Package} \rrbracket$
 $\implies \text{pid } (\text{declclass } m) = \text{pid } C$
 ⟨proof⟩

lemma *member-in-declC*: $G \vdash m \text{ member-in } C \implies G \vdash m \text{ member-in } (\text{declclass } m)$
 ⟨proof⟩

lemma *dyn-accessible-from-commonD*: $G \vdash m \text{ in } C \text{ dyn-accessible-from } S$
 $\implies G \vdash m \text{ member-in } C$
 ⟨proof⟩

lemma *no-Private-stat-override*:

$\llbracket G \vdash \text{new overrides}_S \text{ old} \rrbracket \implies \text{accmodi old} \neq \text{Private}$
 ⟨proof⟩

lemma *no-Private-override*: $\llbracket G \vdash \text{new overrides old} \rrbracket \implies \text{accmodi old} \neq \text{Private}$
 ⟨proof⟩

lemma *permits-acc-inheritance*:
 $\llbracket G \vdash m \text{ in } \text{statC} \text{ permits-acc-from } \text{accC}; G \vdash \text{dynC} \preceq_C \text{statC} \rrbracket$
 $\implies G \vdash m \text{ in } \text{dynC} \text{ permits-acc-from } \text{accC}$
 ⟨proof⟩

lemma *permits-acc-static-declC*:
 $\llbracket G \vdash m \text{ in } C \text{ permits-acc-from } \text{accC}; G \vdash m \text{ member-in } C; \text{is-static } m \rrbracket$
 $\implies G \vdash m \text{ in } (\text{declclass } m) \text{ permits-acc-from } \text{accC}$
 ⟨proof⟩

lemma *dyn-accessible-from-static-declC*:
assumes $\text{acc-C}: G \vdash m \text{ in } C \text{ dyn-accessible-from } \text{accC}$ **and**
 $\text{static: is-static } m$
shows $G \vdash m \text{ in } (\text{declclass } m) \text{ dyn-accessible-from } \text{accC}$
 ⟨proof⟩

lemma *field-accessible-fromD*:
 $\llbracket G \vdash \text{membr of } C \text{ accessible-from } \text{accC}; \text{is-field } \text{membr} \rrbracket$
 $\implies G \vdash \text{membr member-of } C \wedge$
 $G \vdash (\text{Class } C) \text{ accessible-in } (\text{pid } \text{accC}) \wedge$
 $G \vdash \text{membr in } C \text{ permits-acc-from } \text{accC}$
 ⟨proof⟩

lemma *field-accessible-from-permits-acc-inheritance*:
 $\llbracket G \vdash \text{membr of } \text{statC} \text{ accessible-from } \text{accC}; \text{is-field } \text{membr}; G \vdash \text{dynC} \preceq_C \text{statC} \rrbracket$
 $\implies G \vdash \text{membr in } \text{dynC} \text{ permits-acc-from } \text{accC}$
 ⟨proof⟩

lemma *accessible-fieldD*:
 $\llbracket G \vdash \text{membr of } C \text{ accessible-from } \text{accC}; \text{is-field } \text{membr} \rrbracket$
 $\implies G \vdash \text{membr member-of } C \wedge$
 $G \vdash (\text{Class } C) \text{ accessible-in } (\text{pid } \text{accC}) \wedge$
 $G \vdash \text{membr in } C \text{ permits-acc-from } \text{accC}$
 ⟨proof⟩

lemma *member-of-Private*:
 $\llbracket G \vdash m \text{ member-of } C; \text{accmodi } m = \text{Private} \rrbracket \implies \text{declclass } m = C$
 ⟨proof⟩

lemma *member-of-subclseq-declC*:

$G \vdash m$ member-of $C \implies G \vdash C \preceq_C \text{ declclass } m$
 ⟨proof⟩

lemma *member-of-inheritance*:

assumes m : $G \vdash m$ member-of D **and**
 $\text{subclseq-}D\text{-}C$: $G \vdash D \preceq_C C$ **and**
 $\text{subclseq-}C\text{-}m$: $G \vdash C \preceq_C \text{ declclass } m$ **and**
 ws : $ws\text{-prog } G$
shows $G \vdash m$ member-of C
 ⟨proof⟩

lemma *member-of-subcls*:

assumes old : $G \vdash old$ member-of C **and**
 new : $G \vdash new$ member-of D **and**
 $eqid$: $\text{memberid } new = \text{memberid } old$ **and**
 $\text{subclseq-}D\text{-}C$: $G \vdash D \preceq_C C$ **and**
 subcls-new-old : $G \vdash \text{ declclass } new \prec_C \text{ declclass } old$ **and**
 ws : $ws\text{-prog } G$
shows $G \vdash D \prec_C C$
 ⟨proof⟩

corollary *member-of-overrides-subcls*:

$\llbracket G \vdash \text{Methd } sig \text{ old member-of } C; G \vdash \text{Methd } sig \text{ new member-of } D; G \vdash D \preceq_C C;$
 $G, sig \vdash new \text{ overrides } old; ws\text{-prog } G \rrbracket$
 $\implies G \vdash D \prec_C C$
 ⟨proof⟩

corollary *member-of-stat-overrides-subcls*:

$\llbracket G \vdash \text{Methd } sig \text{ old member-of } C; G \vdash \text{Methd } sig \text{ new member-of } D; G \vdash D \preceq_C C;$
 $G, sig \vdash new \text{ overrides}_S old; ws\text{-prog } G \rrbracket$
 $\implies G \vdash D \prec_C C$
 ⟨proof⟩

lemma *inherited-field-access*:

assumes stat-acc : $G \vdash \text{ membr of stat}C$ accessible-from $\text{acc}C$ **and**
 is-field : is-field membr **and**
 subclseq : $G \vdash \text{ dyn}C \preceq_C \text{ stat}C$
shows $G \vdash \text{ membr in dyn}C$ dyn-accessible-from $\text{acc}C$
 ⟨proof⟩

lemma *accessible-inheritance*:

assumes stat-acc : $G \vdash m$ of $\text{stat}C$ accessible-from $\text{acc}C$ **and**
 subclseq : $G \vdash \text{ dyn}C \preceq_C \text{ stat}C$ **and**
 $\text{member-dyn}C$: $G \vdash m$ member-of $\text{dyn}C$ **and**
 $\text{dyn}C\text{-acc}$: $G \vdash (\text{Class } \text{dyn}C)$ accessible-in $(\text{pid } \text{acc}C)$
shows $G \vdash m$ of $\text{dyn}C$ accessible-from $\text{acc}C$
 ⟨proof⟩

fields and methods

types

$f\text{spec} = v\text{name} \times q\text{name}$

translations

$$fspec \leq (type) \ vname \times \ qname$$
constdefs

$$imethds:: \ prog \Rightarrow \ qname \Rightarrow \ (sig, qname \times \ mhead) \ tables$$

$$imethds \ G \ I$$

$$\equiv \ iface-rec \ (G, I) \\ (\lambda I \ i \ ts. \ (Un-tables \ ts) \oplus \oplus \\ (o2s \circ \ table-of \ (map \ (\lambda(s, m). \ (s, I, m)) \ (imethds \ i))))$$

methods of an interface, with overriding and inheritance, cf. 9.2

constdefs

$$accimethds:: \ prog \Rightarrow \ pname \Rightarrow \ qname \Rightarrow \ (sig, qname \times \ mhead) \ tables$$

$$accimethds \ G \ pack \ I$$

$$\equiv \ if \ G \vdash \ I \text{face } I \text{ accessible-in } pack \\ \text{then } imethds \ G \ I \\ \text{else } \lambda k. \ \{\}$$

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

$$dynmethd:: \ prog \Rightarrow \ qname \Rightarrow \ qname \Rightarrow \ (sig, qname \times \ methd) \ table$$

$$dynmethd \ G \ statC \ dynC$$

$$\equiv \ \lambda \ sig. \\ (if \ G \vdash \ dynC \ \leq_C \ statC \\ \text{then } (case \ methd \ G \ statC \ sig \ of \\ \ \ None \Rightarrow \ None \\ \ | \ Some \ statM \\ \ \Rightarrow \ (class-rec \ (G, dynC) \ empty \\ \ (\lambda C \ c \ subcls-mthds. \\ \ \ subcls-mthds \\ \ ++ \\ \ filter-tab$$

$$\begin{aligned}
& (\lambda - \text{dynM}. G, \text{sig} \vdash \text{dynM overrides statM} \vee \text{dynM} = \text{statM}) \\
& (\text{methd } G \ C))) \\
&) \ \text{sig} \\
&) \\
& \text{else None})
\end{aligned}$$

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*

$$\begin{aligned}
& \equiv \lambda \ \text{sig}. \ \text{if } \text{imethds } G \ I \ \text{sig} \neq \{\} \\
& \quad \text{then } \text{methd } G \ \text{dynC} \ \text{sig} \\
& \quad \text{else } \text{dynmethd } G \ \text{Object} \ \text{dynC} \ \text{sig}
\end{aligned}$$

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*

$$\begin{aligned}
& \equiv (\text{case } \text{statT} \ \text{of} \\
& \quad \text{NullT} \quad \Rightarrow \text{empty} \\
& \quad | \ \text{IfaceT } I \quad \Rightarrow \text{dynimethd } G \ I \quad \text{dynC} \\
& \quad | \ \text{ClassT } \text{statC} \Rightarrow \text{dynmethd } G \ \text{statC} \ \text{dynC} \\
& \quad | \ \text{ArrayT } \text{ty} \quad \Rightarrow \text{dynmethd } G \ \text{Object} \ \text{dynC})
\end{aligned}$$

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*

$$\equiv \text{class-rec } (G, C) \ [] \ (\lambda C \ c \ ts. \ \text{map } (\lambda (n, t). \ ((n, C), t)) \ (\text{cfields } c) \ @ \ ts)$$

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*

$$\begin{aligned}
& \equiv \text{let } \text{field-tab} = \text{table-of}((\text{map } (\lambda ((n, d), f). (n, (d, f)))) \ (\text{fields } G \ C)) \\
& \quad \text{in } \text{filter-tab } (\lambda n \ (\text{declC}, f). \ G \vdash \ (\text{declC}, f \ \text{decl } (n, f)) \ \text{of } C \ \text{accessible-from } S) \\
& \quad \text{field-tab}
\end{aligned}$$

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

$is\text{-}methd :: prog \Rightarrow qname \Rightarrow sig \Rightarrow bool$
 $is\text{-}methd\ G \equiv \lambda C\ sig. is\text{-}class\ G\ C \wedge methd\ G\ C\ sig \neq None$

constdefs $efname :: ((vname \times qname) \times field) \Rightarrow (vname \times qname)$
 $efname \equiv fst$

lemma $efname\text{-}simp[simp]: efname\ (n, f) = n$
 $\langle proof \rangle$

19 imethds

lemma $imethds\text{-}rec: \llbracket iface\ G\ I = Some\ i; ws\text{-}prog\ G \rrbracket \implies$
 $imethds\ G\ I = Un\text{-}tables\ ((\lambda J. imethds\ G\ J)\text{'set}\ (isuperIfs\ i)) \oplus \oplus$
 $(o2s \circ table\text{-}of\ (map\ (\lambda(s, mh). (s, I, mh))\ (imethods\ i)))$

$\langle proof \rangle$

lemma $imethds\text{-}norec:$

$\llbracket iface\ G\ md = Some\ i; ws\text{-}prog\ G; table\text{-}of\ (imethods\ i)\ sig = Some\ mh \rrbracket \implies$
 $(md, mh) \in imethds\ G\ md\ sig$
 $\langle proof \rangle$

lemma $imethds\text{-}declI: \llbracket m \in imethds\ G\ I\ sig; ws\text{-}prog\ G; is\text{-}iface\ G\ I \rrbracket \implies$
 $(\exists i. iface\ G\ (decliface\ m) = Some\ i \wedge$
 $table\text{-}of\ (imethods\ i)\ sig = Some\ (methd\ m)) \wedge$
 $(I, decliface\ m) \in (subint1\ G)^* \wedge m \in imethds\ G\ (decliface\ m)\ sig$
 $\langle proof \rangle$

lemma $imethds\text{-}cases$ [*consumes* \mathcal{I} , *case-names* *NewMethod* *InheritedMethod*]:

assumes $im: im \in imethds\ G\ I\ sig$ **and**

$ifI: iface\ G\ I = Some\ i$ **and**

$ws: ws\text{-}prog\ G$ **and**

$hyp\text{-}new: table\text{-}of\ (map\ (\lambda(s, mh). (s, I, mh))\ (imethods\ i))\ sig$
 $= Some\ im \implies P$ **and**

$hyp\text{-}inh: \bigwedge J. \llbracket J \in set\ (isuperIfs\ i); im \in imethds\ G\ J\ sig \rrbracket \implies P$

shows P

$\langle proof \rangle$

20 accimethd

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

$G \vdash iface\ I\ accessible\text{-}in\ pack \implies accimethds\ G\ pack\ I = imethds\ G\ I$
 $\langle proof \rangle$

lemma $accimethdsD:$

$im \in accimethds\ G\ pack\ I\ sig$

$\implies im \in imethds\ G\ I\ sig \wedge G \vdash iface\ I\ accessible\text{-}in\ pack$

$\langle \text{proof} \rangle$

lemma *accimethdsI*:

$\llbracket im \in imethds\ G\ I\ sig; G \vdash I\ \text{iface}\ I\ \text{accessible-in}\ pack \rrbracket$

$\implies im \in accimethds\ G\ pack\ I\ sig$

$\langle \text{proof} \rangle$

21 method

lemma *method-rec*: $\llbracket class\ G\ C = Some\ c; ws\text{-prog}\ G \rrbracket \implies$
 $method\ G\ C$

$= (if\ C = Object$

$then\ empty$

$else\ filter\text{-tab}\ (\lambda sig\ m. G \vdash C\ \text{inherits}\ method\ sig\ m)$

$(method\ G\ (super\ c))$)

$++\ table\text{-of}\ (map\ (\lambda(s,m). (s,C,m))\ (methods\ c))$

$\langle \text{proof} \rangle$

lemma *method-norec*:

$\llbracket class\ G\ declC = Some\ c; ws\text{-prog}\ G; table\text{-of}\ (methods\ c)\ sig = Some\ m \rrbracket$

$\implies method\ G\ declC\ sig = Some\ (declC,\ m)$

$\langle \text{proof} \rangle$

lemma *method-declC*:

$\llbracket method\ G\ C\ sig = Some\ m; ws\text{-prog}\ G; is\text{-class}\ G\ C \rrbracket \implies$

$(\exists d. class\ G\ (declclass\ m) = Some\ d \wedge table\text{-of}\ (methods\ d)\ sig = Some\ (mthd\ m)) \wedge$

$G \vdash C \preceq_C\ (declclass\ m) \wedge method\ G\ (declclass\ m)\ sig = Some\ m$

$\langle \text{proof} \rangle$

lemma *method-inheritedD*:

$\llbracket class\ G\ C = Some\ c; ws\text{-prog}\ G; method\ G\ C\ sig = Some\ m \rrbracket$

$\implies (declclass\ m \neq C \longrightarrow G \vdash C\ \text{inherits}\ method\ sig\ m)$

$\langle \text{proof} \rangle$

lemma *method-diff-cl*s:

$\llbracket ws\text{-prog}\ G; is\text{-class}\ G\ C; is\text{-class}\ G\ D;$

$method\ G\ C\ sig = m; method\ G\ D\ sig = n; m \neq n$

$\rrbracket \implies C \neq D$

$\langle \text{proof} \rangle$

lemma *method-declared-inI*:

$\llbracket table\text{-of}\ (methods\ c)\ sig = Some\ m; class\ G\ C = Some\ c \rrbracket$

$\implies G \vdash m\ \text{decl}\ (sig,m)\ \text{declared-in}\ C$

$\langle \text{proof} \rangle$

lemma *method-declared-in-declclass*:

$\llbracket method\ G\ C\ sig = Some\ m; ws\text{-prog}\ G; is\text{-class}\ G\ C \rrbracket$

$\implies G \vdash Method\ sig\ m\ \text{declared-in}\ (declclass\ m)$

$\langle \text{proof} \rangle$

lemma *member-method*:

assumes *member-of*: $G \vdash \text{Methd sig } m \text{ member-of } C$ **and**
ws: *ws-prog* G
shows *methd* $G C \text{ sig} = \text{Some } m$
 ⟨*proof*⟩

lemma *finite-method:ws-prog* $G \implies \text{finite } \{\text{methd } G C \text{ sig} \mid \text{sig } C. \text{is-class } G C\}$
 ⟨*proof*⟩

lemma *finite-dom-method*:

$\llbracket \text{ws-prog } G; \text{is-class } G C \rrbracket \implies \text{finite } (\text{dom } (\text{methd } G C))$
 ⟨*proof*⟩

22 accmethod

lemma *accmethod-SomeD*:

accmethod $G S C \text{ sig} = \text{Some } m$
 $\implies \text{methd } G C \text{ sig} = \text{Some } m \wedge G \vdash \text{method sig } m \text{ of } C \text{ accessible-from } S$
 ⟨*proof*⟩

lemma *accmethod-SomeI*:

$\llbracket \text{methd } G C \text{ sig} = \text{Some } m; G \vdash \text{method sig } m \text{ of } C \text{ accessible-from } S \rrbracket$
 $\implies \text{accmethod } G S C \text{ sig} = \text{Some } m$
 ⟨*proof*⟩

lemma *accmethod-declC*:

$\llbracket \text{accmethod } G S C \text{ sig} = \text{Some } m; \text{ws-prog } G; \text{is-class } G C \rrbracket \implies$
 $(\exists d. \text{class } G (\text{declclass } m) = \text{Some } d \wedge$
 $\text{table-of } (\text{methods } d) \text{ sig} = \text{Some } (\text{methd } m)) \wedge$
 $G \vdash C \preceq_C (\text{declclass } m) \wedge \text{methd } G (\text{declclass } m) \text{ sig} = \text{Some } m \wedge$
 $G \vdash \text{method sig } m \text{ of } C \text{ accessible-from } S$
 ⟨*proof*⟩

lemma *finite-dom-accmethod*:

$\llbracket \text{ws-prog } G; \text{is-class } G C \rrbracket \implies \text{finite } (\text{dom } (\text{accmethod } G S C))$
 ⟨*proof*⟩

23 dynmethod

lemma *dynmethod-rec*:

$\llbracket \text{class } G \text{ dynC} = \text{Some } c; \text{ws-prog } G \rrbracket \implies$
 $\text{dynmethod } G \text{ statC } \text{ dynC } \text{ sig}$
 $= (\text{if } G \vdash \text{dynC} \preceq_C \text{statC}$
 $\text{then } (\text{case } \text{methd } G \text{ statC } \text{ sig} \text{ of}$
 $\text{None} \Rightarrow \text{None}$
 $\mid \text{Some } \text{statM}$
 $\Rightarrow (\text{case } \text{methd } G \text{ dynC } \text{ sig} \text{ of}$
 $\text{None} \Rightarrow \text{dynmethod } G \text{ statC } (\text{super } c) \text{ sig}$
 $\mid \text{Some } \text{dynM} \Rightarrow$
 $(\text{if } G, \text{sig} \vdash \text{dynM} \text{ overrides } \text{statM} \vee \text{dynM} = \text{statM}$
 $\text{then } \text{Some } \text{dynM}$

else (dynmethd G statC (super c) sig)
)))
 else None)
 (is - \implies - \implies ?Dynmethd-def dynC sig = ?Dynmethd-rec dynC c sig)
 ⟨proof⟩

lemma *dynmethd-C-C*: \llbracket is-class G C; ws-prog G \rrbracket
 \implies dynmethd G C C sig = methd G C sig
 ⟨proof⟩

lemma *dynmethdSomeD*:
 \llbracket dynmethd G statC dynC sig = Some dynM; is-class G dynC; ws-prog G \rrbracket
 \implies $G \vdash$ dynC \preceq_C statC \wedge (\exists statM. methd G statC sig = Some statM)
 ⟨proof⟩

lemma *dynmethd-Some-cases* [consumes 3, case-names Static Overrides]:
assumes dynM: dynmethd G statC dynC sig = Some dynM **and**
 is-cls-dynC: is-class G dynC **and**
 ws: ws-prog G **and**
 hyp-static: methd G statC sig = Some dynM \implies P **and**
 hyp-override: \bigwedge statM. \llbracket methd G statC sig = Some statM; dynM \neq statM;
 G, sig \vdash dynM overrides statM $\rrbracket \implies$ P
shows P
 ⟨proof⟩

lemma *no-override-in-Object*:
assumes dynM: dynmethd G statC dynC sig = Some dynM **and**
 is-cls-dynC: is-class G dynC **and**
 ws: ws-prog G **and**
 statM: methd G statC sig = Some statM **and**
 neq-dynM-statM: dynM \neq statM
shows dynC \neq Object
 ⟨proof⟩

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-prog G **and**
 hyp-static: methd G statC sig = Some dynM \implies P **and**
 hyp-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-recursion: \llbracket dynC \neq Object;
 dynmethd G statC (super c) sig = Some dynM $\rrbracket \implies$ P
shows P
 ⟨proof⟩

lemma *dynmethd-declC*:
 \llbracket dynmethd G statC dynC sig = Some m;
 is-class G statC; ws-prog G

$\llbracket \implies$
 $(\exists d. \text{class } G (\text{declclass } m) = \text{Some } d \wedge \text{table-of } (\text{methods } d) \text{ sig} = \text{Some } (\text{mthd } m)) \wedge$
 $G \vdash \text{dynC} \preceq_C (\text{declclass } m) \wedge \text{methd } G (\text{declclass } m) \text{ sig} = \text{Some } m$
 <proof>

lemma *methd-Some-dynmethd-Some*:

assumes $\text{statM}: \text{methd } G \text{ statC sig} = \text{Some } \text{statM}$ **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 $\exists \text{dynM}. \text{dynmethd } G \text{ statC dynC sig} = \text{Some } \text{dynM}$
 (is ?P dynC)
 <proof>

lemma *dynmethd-cases* [consumes 4, case-names *Static Overrides*]:

assumes $\text{statM}: \text{methd } G \text{ statC sig} = \text{Some } \text{statM}$ **and**
 $\text{subclseq}: G \vdash \text{dynC} \preceq_C \text{statC}$ **and**
 $\text{is-clc-statC}: \text{is-class } G \text{ statC}$ **and**
 $\text{ws}: \text{ws-prog } G$ **and**
 $\text{hyp-static}: \text{dynmethd } G \text{ statC dynC sig} = \text{Some } \text{statM} \implies P$ **and**
 $\text{hyp-override}: \bigwedge \text{dynM}. \llbracket \text{dynmethd } G \text{ statC dynC sig} = \text{Some } \text{dynM};$
 $\text{dynM} \neq \text{statM};$
 $G, \text{sig} \vdash \text{dynM overrides statM} \rrbracket \implies P$
shows P
 <proof>

lemma *ws-dynmethd*:

assumes $\text{statM}: \text{methd } G \text{ statC sig} = \text{Some } \text{statM}$ **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
 $\exists \text{dynM}. \text{dynmethd } G \text{ statC dynC sig} = \text{Some } \text{dynM} \wedge$
 $\text{is-static } \text{dynM} = \text{is-static } \text{statM} \wedge G \vdash \text{resTy } \text{dynM} \preceq_{\text{resTy}} \text{statM}$
 <proof>

24 dynlookup

lemma *dynlookup-cases* [consumes 1, case-names *NullT IfaceT ClassT ArrayT*]:

$\llbracket \text{dynlookup } G \text{ statT dynC sig} = x;$
 $\llbracket \text{statT} = \text{NullT} \quad ; \text{empty sig} = x \quad \rrbracket \implies P;$
 $\bigwedge I. \llbracket \text{statT} = \text{IfaceT } I \quad ; \text{dynimethd } G \text{ } I \quad \text{dynC sig} = x \rrbracket \implies P;$
 $\bigwedge \text{statC}. \llbracket \text{statT} = \text{ClassT } \text{statC}; \text{dynmethd } G \text{ statC dynC sig} = x \rrbracket \implies P;$
 $\bigwedge \text{ty}. \llbracket \text{statT} = \text{ArrayT } \text{ty} \quad ; \text{dynmethd } G \text{ Object dynC sig} = x \rrbracket \implies P$
 $\rrbracket \implies P$
 <proof>

25 fields

lemma *fields-rec*: $\llbracket \text{class } G \text{ } C = \text{Some } c; \text{ws-prog } G \rrbracket \implies$
 $\text{fields } G \text{ } C = \text{map } (\lambda(\text{fn}, \text{ft}). ((\text{fn}, C), \text{ft})) (\text{cfields } c) @$
 (if $C = \text{Object}$ then \llbracket else $\text{fields } G (\text{super } c)$)
 <proof>

lemma *fields-norec*:

$\llbracket \text{class } G \text{ fd} = \text{Some } c; \text{ ws-prog } G; \text{ table-of } (\text{cfields } c) \text{ fn} = \text{Some } f \rrbracket$
 $\implies \text{table-of } (\text{fields } G \text{ fd}) (\text{fn}, \text{fd}) = \text{Some } f$
 ⟨proof⟩

lemma *table-of-fieldsD*:

$\text{table-of } (\text{map } (\lambda(\text{fn}, \text{ft}). ((\text{fn}, C), \text{ft})) (\text{cfields } c)) \text{ efn} = \text{Some } f$
 $\implies (\text{declclassf } \text{efn}) = C \wedge \text{table-of } (\text{cfields } c) (\text{fname } \text{efn}) = \text{Some } f$
 ⟨proof⟩

lemma *fields-declC*:

$\llbracket \text{table-of } (\text{fields } G \text{ C}) \text{ efn} = \text{Some } f; \text{ ws-prog } G; \text{ is-class } G \text{ C} \rrbracket \implies$
 $(\exists d. \text{class } G (\text{declclassf } \text{efn}) = \text{Some } d \wedge$
 $\text{table-of } (\text{cfields } d) (\text{fname } \text{efn}) = \text{Some } f) \wedge$
 $G \vdash C \preceq_C (\text{declclassf } \text{efn}) \wedge \text{table-of } (\text{fields } G (\text{declclassf } \text{efn})) \text{ efn} = \text{Some } f$
 ⟨proof⟩

lemma *fields-emptyI*: $\bigwedge y. \llbracket \text{ws-prog } G; \text{ class } G \text{ C} = \text{Some } c; \text{ cfields } c = [];$
 $C \neq \text{Object} \implies \text{class } G (\text{super } c) = \text{Some } y \wedge \text{fields } G (\text{super } c) = [] \rrbracket \implies$
 $\text{fields } G \text{ C} = []$
 ⟨proof⟩

lemma *fields-mono-lemma*:

$\llbracket x \in \text{set } (\text{fields } G \text{ C}); G \vdash D \preceq_C C; \text{ ws-prog } G \rrbracket$
 $\implies x \in \text{set } (\text{fields } G \text{ D})$
 ⟨proof⟩

lemma *ws-unique-fields-lemma*:

$\llbracket (\text{efn}, \text{fd}) \in \text{set } (\text{fields } G (\text{super } c)); \text{ fc} \in \text{set } (\text{cfields } c); \text{ ws-prog } G;$
 $\text{fname } \text{efn} = \text{fname } \text{fc}; \text{ declclassf } \text{efn} = C;$
 $\text{class } G \text{ C} = \text{Some } c; C \neq \text{Object}; \text{ class } G (\text{super } c) = \text{Some } d \rrbracket \implies R$
 ⟨proof⟩

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

$\bigwedge C c. \llbracket \text{class } G \text{ C} = \text{Some } c \rrbracket \implies \text{unique } (\text{cfields } c) \rrbracket \implies$
 $\text{unique } (\text{fields } G \text{ C})$
 ⟨proof⟩

26 accfield

lemma *accfield-fields*:

$\text{accfield } G \text{ S } C \text{ fn} = \text{Some } f$
 $\implies \text{table-of } (\text{fields } G \text{ C}) (\text{fn}, \text{declclass } f) = \text{Some } (\text{fld } f)$
 ⟨proof⟩

lemma *accfield-declC-is-class*:

$\llbracket \text{is-class } G \text{ C}; \text{ accfield } G \text{ S } C \text{ en} = \text{Some } (\text{fd}, f); \text{ ws-prog } G \rrbracket \implies$
 $\text{is-class } G \text{ fd}$

<proof>

lemma *accfield-accessibleD*:

accfield G S C fn = Some f \implies G-Field fn f of C accessible-from S
<proof>

27 is methd

lemma *is-methdI*:

$\llbracket \text{class } G \ C = \text{Some } y; \text{methd } G \ C \ \text{sig} = \text{Some } b \rrbracket \implies \text{is-methd } G \ C \ \text{sig}$
<proof>

lemma *is-methdD*:

is-methd G C sig \implies class G C \neq None \wedge methd G C sig \neq None
<proof>

lemma *finite-is-methd*:

ws-prog G \implies finite (Collect (split (is-methd G)))
<proof>

calculation of the superclasses of a class

constdefs

superclasses:: prog \Rightarrow qname \Rightarrow qname set
superclasses G C \equiv class-rec (G,C) {}
 (λ C c superclss. (if C=Object
 then {}
 else insert (super c) superclss))

lemma *superclasses-rec*: $\llbracket \text{class } G \ C = \text{Some } c; \text{ws-prog } G \rrbracket \implies$

superclasses G C
 = *(if (C=Object)*
 then {}
 else insert (super c) (superclasses G (super c)))
<proof>

lemma *superclasses-mono*:

$\llbracket G \vdash C \prec_C D; \text{ws-prog } G; \text{class } G \ C = \text{Some } c;$
 $\wedge C \ c. \llbracket \text{class } G \ C = \text{Some } c; C \neq \text{Object} \rrbracket \implies \exists \ sc. \text{class } G \ (\text{super } c) = \text{Some } sc;$
 $x \in \text{superclasses } G \ D$
 $\rrbracket \implies x \in \text{superclasses } G \ C$
<proof>

lemma *subclsEval*:

$\llbracket G \vdash C \prec_C D; \text{ws-prog } G; \text{class } G \ C = \text{Some } c;$
 $\wedge C \ c. \llbracket \text{class } G \ C = \text{Some } c; C \neq \text{Object} \rrbracket \implies \exists \ sc. \text{class } G \ (\text{super } c) = \text{Some } sc$
 $\rrbracket \implies D \in \text{superclasses } G \ C$
<proof>

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:: *qname* — current package and class name
lcl:: *lenv* — local environment

translations

lenv <= (*type*) (*lname*, *ty*) *table*
lenv <= (*type*) *lname* ⇒ *ty option*
env <= (*type*) (*prg*::*prog*, *cls*::*qname*, *lcl*::*lenv*)
env <= (*type*) (*prg*::*prog*, *cls*::*qname*, *lcl*::*lenv*, . . . :: 'a)

syntax

pkg :: *env* ⇒ *pname* — select the current package from an environment

translations

pkg e == *pid (cls e)*

Static overloading: maximally specific methods

types

emhead = *ref-ty* × *mhead*

— Some mnemonic selectors for *emhead*

constdefs

declrefT :: *emhead* ⇒ *ref-ty*

declrefT ≡ *fst*

mhd :: *emhead* ⇒ *mhead*

mhd ≡ *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 :: $prog \Rightarrow qname \Rightarrow qname \Rightarrow sig \Rightarrow emhead \ set$

Objectmheads :: $prog \Rightarrow qname \Rightarrow sig \Rightarrow emhead \ set$

accObjectmheads:: $prog \Rightarrow qname \Rightarrow ref-ty \Rightarrow sig \Rightarrow emhead \ set$

mheads :: $prog \Rightarrow qname \Rightarrow ref-ty \Rightarrow sig \Rightarrow emhead \ set$

defs

cmheads-def:

cmheads $G S C$

$\equiv \lambda sig. (\lambda (Cls, mthd). (ClassT Cls, (mhead mthd))) \text{ ' } o2s (accmethd G S C sig)$

Objectmheads-def:

Objectmheads $G S$

$\equiv \lambda sig. (\lambda (Cls, mthd). (ClassT Cls, (mhead mthd)))$

$\text{ ' } o2s (filter-tab (\lambda sig m. accmodi m \neq Private) (accmethd G S Object) sig)$

accObjectmheads-def:

accObjectmheads $G S T$

$\equiv \text{if } G \vdash RefT T \text{ accessible-in } (pid S)$

$\text{ then } Objectmheads G S$

$\text{ else } \lambda sig. \{ \}$

primrec

mheads $G S NullT = (\lambda sig. \{ \})$

mheads $G S (IfaceT I) = (\lambda sig. (\lambda (I, h). (IfaceT I, h)))$

$\text{ ' } accimethds G (pid S) I sig \cup$

$accObjectmheads G S (IfaceT I) sig)$

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 :: $prog \Rightarrow qname \Rightarrow ref-ty \Rightarrow sig \Rightarrow (emhead \times ty \ list) \ set$

appl-methds $G S rt \equiv \lambda sig.$

$\{ (mh, pTs') \mid mh \ pTs'. mh \in mheads G S rt \ (name=name \ sig, parTs=pTs') \wedge$
 $G \vdash (parTs \ sig) [\preceq] pTs' \}$

— 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

$max-spec \quad :: \text{prog} \Rightarrow \text{qname} \Rightarrow \text{ref-ty} \Rightarrow \text{sig} \Rightarrow (\text{emhead} \times \text{ty list}) \quad \text{set}$

$max-spec \ G \ S \ rt \ sig \equiv \{m. m \in \text{appl-methds} \ G \ S \ rt \ sig \wedge$
 $(\forall m' \in \text{appl-methds} \ G \ S \ rt \ sig. \text{more-spec} \ G \ m' \ m \longrightarrow m' = m)\}$

lemma *max-spec2appl-meths*:

$x \in max-spec \ G \ S \ T \ sig \implies x \in \text{appl-methds} \ G \ S \ T \ sig$

$\langle \text{proof} \rangle$

lemma *appl-methsD*: $(mh, pTs') \in \text{appl-methds} \ G \ S \ T \ (\text{name} = mn, \text{parTs} = pTs) \implies$

$mh \in \text{mheads} \ G \ S \ T \ (\text{name} = mn, \text{parTs} = pTs') \wedge G \vdash pTs [\preceq] pTs'$

$\langle \text{proof} \rangle$

lemma *max-spec2mheads*:

$max-spec \ G \ S \ rt \ (\text{name} = mn, \text{parTs} = pTs) = \text{insert} \ (mh, pTs') \ A$

$\implies mh \in \text{mheads} \ G \ S \ rt \ (\text{name} = mn, \text{parTs} = pTs') \wedge G \vdash pTs [\preceq] pTs'$

$\langle \text{proof} \rangle$

constdefs

$empty-dt \quad :: \text{dyn-ty}$

$empty-dt \equiv \lambda a. \text{None}$

$invmode \quad :: ('a::\text{type})\text{member-scheme} \Rightarrow \text{expr} \Rightarrow \text{inv-mode}$

$invmode \ m \ e \equiv \text{if } is-static \ m$

$\text{then } \text{Static}$

$\text{else if } e = \text{Super} \text{ then } \text{SuperM} \text{ else } \text{IntVir}$

lemma *invmode-nonstatic* [simp]:

$invmode \ (\text{access} = a, \text{static} = \text{False}, \dots = x) \ (\text{Acc} \ (\text{LVar} \ e)) = \text{IntVir}$

$\langle \text{proof} \rangle$

lemma *invmode-Static-eq* [simp]: $(invmode \ m \ e = \text{Static}) = is-static \ m$

$\langle \text{proof} \rangle$

lemma *invmode-IntVir-eq*: $(invmode \ m \ e = \text{IntVir}) = (\neg(is-static \ m) \wedge e \neq \text{Super})$

$\langle \text{proof} \rangle$

lemma *Null-staticD*:

$a' = \text{Null} \longrightarrow (is-static \ m) \implies invmode \ m \ e = \text{IntVir} \longrightarrow a' \neq \text{Null}$

$\langle \text{proof} \rangle$

Typing for unary operations

consts *unop-type* $:: \text{unop} \Rightarrow \text{prim-ty}$

primrec

unop-type $UPlus \quad = \text{Integer}$

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*))

$wt\text{-binop } G \text{ Or } t1 \ t2 = ((t1 = PrimT \ Boolean) \wedge (t2 = PrimT \ Boolean))$
 $wt\text{-binop } G \text{ CondAnd } t1 \ t2 = ((t1 = PrimT \ Boolean) \wedge (t2 = PrimT \ Boolean))$
 $wt\text{-binop } G \text{ CondOr } t1 \ t2 = ((t1 = PrimT \ Boolean) \wedge (t2 = PrimT \ Boolean))$

Typing for terms

types $tys = ty + ty \ list$
translations
 $tys \leq (type) \ ty + ty \ list$

inductive

$wt :: env \Rightarrow dyn\text{-}ty \Rightarrow [term, tys] \Rightarrow bool \ (-, \models\text{-} :: - [51, 51, 51, 51] \ 50)$
and $wt\text{-}stmt :: env \Rightarrow dyn\text{-}ty \Rightarrow stmt \Rightarrow bool \ (-, \models\text{-} :: \surd [51, 51, 51] \ 50)$
and $ty\text{-}expr :: env \Rightarrow dyn\text{-}ty \Rightarrow [expr, ty] \Rightarrow bool \ (-, \models\text{-} :: - [51, 51, 51, 51] \ 50)$
and $ty\text{-}var :: env \Rightarrow dyn\text{-}ty \Rightarrow [var, ty] \Rightarrow bool \ (-, \models\text{-} :: - [51, 51, 51, 51] \ 50)$
and $ty\text{-}exprs :: env \Rightarrow dyn\text{-}ty \Rightarrow [expr \ list, ty \ list] \Rightarrow bool$
 $(-, \models\text{-} :: \dot{-} [51, 51, 51, 51] \ 50)$

where

$E, dt \models s :: \surd \equiv E, dt \models In1r \ s :: Inl \ (PrimT \ Void)$
 $| \ E, dt \models e :: - \ T \equiv E, dt \models In1l \ e :: Inl \ T$
 $| \ E, dt \models e :: = \ T \equiv E, dt \models In2 \ e :: Inl \ T$
 $| \ E, dt \models e :: \dot{-} \ T \equiv E, dt \models In3 \ e :: Inr \ T$

— well-typed statements

$| \ Skip: \quad E, dt \models Skip :: \surd$

$| \ Expr: \llbracket E, dt \models e :: - \ T \rrbracket \Longrightarrow$
 $E, dt \models Expr \ e :: \surd$

— cf. 14.6

$| \ Lab: \ E, dt \models c :: \surd \Longrightarrow$
 $E, dt \models l \cdot c :: \surd$

$| \ Comp: \llbracket E, dt \models c1 :: \surd;$
 $\quad \quad \quad E, dt \models c2 :: \surd \rrbracket \Longrightarrow$
 $E, dt \models c1 ;; c2 :: \surd$

— cf. 14.8

$| \ If: \llbracket E, dt \models e :: - \ PrimT \ Boolean;$
 $\quad \quad \quad E, dt \models c1 :: \surd;$
 $\quad \quad \quad E, dt \models c2 :: \surd \rrbracket \Longrightarrow$
 $E, dt \models If \ (e) \ c1 \ Else \ c2 :: \surd$

— cf. 14.10

$| \ Loop: \llbracket E, dt \models e :: - \ PrimT \ Boolean;$
 $\quad \quad \quad E, dt \models c :: \surd \rrbracket \Longrightarrow$
 $E, dt \models l \cdot While \ (e) \ c :: \surd$

— cf. 14.13, 14.15, 14.16

$| \ Jmp: \quad E, dt \models Jmp \ jump :: \surd$

— cf. 14.16

$| \ Throw: \llbracket E, dt \models e :: - \ Class \ tn;$
 $\quad \quad \quad prg \ E \vdash tn \leq_C \ SXcpt \ Throwable \rrbracket \Longrightarrow$
 $E, dt \models Throw \ e :: \surd$

— cf. 14.18

$| \ Try: \llbracket E, dt \models c1 :: \surd; \ prg \ E \vdash tn \leq_C \ SXcpt \ Throwable;$
 $\quad \quad \quad lcl \ E \ (VName \ vn) = None; \ E \ (lcl \ := \ lcl \ E \ (VName \ vn) \mapsto \ Class \ tn) \rrbracket, dt \models c2 :: \surd$

\implies

$$E, dt \models \text{Try } c1 \text{ Catch}(tn \ vn) \ c2 :: \checkmark$$

— cf. 14.18

$$\begin{array}{l} | \text{Fin: } \llbracket E, dt \models c1 :: \checkmark; E, dt \models c2 :: \checkmark \rrbracket \implies \\ \quad E, dt \models c1 \text{ Finally } c2 :: \checkmark \end{array}$$

$$| \text{Init: } \llbracket \text{is-class } (prg \ E) \ C \rrbracket \implies$$

$$E, dt \models \text{Init } C :: \checkmark$$

— *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

$$\begin{array}{l} | \text{NewC: } \llbracket \text{is-acc-class } (prg \ E) \ (pkg \ E) \ C \rrbracket \implies \\ \quad E, dt \models \text{NewC } C :: \text{--Class } C \end{array}$$

— cf. 15.9

$$\begin{array}{l} | \text{NewA: } \llbracket \text{is-acc-type } (prg \ E) \ (pkg \ E) \ T; \\ \quad E, dt \models i :: \text{--PrimT Integer} \rrbracket \implies \\ \quad E, dt \models \text{New } T[i] :: \text{--T.} \end{array}$$

— cf. 15.15

$$\begin{array}{l} | \text{Cast: } \llbracket E, dt \models e :: \text{--T}; \text{is-acc-type } (prg \ E) \ (pkg \ E) \ T'; \\ \quad prg \ E \vdash T \preceq? T' \rrbracket \implies \\ \quad E, dt \models \text{Cast } T' \ e :: \text{--T}' \end{array}$$

— cf. 15.19.2

$$\begin{array}{l} | \text{Inst: } \llbracket E, dt \models e :: \text{--RefT } T; \text{is-acc-type } (prg \ E) \ (pkg \ E) \ (\text{RefT } T'); \\ \quad prg \ E \vdash \text{RefT } T \preceq? \text{RefT } T' \rrbracket \implies \\ \quad E, dt \models e \text{ InstOf } T' :: \text{--PrimT Boolean} \end{array}$$

— cf. 15.7.1

$$\begin{array}{l} | \text{Lit: } \llbracket \text{typeof } dt \ x = \text{Some } T \rrbracket \implies \\ \quad E, dt \models \text{Lit } x :: \text{--T} \end{array}$$

$$\begin{array}{l} | \text{UnOp: } \llbracket E, dt \models e :: \text{--T}e; \text{wt-unop } unop \ Te; T = \text{PrimT } (\text{unop-type } unop) \rrbracket \\ \implies \\ \quad E, dt \models \text{UnOp } unop \ e :: \text{--T} \end{array}$$

$$\begin{array}{l} | \text{BinOp: } \llbracket E, dt \models e1 :: \text{--T}1; E, dt \models e2 :: \text{--T}2; \text{wt-binop } (prg \ E) \ binop \ T1 \ T2; \\ \quad T = \text{PrimT } (\text{binop-type } binop) \rrbracket \\ \implies \\ \quad E, dt \models \text{BinOp } binop \ e1 \ e2 :: \text{--T} \end{array}$$

— cf. 15.10.2, 15.11.1

$$\begin{array}{l} | \text{Super: } \llbracket \text{lcl } E \ \text{This} = \text{Some } (\text{Class } C); C \neq \text{Object}; \\ \quad \text{class } (prg \ E) \ C = \text{Some } c \rrbracket \implies \\ \quad E, dt \models \text{Super} :: \text{--Class } (\text{super } c) \end{array}$$

— cf. 15.13.1, 15.10.1, 15.12

$$\begin{array}{l} | \text{Acc: } \llbracket E, dt \models va :: \text{--T} \rrbracket \implies \\ \quad E, dt \models \text{Acc } va :: \text{--T} \end{array}$$

— cf. 15.25, 15.25.1

$$\begin{array}{l} | \text{Ass: } \llbracket E, dt \models va :: \text{--T}; va \neq \text{LVar } \text{This}; \\ \quad E, dt \models v :: \text{--T}' \\ \quad prg \ E \vdash T' \preceq T \rrbracket \implies \\ \quad E, dt \models va := v :: \text{--T}' \end{array}$$

— cf. 15.24

| *Cond*: $\llbracket E, dt \models e0 :: - \text{PrimT Boolean};$
 $E, dt \models e1 :: - T1; E, dt \models e2 :: - T2;$
 $\text{prg } E \vdash T1 \preceq T2 \wedge T = T2 \vee \text{prg } E \vdash T2 \preceq T1 \wedge T = T1 \rrbracket \implies$
 $E, dt \models e0 \text{ ? } e1 : e2 :: - T$

— cf. 15.11.1, 15.11.2, 15.11.3

| *Call*: $\llbracket E, dt \models e :: - \text{RefT statT};$
 $E, dt \models ps :: \doteq pTs;$
 $\text{max-spec } (\text{prg } E) (\text{cls } E) \text{ statT } (\{ \text{name} = mn, \text{parTs} = pTs \})$
 $= \{ ((\text{statDeclT}, m), pTs') \}$
 $\rrbracket \implies$
 $E, dt \models \{ \text{cls } E, \text{statT}, \text{invmode } m \} e \cdot mn (\{ pTs' \} ps) :: - (\text{resTy } m)$

| *Method*: $\llbracket \text{is-class } (\text{prg } E) C;$
 $\text{methd } (\text{prg } E) C \text{ sig} = \text{Some } m;$
 $E, dt \models \text{Body } (\text{declclass } m) (\text{stmt } (\text{mbody } (\text{methd } m))) :: - T \rrbracket \implies$
 $E, dt \models \text{Method } C \text{ sig} :: - T$

— 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 $\text{pkg } E$. Only the static class must be accessible (enshured 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*: $\llbracket \text{is-class } (\text{prg } E) D;$
 $E, dt \models \text{blk} :: \checkmark;$
 $(\text{lcl } E) \text{ Result} = \text{Some } T;$
 $\text{is-type } (\text{prg } E) T \rrbracket \implies$
 $E, dt \models \text{Body } D \text{ blk} :: - T$

— The class D implementing the method must not directly be accessible from the current package $\text{pkg } E$, but can also be indirectly accessible due to inheritance (enshured 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*: $\llbracket \text{lcl } E \text{ vn} = \text{Some } T; \text{is-acc-type } (\text{prg } E) (\text{pkg } E) T \rrbracket \implies$
 $E, dt \models \text{LVar } \text{vn} :: = T$

— cf. 15.10.1

| *FVar*: $\llbracket E, dt \models e :: - \text{Class } C;$
 $\text{accfield } (\text{prg } E) (\text{cls } E) C \text{ fn} = \text{Some } (\text{statDeclC}, f) \rrbracket \implies$
 $E, dt \models \{ \text{cls } E, \text{statDeclC}, \text{is-static } f \} e \cdot \text{fn} :: = (\text{type } f)$

— cf. 15.12

| *AVar*: $\llbracket E, dt \models e :: - T. [];$
 $E, dt \models i :: - \text{PrimT Integer} \rrbracket \implies$
 $E, dt \models e \cdot [i] :: = T$

— well-typed expression lists

— cf. 15.11.???

| *Nil*: $E, dt \models [] :: \doteq []$

— cf. 15.11.???

| *Cons*: $\llbracket E, dt \models e :: - T;$
 $E, dt \models es :: \doteq Ts \rrbracket \implies$
 $E, dt \models e \# es :: \doteq T \# Ts$

syntax

$-wt \quad :: \text{env} \Rightarrow [\text{term}, \text{tys}] \Rightarrow \text{bool} \text{ (|-:-: [51,51,51] 50)}$
 $-wt\text{-stmt} \quad :: \text{env} \Rightarrow \text{stmt} \quad \Rightarrow \text{bool} \text{ (|-:-:<> [51,51] 50)}$
 $-ty\text{-expr} \quad :: \text{env} \Rightarrow [\text{expr}, \text{ty}] \Rightarrow \text{bool} \text{ (|-:-:- [51,51,51] 50)}$
 $-ty\text{-var} \quad :: \text{env} \Rightarrow [\text{var}, \text{ty}] \Rightarrow \text{bool} \text{ (|-:-:= [51,51,51] 50)}$
 $-ty\text{-exprs} \quad :: \text{env} \Rightarrow [\text{expr list},$
 $\quad \quad \quad \text{ty list}] \Rightarrow \text{bool} \text{ (|-:-:#- [51,51,51] 50)}$

syntax (*xsymbols*)

$-wt \quad :: \text{env} \Rightarrow [\text{term}, \text{tys}] \Rightarrow \text{bool} \text{ (+-:-: [51,51,51] 50)}$
 $-wt\text{-stmt} \quad :: \text{env} \Rightarrow \text{stmt} \quad \Rightarrow \text{bool} \text{ (+-:-:\surd [51,51] 50)}$
 $-ty\text{-expr} \quad :: \text{env} \Rightarrow [\text{expr}, \text{ty}] \Rightarrow \text{bool} \text{ (+-:-:- [51,51,51] 50)}$
 $-ty\text{-var} \quad :: \text{env} \Rightarrow [\text{var}, \text{ty}] \Rightarrow \text{bool} \text{ (+-:-:= [51,51,51] 50)}$
 $-ty\text{-exprs} \quad :: \text{env} \Rightarrow [\text{expr list},$
 $\quad \quad \quad \text{ty list}] \Rightarrow \text{bool} \text{ (+-:-:\simeq [51,51,51] 50)}$

translations

$E \vdash t :: T == E, \text{empty-dt} \models t :: T$
 $E \vdash s :: \surd == E \vdash \text{In1r } s :: \text{Inl } (\text{PrimT } \text{Void})$
 $E \vdash e :: -T == E \vdash \text{In1l } e :: \text{Inl } T$
 $E \vdash e :: =T == E \vdash \text{In2 } e :: \text{Inl } T$
 $E \vdash e :: \simeq T == E \vdash \text{In3 } e :: \text{Inr } T$

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*]

$\langle \text{ML} \rangle$

inductive-cases *wt-elim-cases* [*cases set*]:

$E, dt \models \text{In2 } (\text{LVar } vn) \quad :: T$
 $E, dt \models \text{In2 } (\{\text{accC}, \text{statDeclC}, s\} e..fn) :: T$
 $E, dt \models \text{In2 } (e.[i]) \quad :: T$
 $E, dt \models \text{In1l } (\text{NewC } C) \quad :: T$
 $E, dt \models \text{In1l } (\text{New } T'[i]) \quad :: T$
 $E, dt \models \text{In1l } (\text{Cast } T' e) \quad :: T$
 $E, dt \models \text{In1l } (e \text{ InstOf } T') \quad :: T$
 $E, dt \models \text{In1l } (\text{Lit } x) \quad :: T$
 $E, dt \models \text{In1l } (\text{UnOp } \text{unop } e) \quad :: T$
 $E, dt \models \text{In1l } (\text{BinOp } \text{binop } e1 e2) \quad :: T$
 $E, dt \models \text{In1l } (\text{Super}) \quad :: T$
 $E, dt \models \text{In1l } (\text{Acc } va) \quad :: T$
 $E, dt \models \text{In1l } (\text{Ass } va v) \quad :: T$
 $E, dt \models \text{In1l } (e0 ? e1 : e2) \quad :: T$
 $E, dt \models \text{In1l } (\{\text{accC}, \text{statT}, \text{mode}\} e.mn(\{pT'\}p)) :: T$
 $E, dt \models \text{In1l } (\text{Methd } C \text{ sig}) \quad :: T$
 $E, dt \models \text{In1l } (\text{Body } D \text{ blk}) \quad :: T$
 $E, dt \models \text{In3 } ([]) \quad :: Ts$
 $E, dt \models \text{In3 } (e \# es) \quad :: Ts$
 $E, dt \models \text{In1r } \text{Skip} \quad :: x$
 $E, dt \models \text{In1r } (\text{Expr } e) \quad :: x$
 $E, dt \models \text{In1r } (c1 ;; c2) \quad :: x$
 $E, dt \models \text{In1r } (l \cdot c) \quad :: x$
 $E, dt \models \text{In1r } (\text{If } (e) c1 \text{ Else } c2) \quad :: x$
 $E, dt \models \text{In1r } (l \cdot \text{While } (e) c) \quad :: x$
 $E, dt \models \text{In1r } (\text{Jmp } \text{jump}) \quad :: x$
 $E, dt \models \text{In1r } (\text{Throw } e) \quad :: x$
 $E, dt \models \text{In1r } (\text{Try } c1 \text{ Catch } (tn \text{ vn}) c2) :: x$

$$\begin{array}{l} E, dt \models \text{In1r } (c1 \text{ Finally } c2) \quad ::x \\ E, dt \models \text{In1r } (\text{Init } C) \quad ::x \\ \text{declare } \text{not-None-eq} \text{ [simp]} \\ \text{declare } \text{split-if} \text{ [split]} \text{ split-if-asm [split]} \\ \text{declare } \text{split-paired-All} \text{ [simp]} \text{ split-paired-Ex [simp]} \\ \langle ML \rangle \end{array}$$

lemma *is-acc-class-is-accessible*:
 $\text{is-acc-class } G P C \implies G \vdash (\text{Class } C) \text{ accessible-in } P$
 $\langle \text{proof} \rangle$

lemma *is-acc-iface-is-iface*: $\text{is-acc-iface } G P I \implies \text{is-iface } G I$
 $\langle \text{proof} \rangle$

lemma *is-acc-iface-Iface-is-accessible*:
 $\text{is-acc-iface } G P I \implies G \vdash (\text{Iface } I) \text{ accessible-in } P$
 $\langle \text{proof} \rangle$

lemma *is-acc-type-is-type*: $\text{is-acc-type } G P T \implies \text{is-type } G T$
 $\langle \text{proof} \rangle$

lemma *is-acc-iface-is-accessible*:
 $\text{is-acc-type } G P T \implies G \vdash T \text{ accessible-in } P$
 $\langle \text{proof} \rangle$

lemma *wt-Methd-is-methd*:
 $E \vdash \text{In1l } (\text{Methd } C \text{ sig}) :: T \implies \text{is-methd } (\text{prg } E) C \text{ sig}$
 $\langle \text{proof} \rangle$

Special versions of some typing rules, better suited to pattern match the conclusion (no selectors in the conclusion)

lemma *wt-Call*:
 $\llbracket E, dt \models e :: - \text{RefT } \text{statT}; E, dt \models ps :: \dot{=} pTs;$
 $\text{max-spec } (\text{prg } E) (\text{cls } E) \text{ statT } (\text{name} = mn, \text{parTs} = pTs)$
 $= \{((\text{statDeclC}, m), pTs'); rT = (\text{resTy } m); \text{accC} = \text{cls } E;$
 $\text{mode} = \text{invmode } m \ e\} \implies E, dt \models \{\text{accC}, \text{statT}, \text{mode}\} e \cdot mn(\{pTs'\} ps) :: - rT$
 $\langle \text{proof} \rangle$

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*:

$\llbracket E, dt \models e :: - \text{Class } C; \text{accfield } (\text{prg } E) (\text{cls } E) C \text{ fn} = \text{Some } (\text{statDecl } C, f);$
 $\quad sf = \text{is-static } f; fT = (\text{type } f); \text{accC} = \text{cls } E \rrbracket$
 $\implies E, dt \models \{ \text{accC}, \text{statDecl } C, sf \} e. \text{fn} :: = fT$
 $\langle \text{proof} \rangle$

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*:
 $\bigwedge E. E, dt \models t :: U \implies \text{case } t \text{ of}$
 $\quad \text{In1 } ec \Rightarrow (\text{case } ec \text{ of}$
 $\quad \quad \text{Inl } e \Rightarrow \exists T. U = \text{Inl } T$
 $\quad \quad | \text{Inr } s \Rightarrow U = \text{Inl } (\text{PrimT } \text{Void}))$
 $\quad | \text{In2 } e \Rightarrow (\exists T. U = \text{Inl } T)$
 $\quad | \text{In3 } e \Rightarrow (\exists T. U = \text{Inr } T)$
 $\langle \text{proof} \rangle$

lemma *wt-expr-eq*: $E, dt \models \text{In1l } t :: U = (\exists T. U = \text{Inl } T \wedge E, dt \models t :: - T)$
 $\langle \text{proof} \rangle$

lemma *wt-var-eq*: $E, dt \models \text{In2 } t :: U = (\exists T. U = \text{Inl } T \wedge E, dt \models t :: = T)$
 $\langle \text{proof} \rangle$

lemma *wt-exprs-eq*: $E, dt \models \text{In3 } t :: U = (\exists Ts. U = \text{Inr } Ts \wedge E, dt \models t :: \doteq Ts)$
 $\langle \text{proof} \rangle$

lemma *wt-stmt-eq*: $E, dt \models \text{In1r } t :: U = (U = \text{Inl } (\text{PrimT } \text{Void}) \wedge E, dt \models t :: \surd)$
 $\langle \text{proof} \rangle$

$\langle \text{ML} \rangle$

lemma *wt-elim-BinOp*:
 $\llbracket E, dt \models \text{In1l } (\text{BinOp } \text{binop } e1 e2) :: T;$
 $\quad \bigwedge T1 T2 T3.$
 $\quad \llbracket E, dt \models e1 :: - T1; E, dt \models e2 :: - T2; \text{wt-binop } (\text{prg } E) \text{binop } T1 T2;$
 $\quad \quad E, dt \models (\text{if } b \text{ then } \text{In1l } e2 \text{ else } \text{In1r } \text{Skip}) :: T3;$
 $\quad \quad T = \text{Inl } (\text{PrimT } (\text{binop-type } \text{binop})) \rrbracket$
 $\implies P \rrbracket$
 $\implies P$
 $\langle \text{proof} \rangle$

lemma *Inj-eq-lemma* [*simp*]:
 $(\forall T. (\exists T'. T = \text{Inj } T' \wedge P T') \longrightarrow Q T) = (\forall T'. P T' \longrightarrow Q (\text{Inj } T'))$

<proof>

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 abruptio (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 ensure that for all initializers $jumpNestingOkS$ holds
 — Dummy analysis for intermediate smallest step 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 $jumpNestingOk\ expr\ simp\ [simp]: jumpNestingOk\ jmps\ (In1l\ e) = True$
 $\langle proof \rangle$

lemma $jumpNestingOk\ expr\ simp1\ [simp]: jumpNestingOk\ jmps\ \langle e::expr \rangle = True$
 $\langle proof \rangle$

lemma $jumpNestingOk\ stmt\ simp\ [simp]:$
 $jumpNestingOk\ jmps\ (In1r\ s) = jumpNestingOkS\ jmps\ s$
 $\langle proof \rangle$

lemma $jumpNestingOk\ stmt\ simp1\ [simp]:$
 $jumpNestingOk\ jmps\ \langle s::stmt \rangle = jumpNestingOkS\ jmps\ s$
 $\langle proof \rangle$

lemma $jumpNestingOk\ var\ simp\ [simp]: jumpNestingOk\ jmps\ (In2\ v) = True$
 $\langle proof \rangle$

lemma $jumpNestingOk\ var\ simp1\ [simp]: jumpNestingOk\ jmps\ \langle v::var \rangle = True$
 $\langle proof \rangle$

lemma $jumpNestingOk\ expr\ list\ simp\ [simp]: jumpNestingOk\ jmps\ (In3\ es) = True$
 $\langle proof \rangle$

lemma $jumpNestingOk\ expr\ list\ simp1\ [simp]:$
 $jumpNestingOk\ jmps\ \langle es::expr\ list \rangle = True$
 $\langle proof \rangle$

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

$assignsE\ (NewC\ c) = \{\}$
 $assignsE\ (NewA\ t\ e) = assignsE\ e$
 $assignsE\ (Cast\ t\ e) = assignsE\ e$
 $assignsE\ (e\ InstOf\ r) = assignsE\ e$
 $assignsE\ (Lit\ val) = \{\}$
 $assignsE\ (UnOp\ unop\ e) = assignsE\ e$
 $assignsE\ (BinOp\ binop\ e1\ e2) = (if\ binop=CondAnd\ \vee\ binop=CondOr$
 $then\ (assignsE\ e1)$
 $else\ (assignsE\ e1) \cup (assignsE\ e2))$
 $assignsE\ (Super) = \{\}$
 $assignsE\ (Acc\ v) = assignsV\ v$
 $assignsE\ (v:=e)$
 $= (assignsV\ v) \cup (assignsE\ e) \cup$
 $(if\ \exists\ n.\ v=(LVar\ n)\ then\ \{the-LVar-name\ v\}$
 $else\ \{\})$
 $assignsE\ (b?\ e1 : e2) = (assignsE\ b) \cup ((assignsE\ e1) \cap (assignsE\ e2))$
 $assignsE\ (\{accC,statT,mode\}objRef.mn(\{pTs\}args))$
 $= (assignsE\ objRef) \cup (assignsEs\ args)$

— 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 $assigns :: term \Rightarrow lname\ set$

$assigns\ t \equiv (case\ t\ of$
 $In1\ se \Rightarrow (case\ se\ of$
 $Inl\ e \Rightarrow assignsE\ e$
 $| Inr\ s \Rightarrow \{\})$
 $| In2\ v \Rightarrow assignsV\ v$
 $| In3\ es \Rightarrow assignsEs\ es)$

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* ⇒ *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 ⇒ *None*
 | *Some v* ⇒ *Some (eval-unop unop v)*)

constVal (*BinOp* *binop e1 e2*) = (case (*constVal* *e1*) of
 None ⇒ *None*
 | *Some v1* ⇒ (case (*constVal* *e2*) of
 None ⇒ *None*
 | *Some v2* ⇒ *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 ⇒ *None*
 | *Some bv* ⇒ (case *the-Bool* *bv* of
 True ⇒ (case (*constVal* *e2*) of
 None ⇒ *None*
 | *Some v* ⇒ *constVal e1*)
 | *False* ⇒ (case (*constVal* *e1*) of
 None ⇒ *None*
 | *Some v* ⇒ *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 (Methd 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: $\bigwedge unop e'. P e' \implies P (UnOp unop e')$ **and**

hyp-BinOp: $\bigwedge binop e1 e2. [P e1; P e2] \implies P (BinOp binop e1 e2)$ **and**

$$\begin{aligned}
\text{hyp-CondL: } & \bigwedge b \, bv \, e1 \, e2. \llbracket \text{constVal } b = \text{Some } bv; \text{the-Bool } bv; P \, b; P \, e1 \rrbracket \\
& \implies P \, (b? \, e1 : e2) \text{ and} \\
\text{hyp-CondR: } & \bigwedge b \, bv \, e1 \, e2. \llbracket \text{constVal } b = \text{Some } bv; \neg \text{the-Bool } bv; P \, b; P \, e2 \rrbracket \\
& \implies P \, (b? \, e1 : e2)
\end{aligned}$$

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 \quad \text{then assigns-if } (\neg b) \, e \\
& \quad \quad \text{else UNIV}) \\
& \quad | \text{Some } v \Rightarrow (\text{if } v = \text{Bool } b \\
& \quad \quad \text{then } \{\} \\
& \quad \quad \text{else UNIV})) \\
\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})) \\
\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)) \\
\text{assigns-if } b \, (\{\text{accC, statT, mode}\} \text{objRef} \cdot \text{mn}(\{\text{pTs}\} \text{args})) & = \text{assignsE } (\{\text{accC, statT, mode}\} \text{objRef} \cdot \text{mn}(\{\text{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* $e = \text{Some } (\text{Bool } b)$ (**is** ?*Const* b e)
shows *assigns-if* b $e = \{\}$ (**is** ?*Ass* b e)
 <proof>

lemma *assigns-if-const-not-b-simp*:

assumes *boolConst*: *constVal* $e = \text{Some } (\text{Bool } b)$ (**is** ?*Const* b e)
shows *assigns-if* $(\neg b)$ $e = \text{UNIV}$ (**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

constdefs *rmlab* :: '*a* \Rightarrow ('*a*, '*b*) *tables* \Rightarrow ('*a*, '*b*) *tables*

rmlab *k A* $\equiv \lambda x. \text{if } x=k \text{ then UNIV else } A x$

constdefs *range-inter-ts* :: ('*a*, '*b*) *tables* \Rightarrow '*b set* ($\Rightarrow \cap$ - 80)

$$\Rightarrow \bigcap A \equiv \{x \mid x. \forall k. x \in A k\}$$

In $E \vdash B \gg t \gg A$, B denotes the "assigned" variables before evaluating term t , whereas A denotes the "assigned" variables after evaluating term t . The environment E is only needed for the conditional - ? - : -. The definite assignment rules refer to the typing rules here to distinguish boolean and other expressions.

inductive

$da :: env \Rightarrow lname set \Rightarrow term \Rightarrow assigned \Rightarrow bool$ (-+ - »-» - [65,65,65,65] 71)

where

$Skip: Env \vdash B \gg \langle Skip \rangle \gg (\text{norm}=B, \text{brk}=\lambda l. UNIV)$

| $Expr: Env \vdash B \gg \langle e \rangle \gg A$

\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

$Env \vdash B \gg \langle Break l \cdot c \rangle \gg A$

| $Comp: \llbracket Env \vdash B \gg \langle c1 \rangle \gg C1; Env \vdash \text{norm } C1 \gg \langle c2 \rangle \gg C2;$

$\text{norm } A = \text{norm } C2; \text{brk } A = (\text{brk } C1) \Rightarrow \bigcap (\text{brk } C2) \rrbracket$

\Rightarrow

$Env \vdash B \gg \langle c1;; c2 \rangle \gg A$

| $If: \llbracket Env \vdash B \gg \langle e \rangle \gg E;$

$Env \vdash (B \cup \text{assigns-if True } e) \gg \langle c1 \rangle \gg C1;$

$Env \vdash (B \cup \text{assigns-if False } e) \gg \langle c2 \rangle \gg C2;$

$\text{norm } A = \text{norm } C1 \cap \text{norm } C2;$

$\text{brk } A = \text{brk } C1 \Rightarrow \bigcap \text{brk } C2 \rrbracket$

\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 therefor $\text{norm } C2 = UNIV$. So finally $\text{norm } A = \text{norm } C1$. For the break maps this trick workd too, because the trival 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

$Env \vdash B \gg \langle l \cdot While(e) c \rangle \gg A$

— 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.

| $Jmp: \llbracket \text{jump}=\text{Ret} \longrightarrow \text{Result} \in B;$

$\text{norm } A = UNIV;$

$\text{brk } A = (\text{case jump of}$

$Break l \Rightarrow \lambda k. \text{if } k=l \text{ then } B \text{ else } UNIV$

| $Cont l \Rightarrow \lambda k. UNIV$

| $Ret \Rightarrow \lambda k. UNIV) \rrbracket$

$$\begin{aligned} &\implies \\ &Env \vdash B \gg \langle \text{Jump } jump \rangle \gg A \end{aligned}$$

— 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.

$$\begin{aligned} | \text{Throw: } &[[Env \vdash B \gg \langle e \rangle \gg E; nrm A = UNIV; brk A = (\lambda l. UNIV)]] \\ &\implies Env \vdash B \gg \langle \text{Throw } e \rangle \gg A \end{aligned}$$

$$\begin{aligned} | \text{Try: } &[[Env \vdash B \gg \langle c1 \rangle \gg C1; \\ &Env(\{lcl := lcl \ Env(VName \ v \mapsto \text{Class } C)\}) \vdash (B \cup \{VName \ v\}) \gg \langle c2 \rangle \gg C2; \\ &nrm A = nrm C1 \cap nrm C2; \\ &brk A = brk C1 \Rightarrow \cap brk C2]] \\ &\implies Env \vdash B \gg \langle \text{Try } c1 \ \text{Catch}(C \ v) \ c2 \rangle \gg A \end{aligned}$$

$$\begin{aligned} | \text{Fin: } &[[Env \vdash B \gg \langle c1 \rangle \gg C1; \\ &Env \vdash B \gg \langle c2 \rangle \gg C2; \\ &nrm A = nrm C1 \cup nrm C2; \\ &brk A = ((brk C1) \Rightarrow \cup_{\vee} (nrm C2)) \Rightarrow \cap (brk C2)]] \\ &\implies \\ &Env \vdash B \gg \langle c1 \ \text{Finally } c2 \rangle \gg A \end{aligned}$$

— 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 $brk C1$ with $nrm C2$ ($\Rightarrow \cup_{\vee}$). 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: $\lambda l. UNIV$. 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*

$$| \text{Init: } Env \vdash B \gg \langle \text{Init } C \rangle \gg (nrm=B, brk=\lambda l. UNIV)$$

— 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.

$$| \text{NewC: } Env \vdash B \gg \langle \text{NewC } C \rangle \gg (nrm=B, brk=\lambda l. UNIV)$$

$$\begin{aligned} | \text{NewA: } &Env \vdash B \gg \langle e \rangle \gg A \\ &\implies \\ &Env \vdash B \gg \langle \text{New } T[e] \rangle \gg A \end{aligned}$$

$$\begin{aligned} | \text{Cast: } &Env \vdash B \gg \langle e \rangle \gg A \\ &\implies \\ &Env \vdash B \gg \langle \text{Cast } T \ e \rangle \gg A \end{aligned}$$

- | *Inst*: $Env \vdash B \gg \langle e \rangle \gg A$
 \implies
 $Env \vdash B \gg \langle e \text{ InstOf } T \rangle \gg A$
- | *Lit*: $Env \vdash B \gg \langle Lit \ v \rangle \gg (\downarrow nrm=B, brk=\lambda l. UNIV)$
- | *UnOp*: $Env \vdash B \gg \langle e \rangle \gg A$
 \implies
 $Env \vdash B \gg \langle UnOp \ unop \ e \rangle \gg A$
- | *CondAnd*: $\llbracket Env \vdash B \gg \langle e1 \rangle \gg E1; Env \vdash (B \cup \text{assigns-if True } e1) \gg \langle e2 \rangle \gg E2;$
 $nrm \ A = B \cup (\text{assigns-if True } (BinOp \ CondAnd \ e1 \ e2) \cap$
 $\text{assigns-if False } (BinOp \ CondAnd \ e1 \ e2));$
 $brk \ A = (\lambda l. UNIV) \rrbracket$
 \implies
 $Env \vdash B \gg \langle BinOp \ CondAnd \ e1 \ e2 \rangle \gg A$
- | *CondOr*: $\llbracket Env \vdash B \gg \langle e1 \rangle \gg E1; Env \vdash (B \cup \text{assigns-if False } e1) \gg \langle e2 \rangle \gg E2;$
 $nrm \ A = B \cup (\text{assigns-if True } (BinOp \ CondOr \ e1 \ e2) \cap$
 $\text{assigns-if False } (BinOp \ CondOr \ e1 \ e2));$
 $brk \ A = (\lambda l. UNIV) \rrbracket$
 \implies
 $Env \vdash B \gg \langle BinOp \ CondOr \ e1 \ e2 \rangle \gg A$
- | *BinOp*: $\llbracket Env \vdash B \gg \langle e1 \rangle \gg E1; Env \vdash nrm \ E1 \gg \langle e2 \rangle \gg A;$
 $binop \neq \text{CondAnd}; binop \neq \text{CondOr} \rrbracket$
 \implies
 $Env \vdash B \gg \langle BinOp \ binop \ e1 \ e2 \rangle \gg A$
- | *Super*: $This \in B$
 \implies
 $Env \vdash B \gg \langle Super \rangle \gg (\downarrow nrm=B, brk=\lambda l. UNIV)$
- | *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
 $Env \vdash B \gg \langle v := e \rangle \gg A$
- | *CondBool*: $\llbracket Env \vdash (c \ ? \ e1 : e2) :: \neg (PrimT \ Boolean);$
 $Env \vdash B \gg \langle c \rangle \gg C;$
 $Env \vdash (B \cup \text{assigns-if True } c) \gg \langle e1 \rangle \gg E1;$
 $Env \vdash (B \cup \text{assigns-if False } c) \gg \langle e2 \rangle \gg E2;$
 $nrm \ A = B \cup (\text{assigns-if True } (c \ ? \ e1 : e2) \cap$
 $\text{assigns-if False } (c \ ? \ e1 : e2)); \rrbracket$

$$\begin{aligned} & \text{brk } A = (\lambda l. \text{UNIV}) \\ \implies & \\ & \text{Env} \vdash B \gg \langle c \text{ ? } e1 : e2 \rangle \gg A \end{aligned}$$

$$\begin{aligned} | \text{Cond: } & \llbracket \neg \text{Env} \vdash (c \text{ ? } e1 : e2) :: \neg (\text{PrimT Boolean}); \\ & \text{Env} \vdash B \gg \langle c \rangle \gg C; \\ & \text{Env} \vdash (B \cup \text{assigns-if True } c) \gg \langle e1 \rangle \gg E1; \\ & \text{Env} \vdash (B \cup \text{assigns-if False } c) \gg \langle e2 \rangle \gg E2; \\ & \text{nrm } A = \text{nrm } E1 \cap \text{nrm } E2; \text{brk } A = (\lambda l. \text{UNIV}) \rrbracket \\ \implies & \\ & \text{Env} \vdash B \gg \langle c \text{ ? } e1 : e2 \rangle \gg A \end{aligned}$$

$$\begin{aligned} | \text{Call: } & \llbracket \text{Env} \vdash B \gg \langle e \rangle \gg E; \text{Env} \vdash \text{nrm } E \gg \langle \text{args} \rangle \gg A \rrbracket \\ \implies & \\ & \text{Env} \vdash B \gg \langle \{ \text{accC}, \text{statT}, \text{mode} \} e \cdot \text{mn}(\{ \text{pTs} \} \text{args}) \rangle \gg A \end{aligned}$$

— 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 therefore 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.

$$\begin{aligned} | \text{Method: } & \llbracket \text{method } (\text{prg } \text{Env}) \text{ } D \text{ sig} = \text{Some } m; \\ & \text{Env} \vdash B \gg \langle \text{Body } (\text{declclass } m) (\text{stmt } (\text{mbody } (\text{mthd } m))) \rangle \gg A \\ & \rrbracket \\ \implies & \\ & \text{Env} \vdash B \gg \langle \text{Method } D \text{ sig} \rangle \gg A \end{aligned}$$

$$\begin{aligned} | \text{Body: } & \llbracket \text{Env} \vdash B \gg \langle c \rangle \gg C; \text{jumpNestingOkS } \{ \text{Ret} \} c; \text{Result} \in \text{nrm } C; \\ & \text{nrm } A = B; \text{brk } A = (\lambda l. \text{UNIV}) \rrbracket \\ \implies & \\ & \text{Env} \vdash B \gg \langle \text{Body } D \text{ } c \rangle \gg A \end{aligned}$$

— 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.

$$| \text{LVar: } \text{Env} \vdash B \gg \langle \text{LVar } vn \rangle \gg (\text{nrm} = B, \text{brk} = \lambda l. \text{UNIV})$$

$$\begin{aligned} | \text{FVar: } & \text{Env} \vdash B \gg \langle e \rangle \gg A \\ \implies & \\ & \text{Env} \vdash B \gg \langle \{ \text{accC}, \text{statDeclC}, \text{stat} \} e \cdot \text{fn} \rangle \gg A \end{aligned}$$

$$\begin{aligned} | \text{AVar: } & \llbracket \text{Env} \vdash B \gg \langle e1 \rangle \gg E1; \text{Env} \vdash \text{nrm } E1 \gg \langle e2 \rangle \gg A \rrbracket \\ \implies & \\ & \text{Env} \vdash B \gg \langle e1.[e2] \rangle \gg A \end{aligned}$$

$$| \text{Nil: } \text{Env} \vdash B \gg \langle [] :: \text{expr list} \rangle \gg (\text{nrm} = B, \text{brk} = \lambda l. \text{UNIV})$$

$$\begin{aligned} | \text{Cons: } & \llbracket \text{Env} \vdash B \gg \langle e :: \text{expr} \rangle \gg E; \text{Env} \vdash \text{nrm } E \gg \langle es \rangle \gg A \rrbracket \\ \implies & \\ & \text{Env} \vdash B \gg \langle e \# es \rangle \gg A \end{aligned}$$

declare *inj-term-sym-simps* [*simp*]
declare *assigns-if.simps* [*simp del*]
declare *split-paired-All* [*simp del*] *split-paired-Ex* [*simp del*]
 ⟨*ML*⟩

inductive-cases *da-elim-cases* [*cases set*]:

$Env \vdash B \gg \langle \text{Skip} \rangle \gg A$
 $Env \vdash B \gg \text{In1r Skip} \gg A$
 $Env \vdash B \gg \langle \text{Expr } e \rangle \gg A$
 $Env \vdash B \gg \text{In1r (Expr } e) \gg A$
 $Env \vdash B \gg \langle l \cdot c \rangle \gg A$
 $Env \vdash B \gg \text{In1r (} l \cdot c) \gg A$
 $Env \vdash B \gg \langle c1 ;; c2 \rangle \gg A$
 $Env \vdash B \gg \text{In1r (} c1 ;; c2) \gg A$
 $Env \vdash B \gg \langle \text{If}(e) \ c1 \ \text{Else } c2 \rangle \gg A$
 $Env \vdash B \gg \text{In1r (If}(e) \ c1 \ \text{Else } c2) \gg A$
 $Env \vdash B \gg \langle l \cdot \text{While}(e) \ c \rangle \gg A$
 $Env \vdash B \gg \text{In1r (} l \cdot \text{While}(e) \ c) \gg A$
 $Env \vdash B \gg \langle \text{Jmp } \text{jump} \rangle \gg A$
 $Env \vdash B \gg \text{In1r (} \text{Jmp } \text{jump}) \gg A$
 $Env \vdash B \gg \langle \text{Throw } e \rangle \gg A$
 $Env \vdash B \gg \text{In1r (} \text{Throw } e) \gg A$
 $Env \vdash B \gg \langle \text{Try } c1 \ \text{Catch}(C \ vn) \ c2 \rangle \gg A$
 $Env \vdash B \gg \text{In1r (} \text{Try } c1 \ \text{Catch}(C \ vn) \ c2) \gg A$
 $Env \vdash B \gg \langle c1 \ \text{Finally } c2 \rangle \gg A$
 $Env \vdash B \gg \text{In1r (} c1 \ \text{Finally } c2) \gg A$
 $Env \vdash B \gg \langle \text{Init } C \rangle \gg A$
 $Env \vdash B \gg \text{In1r (} \text{Init } C) \gg A$
 $Env \vdash B \gg \langle \text{NewC } C \rangle \gg A$
 $Env \vdash B \gg \text{In1l (} \text{NewC } C) \gg A$
 $Env \vdash B \gg \langle \text{New } T[e] \rangle \gg A$
 $Env \vdash B \gg \text{In1l (} \text{New } T[e]) \gg A$
 $Env \vdash B \gg \langle \text{Cast } T \ e \rangle \gg A$
 $Env \vdash B \gg \text{In1l (} \text{Cast } T \ e) \gg A$
 $Env \vdash B \gg \langle e \ \text{InstOf } T \rangle \gg A$
 $Env \vdash B \gg \text{In1l (} e \ \text{InstOf } T) \gg A$
 $Env \vdash B \gg \langle \text{Lit } v \rangle \gg A$
 $Env \vdash B \gg \text{In1l (} \text{Lit } v) \gg A$
 $Env \vdash B \gg \langle \text{UnOp } \text{unop } e \rangle \gg A$
 $Env \vdash B \gg \text{In1l (} \text{UnOp } \text{unop } e) \gg A$
 $Env \vdash B \gg \langle \text{BinOp } \text{binop } e1 \ e2 \rangle \gg A$
 $Env \vdash B \gg \text{In1l (} \text{BinOp } \text{binop } e1 \ e2) \gg A$
 $Env \vdash B \gg \langle \text{Super} \rangle \gg A$
 $Env \vdash B \gg \text{In1l (} \text{Super}) \gg A$
 $Env \vdash B \gg \langle \text{Acc } v \rangle \gg A$
 $Env \vdash B \gg \text{In1l (} \text{Acc } v) \gg A$
 $Env \vdash B \gg \langle v := e \rangle \gg A$
 $Env \vdash B \gg \text{In1l (} v := e) \gg A$
 $Env \vdash B \gg \langle c \ ? \ e1 : e2 \rangle \gg A$
 $Env \vdash B \gg \text{In1l (} c \ ? \ e1 : e2) \gg A$
 $Env \vdash B \gg \langle \{accC, statT, mode\} e \cdot mn(\{pTs\} args) \rangle \gg A$
 $Env \vdash B \gg \text{In1l (} \{accC, statT, mode\} e \cdot mn(\{pTs\} args) \rangle \gg A$
 $Env \vdash B \gg \langle \text{Methd } C \ sig \rangle \gg A$
 $Env \vdash B \gg \text{In1l (} \text{Methd } C \ sig) \gg A$
 $Env \vdash B \gg \langle \text{Body } D \ c \rangle \gg A$
 $Env \vdash B \gg \text{In1l (} \text{Body } D \ c) \gg A$
 $Env \vdash B \gg \langle \text{LVar } vn \rangle \gg A$

$Env \vdash B \gg In2 (LVar\ vn) \gg A$
 $Env \vdash B \gg \langle \{accC, statDeclC, stat\} e..fn \rangle \gg A$
 $Env \vdash B \gg In2 (\{accC, statDeclC, stat\} e..fn) \gg A$
 $Env \vdash B \gg \langle e1.[e2] \rangle \gg A$
 $Env \vdash B \gg In2 (e1.[e2]) \gg A$
 $Env \vdash B \gg \langle []::expr\ list \rangle \gg A$
 $Env \vdash B \gg In3 ([]::expr\ list) \gg A$
 $Env \vdash B \gg \langle e\#es \rangle \gg A$
 $Env \vdash B \gg In3 (e\#es) \gg A$
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*: $E \vdash e :: \text{--PrimT Boolean (is ?Boolean e)}$
shows $assignsE\ e \subseteq assigns\text{-if}\ True\ e \cap assigns\text{-if}\ False\ e$ (**is** *?Incl e*)
 $\langle proof \rangle$

lemma *rmlab-same-label* [*simp*]: $(rmlab\ l\ A)\ l = UNIV$
 $\langle proof \rangle$

lemma *rmlab-same-label1* [*simp*]: $l=l' \implies (rmlab\ l\ A)\ l' = UNIV$
 $\langle proof \rangle$

lemma *rmlab-other-label* [*simp*]: $l \neq l' \implies (rmlab\ l\ A)\ l' = A\ l'$
 $\langle proof \rangle$

lemma *range-inter-ts-subseteq* [intro]: $\forall k. A \subseteq B \implies \Rightarrow \bigcap A \subseteq \Rightarrow \bigcap B$
 ⟨proof⟩

lemma *range-inter-ts-subseteq'*:
 $\llbracket \forall k. A \subseteq B \ k; x \in \Rightarrow \bigcap A \rrbracket \implies x \in \Rightarrow \bigcap B$
 ⟨proof⟩

lemma *da-monotone*:
assumes *da*: $Env \vdash B \gg t \gg A$ **and**
 $B \subseteq B'$ **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** $B \subseteq B'$
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*
 $\equiv \lambda k. (case\ k\ of$
 EName e
 $\Rightarrow (case\ e\ of$
 VNam v
 $\Rightarrow (table-of\ (lcls\ (mbody\ m))((pars\ m)[\mapsto](parTs\ sig)))\ v$
 | *Res* $\Rightarrow Some\ (resTy\ m)$
 | *This* $\Rightarrow if\ is-static\ m\ then\ None\ else\ Some\ (Class\ C)$)

constdefs *parameters* :: *methd* \Rightarrow *lname set*
parameters m $\equiv set\ (map\ (EName\ \circ\ VNam)\ (pars\ m))$
 $\cup\ (if\ (static\ m)\ then\ \{\}\ else\ \{This\})$

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
 $(\downarrow prg = G, cls = C, lcl = callee-lcl\ C\ sig\ m) \vdash (stmt\ (mbody\ m)) :: \surd \wedge$
 $(\exists A.\ (\downarrow prg = G, cls = C, lcl = callee-lcl\ C\ sig\ m)$
 $\vdash parameters\ m \gg \langle stmt\ (mbody\ m) \rangle \gg A$
 $\wedge Result \in nrm\ A)$

lemma *callee-lcl-VNam-simp* [*simp*]:
callee-lcl C sig m (EName (VNam 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*:

$\neg \text{is-static } m \implies \text{callee-lcl } C \text{ sig } m \text{ (This) = Some (Class C)}$

$\langle \text{proof} \rangle$

lemma *wf-mheadI*:

$\llbracket \text{length (parTs sig) = length (pars m)}; \forall T \in \text{set (parTs sig)}. \text{is-acc-type } G \ P \ T;$

$\text{is-acc-type } G \ P \ (\text{resTy } m); \text{distinct (pars m)} \rrbracket \implies$

$\text{wf-mhead } G \ P \ \text{sig } m$

$\langle \text{proof} \rangle$

lemma *wf-mdeclI*: \llbracket

$\text{wf-mhead } G \ (\text{pid } C) \ \text{sig } (\text{mhead } m); \text{unique (lcls (mbody m))};$

$(\forall pn \in \text{set (pars m)}. \text{table-of (lcls (mbody m)) } pn = \text{None});$

$\forall (vn, T) \in \text{set (lcls (mbody m))}. \text{is-acc-type } G \ (\text{pid } C) \ T;$

$\text{jumpNestingOkS } \{\text{Ret}\} \ (\text{stmt (mbody m)});$

$\text{is-class } G \ C;$

$(\text{prg} = G, \text{cls} = C, \text{lcl} = \text{callee-lcl } C \ \text{sig } m) \vdash (\text{stmt (mbody m)}) :: \checkmark;$

$(\exists A. (\text{prg} = G, \text{cls} = C, \text{lcl} = \text{callee-lcl } C \ \text{sig } m) \vdash \text{parameters } m \gg (\text{stmt (mbody m)}) \gg A$
 $\wedge \text{Result} \in \text{nrm } A)$

$\rrbracket \implies$

$\text{wf-mdecl } G \ C \ (\text{sig}, m)$

$\langle \text{proof} \rangle$

lemma *wf-mdeclE* [*consumes 1*]:

$\llbracket \text{wf-mdecl } G \ C \ (\text{sig}, m);$

$\llbracket \text{wf-mhead } G \ (\text{pid } C) \ \text{sig } (\text{mhead } m); \text{unique (lcls (mbody m))};$

$\forall pn \in \text{set (pars m)}. \text{table-of (lcls (mbody m)) } pn = \text{None};$

$\forall (vn, T) \in \text{set (lcls (mbody m))}. \text{is-acc-type } G \ (\text{pid } C) \ T;$

$\text{jumpNestingOkS } \{\text{Ret}\} \ (\text{stmt (mbody m)});$

$\text{is-class } G \ C;$

$(\text{prg} = G, \text{cls} = C, \text{lcl} = \text{callee-lcl } C \ \text{sig } m) \vdash (\text{stmt (mbody m)}) :: \checkmark;$

$(\exists A. (\text{prg} = G, \text{cls} = C, \text{lcl} = \text{callee-lcl } C \ \text{sig } m) \vdash \text{parameters } m \gg (\text{stmt (mbody m)}) \gg A$
 $\wedge \text{Result} \in \text{nrm } A)$

$\rrbracket \implies P$

$\rrbracket \implies P$

$\langle \text{proof} \rangle$

lemma *wf-mdeclD1*:

$\text{wf-mdecl } G \ C \ (\text{sig}, m) \implies$

$\text{wf-mhead } G \ (\text{pid } C) \ \text{sig } (\text{mhead } m) \wedge \text{unique (lcls (mbody m))} \wedge$

$(\forall pn \in \text{set (pars m)}. \text{table-of (lcls (mbody m)) } pn = \text{None}) \wedge$

$(\forall (vn, T) \in \text{set (lcls (mbody m))}. \text{is-acc-type } G \ (\text{pid } C) \ T)$

$\langle \text{proof} \rangle$

lemma *wf-mdecl-bodyD*:

$\text{wf-mdecl } G \ C \ (\text{sig}, m) \implies$

$(\exists T. (\text{prg} = G, \text{cls} = C, \text{lcl} = \text{callee-lcl } C \ \text{sig } m) \vdash \text{Body } C \ (\text{stmt (mbody m)}) :: -T \wedge$

$G \vdash T \preceq (\text{resTy } m))$

$\langle \text{proof} \rangle$

lemma *rT-is-acc-type*:

$wf\text{-mhead } G \ P \ sig \ m \implies is\text{-acc-type } G \ P \ (resTy \ m)$
 ⟨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

$wf\text{-idecl} :: prog \implies idecl \implies bool$
 $wf\text{-idecl } G \equiv$
 $\lambda(I, i).$
 $ws\text{-idecl } G \ I \ (isuperIfs \ i) \wedge$
 $\neg is\text{-class } G \ I \wedge$
 $(\forall (sig, mh) \in set \ (imethods \ i). wf\text{-mhead } G \ (pid \ I) \ sig \ mh \wedge$
 $\neg is\text{-static } mh \wedge$
 $accmodi \ mh = Public) \wedge$
 $unique \ (imethods \ i) \wedge$
 $(\forall J \in set \ (isuperIfs \ i). is\text{-acc-iface } G \ (pid \ I) \ J) \wedge$
 $(table\text{-of } (imethods \ i)$
 $hiding \ (methd \ G \ Object)$
 $under \ (\lambda new \ old. accmodi \ old \neq Private)$
 $entails \ (\lambda new \ old. G \vdash resTy \ new \preceq resTy \ old \wedge$
 $is\text{-static } new = is\text{-static } old)) \wedge$
 $(o2s \circ table\text{-of } (imethods \ i)$
 $hidings \ Un\text{-tables}((\lambda J. (imethds \ G \ J)) 'set \ (isuperIfs \ i))$
 $entails \ (\lambda new \ old. G \vdash resTy \ new \preceq resTy \ old))$

lemma *wf-idecl-mhead*: $\llbracket wf\text{-idecl } G \ (I, i); (sig, mh) \in set \ (imethods \ i) \rrbracket \implies$
 $wf\text{-mhead } G \ (pid \ I) \ sig \ mh \wedge \neg is\text{-static } mh \wedge accmodi \ mh = Public$
 ⟨proof⟩

lemma *wf-idecl-hidings*:

$wf\text{-idecl } G \ (I, i) \implies$
 $(\lambda s. o2s \ (table\text{-of } (imethods \ i) \ s))$
 $hidings \ Un\text{-tables} \ ((\lambda J. imethds \ G \ J) \ 'set \ (isuperIfs \ i))$
 $entails \ \lambda new \ old. G \vdash resTy \ new \preceq resTy \ old$
 ⟨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 $(\lambda \text{ new old. } G \vdash \text{resTy } \text{new} \leq \text{resTy } \text{old} \wedge$
 $\text{is-static } \text{new} = \text{is-static } \text{old}))$

<proof>

lemma *wf-idecl-supD*:

$\llbracket \text{wf-idecl } G (I, i); J \in \text{set } (\text{isuperIfs } i) \rrbracket$
 $\implies \text{is-acc-iface } G (\text{pid } I) J \wedge (J, I) \notin (\text{subint1 } G) \hat{+}$

<proof>

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*:

$\llbracket t \text{ entails } P; t k = \text{Some } x \rrbracket \implies P x$
 ⟨proof⟩

lemma *empty-entails[simp]*: *empty entails P*

⟨proof⟩

constdefs

wf-cdecl :: *prog* \Rightarrow *cdecl* \Rightarrow *bool*

wf-cdecl *G* \equiv

$\lambda(C, c).$
 $\neg \text{is-iface } G \ C \ \wedge$
 $(\forall I \in \text{set } (\text{superIfs } c). \text{is-acc-iface } G \ (\text{pid } C) \ I \ \wedge$
 $(\forall s. \forall im \in \text{imethds } G \ I \ s.$
 $(\exists cm \in \text{methd } G \ C \ s: G \vdash \text{resTy } cm \preceq \text{resTy } im \ \wedge$
 $\neg \text{is-static } cm \ \wedge$
 $\text{accmodi } im \leq \text{accmodi } cm))) \ \wedge$
 $(\forall f \in \text{set } (\text{cfields } c). \text{wf-fdecl } G \ (\text{pid } C) \ f) \ \wedge \text{unique } (\text{cfields } c) \ \wedge$
 $(\forall m \in \text{set } (\text{methods } c). \text{wf-mdecl } G \ C \ m) \ \wedge \text{unique } (\text{methods } c) \ \wedge$
 $\text{jumpNestingOkS } \{\} \ (\text{init } c) \ \wedge$
 $(\exists A. (\text{prg}=G, \text{cls}=C, \text{lcl}=\text{empty}) \vdash \{\} \ \gg \langle \text{init } c \rangle \ \gg A) \ \wedge$
 $(\text{prg}=G, \text{cls}=C, \text{lcl}=\text{empty}) \vdash (\text{init } c) :: \checkmark \ \wedge \text{ws-cdecl } G \ C \ (\text{super } c) \ \wedge$
 $(C \neq \text{Object} \longrightarrow$
 $(\text{is-acc-class } G \ (\text{pid } C) \ (\text{super } c) \ \wedge$
 $(\text{table-of } (\text{map } (\lambda (s, m). (s, C, m)) \ (\text{methods } c))$
 $\text{entails } (\lambda \text{ new. } \forall \text{ old sig.}$
 $(G, \text{sig} \vdash \text{new overrides}_S \text{old}$
 $\longrightarrow (G \vdash \text{resTy } \text{new} \preceq \text{resTy } \text{old} \ \wedge$
 $\text{accmodi } \text{old} \leq \text{accmodi } \text{new} \ \wedge$
 $\neg \text{is-static } \text{old})) \ \wedge$
 $(G, \text{sig} \vdash \text{new hides } \text{old}$
 $\longrightarrow (\text{accmodi } \text{old} \leq \text{accmodi } \text{new} \ \wedge$
 $\text{is-static } \text{old}))))))$
 $)$

lemma *wf-cdeclE* [*consumes 1*]:

$\llbracket \text{wf-cdecl } G \ (C, c);$
 $\llbracket \neg \text{is-iface } G \ C;$
 $(\forall I \in \text{set } (\text{superIfs } c). \text{is-acc-iface } G \ (\text{pid } C) \ I \ \wedge$
 $(\forall s. \forall im \in \text{imethds } G \ I \ s.$
 $(\exists cm \in \text{methd } G \ C \ s: G \vdash \text{resTy } cm \preceq \text{resTy } im \ \wedge$
 $\neg \text{is-static } cm \ \wedge$
 $\text{accmodi } im \leq \text{accmodi } cm)))$;
 $\forall f \in \text{set } (\text{cfields } c). \text{wf-fdecl } G \ (\text{pid } C) \ f; \text{unique } (\text{cfields } c);$
 $\forall m \in \text{set } (\text{methods } c). \text{wf-mdecl } G \ C \ m; \text{unique } (\text{methods } c);$
 $\text{jumpNestingOkS } \{\} \ (\text{init } c);$
 $\exists A. (\text{prg}=G, \text{cls}=C, \text{lcl}=\text{empty}) \vdash \{\} \ \gg \langle \text{init } c \rangle \ \gg A;$
 $(\text{prg}=G, \text{cls}=C, \text{lcl}=\text{empty}) \vdash (\text{init } c) :: \checkmark;$
 $\text{ws-cdecl } G \ C \ (\text{super } c);$
 $(C \neq \text{Object} \longrightarrow$
 $(\text{is-acc-class } G \ (\text{pid } C) \ (\text{super } c) \ \wedge$
 $(\text{table-of } (\text{map } (\lambda (s, m). (s, C, m)) \ (\text{methods } c))$
 $\text{entails } (\lambda \text{ new. } \forall \text{ old sig.}$
 $(G, \text{sig} \vdash \text{new overrides}_S \text{old}$
 $\longrightarrow (G \vdash \text{resTy } \text{new} \preceq \text{resTy } \text{old} \ \wedge$

lemma *wf-cdecl-hides-SomeD*:

$\llbracket wf\text{-}cdecl\ G\ (C, c); C \neq Object; \text{table-of (methods } c) \text{ sig} = \text{Some newM};$
 $G, sig \vdash (C, newM) \text{ hides old}$
 $\rrbracket \implies \text{accmodi old} \leq \text{access newM} \wedge$
 is-static old
 <proof>

lemma *wf-cdecl-wt-init*:

$wf\text{-}cdecl\ G\ (C, c) \implies (\text{prg} = G, \text{cls} = C, \text{lcl} = \text{empty}) \vdash \text{init } c :: \checkmark$
 <proof>

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-dynmethod* and *class-mheadsD*)
- all standard Exceptions are defined
- all defined interfaces are wellformed
- all defined classes are wellformed

constdefs

$wf\text{-}prog :: prog \Rightarrow bool$
 $wf\text{-}prog\ G \equiv \text{let } is = \text{ifaces } G; cs = \text{classes } G \text{ in}$
 $\text{ObjectC} \in \text{set } cs \wedge$
 $(\forall m \in \text{set } Object\text{-}mdecls. \text{accmodi } m \neq \text{Package}) \wedge$
 $(\forall xn. SXcptC\ xn \in \text{set } cs) \wedge$
 $(\forall i \in \text{set } is. wf\text{-}idecl\ G\ i) \wedge \text{unique } is \wedge$
 $(\forall c \in \text{set } cs. wf\text{-}cdecl\ G\ c) \wedge \text{unique } cs$

lemma *wf-prog-idecl*: $\llbracket \text{iface } G\ I = \text{Some } i; wf\text{-}prog\ G \rrbracket \implies wf\text{-}idecl\ G\ (I, i)$
 <proof>

lemma *wf-prog-cdecl*: $\llbracket \text{class } G\ C = \text{Some } c; wf\text{-}prog\ G \rrbracket \implies wf\text{-}cdecl\ G\ (C, c)$
 <proof>

lemma *wf-prog-Object-mdecls*:

$wf\text{-}prog\ G \implies (\forall m \in \text{set } Object\text{-}mdecls. \text{accmodi } m \neq \text{Package})$
 <proof>

lemma *wf-prog-acc-superD*:

$\llbracket wf\text{-}prog\ G; \text{class } G\ C = \text{Some } c; C \neq Object \rrbracket$
 $\implies \text{is-acc-class } G\ (\text{pid } C)\ (\text{super } c)$
 <proof>

lemma *wf-ws-prog [elim!,simp]*: $wf\text{-}prog\ G \implies ws\text{-}prog\ G$

$\langle proof \rangle$

lemma *class-Object* [simp]:

$wf\text{-prog } G \implies$

$$\text{class } G \text{ Object} = \text{Some } (\{\text{access}=\text{Public}, \text{cfields}=[], \text{methods}=\text{Object-mdecls}, \\ \text{init}=\text{Skip}, \text{super}=\text{arbitrary}, \text{superIfs}=[]\})$$

$\langle proof \rangle$

lemma *methd-Object*[simp]: $wf\text{-prog } G \implies \text{methd } G \text{ Object} =$

$$\text{table-of } (\text{map } (\lambda(s,m). (s, \text{Object}, m)) \text{ Object-mdecls})$$

$\langle proof \rangle$

lemma *wf-prog-Object-methd*:

$$\llbracket wf\text{-prog } G; \text{methd } G \text{ Object sig} = \text{Some } m \rrbracket \implies \text{accmodi } m \neq \text{Package}$$

$\langle proof \rangle$

lemma *wf-prog-Object-is-public*[intro]:

$$wf\text{-prog } G \implies \text{is-public } G \text{ Object}$$

$\langle proof \rangle$

lemma *class-SXcpt* [simp]:

$wf\text{-prog } G \implies$

$$\text{class } G \text{ (SXcpt } xn) = \text{Some } (\{\text{access}=\text{Public}, \text{cfields}=[], \text{methods}=\text{SXcpt-mdecls}, \\ \text{init}=\text{Skip}, \\ \text{super}=\text{if } xn = \text{Throwable then Object} \\ \text{else SXcpt Throwable}, \\ \text{superIfs}=[]\})$$

$\langle proof \rangle$

lemma *wf-ObjectC* [simp]:

$$wf\text{-cdecl } G \text{ ObjectC} = (\neg \text{is-iface } G \text{ Object} \wedge \text{Ball } (\text{set } \text{Object-mdecls}))$$

$$(\text{wf-mdecl } G \text{ Object}) \wedge \text{unique } \text{Object-mdecls}$$

$\langle proof \rangle$

lemma *Object-is-class* [simp, elim!]: $wf\text{-prog } G \implies \text{is-class } G \text{ Object}$

$\langle proof \rangle$

lemma *Object-is-acc-class* [simp, elim!]: $wf\text{-prog } G \implies \text{is-acc-class } G \text{ S Object}$

$\langle proof \rangle$

lemma *SXcpt-is-class* [simp, elim!]: $wf\text{-prog } G \implies \text{is-class } G \text{ (SXcpt } xn)$

$\langle proof \rangle$

lemma *SXcpt-is-acc-class* [simp, elim!]:

$$wf\text{-prog } G \implies \text{is-acc-class } G \text{ S (SXcpt } xn)$$

$\langle proof \rangle$

lemma *fields-Object* [simp]: $wf\text{-prog } G \implies \text{DeclConcepts.fields } G \text{ Object} = []$

$\langle \text{proof} \rangle$

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*:
 $\llbracket \text{is-class } G \ C; \text{wf-prog } G \rrbracket \implies \text{unique } (\text{DeclConcepts.fields } G \ C)$
 $\langle \text{proof} \rangle$

lemma *fields-mono*:
 $\llbracket \text{table-of } (\text{DeclConcepts.fields } G \ C) \text{ fn} = \text{Some } f; G \vdash D \preceq_C C; \text{is-class } G \ D; \text{wf-prog } G \rrbracket$
 $\implies \text{table-of } (\text{DeclConcepts.fields } G \ D) \text{ fn} = \text{Some } f$
 $\langle \text{proof} \rangle$

lemma *fields-is-type* [elim]:
 $\llbracket \text{table-of } (\text{DeclConcepts.fields } G \ C) \text{ m} = \text{Some } f; \text{wf-prog } G; \text{is-class } G \ C \rrbracket \implies$
 $\text{is-type } G \ (\text{type } f)$
 $\langle \text{proof} \rangle$

lemma *imethds-wf-mhead* [rule-format (no-asm)]:
 $\llbracket m \in \text{imethds } G \ I \text{ sig}; \text{wf-prog } G; \text{is-iface } G \ I \rrbracket \implies$
 $\text{wf-mhead } G \ (\text{pid } (\text{decliface } m)) \text{ sig } (\text{mthd } m) \wedge$
 $\neg \text{is-static } m \wedge \text{accmodi } m = \text{Public}$
 $\langle \text{proof} \rangle$

lemma *methd-wf-mdecl*:
 $\llbracket \text{methd } G \ C \text{ sig} = \text{Some } m; \text{wf-prog } G; \text{class } G \ C = \text{Some } y \rrbracket \implies$
 $G \vdash C \preceq_C (\text{declclass } m) \wedge \text{is-class } G \ (\text{declclass } m) \wedge$
 $\text{wf-mdecl } G \ (\text{declclass } m) \ (\text{sig}, (\text{mthd } m))$
 $\langle \text{proof} \rangle$

lemma *methd-rT-is-type*:
 $\llbracket wf\text{-prog } G; methd\ G\ C\ sig = Some\ m;$
 $\quad class\ G\ C = Some\ y \rrbracket$
 $\implies is\text{-type } G\ (resTy\ m)$
 $\langle proof \rangle$

lemma *accmethd-rT-is-type*:
 $\llbracket wf\text{-prog } G; accmethd\ G\ S\ C\ sig = Some\ m;$
 $\quad class\ G\ C = Some\ y \rrbracket$
 $\implies is\text{-type } G\ (resTy\ m)$
 $\langle proof \rangle$

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 wellformed program. Dynamic overriding has no restrictions on the access modifiers but enforces conform result types as precondition. But with some effort 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*: $G \vdash \text{Method old inheritable-in (pid C)}$ **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: $G \vdash \text{Method old declared-in (declclass old)}$ **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*: $G \vdash \text{Method old inheritable-in (pid C)}$ **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: $G \vdash \text{Method old declared-in (declclass old)}$ **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 $\text{pid (declclass old)} = \text{pid (declclass new)}$
 $\langle \text{proof} \rangle$

lemma *dyn-override-Package-escape*:

assumes *dyn-override*: $G \vdash \text{new overrides old}$ **and**
accommodi-old: $\text{accommodi old} = \text{Package}$ **and**
outside-pack: $\text{pid}(\text{declclass old}) \neq \text{pid}(\text{declclass new})$ **and**
wf: *wf-prog* G
shows $\exists \text{inter}. G \vdash \text{new overrides inter} \wedge G \vdash \text{inter overrides old} \wedge$
 $\text{pid}(\text{declclass old}) = \text{pid}(\text{declclass inter}) \wedge$
 $\text{Protected} \leq \text{accommodi inter}$

<proof>

lemma *declclass-widen*[*rule-format*]:

wf-prog G
 $\longrightarrow (\forall c m. \text{class } G C = \text{Some } c \longrightarrow \text{methd } G C \text{ sig} = \text{Some } m$
 $\longrightarrow G \vdash C \preceq_C \text{declclass } m) \text{ (is } ?P G C)$

<proof>

lemma *declclass-methd-Object*:

$\llbracket \text{wf-prog } G; \text{methd } G \text{ Object sig} = \text{Some } m \rrbracket \Longrightarrow \text{declclass } m = \text{Object}$
<proof>

lemma *methd-declaredD*:

$\llbracket \text{wf-prog } G; \text{is-class } G C; \text{methd } G C \text{ sig} = \text{Some } m \rrbracket$
 $\Longrightarrow G \vdash (\text{mdecl}(\text{sig}, \text{methd } m)) \text{ declared-in}(\text{declclass } m)$

<proof>

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

$\llbracket \text{table-of}(\text{map}(\lambda(s, m). (s, C, m))(\text{methods } c)) \text{ sig} = \text{Some } m \Longrightarrow P;$
 $\llbracket G \vdash C \text{ inherits}(\text{method } \text{sig } m); \text{methd } G(\text{super } c) \text{ sig} = \text{Some } m \rrbracket \Longrightarrow P$
 $\rrbracket \Longrightarrow P$

<proof>

lemma *methd-member-of*:

assumes *wf*: *wf-prog* G

shows

$\llbracket \text{is-class } G C; \text{methd } G C \text{ sig} = \text{Some } m \rrbracket \Longrightarrow G \vdash \text{Methd sig } m \text{ member-of } C$
 $\text{(is } ?\text{Class } C \Longrightarrow ?\text{Method } C \Longrightarrow ?\text{MemberOf } C)$

<proof>

lemma *current-methd*:

$\llbracket \text{table-of}(\text{methods } c) \text{ sig} = \text{Some } \text{new};$
 $\text{ws-prog } G; \text{class } G C = \text{Some } c; C \neq \text{Object};$
 $\text{methd } G(\text{super } c) \text{ sig} = \text{Some } \text{old} \rrbracket$
 $\Longrightarrow \text{methd } G C \text{ sig} = \text{Some}(C, \text{new})$

<proof>

lemma *wf-prog-staticD*:

assumes *wf*: wf-prog *G* **and**
clsC: class *G* *C* = Some *c* **and**
neq-C-Obj: *C* ≠ Object **and**
old: methd *G* (super *c*) sig = Some *old* **and**
accommodi-old: Protected ≤ 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**
accommodi-old: Protected ≤ accmodi *old* **and**
not-static-old: ¬ *is-static old*
shows
 \exists *new*. methd *G* *C* sig = Some *new* ∧
(*new* = *old* ∨ $G, \text{sig} \vdash \text{new overrides}_S \text{old}$)
(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**
accommodi-old: Protected ≤ accmodi *old* **and**
not-static-old: ¬ *is-static old* **and**
inheritance: methd *G* *C* sig = Some *old* ⇒ *P* **and**
overriding: \bigwedge *new*. $\llbracket \text{methd } G \ C \ \text{sig} = \text{Some } \text{new};$
 $G, \text{sig} \vdash \text{new overrides}_S \text{old} \rrbracket \Rightarrow P$
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**
accommodi-old: Protected ≤ accmodi *old* **and**
not-static-old: ¬ *is-static old*
shows
 \exists *new*. methd *G* *C* sig = Some *new* ∧
¬ *is-static new* ∧ $G \vdash \text{resTy } \text{new} \preceq \text{resTy } \text{old} \wedge \text{accommodi } \text{old} \leq \text{accommodi } \text{new}$
(is (\exists *new*. (?Constraint *C new old*)))
⟨proof⟩

lemma *beXI'*: $x \in A \Rightarrow P \ x \Rightarrow \exists x \in A. P \ x$ ⟨proof⟩

lemma *ballE'*: $\forall x \in A. P \ x \Rightarrow (x \notin A \Rightarrow Q) \Rightarrow (P \ x \Rightarrow Q) \Rightarrow Q$ ⟨proof⟩

lemma *subint-widen-imethds*:

$$\begin{aligned} & \llbracket G \vdash I \preceq I J; \text{wf-prog } G; \text{is-iface } G J; jm \in \text{imethds } G J \text{ sig} \rrbracket \implies \\ & \exists im \in \text{imethds } G I \text{ sig. } \text{is-static } im = \text{is-static } jm \wedge \\ & \quad \text{accmodi } im = \text{accmodi } jm \wedge \\ & \quad G \vdash \text{resTy } im \preceq \text{resTy } jm \end{aligned}$$

$\langle \text{proof} \rangle$

lemma *implmt1-methd*:

$$\begin{aligned} & \wedge \text{sig. } \llbracket G \vdash C \rightsquigarrow I I; \text{wf-prog } G; im \in \text{imethds } G I \text{ sig} \rrbracket \implies \\ & \exists cm \in \text{methd } G C \text{ sig: } \neg \text{is-static } cm \wedge \neg \text{is-static } im \wedge \\ & \quad G \vdash \text{resTy } cm \preceq \text{resTy } im \wedge \\ & \quad \text{accmodi } im = \text{Public} \wedge \text{accmodi } cm = \text{Public} \end{aligned}$$

$\langle \text{proof} \rangle$

lemma *implmt-methd* [rule-format (no-asm)]:

$$\begin{aligned} & \llbracket \text{wf-prog } G; G \vdash C \rightsquigarrow I I \rrbracket \implies \text{is-iface } G I \longrightarrow \\ & (\forall im \in \text{imethds } G I \text{ sig.} \\ & \quad \exists cm \in \text{methd } G C \text{ sig: } \neg \text{is-static } cm \wedge \neg \text{is-static } im \wedge \\ & \quad \quad G \vdash \text{resTy } cm \preceq \text{resTy } im \wedge \\ & \quad \quad \text{accmodi } im = \text{Public} \wedge \text{accmodi } cm = \text{Public}) \end{aligned}$$

$\langle \text{proof} \rangle$

lemma *mheadsD* [rule-format (no-asm)]:

$$\begin{aligned} & emh \in \text{mheads } G S t \text{ sig} \longrightarrow \text{wf-prog } G \longrightarrow \\ & (\exists C D m. t = \text{ClassT } C \wedge \text{declrefT } emh = \text{ClassT } D \wedge \\ & \quad \text{accmethd } G S C \text{ sig} = \text{Some } m \wedge \\ & \quad (\text{declclass } m = D) \wedge \text{mhead } (\text{mthd } m) = (\text{mhd } emh)) \vee \\ & (\exists I. t = \text{IfaceT } I \wedge ((\exists im. im \in \text{accimethds } G (\text{pid } S) I \text{ sig} \wedge \\ & \quad \text{mthd } im = \text{mhd } emh)) \vee \\ & (\exists m. G \vdash \text{Iface } I \text{ accessible-in } (\text{pid } S) \wedge \text{accmethd } G S \text{ Object sig} = \text{Some } m \wedge \\ & \quad \text{accmodi } m \neq \text{Private} \wedge \\ & \quad \text{declrefT } emh = \text{ClassT } \text{Object} \wedge \text{mhead } (\text{mthd } m) = \text{mhd } emh))) \vee \\ & (\exists T m. t = \text{ArrayT } T \wedge G \vdash \text{Array } T \text{ accessible-in } (\text{pid } S) \wedge \\ & \quad \text{accmethd } G S \text{ Object sig} = \text{Some } m \wedge \text{accmodi } m \neq \text{Private} \wedge \\ & \quad \text{declrefT } emh = \text{ClassT } \text{Object} \wedge \text{mhead } (\text{mthd } m) = \text{mhd } emh) \end{aligned}$$

$\langle \text{proof} \rangle$

lemma *mheads-cases* [consumes 2, case-names Class-methd

Iface-methd Iface-Object-methd Array-Object-methd]:

$$\begin{aligned} & \llbracket emh \in \text{mheads } G S t \text{ sig}; \text{wf-prog } G; \\ & \wedge C D m. \llbracket t = \text{ClassT } C; \text{declrefT } emh = \text{ClassT } D; \text{accmethd } G S C \text{ sig} = \text{Some } m; \\ & \quad (\text{declclass } m = D); \text{mhead } (\text{mthd } m) = (\text{mhd } emh) \rrbracket \implies P \text{ emh}; \\ & \wedge I im. \llbracket t = \text{IfaceT } I; im \in \text{accimethds } G (\text{pid } S) I \text{ sig}; \text{mthd } im = \text{mhd } emh \rrbracket \\ & \quad \implies P \text{ emh}; \\ & \wedge I m. \llbracket t = \text{IfaceT } I; G \vdash \text{Iface } I \text{ accessible-in } (\text{pid } S); \\ & \quad \text{accmethd } G S \text{ Object sig} = \text{Some } m; \text{accmodi } m \neq \text{Private}; \end{aligned}$$

$$\begin{aligned} & \text{declrefT } emh = \text{ClassT Object}; \text{mhead (mthd m) = mhd emh} \implies P \text{ emh}; \\ \wedge \text{ T m. } & \llbracket t = \text{ArrayT T}; G \vdash \text{Array T accessible-in (pid S)}; \\ & \text{accmethd G S Object sig = Some m}; \text{accmodi m} \neq \text{Private}; \\ & \text{declrefT } emh = \text{ClassT Object}; \text{mhead (mthd m) = mhd emh} \implies P \text{ emh} \\ & \rrbracket \implies P \text{ emh} \\ & \langle \text{proof} \rangle \end{aligned}$$

lemma *declclassD*[*rule-format*]:
 $\llbracket \text{wf-prog G}; \text{class G C} = \text{Some c}; \text{methd G C sig} = \text{Some m};$
 $\text{class G (declclass m) = Some d} \rrbracket$
 $\implies \text{table-of (methods d) sig} = \text{Some (mthd m)}$
 $\langle \text{proof} \rangle$

lemma *dynmethd-Object*:
assumes *statM*: $\text{methd G Object sig} = \text{Some statM}$ **and**
 $\text{private: accmodi statM} = \text{Private}$ **and**
 $\text{is-cls-C: is-class G C}$ **and**
 wf: wf-prog G
shows $\text{dynmethd G Object C sig} = \text{Some statM}$
 $\langle \text{proof} \rangle$

lemma *wf-imethds-hiding-objmethdsD*:
assumes $\text{old: methd G Object sig} = \text{Some old}$ **and**
 $\text{is-if-I: is-iface G I}$ **and**
 wf: wf-prog G **and**
 $\text{not-private: accmodi old} \neq \text{Private}$ **and**
 $\text{new: new} \in \text{imethds G I sig}$
shows $G \vdash \text{resTy new} \preceq \text{resTy old} \wedge \text{is-static new} = \text{is-static old}$ (**is** ?*P* *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 *statT* and a dynamic class *dynC*. Between both of these types the widening relation holds $G \vdash \text{Class } dynC \preceq \text{statT}$. 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 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-cls*:: prog \Rightarrow ref-ty \Rightarrow qname \Rightarrow bool \Rightarrow bool
 (-, - \vdash - *valid'-lookup'-cls'-for* - [61,61,61,61] 60)

primrec

$G, \text{NullT} \vdash \text{dynC } \text{valid-lookup-cls-for } \text{static-membr} = \text{False}$

$G, \text{IfaceT } I \vdash \text{dynC } \text{valid-lookup-cls-for } \text{static-membr}$
 $= (\text{if } \text{static-membr}$

$\text{then } \text{dynC} = \text{Object}$

$\text{else } G \vdash \text{Class } \text{dynC} \preceq \text{Iface } I)$

$G, \text{ClassT } C \vdash \text{dynC } \text{valid-lookup-cls-for } \text{static-membr} = G \vdash \text{Class } \text{dynC} \preceq \text{Class } C$

$G, \text{ArrayT } T \vdash \text{dynC } \text{valid-lookup-cls-for } \text{static-membr} = (\text{dynC} = \text{Object})$

lemma *valid-lookup-cls-is-class*:

assumes $\text{dynC}: G, \text{statT} \vdash \text{dynC } \text{valid-lookup-cls-for } \text{static-membr}$ **and**

$\text{ty-statT}: \text{isrtype } G \text{ statT}$ **and**

$\text{wf}: \text{wf-prog } G$

shows $\text{is-class } G \text{ dynC}$

$\langle \text{proof} \rangle$

declare *split-paired-All* [simp del] *split-paired-Ex* [simp del]

$\langle \text{ML} \rangle$

lemma *dynamic-mheadsD*:

$\llbracket \text{emh} \in \text{mheads } G \text{ S statT sig};$

$G, \text{statT} \vdash \text{dynC } \text{valid-lookup-cls-for } (\text{is-static } \text{emh});$

$\text{isrtype } G \text{ statT}; \text{wf-prog } G$

$\rrbracket \Longrightarrow \exists m \in \text{dynlookup } G \text{ statT } \text{dynC } \text{sig}:$

$\text{is-static } m = \text{is-static } \text{emh} \wedge G \vdash \text{resTy } m \preceq \text{resTy } \text{emh}$

$\langle \text{proof} \rangle$

declare *split-paired-All* [simp] *split-paired-Ex* [simp]

$\langle \text{ML} \rangle$

lemma *methd-declclass*:

$\llbracket \text{class } G \text{ C} = \text{Some } c; \text{wf-prog } G; \text{methd } G \text{ C } \text{sig} = \text{Some } m \rrbracket$

$\Longrightarrow \text{methd } G (\text{declclass } m) \text{ sig} = \text{Some } m$

$\langle \text{proof} \rangle$

lemma *dynmethd-declclass*:

$\llbracket \text{dynmethd } G \text{ statC } \text{dynC } \text{sig} = \text{Some } m;$

$\text{wf-prog } G; \text{is-class } G \text{ statC}$

$\rrbracket \Longrightarrow \text{methd } G (\text{declclass } m) \text{ sig} = \text{Some } m$

$\langle \text{proof} \rangle$

lemma *dynlookup-declC*:

$\llbracket \text{dynlookup } G \text{ statT } \text{dynC } \text{sig} = \text{Some } m; \text{wf-prog } G;$

$\text{is-class } G \text{ dynC}; \text{isrtype } G \text{ statT}$

$\rrbracket \Longrightarrow G \vdash \text{dynC} \preceq_C (\text{declclass } m) \wedge \text{is-class } G (\text{declclass } m)$

$\langle \text{proof} \rangle$

lemma *dynlookup-Array-declclassD* [simp]:
 $\llbracket \text{dynlookup } G \text{ (ArrayT } T) \text{ Object sig} = \text{Some } dm; \text{wf-prog } G \rrbracket$
 $\implies \text{declclass } dm = \text{Object}$
 <proof>

declare *split-paired-All* [simp del] *split-paired-Ex* [simp del]
 <ML>

lemma *wt-is-type*: $E, dt \models v :: T \implies \text{wf-prog } (\text{prg } E) \longrightarrow$
 $dt = \text{empty-dt} \longrightarrow (\text{case } T \text{ of}$
 $\quad \text{Inl } T \Rightarrow \text{is-type } (\text{prg } E) \ T$
 $\quad | \text{Inr } Ts \Rightarrow \text{Ball } (\text{set } Ts) (\text{is-type } (\text{prg } E)))$

<proof>

declare *split-paired-All* [simp] *split-paired-Ex* [simp]
 <ML>

lemma *ty-expr-is-type*:
 $\llbracket E \vdash e :: -T; \text{wf-prog } (\text{prg } E) \rrbracket \implies \text{is-type } (\text{prg } E) \ T$
 <proof>

lemma *ty-var-is-type*:
 $\llbracket E \vdash v :: T; \text{wf-prog } (\text{prg } E) \rrbracket \implies \text{is-type } (\text{prg } E) \ T$
 <proof>

lemma *ty-exprs-is-type*:
 $\llbracket E \vdash es :: \text{:= } Ts; \text{wf-prog } (\text{prg } E) \rrbracket \implies \text{Ball } (\text{set } Ts) (\text{is-type } (\text{prg } E))$
 <proof>

lemma *static-mheadsD*:
 $\llbracket \text{emh} \in \text{mheads } G \ S \ t \ \text{sig}; \text{wf-prog } G; E \vdash e :: -\text{RefT } t; \text{prg } E = G ;$
 $\quad \text{invmode } (\text{mhd } \text{emh}) \ e \neq \text{IntVir}$
 $\rrbracket \implies \exists m. ((\exists C. t = \text{ClassT } C \wedge \text{accmethd } G \ S \ C \ \text{sig} = \text{Some } m)$
 $\quad \vee (\forall C. t \neq \text{ClassT } C \wedge \text{accmethd } G \ S \ \text{Object } \text{sig} = \text{Some } m)) \wedge$
 $\quad \text{declrefT } \text{emh} = \text{ClassT } (\text{declclass } m) \wedge \text{mhead } (\text{mthd } m) = (\text{mhd } \text{emh})$
 <proof>

lemma *wt-MethdI*:
 $\llbracket \text{methd } G \ C \ \text{sig} = \text{Some } m; \text{wf-prog } G;$
 $\quad \text{class } G \ C = \text{Some } c \rrbracket \implies$
 $\exists T. (\text{prg} = G, \text{cls} = (\text{declclass } m),$
 $\quad \text{lcl} = \text{callee-lcl } (\text{declclass } m) \ \text{sig } (\text{mthd } m)) \vdash \text{Methd } C \ \text{sig} :: -T \wedge G \vdash T \preceq_{\text{resTy}} m$
 <proof>

35 accessibility concerns

lemma *mheads-type-accessible*:
 $\llbracket \text{emh} \in \text{mheads } G \ S \ T \ \text{sig}; \text{wf-prog } G \rrbracket$
 $\implies G \vdash \text{RefT } T \ \text{accessible-in } (\text{pid } S)$
 <proof>

lemma *static-to-dynamic-accessible-from-aux*:

$\llbracket G \vdash m \text{ of } C \text{ accessible-from } accC; wf\text{-prog } G \rrbracket$
 $\implies G \vdash m \text{ in } C \text{ dyn-accessible-from } accC$
 ⟨proof⟩

lemma *static-to-dynamic-accessible-from*:

assumes *stat-acc*: $G \vdash m \text{ of } statC \text{ accessible-from } accC$ **and**
subclseq: $G \vdash dynC \preceq_C statC$ **and**
wf: $wf\text{-prog } G$
shows $G \vdash m \text{ in } dynC \text{ dyn-accessible-from } accC$
 ⟨proof⟩

lemma *static-to-dynamic-accessible-from-static*:

assumes *stat-acc*: $G \vdash m \text{ of } statC \text{ accessible-from } accC$ **and**
static: *is-static* m **and**
wf: $wf\text{-prog } G$
shows $G \vdash m \text{ in } (declclass\ m) \text{ dyn-accessible-from } accC$
 ⟨proof⟩

lemma *dynmethd-member-in*:

assumes m : $dynmethd\ G\ statC\ dynC\ sig = Some\ m$ **and**
iscls-statC: *is-class* $G\ statC$ **and**
wf: $wf\text{-prog } G$
shows $G \vdash Methd\ sig\ m \text{ member-in } dynC$
 ⟨proof⟩

lemma *dynmethd-access-prop*:

assumes *statM*: $methd\ G\ statC\ sig = Some\ statM$ **and**
stat-acc: $G \vdash Methd\ sig\ statM \text{ of } statC \text{ accessible-from } accC$ **and**
dynM: $dynmethd\ G\ statC\ dynC\ sig = Some\ dynM$ **and**
wf: $wf\text{-prog } G$
shows $G \vdash Methd\ sig\ dynM \text{ in } dynC \text{ dyn-accessible-from } accC$
 ⟨proof⟩

lemma *implmt-methd-access*:

fixes $accC::qname$
assumes *iface-methd*: $imethds\ G\ I\ sig \neq \{\}$ **and**
implements: $G \vdash dynC \rightsquigarrow I$ **and**
isif-I: *is-iface* $G\ I$ **and**
wf: $wf\text{-prog } G$
shows $\exists\ dynM. methd\ G\ dynC\ sig = Some\ dynM \wedge$
 $G \vdash Methd\ sig\ dynM \text{ in } dynC \text{ dyn-accessible-from } accC$
 ⟨proof⟩

corollary *implmt-dynimethd-access*:

fixes $accC::qname$
assumes *iface-methd*: $imethds\ G\ I\ sig \neq \{\}$ **and**
implements: $G \vdash dynC \rightsquigarrow I$ **and**
isif-I: *is-iface* $G\ I$ **and**
wf: $wf\text{-prog } G$
shows $\exists\ dynM. dynimethd\ G\ I\ dynC\ sig = Some\ dynM \wedge$
 $G \vdash Methd\ sig\ dynM \text{ in } dynC \text{ dyn-accessible-from } accC$
 ⟨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: $G, statT \vdash dynC\ valid-lookup-cls-for\ is-static\ emh$ **and**
isT-statT: $isrtype\ G\ statT$ **and**
wf: $wf-prog\ G$

shows $G \vdash Methd\ sig\ dynM\ in\ dynC\ dyn-accessible-from\ accC$
 ⟨proof⟩

lemma *dynlookup-access*:

assumes *emh*: $emh \in mheads\ G\ accC\ statT\ sig$ **and**
dynC-prop: $G, statT \vdash dynC\ valid-lookup-cls-for\ (is-static\ emh)$ **and**
isT-statT: $isrtype\ G\ statT$ **and**
wf: $wf-prog\ G$

shows $\exists\ dynM. dynlookup\ G\ statT\ dynC\ sig = Some\ dynM \wedge$
 $G \vdash Methd\ sig\ dynM\ in\ dynC\ dyn-accessible-from\ accC$
 ⟨proof⟩

lemma *stat-overrides-Package-old*:

assumes *stat-override*: $G \vdash new\ overrides_s\ old$ **and**
accmodi-new: $accmodi\ new = Package$ **and**
wf: $wf-prog\ G$

shows $accmodi\ old = Package$
 ⟨proof⟩

Properties of dynamic accessibility

lemma *dyn-accessible-Private*:

assumes *dyn-acc*: $G \vdash m\ in\ C\ dyn-accessible-from\ accC$ **and**
priv: $accmodi\ m = Private$

shows $accC = declclass\ m$
 ⟨proof⟩

dyn-accessible-Package only works with the *wf-prog* assumption. Without it. it is easy to leaf the Package!

lemma *dyn-accessible-Package*:

$\llbracket G \vdash m\ in\ C\ dyn-accessible-from\ accC; accmodi\ m = Package;$
 $wf-prog\ G \rrbracket$

$\implies pid\ accC = pid\ (declclass\ m)$
 ⟨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*: $G \vdash f\ in\ C\ dyn-accessible-from\ accC$ **and**
pack: $accmodi\ f = Package$ **and**

field: $is-field\ f$

shows $pid\ accC = pid\ (declclass\ f)$
 ⟨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*: $G \vdash f\ in\ C\ dyn-accessible-from\ accC$ **and**
prot: $accmodi\ f = Protected$ **and**

field: $is-field\ f$ **and**

instance-field: \neg *is-static* f **and**
outside: $\text{pid}(\text{declclass } f) \neq \text{pid } \text{acc}C$
shows $G \vdash C \preceq_C \text{acc}C$
 ⟨*proof*⟩

lemma *dyn-accessible-static-field-Protected*:
assumes *dyn-acc*: $G \vdash f$ in C *dyn-accessible-from* $\text{acc}C$ **and**
prot: $\text{accmodi } f = \text{Protected}$ **and**
field: *is-field* f **and**
static-field: *is-static* f **and**
outside: $\text{pid}(\text{declclass } f) \neq \text{pid } \text{acc}C$
shows $G \vdash \text{acc}C \preceq_C \text{declclass } f \wedge G \vdash C \preceq_C \text{declclass } f$
 ⟨*proof*⟩
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*) (\downarrow *tag*::*obj-tag*, *values*::*vn* \Rightarrow *val option*)
obj <= (*type*) (\downarrow *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* (\downarrow *tag=Arr T k,values=t)*

lemma *the-Arr-Arr* [*simp*]: *the-Arr* (*Some* (\downarrow *tag=Arr T k,values=cs)) = (*T,k,cs*)
 <*proof*>*

lemma *the-Arr-Arr1* [*simp,intro,dest*]:
 [\downarrow *tag obj = Arr T k] \Longrightarrow *the-Arr* (*Some obj*) = (*T,k,values obj*)
 <*proof*>*

constdefs

upd-obj :: *vn* \Rightarrow *val* \Rightarrow *obj* \Rightarrow *obj*
upd-obj n v \equiv λ *obj* . *obj* (\downarrow *values:=*(*values obj*)(*n* \mapsto *v*))

lemma *upd-obj-def2* [*simp*]:
upd-obj n v obj = *obj* (\downarrow *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!]: $obj\text{-}ty\ (\!|tag=oi,values=x\!) = obj\text{-}ty\ (\!|tag=oi,values=y\!)$
 ⟨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]:
 $obj\text{-}ty\ (obj\ (\!|values:=vs\!)) = obj\text{-}ty\ obj$
 ⟨proof⟩

lemma *obj-ty-CInst* [simp]:
 $obj\text{-}ty\ (\!|tag=CInst\ C,values=vs\!) = Class\ C$
 ⟨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]:
 $obj\text{-}ty\ (\!|tag=Arr\ T\ i,values=vs\!) = T.[]$
 ⟨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]: $obj\text{-}class\ (\!|tag=CInst\ C,values=vs\!) = C$
 ⟨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* ($\text{tag}=\text{Arr } T \ k, \text{values}=\text{vs}$) = *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* ((-/ 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*
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*

$$\begin{aligned} &\equiv \text{case } r \text{ of} \\ &\quad \text{Heap } a \Rightarrow (\text{case } oi \text{ of} \\ &\quad\quad \text{CInst } C \Rightarrow \text{fields-table } G \ C \ (\lambda n \ f. \neg \text{static } f) \ (+) \ \text{empty} \\ &\quad\quad | \text{Arr } T \ k \Rightarrow \text{empty } (+) \ \text{arr-comps } T \ k) \\ &\quad | \text{Stat } C \Rightarrow \text{fields-table } G \ C \ (\lambda fn \ f. \text{declclassf } fn = C \wedge \text{static } f) \\ &\quad\quad (+) \ \text{empty} \end{aligned}$$

lemma *var-tys-Some-eq*:

var-tys $G \ oi \ r \ n = \text{Some } T$

$$\begin{aligned} &= (\text{case } r \text{ of} \\ &\quad \text{Inl } a \Rightarrow (\text{case } oi \text{ of} \\ &\quad\quad \text{CInst } C \Rightarrow (\exists nt. n = \text{Inl } nt \wedge \text{fields-table } G \ C \ (\lambda n \ f. \\ &\quad\quad\quad \neg \text{static } f) \ nt = \text{Some } T)) \\ &\quad\quad | \text{Arr } t \ k \Rightarrow (\exists i. n = \text{Inr } i \wedge i \text{ in-bounds } k \wedge t = T)) \\ &\quad | \text{Inr } C \Rightarrow (\exists nt. n = \text{Inl } nt \wedge \\ &\quad\quad \text{fields-table } G \ C \ (\lambda fn \ f. \text{declclassf } fn = C \wedge \text{static } f) \ nt \\ &\quad\quad = \text{Some } T)) \end{aligned}$$

$\langle \text{proof} \rangle$

stores

types *globs* — global variables: heap and static variables
 $= (\text{oref} \ , \ \text{obj}) \ \text{table}$
heap
 $= (\text{loc} \ , \ \text{obj}) \ \text{table}$

translations

globs $\leq (\text{type}) (\text{oref} \ , \ \text{obj}) \ \text{table}$
heap $\leq (\text{type}) (\text{loc} \ , \ \text{obj}) \ \text{table}$

datatype *st* =

st *globs* *locals*

37 access

constdefs

globs $:: st \Rightarrow \text{globs}$
globs $\equiv \text{st-case } (\lambda g \ l. \ g)$

locals $:: st \Rightarrow \text{locals}$
locals $\equiv \text{st-case } (\lambda g \ l. \ l)$

heap $:: st \Rightarrow \text{heap}$
heap $s \equiv \text{globs } s \circ \text{Heap}$

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*]: $\text{heap } s \text{ a} = \text{globs } s \text{ (Heap } a)$
 ⟨*proof*⟩

syntax

$\text{val-this} \quad :: \text{st} \Rightarrow \text{val}$
 $\text{lookup-obj} \quad :: \text{st} \Rightarrow \text{val} \Rightarrow \text{obj}$

translations

$\text{val-this } s \quad == \text{the (locals } s \text{ This)}$
 $\text{lookup-obj } s \text{ a}' \quad == \text{the (heap } s \text{ (the-Addr } a'))$

38 memory allocation

constdefs

$\text{new-Addr} \quad :: \text{heap} \Rightarrow \text{loc option}$
 $\text{new-Addr } h \quad \equiv \text{if } (\forall a. h \text{ a} \neq \text{None}) \text{ then None else Some (SOME } a. h \text{ a} = \text{None})$

lemma *new-AddrD*: $\text{new-Addr } h = \text{Some } a \implies h \text{ a} = \text{None}$
 ⟨*proof*⟩

lemma *new-AddrD2*: $\text{new-Addr } h = \text{Some } a \implies \forall b. h \text{ b} \neq \text{None} \longrightarrow b \neq a$
 ⟨*proof*⟩

lemma *new-Addr-SomeI*: $h \text{ a} = \text{None} \implies \exists b. \text{new-Addr } h = \text{Some } b \wedge h \text{ b} = \text{None}$
 ⟨*proof*⟩

39 initialization

syntax

$\text{init-vals} \quad :: ('a, \text{ty}) \text{ table} \Rightarrow ('a, \text{val}) \text{ table}$

translations

$\text{init-vals } vs \quad == \text{option-map default-val} \circ vs$

lemma *init-arr-comps-base* [*simp*]: $\text{init-vals (arr-comps } T \ 0) = \text{empty}$
 ⟨*proof*⟩

lemma *init-arr-comps-step* [*simp*]:

$0 < j \implies \text{init-vals (arr-comps } T \ j) =$
 $\quad \text{init-vals (arr-comps } T \ (j - 1))(j - 1 \mapsto \text{default-val } T)$
 ⟨*proof*⟩

40 update

constdefs

$\text{gupd} \quad :: \text{oref} \Rightarrow \text{obj} \Rightarrow \text{st} \Rightarrow \text{st} \quad (\text{gupd}'(\mapsto)[10,10]1000)$
 $\text{gupd } r \text{ obj} \equiv \text{st-case } (\lambda g \text{ l. st } (g(r \mapsto \text{obj}))) \text{ l}$

$\text{lupd} \quad :: \text{lname} \Rightarrow \text{val} \Rightarrow \text{st} \Rightarrow \text{st} \quad (\text{lupd}'(\mapsto)[10,10]1000)$
 $\text{lupd } vn \text{ v} \equiv \text{st-case } (\lambda g \text{ l. st } g \text{ (l(vn} \mapsto \text{v})))$

$\text{upd-gobj} \quad :: \text{oref} \Rightarrow \text{vn} \Rightarrow \text{val} \Rightarrow \text{st} \Rightarrow \text{st}$

$upd-gobj\ r\ n\ v \equiv st-case\ (\lambda g\ l.\ st\ (chg-map\ (upd-obj\ n\ v)\ r\ g)\ l)$

$set-locals\ ::\ locals \Rightarrow st \Rightarrow st$
 $set-locals\ l \equiv st-case\ (\lambda g\ l'.\ st\ g\ l)$

$init-obj\ ::\ prog \Rightarrow obj-tag \Rightarrow oref \Rightarrow st \Rightarrow st$
 $init-obj\ G\ oi\ r \equiv gupd(r \mapsto (\!|tag=oi, values=init-vals\ (var-tys\ G\ oi\ r)\!|))$

syntax

$init-class-obj\ ::\ prog \Rightarrow qname \Rightarrow st \Rightarrow st$

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]: $globs\ (init-obj\ G\ oi\ r\ s)\ t =$
 $(if\ t=r\ then\ Some\ (\!|tag=oi, values=init-vals\ (var-tys\ G\ oi\ r)\!|)\ else\ globs\ s\ t)$
 $\langle proof \rangle$

lemma *locals-init-obj* [simp]: $locals (init-obj\ G\ oi\ r\ s) = locals\ s$
 ⟨proof⟩

lemma *surjective-st* [simp]: $st (globs\ s) (locals\ s) = s$
 ⟨proof⟩

lemma *surjective-st-init-obj*:
 $st (globs (init-obj\ G\ oi\ r\ s)) (locals\ s) = init-obj\ G\ oi\ r\ s$
 ⟨proof⟩

lemma *heap-heap-upd* [simp]:
 $heap (st (g(Inl\ a\mapsto\ obj))\ l) = heap (st\ g\ l)(a\mapsto\ obj)$
 ⟨proof⟩

lemma *heap-stat-upd* [simp]: $heap (st (g(Inr\ C\mapsto\ obj))\ l) = heap (st\ g\ l)$
 ⟨proof⟩

lemma *heap-local-upd* [simp]: $heap (st\ g (l(vn\mapsto\ v))) = heap (st\ g\ l)$
 ⟨proof⟩

lemma *heap-gupd-Heap* [simp]: $heap (gupd(Heap\ a\mapsto\ obj)\ s) = heap\ s(a\mapsto\ obj)$
 ⟨proof⟩

lemma *heap-gupd-Stat* [simp]: $heap (gupd(Stat\ C\mapsto\ obj)\ s) = heap\ s$
 ⟨proof⟩

lemma *heap-lupd* [simp]: $heap (lupd(vn\mapsto\ v)\ s) = heap\ s$
 ⟨proof⟩

lemma *heap-upd-gobj-Stat* [simp]: $heap (upd-gobj (Stat\ C)\ n\ v\ s) = heap\ s$
 ⟨proof⟩

lemma *set-locals-def2* [simp]: $set-locals\ l (st\ g\ l') = st\ g\ l$
 ⟨proof⟩

lemma *set-locals-id* [simp]: $set-locals (locals\ s)\ s = s$
 ⟨proof⟩

lemma *set-set-locals* [simp]: $set-locals\ l (set-locals\ l'\ s) = set-locals\ l\ s$
 ⟨proof⟩

lemma *locals-set-locals* [simp]: $locals (set-locals\ l\ s) = l$
 ⟨proof⟩

lemma *globs-set-locals* [simp]: $globs (set-locals\ l\ s) = globs\ s$
 ⟨proof⟩

lemma *heap-set-locals* [simp]: $heap (set-locals\ l\ s) = heap\ s$

<proof>

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*
<proof>

lemma *abrupt-if-True-not-None* [*simp*]: *x* \neq *None* \implies *abrupt-if* *True* *x* *y* \neq *None*
<proof>

lemma *abrupt-if-False* [*simp*]: *abrupt-if* *False* *x* *y* = *y*
<proof>

lemma *abrupt-if-Some* [*simp*]: *abrupt-if* *c* *x* (*Some* *y*) = *Some* *y*
<proof>

lemma *abrupt-if-not-None* [*simp*]: *y* \neq *None* \implies *abrupt-if* *c* *x* *y* = *y*
<proof>

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*))
<proof>

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

full program state**types**

$state = abopt \times st$ — state including abruptio 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*: $\forall Z. Z = (s::state) \implies False$
 ⟨proof⟩

lemma *state-not-single*: $All (op = (x::state)) \implies R$
 ⟨proof⟩

constdefs

$normal :: state \Rightarrow bool$
 $normal \equiv \lambda s. abrupt\ s = None$

lemma *normal-def2* [simp]: $normal\ s = (abrupt\ s = None)$
 ⟨proof⟩

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]: $heap-free\ n\ s = atleast-free\ (heap\ (store\ s))\ n$
 ⟨proof⟩

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]: $abupd\ f\ (x, s) = (f\ x, s)$

<proof>

lemma *abupd-abrupt-if-False* [simp]: $\bigwedge s. \text{abupd} (\text{abrupt-if False } x) s = s$
<proof>

lemma *supd-def2* [simp]: $\text{supd } f (x, s) = (x, f s)$
<proof>

lemma *supd-lupd* [simp]:
 $\bigwedge s. \text{supd} (\text{lupd } vn \ v) s = (\text{abrupt } s, \text{lupd } vn \ v (\text{store } s))$
<proof>

lemma *supd-gupd* [simp]:
 $\bigwedge s. \text{supd} (\text{gupd } r \ \text{obj}) s = (\text{abrupt } s, \text{gupd } r \ \text{obj} (\text{store } s))$
<proof>

lemma *supd-init-obj* [simp]:
 $\text{supd} (\text{init-obj } G \ \text{oi } r) s = (\text{abrupt } s, \text{init-obj } G \ \text{oi } r (\text{store } s))$
<proof>

lemma *abupd-store-invariant* [simp]: $\text{store} (\text{abupd } f \ s) = \text{store } s$
<proof>

lemma *supd-abrupt-invariant* [simp]: $\text{abrupt} (\text{supd } f \ s) = \text{abrupt } s$
<proof>

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]: $\bigwedge s. \text{set-lvars } l (\text{set-lvars } l' \ s) = \text{set-lvars } l \ s$
<proof>

lemma *set-lvars-id* [simp]: $\bigwedge s. \text{set-lvars} (\text{locals} (\text{store } s)) s = s$
<proof>

initialisation test

constdefs

inited :: *qname* \Rightarrow *globs* \Rightarrow *bool*
inited *C* *g* \equiv *g* (*Stat* *C*) \neq *None*

initd :: *qname* \Rightarrow *state* \Rightarrow *bool*
initd *C* \equiv *initd* *C* \circ *globs* \circ *store*

lemma *not-initd-empty* [*simp*]: \neg *initd* *C* *empty*
 <proof>

lemma *initd-gupdate* [*simp*]: *initd* *C* (*g*(*r* \mapsto *obj*)) = (*initd* *C* *g* \vee *r* = *Stat* *C*)
 <proof>

lemma *initd-init-class-obj* [*intro!*]: *initd* *C* (*globs* (*init-class-obj* *G* *C* *s*))
 <proof>

lemma *not-initdD*: \neg *initd* *C* *g* \Longrightarrow *g* (*Stat* *C*) = *None*
 <proof>

lemma *initdD*: *initd* *C* *g* \Longrightarrow \exists *obj*. *g* (*Stat* *C*) = *Some* *obj*
 <proof>

lemma *initd-def2* [*simp*]: *initd* *C* *s* = *initd* *C* (*globs* (*store* *s*))
 <proof>

error-free

constdefs *error-free*:: *state* \Rightarrow *bool*
error-free *s* \equiv \neg (\exists *err*. *abrupt* *s* = *Some* (*Error* *err*))

lemma *error-free-Norm* [*simp,intro*]: *error-free* (*Norm* *s*)
 <proof>

lemma *error-free-normal* [*simp,intro*]: *normal* *s* \Longrightarrow *error-free* *s*
 <proof>

lemma *error-free-Xcpt* [*simp*]: *error-free* (*Some* (*Xcpt* *x*),*s*)
 <proof>

lemma *error-free-Jump* [*simp,intro*]: *error-free* (*Some* (*Jump* *j*),*s*)
 <proof>

lemma *error-free-Error* [*simp*]: *error-free* (*Some* (*Error* *e*),*s*) = *False*
 <proof>

lemma *error-free-Some* [*simp,intro*]:
 \neg (\exists *err*. *x*=*Error* *err*) \Longrightarrow *error-free* ((*Some* *x*),*s*)
 <proof>

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*]:
 $error\text{-}free\ (a,s) \implies error\text{-}free\ (a,f\ s)$
<proof>

lemma *error-free-set-lvars* [*simp,intro*]:
 $error\text{-}free\ s \implies error\text{-}free\ ((set\text{-}lvars\ l)\ s)$
<proof>

lemma *error-free-set-locals* [*simp,intro*]:
 $error\text{-}free\ (x, s)$
 $\implies error\text{-}free\ (x, set\text{-}locals\ l\ s')$
<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
- `halloc` 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

$$\begin{aligned} arbitrary3 &:: ('a1 + 'ar, 'b, 'c)\ sum3 \Rightarrow vals \\ arbitrary3 &\equiv sum3\text{-case}\ (In1 \circ sum\text{-case}\ (\lambda x. arbitrary))\ (\lambda x. Unit)) \\ &\quad (\lambda x. In2\ arbitrary)\ (\lambda x. In3\ arbitrary) \end{aligned}$$

lemma [simp]: $arbitrary3\ (In1\ x) = In1\ arbitrary$
 <proof>

lemma [simp]: $arbitrary3\ (In1r\ x) = \diamond$
 <proof>

lemma [simp]: $arbitrary3\ (In2\ x) = In2\ arbitrary$
 <proof>

lemma [simp]: $arbitrary3\ (In3\ x) = In3\ arbitrary$
 <proof>

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*:

$$throw\ a'\ x = abrupt\text{-if}\ True\ (Some\ (Xcpt\ (Loc\ (the\text{-}Addr\ a'))))\ (np\ a'\ x)$$
 <proof>
constdefs

$$\begin{aligned} fits &:: prog \Rightarrow st \Rightarrow val \Rightarrow ty \Rightarrow bool\ (-, \text{-} \text{-}\ fits\ \text{-}[61,61,61,61]60) \\ G, s \vdash a'\ fits\ T &\equiv (\exists rt. T = RefT\ rt) \longrightarrow a' = Null \vee G \vdash obj\text{-}ty\ (lookup\text{-}obj\ s\ a') \preceq T \end{aligned}$$

lemma *fits-Null* [simp]: $G, s \vdash Null\ fits\ T$
 <proof>

lemma *fits-Addr-RefT* [simp]:

$$G, s \vdash Addr\ a\ fits\ RefT\ t = G \vdash obj\text{-}ty\ (the\ (heap\ s\ a)) \preceq RefT\ t$$
 <proof>

lemma *fitsD*: $\bigwedge X. G, s \vdash a'\ fits\ T \implies (\exists pt. T = PrimT\ pt) \vee$
 $(\exists t. T = RefT\ t) \wedge a' = Null \vee$
 $(\exists t. T = RefT\ t) \wedge a' \neq Null \wedge G \vdash obj\text{-}ty\ (lookup\text{-}obj\ s\ a') \preceq T$
 <proof>

constdefs

$$\begin{aligned} catch &:: prog \Rightarrow state \Rightarrow qname \Rightarrow bool\ (-, \text{-} \text{-}\ catch\ \text{-}[61,61,61]60) \\ G, s \vdash catch\ C &\equiv \exists xc. abrupt\ s = Some\ (Xcpt\ xc) \wedge \end{aligned}$$

$G, store \vdash Addr (the-Loc xc) \text{ fits Class } C$

lemma *catch-Norm* [simp]: $\neg G, Norm \vdash catch \ tn$
 ⟨proof⟩

lemma *catch-XcptLoc* [simp]:
 $G, (Some (Xcpt (Loc a)), s) \vdash catch \ C = G, s \vdash Addr \ a \ \text{fits Class } C$
 ⟨proof⟩

lemma *catch-Jump* [simp]: $\neg G, (Some (Jump j), s) \vdash catch \ tn$
 ⟨proof⟩

lemma *catch-Error* [simp]: $\neg G, (Some (Error e), s) \vdash catch \ tn$
 ⟨proof⟩

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 $vn \ (x, s) =$
 $Norm (lupd (VName \ vn \mapsto Addr (the-Loc (the-Xcpt (the \ x)))) \ s)$
 ⟨proof⟩

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]:
 $f \ v \ (Norm \ s) = Norm \ s' \Longrightarrow assign \ f \ v \ (Norm \ s) = Norm \ s'$
 ⟨proof⟩

lemma *assign-Norm-Some* [simp]:
 $\llbracket abrupt \ (f \ v \ (Norm \ s)) = Some \ y \rrbracket$
 $\Longrightarrow assign \ f \ v \ (Norm \ s) = (Some \ y, s)$
 ⟨proof⟩

lemma *assign-Some* [simp]:
 $assign \ f \ v \ (Some \ x, s) = (Some \ x, s)$
 ⟨proof⟩

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 :: $ty \Rightarrow \text{stmt}$
 $\text{init-comp-ty } T \equiv \text{if } (\exists C. T = \text{Class } C) \ \text{then } \text{Init } (\text{the-Class } T) \ \text{else } \text{Skip}$

lemma *init-comp-ty-PrimT* [simp]: $\text{init-comp-ty } (\text{PrimT } pt) = \text{Skip}$
 ⟨proof⟩

constdefs

invocation-class :: $\text{inv-mode} \Rightarrow \text{st} \Rightarrow \text{val} \Rightarrow \text{ref-ty} \Rightarrow \text{qname}$
 $\text{invocation-class } m \ s \ a' \ \text{statT}$
 $\equiv (\text{case } m \ \text{of}$
 $\text{Static} \Rightarrow \text{if } (\exists \text{statC}. \text{statT} = \text{ClassT } \text{statC})$
 $\text{then } \text{the-Class } (\text{RefT } \text{statT})$
 $\text{else } \text{Object}$
 $| \text{SuperM} \Rightarrow \text{the-Class } (\text{RefT } \text{statT})$
 $| \text{IntVir} \Rightarrow \text{obj-class } (\text{lookup-obj } s \ a')$

invocation-declclass:: $\text{prog} \Rightarrow \text{inv-mode} \Rightarrow \text{st} \Rightarrow \text{val} \Rightarrow \text{ref-ty} \Rightarrow \text{sig} \Rightarrow \text{qname}$
 $\text{invocation-declclass } G \ m \ s \ a' \ \text{statT } \text{sig}$
 $\equiv \text{declclass } (\text{the } (\text{dynlookup } G \ \text{statT}$
 $(\text{invocation-class } m \ s \ a' \ \text{statT})$
 $\text{sig}))$

lemma *invocation-class-IntVir* [simp]:
 $\text{invocation-class } \text{IntVir } s \ a' \ \text{statT} = \text{obj-class } (\text{lookup-obj } s \ a')$
 ⟨proof⟩

lemma *dynclass-SuperM* [simp]:
 $\text{invocation-class } \text{SuperM } s \ a' \ \text{statT} = \text{the-Class } (\text{RefT } \text{statT})$
 ⟨proof⟩

lemma *invocation-class-Static* [simp]:
 $\text{invocation-class } \text{Static } s \ a' \ \text{statT} = (\text{if } (\exists \text{statC}. \text{statT} = \text{ClassT } \text{statC})$
 $\text{then } \text{the-Class } (\text{RefT } \text{statT})$

else Object)

<proof>

constdefs

init-lvars :: *prog* ⇒ *qname* ⇒ *sig* ⇒ *inv-mode* ⇒ *val* ⇒ *val list* ⇒
state ⇒ *state*

init-lvars *G C sig mode a' pvs*

≡ λ (*x,s*).

let *m* = *mthd* (*the* (*methd* *G C sig*));

l = λ *k*.

(*case* *k* of

EName *e*

⇒ (*case* *e* of

VNam *v* ⇒ (*empty* ((*pars* *m*)[*↦*]*pvs*)) *v*

| *Res* ⇒ *None*)

| *This*

⇒ (*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

(λ *k*.

(*case* *k* of

EName *e*

⇒ (*case* *e* of

VNam *v*

⇒ (*empty* ((*pars* (*mthd* (*the* (*methd* *G C sig*))))[*↦*]*pvs*)) *v*

| *Res* ⇒ *None*)

| *This*

⇒ (*if* *mode*=*Static* then *None* else *Some* *a'*))

(*if* *mode* = *Static* then *x* else *np* *a' x,s*)

<proof>

constdefs

body :: *prog* ⇒ *qname* ⇒ *sig* ⇒ *expr*

body *G C sig* ≡ let *m* = *the* (*methd* *G C sig*)

in *Body* (*declclass* *m*) (*stmt* (*mbody* (*mthd* *m*)))

lemma *body-def2*: — better suited for simplification

body *G C sig* = *Body* (*declclass* (*the* (*methd* *G C sig*)))

(*stmt* (*mbody* (*mthd* (*the* (*methd* *G C sig*))))))

<proof>

variables

constdefs

lvar :: *lname* ⇒ *st* ⇒ *vvar*

lvar *vn s* ≡ (*the* (*locals* *s vn*), λ*v*. *supd* (*lupd*(*vn*→*v*)))

fvar :: *qname* ⇒ *bool* ⇒ *vname* ⇒ *val* ⇒ *state* ⇒ *vvar* × *state*

fvar *C stat fn a' s*

≡ let (*oref*,*xf*) = *if* *stat* then (*Stat* *C*,*id*)

else (*Heap* (*the-Addr* *a'*),*np* *a'*);

```

      n = Inl (fn,C);
      f = (λv. supd (upd-gobj oref n v))
in ((the (values (the (globs (store s) oref)) n),f),abupd xf s)

```

```

avar :: prog ⇒ val ⇒ val ⇒ state ⇒ vvar × state
avar G i' a' s
≡ let oref = Heap (the-Addr a');
    i = the-Intg i';
    n = Inr i;
    (T,k,cs) = the-Arr (globs (store s) oref);
    f = (λv (x,s). (raise-if (¬G,s⊢v fits T)
                          ArrStore x
                          ,upd-gobj oref n v s))
in ((the (cs n),f)
    ,abupd (raise-if (¬i in-bounds k) IndOutOfBounds ∘ np a') s)

```

lemma fvar-def2: — better suited for simplification

```

fvar C stat fn a' s =
((the
  (values
    (the (globs (store s) (if stat then Stat C else Heap (the-Addr a'))))
    (Inl (fn,C)))
  ,(λv. supd (upd-gobj (if stat then Stat C else Heap (the-Addr a'))
                (Inl (fn,C))
                v)))
,abupd (if stat then id else np a') s)

```

⟨proof⟩

lemma avar-def2: — better suited for simplification

```

avar G i' a' s =
((the ((snd(snd(the-Arr (globs (store s) (Heap (the-Addr a'))))))
      (Inr (the-Intg i'))
    ,(λv (x,s'). (raise-if (¬G,s⊢v fits (fst(the-Arr (globs (store s)
      (Heap (the-Addr a'))))))
      ArrStore x
      ,upd-gobj (Heap (the-Addr a'))
                (Inr (the-Intg i')) v s'))
    ,abupd (raise-if (¬(the-Intg i') in-bounds (fst(snd(the-Arr (globs (store s)
      (Heap (the-Addr a')))))) IndOutOfBounds ∘ np a')
    s)

```

⟨proof⟩

constdefs

```

check-field-access::
prog ⇒ qname ⇒ qname ⇒ vname ⇒ bool ⇒ val ⇒ state ⇒ state
check-field-access G accC statDeclC fn stat a' s
≡ let oref = if stat then Stat statDeclC
    else Heap (the-Addr a');
    dynC = case oref of
      Heap a ⇒ obj-class (the (globs (store s) oref))
    | Stat C ⇒ C;
    f = (the (table-of (DeclConcepts.fields G dynC) (fn,statDeclC)))
in abupd
  (error-if (¬ G⊢Field fn (statDeclC,f) in dynC dyn-accessible-from accC)
    AccessViolation)

```

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 $G accC statT mode sig a' s$
 $\equiv let\ invC = invocation\text{-}class\ mode\ (store\ s)\ a'\ statT;$
 $\quad dynM = the\ (dynlookup\ G\ statT\ invC\ sig)$
 in *abupd*
 $(error\text{-}if\ (\neg\ G \vdash Methd\ sig\ dynM\ in\ invC\ dyn\text{-}accessible\text{-}from\ accC)$
 $\quad AccessViolation)$
 s

evaluation judgments**inductive**

halloc :: $[prog, state, obj\text{-}tag, loc, state] \Rightarrow bool$ ($\vdash -\text{-}halloc \text{-}\text{-}\rightarrow -[61, 61, 61, 61, 61]60$) **for** $G::prog$
where — allocating objects on the heap, cf. 12.5

Abrupt:

$G \vdash (Some\ x, s) \text{-}halloc\ oi \text{-}\text{-}arbitrary \rightarrow (Some\ x, s)$

| *New*: $\llbracket new\text{-}Addr\ (heap\ s) = Some\ a;$
 $(x, oi') = (if\ atleast\text{-}free\ (heap\ s)\ (Suc\ (Suc\ 0))\ then\ (None, oi)$
 $\quad else\ (Some\ (Xcpt\ (Loc\ a)), CInst\ (SXcpt\ OutOfMemory)) \rrbracket$
 \implies
 $G \vdash Norm\ s \text{-}halloc\ oi \text{-}\text{-}a \rightarrow (x, init\text{-}obj\ G\ oi'\ (Heap\ a)\ s)$

inductive *sxalloc* :: $[prog, state, state] \Rightarrow bool$ ($\vdash -\text{-}sxalloc \text{-}\text{-}\rightarrow -[61, 61, 61]60$) **for** $G::prog$
where — allocating exception objects for standard exceptions (other than OutOfMemory)

Norm: $G \vdash Norm \quad s \text{-}sxalloc \rightarrow Norm \quad s$

| *Jmp*: $G \vdash (Some\ (Jump\ j), s) \text{-}sxalloc \rightarrow (Some\ (Jump\ j), s)$

| *Error*: $G \vdash (Some\ (Error\ e), s) \text{-}sxalloc \rightarrow (Some\ (Error\ e), s)$

| *XcptL*: $G \vdash (Some\ (Xcpt\ (Loc\ a)), s) \text{-}sxalloc \rightarrow (Some\ (Xcpt\ (Loc\ a)), s)$

| *SXcpt*: $\llbracket G \vdash Norm\ s0 \text{-}halloc\ (CInst\ (SXcpt\ xn)) \text{-}\text{-}a \rightarrow (x, s1) \rrbracket \implies$
 $G \vdash (Some\ (Xcpt\ (Std\ xn)), s0) \text{-}sxalloc \rightarrow (Some\ (Xcpt\ (Loc\ a)), s1)$

inductive

eval :: $[prog, state, term, vals, state] \Rightarrow bool$ ($\vdash -\text{-}eval \text{-}\text{-}\rightarrow '(-, -)' [61, 61, 80, 0, 0]60$)
and *exec* :: $[prog, state, stmt, state] \Rightarrow bool$ ($\vdash -\text{-}exec \text{-}\text{-}\rightarrow - [61, 61, 65, 61]60$)
and *eval'* :: $[prog, state, var, vvar, state] \Rightarrow bool$ ($\vdash -\text{-}eval' \text{-}\text{-}\rightarrow - [61, 61, 90, 61, 61]60$)
and *eval''* :: $[prog, state, expr, val, state] \Rightarrow bool$ ($\vdash -\text{-}eval'' \text{-}\text{-}\rightarrow - [61, 61, 80, 61, 61]60$)
and *evals* :: $[prog, state, expr\ list, val\ list, state] \Rightarrow bool$ ($\vdash -\text{-}evals \text{-}\text{-}\rightarrow - [61, 61, 61, 61, 61]60$)

for $G::prog$

where

$G \vdash s \text{-}c \rightarrow s' \equiv G \vdash s \text{-}In1r\ c \text{-}\text{-}\rightarrow (\diamond, s')$
| $G \vdash s \text{-}e \text{-}\text{-}v \rightarrow s' \equiv G \vdash s \text{-}In1l\ e \text{-}\text{-}\rightarrow (In1\ v, s')$
| $G \vdash s \text{-}e \text{-}\text{-}vf \rightarrow s' \equiv G \vdash s \text{-}In2\ e \text{-}\text{-}\rightarrow (In2\ vf, s')$
| $G \vdash s \text{-}e \text{-}\text{-}v \rightarrow s' \equiv G \vdash s \text{-}In3\ e \text{-}\text{-}\rightarrow (In3\ v, s')$

— propagation of abrupt completion

— cf. 14.1, 15.5

| *Abrupt*:
 $G \vdash (\text{Some } xc, s) -t \succ \rightarrow (\text{arbitrary} \exists t, (\text{Some } xc, s))$

— execution of statements

— cf. 14.5

| *Skip*: $G \vdash \text{Norm } s -\text{Skip} \rightarrow \text{Norm } s$

— cf. 14.7

| *Expr*: $\llbracket G \vdash \text{Norm } s0 -e \succ v \rightarrow s1 \rrbracket \implies$
 $G \vdash \text{Norm } s0 -\text{Expr } e \rightarrow s1$

| *Lab*: $\llbracket G \vdash \text{Norm } s0 -c \rightarrow s1 \rrbracket \implies$
 $G \vdash \text{Norm } s0 -l \cdot c \rightarrow \text{abupd } (\text{absorb } l) s1$

— cf. 14.2

| *Comp*: $\llbracket G \vdash \text{Norm } s0 -c1 \rightarrow s1;$
 $G \vdash s1 -c2 \rightarrow s2 \rrbracket \implies$
 $G \vdash \text{Norm } s0 -c1;; c2 \rightarrow s2$

— cf. 14.8.2

| *If*: $\llbracket G \vdash \text{Norm } s0 -e \succ b \rightarrow s1;$
 $G \vdash s1 -(\text{if the-Bool } b \text{ then } c1 \text{ else } c2) \rightarrow s2 \rrbracket \implies$
 $G \vdash \text{Norm } s0 -\text{If}(e) c1 \text{ Else } c2 \rightarrow s2$

— 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*: $\llbracket G \vdash \text{Norm } s0 -e \succ b \rightarrow s1;$
 $\text{if the-Bool } b$
 $\text{then } (G \vdash s1 -c \rightarrow s2 \wedge$
 $G \vdash (\text{abupd } (\text{absorb } (\text{Cont } l)) s2) -l \cdot \text{While}(e) c \rightarrow s3)$
 $\text{else } s3 = s1 \rrbracket \implies$
 $G \vdash \text{Norm } s0 -l \cdot \text{While}(e) c \rightarrow s3$

| *Jmp*: $G \vdash \text{Norm } s -\text{Jmp } j \rightarrow (\text{Some } (\text{Jump } j), s)$

— cf. 14.16

| *Throw*: $\llbracket G \vdash \text{Norm } s0 -e \succ a' \rightarrow s1 \rrbracket \implies$
 $G \vdash \text{Norm } s0 -\text{Throw } e \rightarrow \text{abupd } (\text{throw } a') s1$

— cf. 14.18.1

| *Try*: $\llbracket G \vdash \text{Norm } s0 -c1 \rightarrow s1; G \vdash s1 -\text{salloc} \rightarrow s2;$
 $\text{if } G, s2 \vdash \text{catch } C \text{ then } G \vdash \text{new-xcpt-var } vn s2 -c2 \rightarrow s3 \text{ else } s3 = s2 \rrbracket \implies$
 $G \vdash \text{Norm } s0 -\text{Try } c1 \text{ Catch}(C vn) c2 \rightarrow s3$

— cf. 14.18.2

| *Fin*: $\llbracket G \vdash \text{Norm } s0 -c1 \rightarrow (x1, s1);$
 $G \vdash \text{Norm } s1 -c2 \rightarrow s2;$
 $s3 = (\text{if } (\exists \text{err. } x1 = \text{Some } (\text{Error } \text{err}))$
 $\text{then } (x1, s1)$
 $\text{else } \text{abupd } (\text{abrupt-if } (x1 \neq \text{None}) x1) s2) \rrbracket$
 \implies
 $G \vdash \text{Norm } s0 -c1 \text{ Finally } c2 \rightarrow s3$

— cf. 12.4.2, 8.5

| *Init*: $\llbracket \text{the } (\text{class } G C) = c; \rrbracket$

$$G \vdash \quad s1 \text{ --}e\text{--}\gamma v \rightarrow s2 \rceil \implies \\ G \vdash \text{Norm } s0 \text{ --}va\text{:}:=e\text{--}\gamma v \rightarrow \text{assign } f \ v \ s2$$

— cf. 15.24

$$\mid \text{Cond: } \llbracket G \vdash \text{Norm } s0 \text{ --}e0\text{--}\gamma b \rightarrow s1; \\ G \vdash \quad s1 \text{ --(if the-Bool } b \text{ then } e1 \text{ else } e2)\text{--}\gamma v \rightarrow s2 \rceil \implies \\ G \vdash \text{Norm } s0 \text{ --}e0 \ ? \ e1 : e2\text{--}\gamma v \rightarrow s2$$

— The interplay of *Call*, *Method* 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 *Method* rule is evaluated on the calculated declaration class of the method invocation.

Method 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

$$\mid \text{Call:} \\ \llbracket G \vdash \text{Norm } s0 \text{ --}e\text{--}\gamma a' \rightarrow s1; G \vdash s1 \text{ --}args\text{--}\gamma vs \rightarrow s2; \\ D = \text{invocation-declclass } G \text{ mode } (store \ s2) \ a' \ \text{statT } (\llbracket name=mn, parTs=pTs \rceil); \\ s3 = \text{init-lvars } G \ D \ (\llbracket name=mn, parTs=pTs \rceil) \ \text{mode } a' \ vs \ s2; \\ s3' = \text{check-method-access } G \ \text{accC} \ \text{statT} \ \text{mode } (\llbracket name=mn, parTs=pTs \rceil) \ a' \ s3; \\ G \vdash s3' \text{ --Method } D \ (\llbracket name=mn, parTs=pTs \rceil) \text{--}\gamma v \rightarrow s4 \rceil \\ \implies \\ G \vdash \text{Norm } s0 \text{ --}\{accC, statT, mode\}e.mn(\{pTs\}args)\text{--}\gamma v \rightarrow (\text{restore-lvars } s2 \ s4)$$

— The accessibility check is after *init-lvars*, to keep it simple. *init-lvars* already tests for the absence of a null-pointer reference in case of an instance method invocation.

$$\mid \text{Method:} \quad \llbracket G \vdash \text{Norm } s0 \text{ --body } G \ D \ sig\text{--}\gamma v \rightarrow s1 \rceil \implies \\ G \vdash \text{Norm } s0 \text{ --Method } D \ sig\text{--}\gamma v \rightarrow s1$$

$$\mid \text{Body: } \llbracket G \vdash \text{Norm } s0 \text{ --Init } D \rightarrow s1; G \vdash s1 \text{ --}c \rightarrow s2; \\ s3 = (\text{if } (\exists l. \text{abrupt } s2 = \text{Some } (\text{Jump } (\text{Break } l))) \vee \\ \text{abrupt } s2 = \text{Some } (\text{Jump } (\text{Cont } l))) \\ \text{then } \text{abupd } (\lambda x. \text{Some } (\text{Error CrossMethodJump})) \ s2 \\ \text{else } s2 \rceil \implies \\ G \vdash \text{Norm } s0 \text{ --Body } D \ c\text{--}\gamma \text{the } (locals \ (store \ s2) \ \text{Result}) \\ \rightarrow \text{abupd } (\text{absorb } Ret) \ s3$$

— cf. 14.15, 12.4.1

— We filter out a break/continue in *s2*, so that we can proof definite assignment correct, without the need of conformance of the state. By this the different parts of the typesafety proof can be disentangled a little.

— evaluation of variables

— cf. 15.13.1, 15.7.2

$$\mid \text{LVar: } G \vdash \text{Norm } s \text{ --LVar } vn\text{--}\gamma lvar \ vn \ s \rightarrow \text{Norm } s$$

— cf. 15.10.1, 12.4.1

$$\mid \text{FVar: } \llbracket G \vdash \text{Norm } s0 \text{ --Init } statDeclC \rightarrow s1; G \vdash s1 \text{ --}e\text{--}\gamma a \rightarrow s2; \\ (v, s2') = \text{fvar } statDeclC \ \text{stat } fn \ a \ s2; \\ s3 = \text{check-field-access } G \ \text{accC} \ \text{statDeclC} \ fn \ \text{stat } a \ s2' \rceil \implies \\ G \vdash \text{Norm } s0 \text{ --}\{accC, statDeclC, stat\}e..fn\text{--}\gamma v \rightarrow s3$$

— The accessibility check is after *fvar*, to keep it simple. *fvar* already tests for the absence of a null-pointer reference in case of an instance field

$G\vdash\text{Norm } s -\text{In1r } (\text{Jmp } j)$	$\succrightarrow (x, s')$
$G\vdash\text{Norm } s -\text{In1r } (l \cdot c)$	$\succrightarrow (x, s')$
$G\vdash\text{Norm } s -\text{In3 } (\square)$	$\succrightarrow (v, s')$
$G\vdash\text{Norm } s -\text{In3 } (e\#es)$	$\succrightarrow (v, s')$
$G\vdash\text{Norm } s -\text{In1l } (\text{Lit } w)$	$\succrightarrow (v, s')$
$G\vdash\text{Norm } s -\text{In1l } (\text{UnOp } \text{unop } e)$	$\succrightarrow (v, s')$
$G\vdash\text{Norm } s -\text{In1l } (\text{BinOp } \text{binop } e1 e2)$	$\succrightarrow (v, s')$
$G\vdash\text{Norm } s -\text{In2 } (\text{LVar } vn)$	$\succrightarrow (v, s')$
$G\vdash\text{Norm } s -\text{In1l } (\text{Cast } T e)$	$\succrightarrow (v, s')$
$G\vdash\text{Norm } s -\text{In1l } (e \text{ InstOf } T)$	$\succrightarrow (v, s')$
$G\vdash\text{Norm } s -\text{In1l } (\text{Super})$	$\succrightarrow (v, s')$
$G\vdash\text{Norm } s -\text{In1l } (\text{Acc } va)$	$\succrightarrow (v, s')$
$G\vdash\text{Norm } s -\text{In1r } (\text{Expr } e)$	$\succrightarrow (x, s')$
$G\vdash\text{Norm } s -\text{In1r } (c1;; c2)$	$\succrightarrow (x, s')$
$G\vdash\text{Norm } s -\text{In1l } (\text{Methd } C \text{ sig})$	$\succrightarrow (x, s')$
$G\vdash\text{Norm } s -\text{In1l } (\text{Body } D c)$	$\succrightarrow (x, s')$
$G\vdash\text{Norm } s -\text{In1l } (e0 ? e1 : e2)$	$\succrightarrow (v, s')$
$G\vdash\text{Norm } s -\text{In1r } (\text{If}(e) c1 \text{ Else } c2)$	$\succrightarrow (x, s')$
$G\vdash\text{Norm } s -\text{In1r } (l \cdot \text{While}(e) c)$	$\succrightarrow (x, s')$
$G\vdash\text{Norm } s -\text{In1r } (c1 \text{ Finally } c2)$	$\succrightarrow (x, s')$
$G\vdash\text{Norm } s -\text{In1r } (\text{Throw } e)$	$\succrightarrow (x, s')$
$G\vdash\text{Norm } s -\text{In1l } (\text{NewC } C)$	$\succrightarrow (v, s')$
$G\vdash\text{Norm } s -\text{In1l } (\text{New } T[e])$	$\succrightarrow (v, s')$
$G\vdash\text{Norm } s -\text{In1l } (\text{Ass } va e)$	$\succrightarrow (v, s')$
$G\vdash\text{Norm } s -\text{In1r } (\text{Try } c1 \text{ Catch}(tn vn) c2)$	$\succrightarrow (x, s')$
$G\vdash\text{Norm } s -\text{In2 } (\{\text{accC}, \text{statDeclC}, \text{stat}\}e..fn)$	$\succrightarrow (v, s')$
$G\vdash\text{Norm } s -\text{In2 } (e1.[e2])$	$\succrightarrow (v, s')$
$G\vdash\text{Norm } s -\text{In1l } (\{\text{accC}, \text{statT}, \text{mode}\}e.mn(\{pT\}p))$	$\succrightarrow (v, s')$
$G\vdash\text{Norm } s -\text{In1r } (\text{Init } C)$	$\succrightarrow (x, s')$

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*:

$G\vdash s -t \succrightarrow (w, s')$
 $\implies \text{case } t \text{ of}$
 $\text{In1 } ec \Rightarrow (\text{case } ec \text{ of}$
 $\text{Inl } e \Rightarrow (\exists v. w = \text{In1 } v)$
 $\mid \text{Inr } c \Rightarrow w = \diamond)$
 $\mid \text{In2 } e \Rightarrow (\exists v. w = \text{In2 } v)$
 $\mid \text{In3 } e \Rightarrow (\exists v. w = \text{In3 } v)$
 $\langle \text{proof} \rangle$

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*).

lemma *eval-expr-eq*: $G\vdash s -\text{In1l } t \succrightarrow (w, s') = (\exists v. w = \text{In1 } v \wedge G\vdash s -t \succrightarrow v \rightarrow s')$
 $\langle \text{proof} \rangle$

lemma *eval-var-eq*: $G\vdash s -\text{In2 } t \succrightarrow (w, s') = (\exists vf. w = \text{In2 } vf \wedge G\vdash s -t = \succrightarrow vf \rightarrow s')$
 $\langle \text{proof} \rangle$

lemma *eval-exprs-eq*: $G\vdash s -\text{In3 } t \succrightarrow (w, s') = (\exists vs. w = \text{In3 } vs \wedge G\vdash s -t = \succrightarrow vs \rightarrow s')$

<proof>

lemma *eval-stmt-eq*: $G \vdash s - \text{In1r } t \succ \rightarrow (w, s') = (w = \diamond \wedge G \vdash s - t \rightarrow s')$
<proof>

<ML>

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 their is no valid evaluation of these terms

lemma *eval-Callee*: $G \vdash \text{Norm } s - \text{Callee } l \ e - \succ v \rightarrow s' = \text{False}$
<proof>

lemma *eval-InsInitE*: $G \vdash \text{Norm } s - \text{InsInitE } c \ e - \succ v \rightarrow s' = \text{False}$
<proof>

lemma *eval-InsInitV*: $G \vdash \text{Norm } s - \text{InsInitV } c \ w - \succ v \rightarrow s' = \text{False}$
<proof>

lemma *eval-FinA*: $G \vdash \text{Norm } s - \text{FinA } a \ c \rightarrow s' = \text{False}$
<proof>

lemma *eval-no-abrupt-lemma*:
 $\bigwedge s \ s'. G \vdash s - t \succ \rightarrow (w, s') \implies \text{normal } s' \longrightarrow \text{normal } s$
<proof>

lemma *eval-no-abrupt*:
 $G \vdash (x, s) - t \succ \rightarrow (w, \text{Norm } s') =$
 $(x = \text{None} \wedge G \vdash \text{Norm } s - t \succ \rightarrow (w, \text{Norm } s'))$
<proof>

<ML>

lemma *eval-abrupt-lemma*:
 $G \vdash s - t \succ \rightarrow (v, s') \implies \text{abrupt } s = \text{Some } xc \longrightarrow s' = s \wedge v = \text{arbitrary3 } t$
<proof>

lemma *eval-abrupt*:
 $G \vdash (\text{Some } xc, s) - t \succ \rightarrow (w, s') =$
 $(s' = (\text{Some } xc, s) \wedge w = \text{arbitrary3 } t \wedge$
 $G \vdash (\text{Some } xc, s) - t \succ \rightarrow (\text{arbitrary3 } t, (\text{Some } xc, s)))$
<proof>

<ML>

lemma *LitI*: $G \vdash s - \text{Lit } v - \succ (\text{if normal } s \text{ then } v \text{ else arbitrary}) \rightarrow s$
<proof>

lemma *SkipI* [*intro!*]: $G \vdash s \text{ --Skip} \rightarrow s$
 ⟨*proof*⟩

lemma *ExprI*: $G \vdash s \text{ --}e \text{ --}\triangleright v \rightarrow s' \implies G \vdash s \text{ --Expr } e \rightarrow s'$
 ⟨*proof*⟩

lemma *CompI*: $\llbracket G \vdash s \text{ --}c1 \rightarrow s1; G \vdash s1 \text{ --}c2 \rightarrow s2 \rrbracket \implies G \vdash s \text{ --}c1;; c2 \rightarrow s2$
 ⟨*proof*⟩

lemma *CondI*:
 $\bigwedge s1. \llbracket G \vdash s \text{ --}e \text{ --}\triangleright b \rightarrow s1; G \vdash s1 \text{ --(if the-Bool } b \text{ then } e1 \text{ else } e2) \text{ --}\triangleright v \rightarrow s2 \rrbracket \implies$
 $G \vdash s \text{ --}e \text{ ? } e1 : e2 \text{ --}\triangleright \text{(if normal } s1 \text{ then } v \text{ else arbitrary)} \rightarrow s2$
 ⟨*proof*⟩

lemma *IfI*: $\llbracket G \vdash s \text{ --}e \text{ --}\triangleright v \rightarrow s1; G \vdash s1 \text{ --(if the-Bool } v \text{ then } c1 \text{ else } c2) \rightarrow s2 \rrbracket$
 $\implies G \vdash s \text{ --If}(e) \ c1 \ \text{Else } c2 \rightarrow s2$
 ⟨*proof*⟩

lemma *MethdI*: $G \vdash s \text{ --body } G \ C \ \text{sig} \text{ --}\triangleright v \rightarrow s'$
 $\implies G \vdash s \text{ --Methd } C \ \text{sig} \text{ --}\triangleright v \rightarrow s'$
 ⟨*proof*⟩

lemma *eval-Call*:
 $\llbracket G \vdash \text{Norm } s0 \text{ --}e \text{ --}\triangleright a' \rightarrow s1; G \vdash s1 \text{ --ps} \dot{=} \triangleright pvs \rightarrow s2;$
 $D = \text{invocation-declclass } G \ \text{mode} \ (\text{store } s2) \ a' \ \text{statT} \ (\{\text{name=mn,parTs=pTs}\});$
 $s3 = \text{init-lvars } G \ D \ (\{\text{name=mn,parTs=pTs}\}) \ \text{mode} \ a' \ pvs \ s2;$
 $s3' = \text{check-method-access } G \ \text{accC} \ \text{statT} \ \text{mode} \ (\{\text{name=mn,parTs=pTs}\}) \ a' \ s3;$
 $G \vdash s3' \text{ --Methd } D \ (\{\text{name=mn,parTs=pTs}\}) \text{ --}\triangleright v \rightarrow s4;$
 $s4' = \text{restore-lvars } s2 \ s4 \rrbracket \implies$
 $G \vdash \text{Norm } s0 \text{ --}\{\text{accC,statT,mode}\}e \cdot \text{mn}(\{\text{pTs}\}ps) \text{ --}\triangleright v \rightarrow s4'$
 ⟨*proof*⟩

lemma *eval-Init*:
 $\llbracket \text{if initd } C \ (\text{globs } s0) \ \text{then } s3 = \text{Norm } s0$
 $\ \text{else } G \vdash \text{Norm} \ (\text{init-class-obj } G \ C \ s0)$
 $\ \text{--(if } C = \text{Object then Skip else Init (super (the (class } G \ C)))} \rrbracket \rightarrow s1 \wedge$
 $G \vdash \text{set-lvars empty } s1 \text{ --(init (the (class } G \ C)))} \rightarrow s2 \wedge$
 $s3 = \text{restore-lvars } s1 \ s2 \rrbracket \implies$
 $G \vdash \text{Norm } s0 \text{ --Init } C \rightarrow s3$
 ⟨*proof*⟩

lemma *init-done*: $\text{initd } C \ s \implies G \vdash s \text{ --Init } C \rightarrow s$
 ⟨*proof*⟩

lemma *eval-StatRef*:
 $G \vdash s \text{ --StatRef } rt \text{ --}\triangleright \text{(if abrupt } s = \text{None then Null else arbitrary)} \rightarrow s$
 ⟨*proof*⟩

lemma *SkipD* [*dest!*]: $G \vdash s \text{ --Skip} \rightarrow s' \implies s' = s$
 ⟨*proof*⟩

lemma *Skip-eq* [*simp*]: $G \vdash s \text{ --Skip} \rightarrow s' = (s = s')$
 ⟨*proof*⟩

lemma *init-retains-locals* [*rule-format (no-asm)*]: $G \vdash s \text{ --}t \rightarrow (w, s') \implies$
 $(\forall C. t = \text{In1r} (\text{Init } C) \longrightarrow \text{locals} (\text{store } s) = \text{locals} (\text{store } s'))$
 ⟨*proof*⟩

lemma *halloc-xcpt* [*dest!*]:
 $\bigwedge s'. G \vdash (\text{Some } xc, s) \text{ --halloc } oi \rightarrow a \rightarrow s' \implies s' = (\text{Some } xc, s)$
 ⟨*proof*⟩

lemma *eval-Method*:
 $G \vdash s \text{ --In1l}(\text{body } G \ C \ \text{sig}) \rightarrow (w, s')$
 $\implies G \vdash s \text{ --In1l}(\text{Method } C \ \text{sig}) \rightarrow (w, s')$
 ⟨*proof*⟩

lemma *eval-Body*: $\llbracket G \vdash \text{Norm } s0 \text{ --Init } D \rightarrow s1; G \vdash s1 \text{ --}c \rightarrow s2;$
 $\text{res} = \text{the} (\text{locals} (\text{store } s2) \ \text{Result});$
 $s3 = (\text{if } (\exists l. \text{abrupt } s2 = \text{Some} (\text{Jump} (\text{Break } l))) \vee$
 $\text{abrupt } s2 = \text{Some} (\text{Jump} (\text{Cont } l)))$
 $\text{then } \text{abupd} (\lambda x. \text{Some} (\text{Error } \text{CrossMethodJump})) s2$
 $\text{else } s2);$
 $s4 = \text{abupd} (\text{absorb } \text{Ret}) s3 \rrbracket \implies$
 $G \vdash \text{Norm } s0 \text{ --Body } D \ c \rightarrow \text{res} \rightarrow s4$
 ⟨*proof*⟩

lemma *eval-binop-arg2-indep*:
 $\neg \text{need-second-arg } \text{binop } v1 \implies \text{eval-binop } \text{binop } v1 \ x = \text{eval-binop } \text{binop } v1 \ y$
 ⟨*proof*⟩

lemma *eval-BinOp-arg2-indepI*:
assumes *eval-e1*: $G \vdash \text{Norm } s0 \text{ --}e1 \rightarrow v1 \rightarrow s1$ **and**
no-need: $\neg \text{need-second-arg } \text{binop } v1$
shows $G \vdash \text{Norm } s0 \text{ --BinOp } \text{binop } e1 \ e2 \rightarrow (\text{eval-binop } \text{binop } v1 \ v2) \rightarrow s1$
 (**is** *?EvalBinOp* *v2*)
 ⟨*proof*⟩

single valued

lemma *unique-halloc* [*rule-format (no-asm)*]:
 $G \vdash s \text{ --halloc } oi \rightarrow a \rightarrow s' \implies G \vdash s \text{ --halloc } oi \rightarrow a' \rightarrow s'' \longrightarrow a' = a \wedge s'' = s'$
 ⟨*proof*⟩

lemma *single-valued-halloc*:

single-valued $\{(s, oi), (a, s')\}. G \vdash s \text{ -halloc } oi \succ a \rightarrow s'\}$
 ⟨proof⟩

lemma *unique-sxalloc* [rule-format (no-asm)]:

$G \vdash s \text{ -sxalloc} \rightarrow s' \implies G \vdash s \text{ -sxalloc} \rightarrow s'' \longrightarrow s'' = s'$
 ⟨proof⟩

lemma *single-valued-sxalloc*: *single-valued* $\{(s, s')\}. G \vdash s \text{ -sxalloc} \rightarrow s'\}$

⟨proof⟩

lemma *split-pairD*: $(x, y) = p \implies x = \text{fst } p \ \& \ y = \text{snd } p$

⟨proof⟩

lemma *unique-eval* [rule-format (no-asm)]:

$G \vdash s \text{ -t} \rightarrow (w, s') \implies (\forall w' s''. G \vdash s \text{ -t} \rightarrow (w', s'') \longrightarrow w' = w \wedge s'' = s')$
 ⟨proof⟩

lemma *single-valued-eval*:

single-valued $\{(s, t), (v, s')\}. G \vdash s \text{ -t} \rightarrow (v, s')\}$
 ⟨proof⟩

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

$inj-tnam' [simp]: (tnam' x = tnam' y) = (x = y)$

$inj-vnam' [simp]: (vnam' x = vnam' y) = (x = y)$

$inj-label' [simp]: (label' x = label' y) = (x = y)$

$surj-tnam': \exists m. n = tnam' m$

$surj-vnam': \exists m. n = vnam' m$

$surj-label': \exists m. n = label' m$

abbreviation

$HasFoo :: qname$ **where**

$HasFoo == (\text{pid}=\text{java-lang}, \text{tid}=\text{TName } (tnam' HasFoo'))$

abbreviation

$Base :: qname$ **where**

$Base == (\text{pid}=\text{java-lang}, \text{tid}=\text{TName } (tnam' Base'))$

abbreviation

$Ext :: qname$ **where**

$Ext == (\text{pid}=\text{java-lang}, \text{tid}=\text{TName } (tnam' Ext'))$

abbreviation

$Main :: qname$ **where**

$Main == (\text{pid}=\text{java-lang}, \text{tid}=\text{TName } (tnam' Main'))$

abbreviation

$arr :: vname$ **where**

$arr == (vnam' arr')$

abbreviation

$vee :: vname$ **where**

$vee == (vnam' vee')$

abbreviation

$z :: vname$ **where**

$z == (vnam' z')$

abbreviation

$e :: vname$ **where**

$e == (vnam' e')$

abbreviation

$lab1 :: label$ **where**

$lab1 == label' lab1'$

constdefs

arr-viewed-from :: *qname* \Rightarrow *qname* \Rightarrow *var*
arr-viewed-from *accC* *C* \equiv {*accC*,*Base*,*True*}*StatRef* (*ClassT* *C*)..*arr*

BaseCl :: *class*
BaseCl \equiv (*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 \equiv (*access=Public*,
cfields=[(*vee*, (*access=Public*,*static=False*,*type= PrimT Integer*)]),
methods=[*Ext-foo*],
init=Skip,
super=Base,
superIfs=[])

MainCl :: *class*
MainCl \equiv (*access=Public*,
cfields=[],
methods=[],
init=Skip,
super=Object,
superIfs=[])

constdefs

HasFooInt :: *iface*
HasFooInt \equiv (*access=Public*,*imethods*=[(*foo-sig*, *foo-mhead*)],*isuperIfs*=[])

Ifaces :: *idecl list*
Ifaces \equiv [(*HasFoo*,*HasFooInt*)]

Classes :: *cdecl list*
Classes \equiv [(*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* \rightarrow *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 (\backslash 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
 ⟨proof⟩

lemma *table-classes-Ext* [simp]: *table-of Classes Ext* = Some ExtCl
 ⟨proof⟩

lemma *table-classes-Main* [simp]: *table-of Classes Main* = Some MainCl
 ⟨proof⟩

program

abbreviation

tprg :: prog where
tprg == (\backslash 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) $^+$ \Longrightarrow R
 ⟨proof⟩

lemma *not-Throwable-subcls-SXcpt* [elim!]:
 (SXcpt Throwable, SXcpt xn) \in (subcls1 tprg) $^+$ \Longrightarrow R
 ⟨proof⟩

lemma *not-SXcpt-n-subcls-SXcpt-n* [elim!]:
 (SXcpt xn, SXcpt xn) \in (subcls1 tprg) $^+$ \Longrightarrow R
 ⟨proof⟩

lemma *not-Base-subcls-Ext* [elim!]: (Base, Ext) \in (subcls1 tprg) $^+$ \Longrightarrow R
 ⟨proof⟩

lemma *not-TName-n-subcls-TName-n* [rule-format (no-asm), elim!]:
 (\backslash pid=java-lang,tid=TName tn), (\backslash pid=java-lang,tid=TName tn)

$\in (\text{subcls1 } \text{tprg})^+ \longrightarrow R$
 ⟨proof⟩

lemma *ws-idecl-HasFoo*: *ws-idecl tprg HasFoo* []
 ⟨proof⟩

lemma *ws-cdecl-Object*: *ws-cdecl tprg Object any*
 ⟨proof⟩

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*:
 $G = \text{tprg} \implies (\forall (I, i) \in \text{set } \text{Ifaces}. \text{ws-idecl } G \ I \ (\text{isuperIfs } i))$
 ⟨proof⟩

lemma *ws-cdecl-all*: $G = \text{tprg} \implies (\forall (C, c) \in \text{set } \text{Classes}. \text{ws-cdecl } G \ C \ (\text{super } c))$
 ⟨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]: *tprg ⊢ SXcpt xn ≤_C SXcpt Throwable*
 ⟨proof⟩

lemma *Ext-subclseq-Base* [simp]: *tprg ⊢ Ext ≤_C Base*
 ⟨proof⟩

lemma *Ext-subcls-Base* [simp]: *tprg ⊢ Ext <_C Base*
 ⟨proof⟩

fields and method lookup

lemma *fields-tprg-Object* [simp]: *DeclConcepts.fields tprg Object = []*
 ⟨proof⟩

lemma *fields-tprg-Throwable* [simp]:
DeclConcepts.fields tprg (SXcpt Throwable) = []
 ⟨proof⟩

lemma *fields-tprg-SXcpt* [simp]: *DeclConcepts.fields tprg (SXcpt xn) = []*
 ⟨proof⟩

lemmas *fields-rec' = fields-rec* [OF - ws-tprg]

lemma *fields-Base* [simp]:
DeclConcepts.fields tprg Base
 = [((arr, Base), (access=Public, static=True, type=PrimT Boolean.[])),
 ((vee, Base), (access=Public, static=False, type=Iface HasFoo []))]
 ⟨proof⟩

lemma *fields-Ext* [simp]:
DeclConcepts.fields tprg Ext
 = [((vee, Ext), (access=Public, static=False, type= PrimT Integer))]
 @ *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]:
imethds tprg HasFoo = o2s ∘ empty(foo-sig ⇒ (HasFoo, foo-mhead))

$\langle \text{proof} \rangle$

lemma *methd-tprg-Object* [simp]: *methd tprg Object = empty*
 $\langle \text{proof} \rangle$

lemma *methd-Base* [simp]:
methd tprg Base = table-of $[(\lambda(s,m). (s, \text{Base}, m)) \text{Base-foo}]$
 $\langle \text{proof} \rangle$

lemma *memberid-Base-foo-simp* [simp]:
memberid (mdecl Base-foo) = mid foo-sig
 $\langle \text{proof} \rangle$

lemma *memberid-Ext-foo-simp* [simp]:
memberid (mdecl Ext-foo) = mid foo-sig
 $\langle \text{proof} \rangle$

lemma *Base-declares-foo*:
tprg \vdash mdecl Base-foo declared-in Base
 $\langle \text{proof} \rangle$

lemma *foo-sig-not-undeclared-in-Base*:
 $\neg \text{tprg} \vdash \text{mid } \text{foo-sig } \text{undeclared-in } \text{Base}$
 $\langle \text{proof} \rangle$

lemma *Ext-declares-foo*:
tprg \vdash mdecl Ext-foo declared-in Ext
 $\langle \text{proof} \rangle$

lemma *foo-sig-not-undeclared-in-Ext*:
 $\neg \text{tprg} \vdash \text{mid } \text{foo-sig } \text{undeclared-in } \text{Ext}$
 $\langle \text{proof} \rangle$

lemma *Base-foo-not-inherited-in-Ext*:
 $\neg \text{tprg} \vdash \text{Ext inherits } (\text{Base}, \text{mdecl } \text{Base-foo})$
 $\langle \text{proof} \rangle$

lemma *Ext-method-inheritance*:
filter-tab $(\lambda \text{sig } m. \text{tprg} \vdash \text{Ext inherits method sig } m)$
 $(\text{empty}(\text{fst } ((\lambda(s, m). (s, \text{Base}, m)) \text{Base-foo}) \mapsto$
 $\text{snd } ((\lambda(s, m). (s, \text{Base}, m)) \text{Base-foo})))$
 $= \text{empty}$
 $\langle \text{proof} \rangle$

lemma *methd-Ext* [simp]: *methd tprg Ext =*
table-of $[(\lambda(s,m). (s, \text{Ext}, m)) \text{Ext-foo}]$
 $\langle \text{proof} \rangle$

accessibility**lemma** *classesDefined*:
$$\llbracket \text{class tprg } C = \text{Some } c; C \neq \text{Object} \rrbracket \implies \exists \text{ sc. class tprg (super } c) = \text{Some } sc$$

<proof>

lemma *superclassesBase* [simp]: *superclasses tprg Base*={ *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*: $\text{snd } ((\lambda(s, m). (s, a, m)) x) = (a, \text{snd } x)$
 ⟨proof⟩

lemma *fst-special-simp*: $\text{fst } ((\lambda(s, m). (s, a, m)) x) = \text{fst } x$
 ⟨proof⟩

lemma *foo-sig-undeclared-in-Object*:
 $\text{tpg} \vdash \text{mid } \text{foo-sig } \text{undeclared-in } \text{Object}$
 ⟨proof⟩

lemma *unique-sig-Base-foo*:
 $\text{tpg} \vdash \text{mdecl } (\text{sig}, \text{snd } \text{Base-foo}) \text{ declared-in } \text{Base} \implies \text{sig} = \text{foo-sig}$
 ⟨proof⟩

lemma *Base-foo-no-override*:
 $\text{tpg}, \text{sig} \vdash (\text{Base}, (\text{snd } \text{Base-foo})) \text{ overrides } \text{old} \implies P$
 ⟨proof⟩

lemma *Base-foo-no-stat-override*:
 $\text{tpg}, \text{sig} \vdash (\text{Base}, (\text{snd } \text{Base-foo})) \text{ overrides}_S \text{old} \implies P$
 ⟨proof⟩

lemma *Base-foo-no-hide*:
 $\text{tpg}, \text{sig} \vdash (\text{Base}, (\text{snd } \text{Base-foo})) \text{ hides } \text{old} \implies P$
 ⟨proof⟩

lemma *Ext-foo-no-hide*:
 $\text{tpg}, \text{sig} \vdash (\text{Ext}, (\text{snd } \text{Ext-foo})) \text{ hides } \text{old} \implies P$
 ⟨proof⟩

lemma *unique-sig-Ext-foo*:
 $\text{tpg} \vdash \text{mdecl } (\text{sig}, \text{snd } \text{Ext-foo}) \text{ declared-in } \text{Ext} \implies \text{sig} = \text{foo-sig}$
 ⟨proof⟩

lemma *Ext-foo-override*:
 $\text{tpg}, \text{sig} \vdash (\text{Ext}, (\text{snd } \text{Ext-foo})) \text{ overrides } \text{old}$
 $\implies \text{old} = (\text{Base}, (\text{snd } \text{Base-foo}))$
 ⟨proof⟩

lemma *Ext-foo-stat-override*:
 $\text{tpg}, \text{sig} \vdash (\text{Ext}, (\text{snd } \text{Ext-foo})) \text{ overrides}_S \text{old}$
 $\implies \text{old} = (\text{Base}, (\text{snd } \text{Base-foo}))$
 ⟨proof⟩

lemma *Base-foo-member-of-Base*:
 $\text{tpg} \vdash (\text{Base}, \text{mdecl } \text{Base-foo}) \text{ member-of } \text{Base}$
 ⟨proof⟩

lemma *Base-foo-member-in-Base*:

$tprg \vdash (Base, mdecl\ Base-foo)\ member-in\ Base$
 $\langle proof \rangle$

lemma *Ext-foo-member-of-Ext*:

$tprg \vdash (Ext, mdecl\ Ext-foo)\ member-of\ Ext$
 $\langle proof \rangle$

lemma *Ext-foo-member-in-Ext*:

$tprg \vdash (Ext, mdecl\ Ext-foo)\ member-in\ Ext$
 $\langle proof \rangle$

lemma *Base-foo-permits-acc*:

$tprg \vdash (Base, mdecl\ Base-foo)\ in\ Base\ permits-acc-from\ S$
 $\langle proof \rangle$

lemma *Base-foo-accessible* [simp]:

$tprg \vdash (Base, mdecl\ Base-foo)\ of\ Base\ accessible-from\ S$
 $\langle proof \rangle$

lemma *Base-foo-dyn-accessible* [simp]:

$tprg \vdash (Base, mdecl\ Base-foo)\ in\ Base\ dyn-accessible-from\ S$
 $\langle proof \rangle$

lemma *accmethd-Base* [simp]:

$accmethd\ tprg\ S\ Base = methd\ tprg\ Base$
 $\langle proof \rangle$

lemma *Ext-foo-permits-acc*:

$tprg \vdash (Ext, mdecl\ Ext-foo)\ in\ Ext\ permits-acc-from\ S$
 $\langle proof \rangle$

lemma *Ext-foo-accessible* [simp]:

$tprg \vdash (Ext, mdecl\ Ext-foo)\ of\ Ext\ accessible-from\ S$
 $\langle proof \rangle$

lemma *Ext-foo-dyn-accessible* [simp]:

$tprg \vdash (Ext, mdecl\ Ext-foo)\ in\ Ext\ dyn-accessible-from\ S$
 $\langle proof \rangle$

lemma *Ext-foo-overrides-Base-foo*:

$tprg \vdash (Ext, Ext-foo)\ overrides\ (Base, Base-foo)$
 $\langle proof \rangle$

lemma *accmethd-Ext* [simp]:

$accmethd\ tprg\ S\ Ext = methd\ tprg\ Ext$

⟨proof⟩

lemma *cls-Ext*: class tprg Ext = Some ExtCl

⟨proof⟩

lemma *dynmethd-Ext-foo*:

dynmethd tprg Base Ext (⟦name = foo, parTs = [Class Base]⟧)
= Some (Ext,snd Ext-foo)

⟨proof⟩

lemma *Base-fields-accessible[simp]*:

accfield tprg S Base
= table-of((map (λ((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 (λ((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*, (λ *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 ⇒ 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=tprg ⇒ 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*: *(prg=tprg,cls=Main,lcl=empty(VName e→Class Base)) ⊢ test ?pTs::√*
 ⟨proof⟩

definite assignment

lemma *da-test*: *(prg=tprg,cls=Main,lcl=empty(VName e→Class Base))*
⊢ { } » (test ?pTs) » (nrm={ VName e },brk=λ l. UNIV)
 ⟨proof⟩

execution

lemma *alloc-one*: *∧ a obj. [the (new-Addr h) = a; atleast-free h (Suc n)] ⇒*
new-Addr h = Some a ∧ atleast-free (h(a→obj)) 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*

abbreviation *one* == *Suc 0*
abbreviation *two* == *Suc one*
abbreviation *tree* == *Suc two*
abbreviation *four* == *Suc tree*

syntax

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

obj-a <= ($\text{tag} = \text{Arr } (\text{PrimT Boolean}) (\text{CONST two})$
 $\text{,values} = \text{CONST empty}(\text{Inr } 0 \mapsto \text{Bool False})(\text{Inr } (\text{CONST one}) \mapsto \text{Bool False})$)
obj-b <= ($\text{tag} = \text{CInst } (\text{CONST Ext})$
 $\text{,values} = (\text{CONST empty}(\text{Inl } (\text{CONST vee}, \text{CONST Base}) \mapsto \text{Null } \quad)$
 $(\text{Inl } (\text{CONST vee}, \text{CONST Ext}) \mapsto \text{Intg } 0))$)
obj-c == ($\text{tag} = \text{CInst } (\text{SXcpt NullPointer})$, $\text{values} = \text{CONST empty}$)
arr-N == $\text{CONST empty}(\text{Inl } (\text{CONST arr}, \text{CONST Base}) \mapsto \text{Null})$
arr-a == $\text{CONST empty}(\text{Inl } (\text{CONST arr}, \text{CONST Base}) \mapsto \text{Addr } a)$
globs1 == $\text{CONST empty}(\text{Inr } (\text{CONST Ext}) \mapsto (\text{tag} = \text{arbitrary}, \text{values} = \text{CONST empty}))$
 $(\text{Inr } (\text{CONST Base}) \mapsto (\text{tag} = \text{arbitrary}, \text{values} = \text{arr-N}))$
 $(\text{Inr } \text{Object} \mapsto (\text{tag} = \text{arbitrary}, \text{values} = \text{CONST empty}))$
globs2 == $\text{CONST empty}(\text{Inr } (\text{CONST Ext}) \mapsto (\text{tag} = \text{arbitrary}, \text{values} = \text{CONST empty}))$
 $(\text{Inr } \text{Object} \mapsto (\text{tag} = \text{arbitrary}, \text{values} = \text{CONST empty}))$
 $(\text{Inl } a \mapsto \text{obj-a})$
 $(\text{Inr } (\text{CONST Base}) \mapsto (\text{tag} = \text{arbitrary}, \text{values} = \text{arr-a}))$
globs3 == *globs2*(*Inl b* \mapsto *obj-b*)
globs8 == *globs3*(*Inl c* \mapsto *obj-c*)
locs3 == $\text{CONST empty}(\text{VName } (\text{CONST } e) \mapsto \text{Addr } b)$
locs4 == $\text{CONST empty}(\text{VName } (\text{CONST } z) \mapsto \text{Null})(\text{Inr } () \mapsto \text{Addr } b)$
locs8 == *locs3*(*VName* (*CONST z*) \mapsto *Addr c*)

```

s0 ==      st (CONST empty) (CONST empty)
s0' == Norm s0
s1 ==      st globs1 (CONST empty)
s1' == Norm s1
s2 ==      st globs2 (CONST 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)

```

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

$$\begin{aligned} gext \quad &:: st \Rightarrow st \Rightarrow bool && (-\leq|- \quad [71,71] \quad 70) \\ s\leq|s' \equiv &\forall r. \forall obj \in globs \ s \ r: \exists obj' \in globs \ s' \ r: tag \ obj' = tag \ obj \end{aligned}$$

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*:

$$\begin{aligned} &init-class-obj \ G \ C \ s1 \leq|s2 \implies inited \ C \ (globs \ s2) \\ &\langle proof \rangle \end{aligned}$$

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*]: $set_locals\ l\ s1 \leq |s2 = s1 \leq |s2$
 ⟨*proof*⟩

lemma *gext-cong2* [*simp*]: $s1 \leq |set_locals\ l\ s2 = s1 \leq |s2$
 ⟨*proof*⟩

lemma *gext-lupd1* [*simp*]: $lupd(vn \mapsto v) s1 \leq |s2 = s1 \leq |s2$
 ⟨*proof*⟩

lemma *gext-lupd2* [*simp*]: $s1 \leq |lupd(vn \mapsto v) s2 = s1 \leq |s2$
 ⟨*proof*⟩

lemma *inited-gext*: $\llbracket inited\ C\ (globs\ s);\ s \leq |s \rrbracket \implies inited\ C\ (globs\ s')$
 ⟨*proof*⟩

value conformance

constdefs

conf :: *prog* \Rightarrow *st* \Rightarrow *val* \Rightarrow *ty* \Rightarrow *bool* ($-, \vdash :: \preceq -$ [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*]: $G, set_locals\ l\ s \vdash v :: \preceq T = G, s \vdash v :: \preceq T$
 ⟨*proof*⟩

lemma *conf-lupd* [*simp*]: $G, lupd(vn \mapsto va) s \vdash v :: \preceq T = G, s \vdash v :: \preceq T$
 ⟨*proof*⟩

lemma *conf-PrimT* [*simp*]: $\forall dt. typeof\ dt\ v = Some\ (PrimT\ t) \implies G, s \vdash v :: \preceq PrimT\ t$
 ⟨*proof*⟩

lemma *conf-Boolean*: $G, s \vdash v :: \preceq PrimT\ Boolean \implies \exists b. v = Bool\ b$
 ⟨*proof*⟩

lemma *conf-litval* [*rule-format* (*no-asm*)]:
 $typeof\ (\lambda a. None)\ v = Some\ T \longrightarrow G, s \vdash v :: \preceq T$
 ⟨*proof*⟩

lemma *conf-Null* [*simp*]: $G, s \vdash Null :: \preceq T = G \vdash NT \preceq T$
 ⟨*proof*⟩

lemma *conf-Addr*:
 $G, s \vdash Addr\ a :: \preceq T = (\exists obj. heap\ s\ a = Some\ obj \wedge G \vdash obj_ty\ obj \preceq T)$
 ⟨*proof*⟩

lemma *conf-AddrI*: $\llbracket \text{heap } s \ a = \text{Some } \text{obj}; G \vdash \text{obj-ty } \text{obj} \preceq T \rrbracket \implies G, s \vdash \text{Addr } a :: \preceq T$
 ⟨proof⟩

lemma *defval-conf* [*rule-format (no-asm), elim*]:
is-type $G \ T \longrightarrow G, s \vdash \text{default-val } T :: \preceq T$
 ⟨proof⟩

lemma *conf-widen* [*rule-format (no-asm), elim*]:
 $G \vdash T \preceq T' \implies G, s \vdash x :: \preceq T \longrightarrow \text{ws-prog } G \longrightarrow G, s \vdash x :: \preceq T'$
 ⟨proof⟩

lemma *conf-gext* [*rule-format (no-asm), elim*]:
 $G, s \vdash v :: \preceq T \longrightarrow s \leq |s' \longrightarrow G, s \uparrow v :: \preceq T$
 ⟨proof⟩

lemma *conf-list-widen* [*rule-format (no-asm)*]:
 $\text{ws-prog } G \implies$
 $\forall Ts \ Ts'. \text{list-all2 } (\text{conf } G \ s) \ \text{vs } Ts$
 $\longrightarrow G \vdash Ts[\preceq] \ Ts' \longrightarrow \text{list-all2 } (\text{conf } G \ s) \ \text{vs } Ts'$
 ⟨proof⟩

lemma *conf-RefTD* [*rule-format (no-asm)*]:
 $G, s \vdash a' :: \preceq \text{RefT } T$
 $\longrightarrow a' = \text{Null} \vee (\exists a \ \text{obj } T'. a' = \text{Addr } a \wedge \text{heap } s \ a = \text{Some } \text{obj} \wedge$
 $\text{obj-ty } \text{obj} = T' \wedge G \vdash T' \preceq \text{RefT } T)$
 ⟨proof⟩

value list conformance

constdefs

$\text{lconf} :: \text{prog} \Rightarrow \text{st} \Rightarrow ('a, \text{val}) \ \text{table} \Rightarrow ('a, \text{ty}) \ \text{table} \Rightarrow \text{bool}$
 $(-, +, -[\preceq]) - [71, 71, 71, 71] \ 70)$
 $G, s \vdash \text{vs}[\preceq] \ Ts \equiv \forall n. \forall T \in Ts \ n: \exists v \in \text{vs } n: G, s \vdash v :: \preceq T$

lemma *lconfD*: $\llbracket G, s \vdash \text{vs}[\preceq] \ Ts; Ts \ n = \text{Some } T \rrbracket \implies G, s \vdash (\text{the } (\text{vs } n)) :: \preceq T$
 ⟨proof⟩

lemma *lconf-cong* [*simp*]: $\bigwedge s. G, \text{set-locals } x \ s \uparrow l[\preceq] \ L = G, s \uparrow l[\preceq] \ L$
 ⟨proof⟩

lemma *lconf-lupd* [*simp*]: $G, \text{lupd}(vn \mapsto v) \ s \uparrow l[\preceq] \ L = G, s \uparrow l[\preceq] \ L$
 ⟨proof⟩

lemma *lconf-new*: $\llbracket L \ vn = \text{None}; G, s \uparrow l[\preceq] \ L \rrbracket \implies G, s \uparrow l(vn \mapsto v)[\preceq] \ L$

<proof>

lemma *lconf-upd*: $\llbracket G, s \vdash l[::\preceq]L; G, s \vdash v::\preceq T; L \text{ vn} = \text{Some } T \rrbracket \implies$
 $G, s \vdash l(\text{vn} \mapsto v)[::\preceq]L$

<proof>

lemma *lconf-ext*: $\llbracket G, s \vdash l[::\preceq]L; G, s \vdash v::\preceq T \rrbracket \implies G, s \vdash l(\text{vn} \mapsto v)[::\preceq]L(\text{vn} \mapsto T)$

<proof>

lemma *lconf-map-sum* [*simp*]:

$G, s \vdash l1 (+) l2[::\preceq]L1 (+) L2 = (G, s \vdash l1[::\preceq]L1 \wedge G, s \vdash l2[::\preceq]L2)$

<proof>

lemma *lconf-ext-list* [*rule-format (no-asm)*]:

$\bigwedge X. \llbracket G, s \vdash l[::\preceq]L \rrbracket \implies$

$\forall vs \ Ts. \text{distinct } vns \longrightarrow \text{length } Ts = \text{length } vns$

$\longrightarrow \text{list-all2 } (\text{conf } G \ s) \ vs \ Ts \longrightarrow G, s \vdash l(vns \mapsto vs)[::\preceq]L(vns \mapsto Ts)$

<proof>

lemma *lconf-deallocL*: $\llbracket G, s \vdash l[::\preceq]L(\text{vn} \mapsto T); L \text{ vn} = \text{None} \rrbracket \implies G, s \vdash l[::\preceq]L$

<proof>

lemma *lconf-geat* [*elim*]: $\llbracket G, s \vdash l[::\preceq]L; s \leq |s^\top \rrbracket \implies G, s \vdash l[::\preceq]L$

<proof>

lemma *lconf-empty* [*simp, intro!*]: $G, s \vdash vs[::\preceq]\text{empty}$

<proof>

lemma *lconf-init-vals* [*intro!*]:

$\forall n. \forall T \in fs \ n:\text{is-type } G \ T \implies G, s \vdash \text{init-vals } fs[::\preceq]fs$

<proof>

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

$wlconf :: \text{prog} \Rightarrow st \Rightarrow ('a, val) \text{ table} \Rightarrow ('a, ty) \text{ table} \Rightarrow bool$
 $(-, + - [\sim::\preceq] - [71, 71, 71, 71] \ 70)$
 $G, s \vdash vs[\sim::\preceq]Ts \equiv \forall n. \forall T \in Ts \ n: \forall v \in vs \ n: G, s \vdash v::\preceq T$

lemma *wlconfD*: $\llbracket G, s \vdash vs[\sim::\preceq]Ts; Ts \ n = \text{Some } T; vs \ n = \text{Some } v \rrbracket \implies G, s \vdash v::\preceq T$

<proof>

lemma *wlconf-cong* [simp]: $\bigwedge s. G, \text{set-locals } x \text{ st-l}[\sim::\preceq]L = G, \text{st-l}[\sim::\preceq]L$
 ⟨proof⟩

lemma *wlconf-lupd* [simp]: $G, \text{lupd}(vn \mapsto v) \text{st-l}[\sim::\preceq]L = G, \text{st-l}[\sim::\preceq]L$
 ⟨proof⟩

lemma *wlconf-upd*: $\llbracket G, \text{st-l}[\sim::\preceq]L; G, \text{st-v}::\preceq T; L \text{ vn} = \text{Some } T \rrbracket \implies$
 $G, \text{st-l}(vn \mapsto v)[\sim::\preceq]L$
 ⟨proof⟩

lemma *wlconf-ext*: $\llbracket G, \text{st-l}[\sim::\preceq]L; G, \text{st-v}::\preceq T \rrbracket \implies G, \text{st-l}(vn \mapsto v)[\sim::\preceq]L(vn \mapsto T)$
 ⟨proof⟩

lemma *wlconf-map-sum* [simp]:
 $G, \text{st-l1} (+) \text{l2}[\sim::\preceq]L1 (+) L2 = (G, \text{st-l1}[\sim::\preceq]L1 \wedge G, \text{st-l2}[\sim::\preceq]L2)$
 ⟨proof⟩

lemma *wlconf-ext-list* [rule-format (no-asm)]:
 $\bigwedge X. \llbracket G, \text{st-l}[\sim::\preceq]L \rrbracket \implies$
 $\forall vs \ Ts. \text{distinct } vns \longrightarrow \text{length } Ts = \text{length } vns$
 $\longrightarrow \text{list-all2 } (\text{conf } G \ s) \ vs \ Ts \longrightarrow G, \text{st-l}(vns[\mapsto]vs)[\sim::\preceq]L(vns[\mapsto]Ts)$
 ⟨proof⟩

lemma *wlconf-deallocL*: $\llbracket G, \text{st-l}[\sim::\preceq]L(vn \mapsto T); L \text{ vn} = \text{None} \rrbracket \implies G, \text{st-l}[\sim::\preceq]L$
 ⟨proof⟩

lemma *wlconf-geat* [elim]: $\llbracket G, \text{st-l}[\sim::\preceq]L; s \leq |s' \rrbracket \implies G, \text{st-l}[\sim::\preceq]L$
 ⟨proof⟩

lemma *wlconf-empty* [simp, intro!]: $G, \text{st-vs}[\sim::\preceq] \text{empty}$
 ⟨proof⟩

lemma *wlconf-empty-vals*: $G, \text{st-empty}[\sim::\preceq]ts$
 ⟨proof⟩

lemma *wlconf-init-vals* [intro!]:
 $\forall n. \forall T \in fs \ n: \text{is-type } G \ T \implies G, \text{st-init-vals } fs[\sim::\preceq]fs$
 ⟨proof⟩

lemma *lconf-wlconf*:
 $G, \text{st-l}[\preceq]L \implies G, \text{st-l}[\sim::\preceq]L$
 ⟨proof⟩

object conformance**constdefs**

$$\begin{aligned}
\text{oconf} &:: \text{prog} \Rightarrow \text{st} \Rightarrow \text{obj} \Rightarrow \text{oref} \Rightarrow \text{bool} \quad (-, \vdash :: \preceq \sqrt{-} \quad [71, 71, 71, 71] \quad 70) \\
&G, \text{s} \vdash \text{obj} :: \preceq \sqrt{r} \equiv G, \text{s} \vdash \text{values } \text{obj} [:: \preceq] \text{var-tys } G \text{ (tag obj) } r \wedge \\
&\quad (\text{case } r \text{ of} \\
&\quad \quad \text{Heap } a \Rightarrow \text{is-type } G \text{ (obj-ty obj)} \\
&\quad \quad | \text{Stat } C \Rightarrow \text{True})
\end{aligned}$$

lemma *oconf-is-type*: $G, \text{s} \vdash \text{obj} :: \preceq \sqrt{\text{Heap } a} \Longrightarrow \text{is-type } G \text{ (obj-ty obj)}$
 $\langle \text{proof} \rangle$

lemma *oconf-lconf*: $G, \text{s} \vdash \text{obj} :: \preceq \sqrt{r} \Longrightarrow G, \text{s} \vdash \text{values } \text{obj} [:: \preceq] \text{var-tys } G \text{ (tag obj) } r$
 $\langle \text{proof} \rangle$

lemma *oconf-cong [simp]*: $G, \text{set-locals } l \text{ s} \vdash \text{obj} :: \preceq \sqrt{r} = G, \text{s} \vdash \text{obj} :: \preceq \sqrt{r}$
 $\langle \text{proof} \rangle$

lemma *oconf-init-obj-lemma*:

$$\begin{aligned}
&\llbracket \wedge C \text{ c. class } G \text{ C} = \text{Some } c \Longrightarrow \text{unique (DeclConcepts.fields } G \text{ C)}; \\
&\quad \wedge C \text{ c f fld. } \llbracket \text{class } G \text{ C} = \text{Some } c; \\
&\quad \quad \text{table-of (DeclConcepts.fields } G \text{ C) } f = \text{Some fld} \rrbracket \\
&\quad \quad \Longrightarrow \text{is-type } G \text{ (type fld)}; \\
&\quad (\text{case } r \text{ of} \\
&\quad \quad \text{Heap } a \Rightarrow \text{is-type } G \text{ (obj-ty obj)} \\
&\quad \quad | \text{Stat } C \Rightarrow \text{is-class } G \text{ C}) \\
&\rrbracket \Longrightarrow G, \text{s} \vdash \text{obj} (\llbracket \text{values} := \text{init-vals (var-tys } G \text{ (tag obj) } r) \rrbracket) :: \preceq \sqrt{r} \\
&\langle \text{proof} \rangle
\end{aligned}$$
state conformance**constdefs**

$$\begin{aligned}
\text{conforms} &:: \text{state} \Rightarrow \text{env}' \Rightarrow \text{bool} \quad (\quad - :: \preceq - \quad [71, 71] \quad 70) \\
&\text{xs} :: \preceq E \equiv \text{let } (G, L) = E; \text{ s} = \text{snd xs}; \text{ l} = \text{locals } \text{s} \text{ in} \\
&\quad (\forall r. \forall \text{obj} \in \text{globs } \text{s} \text{ r:} \quad G, \text{s} \vdash \text{obj} \quad :: \preceq \sqrt{r}) \wedge \\
&\quad \quad G, \text{s} \vdash \text{l} \quad [\sim :: \preceq] L \quad \wedge \\
&\quad (\forall a. \text{fst xs} = \text{Some}(\text{Xcpt (Loc } a)) \longrightarrow G, \text{s} \vdash \text{Addr } a :: \preceq \text{Class (SXcpt Throwable)}) \wedge \\
&\quad (\text{fst xs} = \text{Some}(\text{Jump Ret}) \longrightarrow \text{l Result} \neq \text{None})
\end{aligned}$$
conforms

lemma *conforms-globsD*:

$$\llbracket (x, \text{s}) :: \preceq (G, L); \text{globs } \text{s} \text{ r} = \text{Some obj} \rrbracket \Longrightarrow G, \text{s} \vdash \text{obj} :: \preceq \sqrt{r} \\
\langle \text{proof} \rangle$$

lemma *conforms-localD*: $(x, \text{s}) :: \preceq (G, L) \Longrightarrow G, \text{s} \vdash \text{locals } \text{s} [\sim :: \preceq] L$
 $\langle \text{proof} \rangle$

lemma *conforms-XcptLocD*: $\llbracket (x, \text{s}) :: \preceq (G, L); x = \text{Some}(\text{Xcpt (Loc } a)) \rrbracket \Longrightarrow$
 $G, \text{s} \vdash \text{Addr } a :: \preceq \text{Class (SXcpt Throwable)}$
 $\langle \text{proof} \rangle$

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]:
 $((\text{raise-if } c \text{ xn } x, s)::\preceq(G, L)) = ((x, s)::\preceq(G, L))$
 $\langle \text{proof} \rangle$

lemma conforms-error-if [iff]:
 $((\text{error-if } c \text{ err } x, s)::\preceq(G, L)) = ((x, s)::\preceq(G, L))$
 $\langle \text{proof} \rangle$

lemma conforms-NormI: $(x, s)::\preceq(G, L) \implies \text{Norm } s::\preceq(G, L)$
 $\langle \text{proof} \rangle$

lemma conforms-absorb [rule-format]:
 $(a, b)::\preceq(G, L) \longrightarrow (\text{absorb } j \ a, b)::\preceq(G, L)$
 $\langle \text{proof} \rangle$

lemma conformsI: $\llbracket \forall r. \forall \text{obj} \in \text{globs } s \ r: G, s \vdash \text{obj}::\preceq \sqrt{r};$
 $G, s \vdash \text{locals } s [\sim::\preceq] L;$
 $\forall a. x = \text{Some } (\text{Xcpt } (\text{Loc } a)) \longrightarrow G, s \vdash \text{Addr } a::\preceq \text{Class } (\text{SXcpt Throwable});$
 $x = \text{Some } (\text{Jump Ret}) \longrightarrow \text{locals } s \text{ Result} \neq \text{None} \rrbracket \implies$
 $(x, s)::\preceq(G, L)$
 $\langle \text{proof} \rangle$

lemma conforms-xconf: $\llbracket (x, s)::\preceq(G, L);$
 $\forall a. x' = \text{Some } (\text{Xcpt } (\text{Loc } a)) \longrightarrow G, s \vdash \text{Addr } a::\preceq \text{Class } (\text{SXcpt Throwable});$

$x' = \text{Some } (\text{Jump Ret}) \longrightarrow \text{locals } s \text{ Result } \neq \text{None}] \Longrightarrow$
 $\llbracket (x', s) :: \preceq (G, L) \rrbracket$
 $\langle \text{proof} \rangle$

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: $\llbracket (x, s) :: \preceq (G, L); \text{globs } s r = \text{Some } \text{obj};$

$\text{var-tys } G (\text{tag } \text{obj}) r n = \text{Some } T; G, s \vdash v :: \preceq T \rrbracket \Longrightarrow (x, \text{upd-gobj } r n v s) :: \preceq (G, L)$

$\langle \text{proof} \rangle$

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:

$\llbracket (a, b) :: \preceq (G, L); L x = \text{Some } T; \text{locals } b x \neq \text{None} \rrbracket$

$\Longrightarrow G, b \vdash \text{the } (\text{locals } b x) :: \preceq T$

$\langle \text{proof} \rangle$

lemma *conforms-return*:

$\bigwedge s'. \llbracket (x,s)::\preceq(G, L); (x',s')::\preceq(G, L'); s \leq |s'; x' \neq \text{Some } (\text{Jump Ret}) \rrbracket \implies$

$(x', \text{set-locals } (\text{locals } s) s')::\preceq(G, L)$

<proof>

end

Chapter 18

DefiniteAssignmentCorrect

45 Correctness of Definite Assignment

theory *DefiniteAssignmentCorrect* **imports** *WellForm Eval begin*

declare $[[\text{simproc del: wt-expr wt-var wt-exprs wt-stmt}]]$

lemma *sxalloc-no-jump*:

assumes *sxalloc*: $G \vdash s0 \text{ --sxalloc} \rightarrow s1$ **and**
no-jmp: $\text{abrupt } s0 \neq \text{Some } (Jump\ j)$
shows $\text{abrupt } s1 \neq \text{Some } (Jump\ j)$

$\langle \text{proof} \rangle$

lemma *sxalloc-no-jump'*:

assumes *sxalloc*: $G \vdash s0 \text{ --sxalloc} \rightarrow s1$ **and**
jump: $\text{abrupt } s1 = \text{Some } (Jump\ j)$
shows $\text{abrupt } s0 = \text{Some } (Jump\ j)$

$\langle \text{proof} \rangle$

lemma *halloc-no-jump*:

assumes *halloc*: $G \vdash s0 \text{ --halloc } oi \succ a \rightarrow s1$ **and**
no-jmp: $\text{abrupt } s0 \neq \text{Some } (Jump\ j)$
shows $\text{abrupt } s1 \neq \text{Some } (Jump\ j)$

$\langle \text{proof} \rangle$

lemma *halloc-no-jump'*:

assumes *halloc*: $G \vdash s0 \text{ --halloc } oi \succ a \rightarrow s1$ **and**
jump: $\text{abrupt } s1 = \text{Some } (Jump\ j)$
shows $\text{abrupt } s0 = \text{Some } (Jump\ j)$

$\langle \text{proof} \rangle$

lemma *Body-no-jump*:

assumes *eval*: $G \vdash s0 \text{ --Body } D\ c \text{ --} \succ v \rightarrow s1$ **and**
jump: $\text{abrupt } s0 \neq \text{Some } (Jump\ j)$
shows $\text{abrupt } s1 \neq \text{Some } (Jump\ j)$

$\langle \text{proof} \rangle$

lemma *Methd-no-jump*:

assumes *eval*: $G \vdash s0 \text{ --Methd } D\ sig \text{ --} \succ v \rightarrow s1$ **and**
jump: $\text{abrupt } s0 \neq \text{Some } (Jump\ j)$
shows $\text{abrupt } s1 \neq \text{Some } (Jump\ j)$

$\langle \text{proof} \rangle$

lemma *jumpNestingOkS-mono*:

assumes *jumpNestingOk-l'*: $\text{jumpNestingOkS } jmps' \ c$
and *subset*: $jmps' \subseteq jmps$

shows $\text{jumpNestingOkS } jmps \ c$

$\langle \text{proof} \rangle$

corollary *jumpNestingOk-mono*:

assumes *jmpOk*: $\text{jumpNestingOk } jmps' \ t$
and *subset*: $jmps' \subseteq jmps$

shows $\text{jumpNestingOk } jmps \ t$

<proof>

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 $\text{abrupt } s = x$

<proof>

lemma *wt-init-comp-ty'*:

is-acc-type (*prg Env*) (*pid* (*cls Env*)) $T \implies \text{Env} \vdash \text{init-comp-ty } T :: \checkmark$

<proof>

lemma *fvar-upd-no-jump:*

assumes *upd*: $\text{upd} = \text{snd } (\text{fst } (\text{fvar } \text{statDeclC } \text{stat } \text{fn } a \ s'))$
and *noJmp*: $\text{abrupt } s \neq \text{Some } (\text{Jump } j)$
shows $\text{abrupt } (\text{upd } \text{val } s) \neq \text{Some } (\text{Jump } j)$

<proof>

lemma *avar-state-no-jump:*

assumes *jmp*: $\text{abrupt } (\text{snd } (\text{avar } G \ i \ a \ s)) = \text{Some } (\text{Jump } j)$
shows $\text{abrupt } s = \text{Some } (\text{Jump } j)$

<proof>

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 $\text{abrupt } (\text{upd } \text{val } s) \neq \text{Some } (\text{Jump } j)$

<proof>

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*: $G \vdash s0 \text{ -t> } \rightarrow (v, s1)$
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 $\text{no-jmp}: \forall j. \text{abrupt } s0 = \text{Some } (\text{Jump } j) \longrightarrow j \in \text{jmps}$
 $(\text{is } ?\text{Jmp } \text{jmps } s0)$
shows $(\forall j. \text{fst } s1 = \text{Some } (\text{Jump } j) \longrightarrow j \in \text{jmps}) \wedge$
 $(\text{normal } s1 \longrightarrow$
 $(\forall w \text{ upd}. v = \text{In2 } (w, \text{upd})$
 $\longrightarrow (\forall s \text{ j val.}$
 $\text{abrupt } s \neq \text{Some } (\text{Jump } j) \longrightarrow$
 $\text{abrupt } (\text{upd val } s) \neq \text{Some } (\text{Jump } j))))$
 $(\text{is } ?\text{Jmp } \text{jmps } s1 \wedge ?\text{Upd } v \text{ } s1)$
 $\langle \text{proof} \rangle$

lemmas $\text{jumpNestingOk-evalE} = \text{jumpNestingOk-eval} [\text{THEN } \text{conjE}, \text{rule-format}]$

lemma $\text{jumpNestingOk-eval-no-jump}$:
assumes $\text{eval}: \text{prg Env} \vdash s0 -t \succ \rightarrow (v, s1)$ **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 $\text{eval}: \text{prg Env} \vdash s0 -e \succ v \rightarrow s1$ **and**
 $\text{no-jmp}: \text{abrupt } s0 \neq \text{Some } (\text{Jump } j)$ **and**
 $\text{wt}: \text{Env} \vdash e :: -T$ **and**
 $\text{wf}: \text{wf-prog } (\text{prg Env})$
shows $\text{abrupt } s1 \neq \text{Some } (\text{Jump } j)$
 $\langle \text{proof} \rangle$

corollary eval-var-no-jump :
assumes $\text{eval}: \text{prg Env} \vdash s0 -\text{var} \Rightarrow \succ (w, \text{upd}) \rightarrow s1$ **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 $\text{eval}: \text{prg Env} \vdash s0 -c \rightarrow s1$ **and**
 $\text{jmpOk}: \text{jumpNestingOkS } \{ \} c$ **and**
 $\text{no-jmp}: \text{abrupt } s0 \neq \text{Some } (\text{Jump } j)$ **and**
 $\text{wt}: \text{Env} \vdash c :: \surd$ **and**
 $\text{wf}: \text{wf-prog } (\text{prg Env})$
shows $\text{abrupt } s1 \neq \text{Some } (\text{Jump } j)$

\langle proof \rangle

corollary *eval-expression-list-no-jump*:

assumes *eval*: $\text{prg Env} \vdash s0 \text{ --es--> } v \rightarrow s1$ **and**

no-jmp: $\text{abrupt } s0 \neq \text{Some } (\text{Jump } j)$ **and**

wt: $\text{Env} \vdash \text{es} :: \doteq T$ **and**

wf: $\text{wf-prog } (\text{prg Env})$

shows $\text{abrupt } s1 \neq \text{Some } (\text{Jump } j)$

\langle 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 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 proof \rangle

lemma *dom-locals-sxalloc-mono*:

assumes *sxalloc*: $G \vdash s0 \text{ --sxalloc--> } s1$

shows $\text{dom } (\text{locals } (\text{store } s0)) \subseteq \text{dom } (\text{locals } (\text{store } s1))$

\langle 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 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 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 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 proof \rangle

lemma *dom-locals-avar-vvar-mono*:

$\text{dom } (\text{locals } (\text{store } s))$

$\subseteq \text{dom } (\text{locals } (\text{store } (\text{snd } (\text{fst } (\text{avar } G \ i \ a \ s') \ \text{val } s))))$

<proof>

lemma *dom-locals-avar-mono*:

$dom (locals (store s))$
 $\subseteq dom (locals (store (snd (avar G i a s))))$
<proof>

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 $eval: G \vdash s0 \dashv t \rightarrow (v, s1)$
shows $dom (locals (store s0)) \subseteq dom (locals (store s1)) \wedge$
 $(\forall vv. v = In2 vv \wedge normal s1$
 $\rightarrow (\forall s val. dom (locals (store s))$
 $\subseteq dom (locals (store ((snd vv) val s))))$

<proof>

lemma *dom-locals-eval-mono-elim*:

assumes $eval: G \vdash s0 \dashv t \rightarrow (v, s1)$
obtains $dom (locals (store s0)) \subseteq dom (locals (store s1))$ **and**
 $\wedge vv s val. \llbracket v = In2 vv; normal s1 \rrbracket$
 $\implies dom (locals (store s))$
 $\subseteq dom (locals (store ((snd vv) val s)))$

<proof>

lemma *halloc-no-abrupt*:

assumes $halloc: G \vdash s0 \dashv halloc oi \rightarrow a \rightarrow s1$ **and**
 $normal: normal s1$

shows $normal s0$

<proof>

lemma *sxalloc-mono-no-abrupt*:

assumes $sxalloc: G \vdash s0 \dashv sxalloc \rightarrow s1$ **and**
 $normal: normal s1$

shows $normal s0$

<proof>

lemma *union-subseteqI*: $\llbracket A \cup B \subseteq C; A' \subseteq A; B' \subseteq B \rrbracket \implies A' \cup B' \subseteq C$

<proof>

lemma *union-subseteqII*: $\llbracket A \cup B \subseteq C; A' \subseteq A \rrbracket \implies A' \cup B \subseteq C$

<proof>

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: $G \vdash s0 \text{ -t> } \rightarrow (v, s1)$ **and**
normal: *normal s1*
shows $\text{assigns } t \subseteq \text{dom } (\text{locals } (\text{store } s1))$
 ⟨proof⟩

corollary *assignsE-good-approx*:
assumes
eval: $\text{prg Env} \vdash s0 \text{ -e-> } v \rightarrow s1$ **and**
normal: *normal s1*
shows $\text{assignsE } e \subseteq \text{dom } (\text{locals } (\text{store } s1))$
 ⟨proof⟩

corollary *assignsV-good-approx*:
assumes
eval: $\text{prg Env} \vdash s0 \text{ -v=> } vf \rightarrow s1$ **and**
normal: *normal s1*
shows $\text{assignsV } v \subseteq \text{dom } (\text{locals } (\text{store } s1))$
 ⟨proof⟩

corollary *assignsEs-good-approx*:
assumes
eval: $\text{prg Env} \vdash s0 \text{ -es=> } vs \rightarrow s1$ **and**
normal: *normal s1*
shows $\text{assignsEs } es \subseteq \text{dom } (\text{locals } (\text{store } s1))$
 ⟨proof⟩

lemma *constVal-eval*:
assumes *const*: $\text{constVal } e = \text{Some } c$ **and**
eval: $G \vdash \text{Norm } s0 \text{ -e-> } v \rightarrow s$
shows $v = c \wedge \text{normal } s$
 ⟨proof⟩

lemmas *constVal-eval-elim* = *constVal-eval* [THEN conjE]

lemma *eval-unop-type*:
typeof dt ($\text{eval-unop } \text{unop } v$) = *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-⋗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-⋗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⋗→ (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*:

assumes
prg Env⊢s0 -t⋗→ (v, s1) **and** *Env⊢t::T* **and**
Env⊢ dom (locals (store s0)) »t» A **and** *wf-prog (prg Env)*
obtains
normal s1 ⇒ nrm A ⊆ dom (locals (store s1)) **and**
 ∧ *l. [abrupt s1 = Some (Jump (Break l)); normal s0]*

$\implies \text{brk } A \ l \subseteq \text{dom } (\text{locals } (\text{store } s1))$ **and**
 $\llbracket \text{abrupt } s1 = \text{Some } (\text{Jump } \text{Ret}); \text{normal } s0 \rrbracket \implies \text{Result} \in \text{dom } (\text{locals } (\text{store } s1))$
 <proof>

lemma *da-good-approxE'*:

assumes *eval*: $G \vdash s0 \text{ -t>-} \rightarrow (v, s1)$
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*
obtains *normal* *s1* $\implies \text{nrm } A \subseteq \text{dom } (\text{locals } (\text{store } s1))$ **and**
 $\wedge l. \llbracket \text{abrupt } s1 = \text{Some } (\text{Jump } (\text{Break } l)); \text{normal } s0 \rrbracket$
 $\implies \text{brk } A \ l \subseteq \text{dom } (\text{locals } (\text{store } s1))$ **and**
 $\llbracket \text{abrupt } s1 = \text{Some } (\text{Jump } \text{Ret}); \text{normal } s0 \rrbracket$
 $\implies \text{Result} \in \text{dom } (\text{locals } (\text{store } s1))$

<proof>

declare $\llbracket \text{simproc } \text{add}: \text{wt-expr } \text{wt-var } \text{wt-exprs } \text{wt-stmt} \rrbracket$

end

Chapter 19

TypeSafe

46 The type soundness proof for Java

theory *TypeSafe*

imports *DefiniteAssignmentCorrect Conform*

begin

error free

hide *const field*

lemma *error-free-halloc:*

assumes *halloc: $G \vdash s0 \text{ --halloc } oi \triangleright a \rightarrow s1$ and*
error-free-s0: error-free s0

shows *error-free s1*

<proof>

lemma *error-free-sxalloc:*

assumes *sxalloc: $G \vdash s0 \text{ --sxalloc} \rightarrow s1$ and* *error-free-s0: error-free s0*
shows *error-free s1*

<proof>

lemma *error-free-check-field-access-eq:*

error-free (check-field-access G accC statDeclC fn stat a s)
 \implies *(check-field-access G accC statDeclC fn stat a s) = s*

<proof>

lemma *error-free-check-method-access-eq:*

error-free (check-method-access G accC statT mode sig a' s)
 \implies *(check-method-access G accC statT mode sig a' s) = s*

<proof>

lemma *error-free-FVar-lemma:*

error-free s
 \implies *error-free (abupd (if stat then id else np a) s)*

<proof>

lemma *error-free-init-lvars [simp,intro]:*

error-free s \implies
error-free (init-lvars G C sig mode a pvs s)

<proof>

lemma *error-free-LVar-lemma:*

error-free s \implies error-free (assign ($\lambda v. \text{supd lupd}(v \mapsto v)$) w s)

<proof>

lemma *error-free-throw [simp,intro]:*

error-free s \implies error-free (abupd (throw x) s)

<proof>

result conformance

constdefs

$$\begin{aligned}
& \text{assign-conforms} :: st \Rightarrow (val \Rightarrow state \Rightarrow state) \Rightarrow ty \Rightarrow env' \Rightarrow bool \\
& \quad (-\leq | -\leq :: \leq - \quad [71, 71, 71, 71] \ 70) \\
s \leq | f \leq T :: \leq E & \equiv \\
& (\forall s' w. Norm\ s' :: \leq E \longrightarrow fst\ E, s' \vdash w :: \leq T \longrightarrow s \leq | s' \longrightarrow assign\ f\ w\ (Norm\ s') :: \leq E) \wedge \\
& (\forall s' w. error-free\ s' \longrightarrow (error-free\ (assign\ f\ w\ s')))
\end{aligned}$$
constdefs

$$\begin{aligned}
rconf & :: prog \Rightarrow lenv \Rightarrow st \Rightarrow term \Rightarrow vals \Rightarrow tys \Rightarrow bool \\
& \quad (-, -, +, \succ :: \leq - \quad [71, 71, 71, 71, 71, 71] \ 70) \\
G, L, s \vdash t \succ v :: \leq T & \\
\equiv \text{case } T \text{ of} & \\
\quad Inl\ T \Rightarrow \text{if } (\exists\ var. t = In2\ var) & \\
\quad \text{then } (\forall\ n. (the-In2\ t) = LVar\ n & \\
\quad \longrightarrow (fst\ (the-In2\ v) = the\ (locals\ s\ n)) \wedge & \\
\quad \quad (locals\ s\ n \neq None \longrightarrow G, s \vdash fst\ (the-In2\ v) :: \leq T)) \wedge & \\
\quad \quad (\neg (\exists\ n. the-In2\ t = LVar\ n) \longrightarrow (G, s \vdash fst\ (the-In2\ v) :: \leq T)) \wedge & \\
\quad \quad (s \leq | snd\ (the-In2\ v) \leq T :: \leq (G, L)) & \\
\quad \text{else } G, s \vdash the-In1\ v :: \leq T & \\
| Inr\ Ts \Rightarrow list-all2\ (conf\ G\ s)\ (the-In3\ v)\ Ts &
\end{aligned}$$

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 $\exists var. t = In2\ var$ 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]:
$$G, L, s \vdash In1\ ec \succ In1\ v :: \leq Inl\ T = G, s \vdash v :: \leq T$$

<proof>

lemma *rconf-In2-no-LVar* [simp]:
$$\forall n. va \neq LVar\ n \implies$$

$$G, L, s \vdash In2\ va \succ In2\ vf :: \leq Inl\ T = (G, s \vdash fst\ vf :: \leq T \wedge s \leq | snd\ vf \leq T :: \leq (G, L))$$

<proof>

lemma *rconf-In2-LVar* [simp]:
$$va = LVar\ n \implies$$

$$G, L, s \vdash In2\ va \succ In2\ vf :: \leq Inl\ T$$

$$= ((fst\ vf = the\ (locals\ s\ n)) \wedge$$

$$(locals\ s\ n \neq None \longrightarrow G, s \vdash fst\ vf :: \leq T) \wedge s \leq | snd\ vf \leq T :: \leq (G, L))$$

<proof>

lemma *rconf-In3* [simp]:
$$G, L, s \vdash In3\ es \succ In3\ vs :: \leq Inr\ Ts = list-all2\ (\lambda v\ T. G, s \vdash v :: \leq T)\ vs\ Ts$$

<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*:

$\llbracket G, s \vdash v :: \preceq T; G \vdash T \preceq? T'; G, s \vdash v \text{ fits } T'; \text{ws-prog } G \rrbracket \implies G, s \vdash v :: \preceq T'$
 ⟨proof⟩

lemma *fits-Array*:

$\llbracket G, s \vdash v :: \preceq T; G \vdash T'. [] \preceq T. []; G, s \vdash v \text{ fits } T'; \text{ws-prog } G \rrbracket \implies G, s \vdash v :: \preceq T'$
 ⟨proof⟩

gext

lemma *halloc-gext*: $\bigwedge s1\ s2. G \vdash s1 \text{ -halloc } oi \succ a \rightarrow s2 \implies \text{snd } s1 \leq | \text{snd } s2$
 ⟨proof⟩

lemma *sxalloc-gext*: $\bigwedge s1\ s2. G \vdash s1 \text{ -sxalloc } \rightarrow s2 \implies \text{snd } s1 \leq | \text{snd } s2$
 ⟨proof⟩

lemma *eval-gext-lemma* [*rule-format (no-asm)*]:

$G \vdash s \text{ -t} \succ \rightarrow (w, s') \implies \text{snd } s \leq | \text{snd } s' \wedge (\text{case } w \text{ of}$
 $\text{In1 } v \Rightarrow \text{True}$
 $| \text{In2 } vf \Rightarrow \text{normal } s \rightarrow (\forall v\ x\ s. \text{snd } s \leq | \text{snd } (\text{assign } (\text{snd } vf)\ v\ (x, s)))$
 $| \text{In3 } vs \Rightarrow \text{True})$
 ⟨proof⟩

lemma *eval-gext-f*:

$G \vdash \text{Norm } s1 \text{ -e} \succ vf \rightarrow s2 \implies \text{snd } s1 \leq | \text{snd } (\text{assign } (\text{snd } vf)\ v\ (x, s))$
 ⟨proof⟩

lemmas *eval-gext* = *eval-gext-lemma* [*THEN conjunct1*]

lemma *eval-gext'*: $G \vdash (x1, s1) \text{ -t} \succ \rightarrow (w, (x2, s2)) \implies \text{snd } s1 \leq | \text{snd } s2$
 ⟨proof⟩

lemma *init-yields-initd*: $G \vdash \text{Norm } s1 \text{ -Init } C \rightarrow s2 \implies \text{initd } C\ s2$
 ⟨proof⟩

Lemmas

lemma *obj-ty-obj-class1*:

$\llbracket \text{wf-prog } G; \text{is-type } G\ (\text{obj-ty } \text{obj}) \rrbracket \implies \text{is-class } G\ (\text{obj-class } \text{obj})$
 ⟨proof⟩

lemma *oconf-init-obj*:

$\llbracket \text{wf-prog } G;$
 (case r of $\text{Heap } a \Rightarrow \text{is-type } G\ (\text{obj-ty } \text{obj}) \mid \text{Stat } C \Rightarrow \text{is-class } G\ C$)
 $\rrbracket \implies G, s \vdash \text{obj } (\text{values} := \text{init-vals } (\text{var-tys } G\ (\text{tag } \text{obj})\ r)) :: \preceq \sqrt{r}$
 ⟨proof⟩

lemma conforms-newG: $\llbracket \text{globs } s \text{ oref} = \text{None}; (x, s)::\preceq(G, L);$
 $\text{wf-prog } G; \text{ case oref of Heap } a \Rightarrow \text{is-type } G \text{ (obj-ty } (\text{tag=oi, values=vs}))$
 $\quad | \text{Stat } C \Rightarrow \text{is-class } G \text{ } C \rrbracket \Longrightarrow$
 $(x, \text{init-obj } G \text{ oi oref } s)::\preceq(G, L)$
 $\langle \text{proof} \rangle$

lemma conforms-init-class-obj:
 $\llbracket (x, s)::\preceq(G, L); \text{wf-prog } G; \text{ class } G \text{ } C = \text{Some } y; \neg \text{inited } C \text{ (globs } s) \rrbracket \Longrightarrow$
 $(x, \text{init-class-obj } G \text{ } C \text{ } s)::\preceq(G, L)$
 $\langle \text{proof} \rangle$

lemma fst-init-lvars[simp]:
 $\text{fst (init-lvars } G \text{ } C \text{ sig (invmode } m \text{ } e) \text{ } a' \text{ pvs } (x, s)) =$
 $(\text{if is-static } m \text{ then } x \text{ else (np } a') \text{ } x)$
 $\langle \text{proof} \rangle$

lemma halloc-conforms: $\bigwedge s1. \llbracket G \vdash s1 \text{ -halloc oi} \succ a \rightarrow s2; \text{wf-prog } G; s1::\preceq(G, L);$
 $\text{is-type } G \text{ (obj-ty } (\text{tag=oi, values=fs})) \rrbracket \Longrightarrow s2::\preceq(G, L)$
 $\langle \text{proof} \rangle$

lemma halloc-type-sound:
 $\bigwedge s1. \llbracket G \vdash s1 \text{ -halloc oi} \succ a \rightarrow (x, s); \text{wf-prog } G; s1::\preceq(G, L);$
 $T = \text{obj-ty } (\text{tag=oi, values=fs}); \text{is-type } G \text{ } T \rrbracket \Longrightarrow$
 $(x, s)::\preceq(G, L) \wedge (x = \text{None} \longrightarrow G, s \vdash \text{Addr } a::\preceq T)$
 $\langle \text{proof} \rangle$

lemma sxalloc-type-sound:
 $\bigwedge s1 \text{ } s2. \llbracket G \vdash s1 \text{ -sxalloc} \rightarrow s2; \text{wf-prog } G \rrbracket \Longrightarrow$
 $\text{case fst } s1 \text{ of}$
 $\quad \text{None} \Rightarrow s2 = s1$
 $\quad | \text{Some } \text{abr} \Rightarrow (\text{case } \text{abr} \text{ of}$
 $\quad \quad \text{Xcpt } x \Rightarrow (\exists a. \text{fst } s2 = \text{Some}(\text{Xcpt } (\text{Loc } a)) \wedge$
 $\quad \quad \quad (\forall L. s1::\preceq(G, L) \longrightarrow s2::\preceq(G, L)))$
 $\quad \quad | \text{Jump } j \Rightarrow s2 = s1$
 $\quad \quad | \text{Error } e \Rightarrow s2 = s1)$
 $\langle \text{proof} \rangle$

lemma wt-init-comp-ty:
 $\text{is-acc-type } G \text{ (pid } C) \text{ } T \Longrightarrow (\text{prg}=G, \text{cls}=C, \text{lcl}=L) \vdash \text{init-comp-ty } T::\checkmark$
 $\langle \text{proof} \rangle$

declare fun-upd-same [simp]

declare fun-upd-apply [simp del]

constdefs

$\text{DynT-prop}::[\text{prog, inv-mode, qname, ref-ty}] \Rightarrow \text{bool}$
 $(\text{-} \vdash \text{-} \rightarrow \text{-} \preceq \text{-} [\text{?1, ?1, ?1, ?1}] \text{ } ?0)$

$$G \vdash \text{mode} \rightarrow D \preceq t \equiv \text{mode} = \text{IntVir} \longrightarrow \text{is-class } G \ D \wedge \\ (\text{if } (\exists T. t = \text{ArrayT } T) \text{ then } D = \text{Object} \text{ else } G \vdash \text{Class } D \preceq \text{RefT } t)$$

lemma *DynT-propI*:

$$\llbracket (x, s) :: \preceq (G, L); G, s \vdash a' :: \preceq \text{RefT } \text{statT}; \text{wf-prog } G; \text{mode} = \text{IntVir} \longrightarrow a' \neq \text{Null} \rrbracket \\ \implies G \vdash \text{mode} \rightarrow \text{invocation-class } \text{mode } s \ a' \ \text{statT} \preceq \text{statT}$$

<proof>

lemma *invocation-methd*:

$$\llbracket \text{wf-prog } G; \text{statT} \neq \text{NullT}; \\ (\forall \text{statC}. \text{statT} = \text{ClassT } \text{statC} \longrightarrow \text{is-class } G \ \text{statC}); \\ (\forall I. \text{statT} = \text{IfaceT } I \longrightarrow \text{is-iface } G \ I \wedge \text{mode} \neq \text{SuperM}); \\ (\forall T. \text{statT} = \text{ArrayT } T \longrightarrow \text{mode} \neq \text{SuperM}); \\ G \vdash \text{mode} \rightarrow \text{invocation-class } \text{mode } s \ a' \ \text{statT} \preceq \text{statT}; \\ \text{dynlookup } G \ \text{statT} \ (\text{invocation-class } \text{mode } s \ a' \ \text{statT}) \ \text{sig} = \text{Some } m \rrbracket \\ \implies \text{methd } G \ (\text{invocation-declclass } G \ \text{mode } s \ a' \ \text{statT} \ \text{sig}) \ \text{sig} = \text{Some } m$$

<proof>

lemma *DynT-mheadsD*:

$$\llbracket G \vdash \text{invmode } sm \ e \rightarrow \text{invC} \preceq \text{statT}; \\ \text{wf-prog } G; (\text{prg} = G, \text{cls} = C, \text{lcl} = L) \vdash e :: -\text{RefT } \text{statT}; \\ (\text{statDeclT}, sm) \in \text{mheads } G \ C \ \text{statT} \ \text{sig}; \\ \text{invC} = \text{invocation-class } (\text{invmode } sm \ e) \ s \ a' \ \text{statT}; \\ \text{declC} = \text{invocation-declclass } G \ (\text{invmode } sm \ e) \ s \ a' \ \text{statT} \ \text{sig} \rrbracket \\ \implies \\ \exists dm. \\ \text{methd } G \ \text{declC} \ \text{sig} = \text{Some } dm \wedge \text{dynlookup } G \ \text{statT} \ \text{invC} \ \text{sig} = \text{Some } dm \wedge \\ G \vdash \text{resTy} \ (\text{methd } dm) \preceq \text{resTy } sm \wedge \\ \text{wf-mdecl } G \ \text{declC} \ (\text{sig}, \text{methd } dm) \wedge \\ \text{declC} = \text{declclass } dm \wedge \\ \text{is-static } dm = \text{is-static } sm \wedge \\ \text{is-class } G \ \text{invC} \wedge \text{is-class } G \ \text{declC} \wedge G \vdash \text{invC} \preceq_C \ \text{declC} \wedge \\ (\text{if } \text{invmode } sm \ e = \text{IntVir} \\ \text{then } (\forall \text{statC}. \text{statT} = \text{ClassT } \text{statC} \longrightarrow G \vdash \text{invC} \preceq_C \ \text{statC}) \\ \text{else } ((\exists \text{statC}. \text{statT} = \text{ClassT } \text{statC} \wedge G \vdash \text{statC} \preceq_C \ \text{declC}) \\ \vee (\forall \text{statC}. \text{statT} \neq \text{ClassT } \text{statC} \wedge \text{declC} = \text{Object})) \wedge \\ \text{statDeclT} = \text{ClassT} \ (\text{declclass } dm))$$

<proof>

corollary *DynT-mheadsE* [consumes γ]:

— 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*: $\text{invC} = \text{invocation-class } \text{mode } s \ a' \ \text{statT}$

and *declC*: $\text{declC} = \text{invocation-declclass } G \ \text{mode } s \ a' \ \text{statT} \ \text{sig}$

and *dm*: $\bigwedge dm. \llbracket \text{methd } G \ \text{declC} \ \text{sig} = \text{Some } dm;$

$\text{dynlookup } G \ \text{statT} \ \text{invC} \ \text{sig} = \text{Some } dm;$

$G \vdash \text{resTy} \ (\text{methd } dm) \preceq \text{resTy } sm;$

$\text{wf-mdecl } G \ \text{declC} \ (\text{sig}, \text{methd } dm);$

$\text{declC} = \text{declclass } dm;$

$\text{is-static } dm = \text{is-static } sm;$

$\text{is-class } G \ \text{invC}; \text{is-class } G \ \text{declC}; G \vdash \text{invC} \preceq_C \ \text{declC};$

$(\text{if } \text{invmode } sm \ e = \text{IntVir}$

$$\begin{aligned} & \text{then } (\forall \text{ statC}. \text{statT} = \text{ClassT statC} \longrightarrow G \vdash \text{invC} \preceq_C \text{statC}) \\ & \text{else } ((\exists \text{ statC}. \text{statT} = \text{ClassT statC} \wedge G \vdash \text{statC} \preceq_C \text{declC}) \\ & \quad \vee (\forall \text{ statC}. \text{statT} \neq \text{ClassT statC} \wedge \text{declC} = \text{Object})) \wedge \\ & \quad \text{statDeclT} = \text{ClassT (declclass dm)} \implies P \end{aligned}$$

shows P

$\langle \text{proof} \rangle$

lemma *DynT-conf*: $\llbracket G \vdash \text{invocation-class mode } s \text{ a' statT} \preceq_C \text{declC}; \text{wf-prog } G;$
 $\text{isrtype } G \text{ (statT)};$

$G, s \vdash a' :: \preceq \text{RefT statT}; \text{mode} = \text{IntVir} \longrightarrow a' \neq \text{Null};$

$\text{mode} \neq \text{IntVir} \longrightarrow ((\exists \text{ statC}. \text{statT} = \text{ClassT statC} \wedge G \vdash \text{statC} \preceq_C \text{declC})$
 $\quad \vee (\forall \text{ statC}. \text{statT} \neq \text{ClassT statC} \wedge \text{declC} = \text{Object})) \rrbracket$

$\implies G, s \vdash a' :: \preceq \text{Class declC}$

$\langle \text{proof} \rangle$

lemma *Ass-lemma*:

$\llbracket G \vdash \text{Norm } s0 \text{ -var} \Rightarrow (w, f) \rightarrow \text{Norm } s1; G \vdash \text{Norm } s1 \text{ -e} \rightarrow v \rightarrow \text{Norm } s2;$

$G, s2 \vdash v :: \preceq eT; s1 \leq |s2 \longrightarrow \text{assign } f v \text{ (Norm } s2) :: \preceq (G, L) \rrbracket$

$\implies \text{assign } f v \text{ (Norm } s2) :: \preceq (G, L) \wedge$

$(\text{normal } (\text{assign } f v \text{ (Norm } s2)) \longrightarrow G, \text{store } (\text{assign } f v \text{ (Norm } s2)) \vdash v :: \preceq eT)$

$\langle \text{proof} \rangle$

lemma *Throw-lemma*: $\llbracket G \vdash \text{tn} \preceq_C \text{SXcpt Throwable}; \text{wf-prog } G; (x1, s1) :: \preceq (G, L);$

$x1 = \text{None} \longrightarrow G, s1 \vdash a' :: \preceq \text{Class tn} \rrbracket \implies (\text{throw } a' x1, s1) :: \preceq (G, L)$

$\langle \text{proof} \rangle$

lemma *Try-lemma*: $\llbracket G \vdash \text{obj-ty (the (globs } s1' \text{ (Heap a)))} \preceq \text{Class tn};$

$(\text{Some } (\text{Xcpt } (\text{Loc } a)), s1') :: \preceq (G, L); \text{wf-prog } G \rrbracket$

$\implies \text{Norm } (\text{lupd}(vn \mapsto \text{Addr } a) s1') :: \preceq (G, L(vn \mapsto \text{Class tn}))$

$\langle \text{proof} \rangle$

lemma *Fin-lemma*:

$\llbracket G \vdash \text{Norm } s1 \text{ -c2} \rightarrow (x2, s2); \text{wf-prog } G; (\text{Some } a, s1) :: \preceq (G, L); (x2, s2) :: \preceq (G, L);$
 $\text{dom } (\text{locals } s1) \subseteq \text{dom } (\text{locals } s2) \rrbracket$

$\implies (\text{abrupt-if True } (\text{Some } a) x2, s2) :: \preceq (G, L)$

$\langle \text{proof} \rangle$

lemma *FVar-lemma1*:

$\llbracket \text{table-of } (\text{DeclConcepts.fields } G \text{ statC}) \text{ (fn, statDeclC)} = \text{Some } f ;$

$x2 = \text{None} \longrightarrow G, s2 \vdash a :: \preceq \text{Class statC}; \text{wf-prog } G; G \vdash \text{statC} \preceq_C \text{statDeclC};$

$\text{statDeclC} \neq \text{Object};$

$\text{class } G \text{ statDeclC} = \text{Some } y; (x2, s2) :: \preceq (G, L); s1 \leq |s2;$

$\text{inited } \text{statDeclC} \text{ (globs } s1);$

$(\text{if static } f \text{ then id else np } a) x2 = \text{None} \rrbracket$

\implies

$\exists \text{obj. globs } s2 \text{ (if static } f \text{ then Inr statDeclC else Inl (the-Addr } a))$

$= \text{Some obj} \wedge$

$\text{var-tys } G \text{ (tag obj) (if static } f \text{ then Inr statDeclC else Inl (the-Addr } a))$

$(\text{Inl (fn, statDeclC)}) = \text{Some (type } f)$

$\langle \text{proof} \rangle$

lemma *FVar-lemma2: error-free state*

\implies *error-free*
(assign
*($\lambda v.$ *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:*

$\llbracket ((v, f), \text{Norm } s2') = \text{fvar statDeclC (static field) fn a (x2, s2);$
 $G \vdash \text{statC} \preceq_C \text{statDeclC};$
 $\text{table-of (DeclConcepts.fields } G \text{ statC) (fn, statDeclC) = Some field};$
 $\text{wf-prog } G;$
 $x2 = \text{None} \implies G, s2 \vdash a :: \preceq \text{Class statC};$
 $\text{statDeclC} \neq \text{Object}; \text{class } G \text{ statDeclC} = \text{Some } y;$
 $(x2, s2) :: \preceq (G, L); s1 \leq |s2; \text{inited statDeclC (globs } s1) \rrbracket \implies$
 $G, s2 \uparrow v :: \preceq \text{type field} \wedge s2' \leq |f \preceq \text{type field} :: \preceq (G, L)$

<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:* $\llbracket \text{globs } s \text{ (Inl } a) = \text{Some obj}; \text{tag obj} = \text{Arr } ty \text{ } i;$
 $\text{the-Intg } i' \text{ in-bounds } i; \text{wf-prog } G; G \vdash ty.[] \preceq Tb.[]; \text{Norm } s :: \preceq (G, L)$
 $\rrbracket \implies G, s \vdash \text{the ((values obj) (Inr (the-Intg } i')) :: \preceq Tb$

<proof>

lemma *obj-split:* $\exists t \text{ vs. obj} = (\text{tag} = t, \text{values} = \text{vs})$

<proof>

lemma *AVar-lemma2: error-free state*

\implies *error-free*
(assign
(λv (x, s').
*(*raise-if* ($\neg G, s \uparrow v \text{ fits } T$) *ArrStore*) $x,$*
**upd-gobj (Inl a) (Inr (the-Intg } i)) v s'*)*
**w state*)*

<proof>

lemma *AVar-lemma:* $\llbracket \text{wf-prog } G; G \vdash (x1, s1) -e2 \rightarrow i \rightarrow (x2, s2);$

$((v, f), \text{Norm } s2') = \text{avar } G \text{ } i \text{ } a (x2, s2); x1 = \text{None} \implies G, s1 \vdash a :: \preceq Ta.[];$
 $(x2, s2) :: \preceq (G, L); s1 \leq |s2 \rrbracket \implies G, s2 \uparrow v :: \preceq Ta \wedge s2' \leq |f \preceq Ta :: \preceq (G, L)$

<proof>

Call

lemma *conforms-init-lvars-lemma*: $\llbracket wf\text{-prog } G;$
 $wf\text{-mhead } G \ P \ sig \ mh;$
 $list\text{-all2 } (conf \ G \ s) \ pvs \ pTsa; \ G \vdash pTsa[\preceq](parTs \ sig) \rrbracket \implies$
 $G, s \vdash empty \ (pars \ mh \mapsto pvs)$
 $[\sim::\preceq] \ table\text{-of } lvars(pars \ mh \mapsto parTs \ sig)$
 $\langle proof \rangle$

lemma *lconf-map-lname* [simp]:
 $G, s \vdash (lname\text{-case } l1 \ l2)[::\preceq](lname\text{-case } L1 \ L2)$
 $=$
 $(G, s \vdash l1[::\preceq]L1 \wedge G, s \vdash (\lambda x::unit . l2)[::\preceq](\lambda x::unit. L2))$
 $\langle proof \rangle$

lemma *wlconf-map-lname* [simp]:
 $G, s \vdash (lname\text{-case } l1 \ l2)[\sim::\preceq](lname\text{-case } L1 \ L2)$
 $=$
 $(G, s \vdash l1[\sim::\preceq]L1 \wedge G, s \vdash (\lambda x::unit . l2)[\sim::\preceq](\lambda x::unit. L2))$
 $\langle proof \rangle$

lemma *lconf-map-ename* [simp]:
 $G, s \vdash (ename\text{-case } l1 \ l2)[::\preceq](ename\text{-case } L1 \ L2)$
 $=$
 $(G, s \vdash l1[::\preceq]L1 \wedge G, s \vdash (\lambda x::unit. l2)[::\preceq](\lambda x::unit. L2))$
 $\langle proof \rangle$

lemma *wlconf-map-ename* [simp]:
 $G, s \vdash (ename\text{-case } l1 \ l2)[\sim::\preceq](ename\text{-case } L1 \ L2)$
 $=$
 $(G, s \vdash l1[\sim::\preceq]L1 \wedge G, s \vdash (\lambda x::unit. l2)[\sim::\preceq](\lambda x::unit. L2))$
 $\langle proof \rangle$

lemma *defval-conf1* [rule-format (no-asm), elim]:
 $is\text{-type } G \ T \longrightarrow (\exists v \in Some \ (default\text{-val } T): G, s \vdash v::\preceq T)$
 $\langle proof \rangle$

lemma *np-no-jump*: $x \neq Some \ (Jump \ j) \implies (np \ a') \ x \neq Some \ (Jump \ j)$
 $\langle proof \rangle$

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] $option.split\text{-asm}$ [split del]
 $\langle ML \rangle$

lemma *conforms-init-lvars*:
 $\llbracket wf\text{-mhead } G \ (pid \ declC) \ sig \ (mhead \ (mthd \ dm)); \ wf\text{-prog } G;$
 $list\text{-all2 } (conf \ G \ s) \ pvs \ pTsa; \ G \vdash pTsa[\preceq](parTs \ sig);$
 $(x, s)::\preceq(G, L);$

```

methd G declC sig = Some dm;
isrtype G statT;
G⊢invC≤C declC;
G,s⊢a'::≤RefT statT;
invmode (mhd sm) e = IntVir → a' ≠ Null;
invmode (mhd sm) e ≠ IntVir →
  (∃ statC. statT=ClassT statC ∧ G⊢statC≤C declC)
  ∨ (∀ statC. statT≠ClassT statC ∧ declC=Object);
invC = invocation-class (invmode (mhd sm) e) s a' statT;
declC = invocation-declclass G (invmode (mhd sm) e) s a' statT sig;
x≠Some (Jump Ret)
] ⇒
init-lvars G declC sig (invmode (mhd sm) e) a'
pvs (x,s)::≤(G,λ k.
  (case k of
    EName e ⇒ (case e of
      VName v
        ⇒ (table-of (lcls (mbody (mthd dm)))
          (pars (mthd dm)[→]parTs sig)) v
      | Res ⇒ Some (resTy (mthd dm)))
    | This ⇒ if (is-static (mthd sm))
      then None else Some (Class declC)))
⟨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⟩

```

47 accessibility

theorem *dynamic-field-access-ok:*

```

assumes wf: wf-prog G and
  not-Null: ¬ stat → a≠Null and
  conform-a: G,(store s)⊢a::≤ Class statC and
  conform-s: s::≤(G, L) and
  normal-s: normal s and
  wt-e: (⟦prg=G,cls=accC,lcl=L⟧)⊢e::-Class statC and
  f: accfield G accC statC fn = Some f and
  dynC: if stat then dynC=declclass f
    else dynC=obj-class (lookup-obj (store s) a) and
  stat: if stat then (is-static f) else (¬ is-static f)
shows table-of (DeclConcepts.fields G dynC) (fn,declclass f) = Some (fld f) ∧
  G⊢Field fn f in dynC dyn-accessible-from accC

```

⟨proof⟩

lemma *error-free-field-access:*

```

assumes accfield: accfield G accC statC fn = Some (statDeclC, f) and
  wt-e: (⟦prg = G, cls = accC, lcl = L⟧)⊢e::-Class statC and
  eval-init: G⊢Norm s0 -Init statDeclC → s1 and
  eval-e: G⊢s1 -e→ a → s2 and
  conf-s2: s2::≤(G, L) and
  conf-a: normal s2 ⇒ G, store s2⊢a::≤Class statC and
  fvar: (v,s2')=fvar statDeclC (is-static f) fn a s2 and
  wf: wf-prog G

```

shows check-field-access G accC statDeclC fn (is-static f) a s2' = s2'

⟨proof⟩

lemma *call-access-ok*:

assumes *invC-prop*: $G \vdash \text{invmode } \text{statM } e \rightarrow \text{invC} \preceq \text{statT}$
and *wf*: *wf-prog* *G*
and *wt-e*: $(\text{prg} = G, \text{cls} = C, \text{lcl} = L) \vdash e :: \text{--RefT } \text{statT}$
and *statM*: $(\text{statDeclT}, \text{statM}) \in \text{mheads } G \text{ accC } \text{statT } \text{sig}$
and *invC*: $\text{invC} = \text{invocation-class } (\text{invmode } \text{statM } e) \text{ s a } \text{statT}$
shows $\exists \text{ dynM}. \text{dynlookup } G \text{ statT } \text{invC } \text{sig} = \text{Some } \text{dynM} \wedge$
 $G \vdash \text{Methd } \text{sig } \text{dynM} \text{ in } \text{invC } \text{dyn-accessible-from } \text{accC}$
 $\langle \text{proof} \rangle$

lemma *error-free-call-access*:

assumes
eval-args: $G \vdash s1 \text{ --args} \Rightarrow vs \rightarrow s2$ **and**
wt-e: $(\text{prg} = G, \text{cls} = \text{accC}, \text{lcl} = L) \vdash e :: \text{--(RefT } \text{statT})$ **and**
statM: $\text{max-spec } G \text{ accC } \text{statT } (\text{name} = \text{mn}, \text{parTs} = \text{pTs})$
 $= \{(\text{statDeclT}, \text{statM}), \text{pTs}'\}$ **and**
conf-s2: $s2 :: \preceq(G, L)$ **and**
conf-a: $\text{normal } s1 \Rightarrow G, \text{store } s1 \vdash a :: \preceq \text{RefT } \text{statT}$ **and**
invProp: $\text{normal } s3 \Rightarrow$
 $G \vdash \text{invmode } \text{statM } e \rightarrow \text{invC} \preceq \text{statT}$ **and**
s3: $s3 = \text{init-lvars } G \text{ invDeclC } (\text{name} = \text{mn}, \text{parTs} = \text{pTs}')$
 $(\text{invmode } \text{statM } e) \text{ a } vs \text{ s2}$ **and**
invC: $\text{invC} = \text{invocation-class } (\text{invmode } \text{statM } e) (\text{store } s2) \text{ a } \text{statT}$ **and**
invDeclC: $\text{invDeclC} = \text{invocation-declclass } G (\text{invmode } \text{statM } e) (\text{store } s2)$
 $\text{a } \text{statT } (\text{name} = \text{mn}, \text{parTs} = \text{pTs}')$ **and**
wf: *wf-prog* *G*
shows $\text{check-method-access } G \text{ accC } \text{statT } (\text{invmode } \text{statM } e) (\text{name} = \text{mn}, \text{parTs} = \text{pTs}') \text{ a } s3$
 $= s3$
 $\langle \text{proof} \rangle$

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*:

$\bigwedge \text{tab } qs \ x \ y. \text{length } ps = \text{length } qs$
 $\Rightarrow \text{tab}(ps[\mapsto]qs)(x \mapsto y) = \text{tab}(ps @ [x][\mapsto]qs @ [y])$
 $\langle \text{proof} \rangle$

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*:

$\bigwedge \text{tab } \text{tab}' \ ys. \text{tab } z = \text{tab}' z \Rightarrow (\text{tab}(xs[\mapsto]ys)) z = (\text{tab}'(xs[\mapsto]ys)) z$
 $\langle \text{proof} \rangle$

lemma *map-upd-override*: $(\text{tab}(x \mapsto y)) x = (\text{tab}'(x \mapsto y)) x$
 ⟨proof⟩

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*:
 $\bigwedge \text{tab } qs. \text{length } ps = \text{length } qs$
 $\implies \text{tab}(ps [\mapsto] qs)(xs [\mapsto] ys) = \text{tab}(ps @ xs [\mapsto] qs @ ys)$
 ⟨proof⟩

lemma *map-upd-cut-irrelevant*:
 $\llbracket (\text{tab}(x \mapsto y)) vn = \text{Some } el; (\text{tab}'(x \mapsto y)) vn = \text{None} \rrbracket$
 $\implies \text{tab } vn = \text{Some } el$
 ⟨proof⟩

lemma *map-upd-Some-expand*:
 $\llbracket \text{tab } vn = \text{Some } z \rrbracket$
 $\implies \exists z. (\text{tab}(x \mapsto y)) vn = \text{Some } z$
 ⟨proof⟩

lemma *map-upds-Some-expand*:
 $\bigwedge \text{tab } ys z. \llbracket \text{tab } vn = \text{Some } z \rrbracket$
 $\implies \exists z. (\text{tab}(xs [\mapsto] ys)) vn = \text{Some } z$
 ⟨proof⟩

lemma *map-upd-Some-swap*:
 $(\text{tab}(r \mapsto w)(u \mapsto v)) vn = \text{Some } z \implies \exists z. (\text{tab}(u \mapsto v)(r \mapsto w)) vn = \text{Some } z$
 ⟨proof⟩

lemma *map-upd-None-swap*:
 $(\text{tab}(r \mapsto w)(u \mapsto v)) vn = \text{None} \implies (\text{tab}(u \mapsto v)(r \mapsto w)) vn = \text{None}$
 ⟨proof⟩

lemma *map-eq-upd-eq*: $\text{tab } vn = \text{tab}' vn \implies (\text{tab}(x \mapsto y)) vn = (\text{tab}'(x \mapsto y)) vn$
 ⟨proof⟩

lemma *map-upd-in-expansion-map-swap*:
 $\llbracket (\text{tab}(x \mapsto y)) vn = \text{Some } z; \text{tab } vn \neq \text{Some } z \rrbracket$
 $\implies (\text{tab}'(x \mapsto y)) vn = \text{Some } z$
 ⟨proof⟩

lemma *map-upds-in-expansion-map-swap*:

$$\wedge \text{tab } \text{tab}' \text{ } \text{ys } z. \llbracket (\text{tab}(xs[\mapsto]ys)) \text{ } vn = \text{Some } z; \text{tab } vn \neq \text{Some } z \rrbracket \\ \implies (\text{tab}'(xs[\mapsto]ys)) \text{ } vn = \text{Some } z$$

<proof>

lemma *map-upds-Some-swap*:

assumes *r-u*: $(\text{tab}(r \mapsto w)(u \mapsto v)(xs[\mapsto]ys)) \text{ } vn = \text{Some } z$

shows $\exists z. (\text{tab}(u \mapsto v)(r \mapsto w)(xs[\mapsto]ys)) \text{ } vn = \text{Some } z$

<proof>

lemma *map-upds-Some-insert*:

assumes *z*: $(\text{tab}(xs[\mapsto]ys)) \text{ } vn = \text{Some } z$

shows $\exists z. (\text{tab}(u \mapsto v)(xs[\mapsto]ys)) \text{ } vn = \text{Some } z$

<proof>

lemma *map-upds-None-cut*:

assumes *expand-None*: $(\text{tab}(xs[\mapsto]ys)) \text{ } vn = \text{None}$

shows $\text{tab } vn = \text{None}$

<proof>

lemma *map-upds-cut-irrelevant*:

$$\wedge \text{tab } \text{tab}' \text{ } \text{ys}. \llbracket (\text{tab}(xs[\mapsto]ys)) \text{ } vn = \text{Some } \text{el}; (\text{tab}'(xs[\mapsto]ys)) \text{ } vn = \text{None} \rrbracket \\ \implies \text{tab } vn = \text{Some } \text{el}$$

<proof>

lemma *dom-vname-split*:

$$\text{dom } (\text{lname-case } (\text{ename-case } (\text{tab}(x \mapsto y)(xs[\mapsto]ys)) \text{ } a) \text{ } b) \\ = \text{dom } (\text{lname-case } (\text{ename-case } (\text{tab}(x \mapsto y)) \text{ } a) \text{ } b) \cup \\ \text{dom } (\text{lname-case } (\text{ename-case } (\text{tab}(xs[\mapsto]ys)) \text{ } a) \text{ } b) \\ (\text{is } ?\text{List } x \text{ } xs \text{ } y \text{ } ys = ?\text{Hd } x \text{ } y \cup ?\text{Tl } xs \text{ } ys)$$

<proof>

lemma *dom-map-upd*: $\wedge \text{tab}. \text{dom } (\text{tab}(x \mapsto y)) = \text{dom } \text{tab} \cup \{x\}$

<proof>

lemma *dom-map-upds*: $\wedge \text{tab } \text{ys}. \text{length } xs = \text{length } ys$

$\implies \text{dom } (\text{tab}(xs[\mapsto]ys)) = \text{dom } \text{tab} \cup \text{set } xs$

<proof>

lemma *dom-ename-case-None-simp*:

$\text{dom } (\text{ename-case } \text{vname-tab } \text{None}) = \text{VName } \text{' } (\text{dom } \text{vname-tab})$

<proof>

lemma *dom-ename-case-Some-simp*:

$\text{dom } (\text{ename-case } \text{vname-tab } (\text{Some } a)) = \text{VName } \text{' } (\text{dom } \text{vname-tab}) \cup \{\text{Res}\}$

<proof>

lemma *dom-lname-case-None-simp*:

$dom (lname-case\ ename-tab\ None) = EName \text{ ' } (dom\ ename-tab)$
 ⟨proof⟩

lemma *dom-lname-case-Some-simp*:

$dom (lname-case\ ename-tab\ (Some\ a)) = EName \text{ ' } (dom\ ename-tab) \cup \{This\}$
 ⟨proof⟩

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*:

$f \text{ ' } g \text{ ' } A = (f \circ g) \text{ ' } A$
 ⟨proof⟩

lemma *dom-locals-init-lvars*:

assumes *m*: $m = (mthd\ (the\ (methd\ G\ C\ sig)))$
assumes *len*: $length\ (pars\ m) = length\ pvs$
shows $dom\ (locals\ (store\ (init-lvars\ G\ C\ sig\ (invmode\ m\ e)\ a\ pvs\ s)))$
 $= parameters\ m$

⟨proof⟩

lemma *da-e2-BinOp*:

assumes *da*: $(\langle prg = G, cls = accC, lcl = L \rangle \vdash dom\ (locals\ (store\ s0)) \gg \langle BinOp\ binop\ e1\ e2 \rangle_e) \gg A$
and *wt-e1*: $(\langle prg = G, cls = accC, lcl = L \rangle \vdash e1 :: -e1T)$
and *wt-e2*: $(\langle prg = G, cls = accC, lcl = L \rangle \vdash e2 :: -e2T)$
and *wt-binop*: $wt\ binop\ G\ binop\ e1T\ e2T$
and *conf-s0*: $s0 :: \preceq (G, L)$
and *normal-s1*: $normal\ s1$
and *eval-e1*: $G \vdash s0 -e1 \multimap v1 \rightarrow s1$
and *conf-v1*: $G, store\ s1 \vdash v1 :: \preceq e1T$
and *wf*: $wf\ prog\ G$
shows $\exists E2. (\langle prg = G, cls = accC, lcl = L \rangle \vdash dom\ (locals\ (store\ s1))$
 $\gg (if\ need\ second\ arg\ binop\ v1\ then\ \langle e2 \rangle_e\ else\ \langle Skip \rangle_s) \gg E2$

⟨proof⟩

main proof of type safety

lemma *eval-type-sound*:

assumes *eval*: $G \vdash s0 -t \multimap (v, s1)$
and *wt*: $(\langle prg = G, cls = accC, lcl = L \rangle \vdash t :: T)$
and *da*: $(\langle prg = G, cls = accC, lcl = L \rangle \vdash dom\ (locals\ (store\ s0)) \gg t \gg A)$
and *wf*: $wf\ prog\ G$
and *conf-s0*: $s0 :: \preceq (G, L)$
shows $s1 :: \preceq (G, L) \wedge (normal\ s1 \longrightarrow G, L, store\ s1 \vdash t \multimap v :: \preceq T) \wedge$
 $(error\ free\ s0 = error\ free\ s1)$

⟨proof⟩

corollary *eval-type-soundE* [consumes 5]:

assumes *eval*: $G \vdash s0 -t \multimap (v, s1)$
and *conf*: $s0 :: \preceq (G, L)$
and *wt*: $(\langle prg = G, cls = accC, lcl = L \rangle \vdash t :: T)$

and $da: (\text{prg} = G, \text{cls} = \text{acc}C, \text{lcl} = L) \vdash \text{dom} (\text{locals} (\text{snd } s0)) \gg t \gg A$
and $wf: wf\text{-prog } G$
and $\text{elim}: \llbracket s1 :: \preceq(G, L); \text{normal } s1 \implies G, L, \text{snd } s1 \vdash t \gg v :: \preceq T; \text{error-free } s0 = \text{error-free } s1 \rrbracket \implies P$

shows P

$\langle \text{proof} \rangle$

corollary eval-ts :

$\llbracket G \vdash s - e - \gg v \rightarrow s'; wf\text{-prog } G; s :: \preceq(G, L); (\text{prg} = G, \text{cls} = C, \text{lcl} = L) \vdash e :: -T;$
 $(\text{prg} = G, \text{cls} = C, \text{lcl} = L) \vdash \text{dom} (\text{locals} (\text{store } s)) \gg \text{In1 } e \gg A \rrbracket$
 $\implies s' :: \preceq(G, L) \wedge (\text{normal } s' \longrightarrow G, \text{store } s \vdash v :: \preceq T) \wedge$
 $(\text{error-free } s = \text{error-free } s')$

$\langle \text{proof} \rangle$

corollary evals-ts :

$\llbracket G \vdash s - es \doteq \gg vs \rightarrow s'; wf\text{-prog } G; s :: \preceq(G, L); (\text{prg} = G, \text{cls} = C, \text{lcl} = L) \vdash es :: \doteq Ts;$
 $(\text{prg} = G, \text{cls} = C, \text{lcl} = L) \vdash \text{dom} (\text{locals} (\text{store } s)) \gg \text{In3 } es \gg A \rrbracket$
 $\implies s' :: \preceq(G, L) \wedge (\text{normal } s' \longrightarrow \text{list-all2} (\text{conf } G (\text{store } s')) vs Ts) \wedge$
 $(\text{error-free } s = \text{error-free } s')$

$\langle \text{proof} \rangle$

corollary eval-ts :

$\llbracket G \vdash s - v \doteq \gg vf \rightarrow s'; wf\text{-prog } G; s :: \preceq(G, L); (\text{prg} = G, \text{cls} = C, \text{lcl} = L) \vdash v :: =T;$
 $(\text{prg} = G, \text{cls} = C, \text{lcl} = L) \vdash \text{dom} (\text{locals} (\text{store } s)) \gg \text{In2 } v \gg A \rrbracket \implies$
 $s' :: \preceq(G, L) \wedge (\text{normal } s' \longrightarrow G, L, (\text{store } s') \vdash \text{In2 } v \gg \text{In2 } vf :: \preceq \text{In1 } T) \wedge$
 $(\text{error-free } s = \text{error-free } s')$

$\langle \text{proof} \rangle$

theorem exec-ts :

$\llbracket G \vdash s - c \rightarrow s'; wf\text{-prog } G; s :: \preceq(G, L); (\text{prg} = G, \text{cls} = C, \text{lcl} = L) \vdash c :: \checkmark;$
 $(\text{prg} = G, \text{cls} = C, \text{lcl} = L) \vdash \text{dom} (\text{locals} (\text{store } s)) \gg \text{In1r } c \gg A \rrbracket$
 $\implies s' :: \preceq(G, L) \wedge (\text{error-free } s \longrightarrow \text{error-free } s')$

$\langle \text{proof} \rangle$

lemma $wf\text{-eval-Fin}$:

assumes $wf: wf\text{-prog } G$
and $wt\text{-}c1: (\text{prg} = G, \text{cls} = C, \text{lcl} = L) \vdash \text{In1r } c1 :: \text{In1} (\text{PrimT } \text{Void})$
and $da\text{-}c1: (\text{prg} = G, \text{cls} = C, \text{lcl} = L) \vdash \text{dom} (\text{locals} (\text{store} (\text{Norm } s0))) \gg \text{In1r } c1 \gg A$
and $\text{conf-s0}: \text{Norm } s0 :: \preceq(G, L)$
and $\text{eval-c1}: G \vdash \text{Norm } s0 - c1 \rightarrow (x1, s1)$
and $\text{eval-c2}: G \vdash \text{Norm } s1 - c2 \rightarrow s2$
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*: $G \vdash s0 \text{ -}t \rightarrow (v, s1)$
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*: $\bigwedge s \ t \ \text{abr} \ L \ \text{acc}C \ T \ A.$
 $\llbracket (\text{prg}=G, \text{cls}=\text{acc}C, \text{lcl}=L) \vdash t :: T;$
 $\llbracket (\text{prg}=G, \text{cls}=\text{acc}C, \text{lcl}=L) \vdash \text{dom} (\text{locals} (\text{store} (\text{Some } \text{abr}, s))) \gg t \gg A$
 $\rrbracket \Longrightarrow P \ L \ \text{acc}C \ (\text{Some } \text{abr}, s) \ t \ (\text{arbitrary3 } t) \ (\text{Some } \text{abr}, s)$
and *skip*: $\bigwedge s \ L \ \text{acc}C. P \ L \ \text{acc}C \ (\text{Norm } s) \ \langle \text{Skip} \rangle_s \ \diamond (\text{Norm } s)$
and *expr*: $\bigwedge e \ s0 \ s1 \ v \ L \ \text{acc}C \ eT \ E.$
 $\llbracket (\text{prg}=G, \text{cls}=\text{acc}C, \text{lcl}=L) \vdash e :: \text{-}eT;$
 $\llbracket (\text{prg}=G, \text{cls}=\text{acc}C, \text{lcl}=L)$
 $\vdash \text{dom} (\text{locals} (\text{store} ((\text{Norm } s0)::\text{state}))) \gg \langle e \rangle_e \gg E;$
 $P \ L \ \text{acc}C \ (\text{Norm } s0) \ \langle e \rangle_e \ [v]_e \ s1 \rrbracket$
 $\Longrightarrow P \ L \ \text{acc}C \ (\text{Norm } s0) \ \langle \text{Expr } e \rangle_s \ \diamond s1$
and *lab*: $\bigwedge c \ l \ s0 \ s1 \ L \ \text{acc}C \ C.$
 $\llbracket (\text{prg}=G, \text{cls}=\text{acc}C, \text{lcl}=L) \vdash c :: \checkmark;$
 $\llbracket (\text{prg}=G, \text{cls}=\text{acc}C, \text{lcl}=L)$
 $\vdash \text{dom} (\text{locals} (\text{store} ((\text{Norm } s0)::\text{state}))) \gg \langle c \rangle_s \gg C;$
 $P \ L \ \text{acc}C \ (\text{Norm } s0) \ \langle c \rangle_s \ \diamond s1 \rrbracket$
 $\Longrightarrow P \ L \ \text{acc}C \ (\text{Norm } s0) \ \langle l \cdot c \rangle_s \ \diamond (\text{abupd} (\text{absorb } l) \ s1)$
and *comp*: $\bigwedge c1 \ c2 \ s0 \ s1 \ s2 \ L \ \text{acc}C \ C1.$
 $\llbracket G \vdash \text{Norm } s0 \text{ -}c1 \rightarrow s1; G \vdash s1 \text{ -}c2 \rightarrow s2;$
 $\llbracket (\text{prg}=G, \text{cls}=\text{acc}C, \text{lcl}=L) \vdash c1 :: \checkmark;$
 $\llbracket (\text{prg}=G, \text{cls}=\text{acc}C, \text{lcl}=L) \vdash c2 :: \checkmark;$
 $\llbracket (\text{prg}=G, \text{cls}=\text{acc}C, \text{lcl}=L) \vdash$
 $\text{dom} (\text{locals} (\text{store} ((\text{Norm } s0)::\text{state}))) \gg \langle c1 \rangle_s \gg C1;$
 $P \ L \ \text{acc}C \ (\text{Norm } s0) \ \langle c1 \rangle_s \ \diamond s1;$
 $\bigwedge Q. \llbracket \text{normal } s1;$
 $\bigwedge C2. \llbracket (\text{prg}=G, \text{cls}=\text{acc}C, \text{lcl}=L)$
 $\vdash \text{dom} (\text{locals} (\text{store } s1)) \gg \langle c2 \rangle_s \gg C2;$
 $P \ L \ \text{acc}C \ s1 \ \langle c2 \rangle_s \ \diamond s2 \rrbracket \Longrightarrow Q$
 $\rrbracket \Longrightarrow Q$
 $\rrbracket \Longrightarrow P \ L \ \text{acc}C \ (\text{Norm } s0) \ \langle c1;; c2 \rangle_s \ \diamond s2$
and *if*: $\bigwedge b \ c1 \ c2 \ e \ s0 \ s1 \ s2 \ L \ \text{acc}C \ E.$
 $\llbracket G \vdash \text{Norm } s0 \text{ -}e \text{ -} \succ b \rightarrow s1;$
 $G \vdash s1 \text{ -}(\text{if the-Bool } b \text{ then } c1 \text{ else } c2) \rightarrow s2;$
 $\llbracket (\text{prg}=G, \text{cls}=\text{acc}C, \text{lcl}=L) \vdash e :: \text{-PrimT Boolean};$
 $\llbracket (\text{prg}=G, \text{cls}=\text{acc}C, \text{lcl}=L) \vdash (\text{if the-Bool } b \text{ then } c1 \text{ else } c2) :: \checkmark;$
 $\llbracket (\text{prg}=G, \text{cls}=\text{acc}C, \text{lcl}=L) \vdash$
 $\text{dom} (\text{locals} (\text{store} ((\text{Norm } s0)::\text{state}))) \gg \langle e \rangle_e \gg E;$
 $P \ L \ \text{acc}C \ (\text{Norm } s0) \ \langle e \rangle_e \ [b]_e \ s1;$
 $\bigwedge Q. \llbracket \text{normal } s1;$
 $\bigwedge C. \llbracket (\text{prg}=G, \text{cls}=\text{acc}C, \text{lcl}=L) \vdash (\text{dom} (\text{locals} (\text{store } s1)))$
 $\gg \langle \text{if the-Bool } b \text{ then } c1 \text{ else } c2 \rangle_s \gg C;$
 $P \ L \ \text{acc}C \ s1 \ \langle \text{if the-Bool } b \text{ then } c1 \text{ else } c2 \rangle_s \ \diamond s2$
 $\rrbracket \Longrightarrow Q$
 $\rrbracket \Longrightarrow Q$
 $\rrbracket \Longrightarrow P \ L \ \text{acc}C \ (\text{Norm } s0) \ \langle \text{If}(e) \ c1 \ \text{Else } c2 \rangle_s \ \diamond s2$

shows $P \ L \ \text{acc}C \ s0 \ t \ v \ s1$

<proof>

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.

inductive

```

evaln :: [prog, state, term, nat, vals, state] => bool
  (+- -->----> '(-, -') [61,61,80,61,0,0] 60)
and evaln :: [prog, state, var, vvar, nat, state] => bool
  (+- --=>----> - [61,61,90,61,61,61] 60)
and evaln :: [prog, state, expr, val, nat, state] => bool
  (+- ---->----> - [61,61,80,61,61,61] 60)
and evaln :: [prog, state, expr list, val list, nat, state] => bool
  (+- --≡>----> - [61,61,61,61,61,61] 60)
and execn :: [prog, state, stmt, nat, state] => bool
  (+- ---->----> - [61,61,65, 61,61] 60)
for G :: prog

```

where

```

  G⊢s -c -n→ s' ≡ G⊢s -In1r c>-n→ (◇ , s')
| G⊢s -e->v -n→ s' ≡ G⊢s -In1l e>-n→ (In1 v , s')
| G⊢s -e=>vf -n→ s' ≡ G⊢s -In2 e>-n→ (In2 vf , s')
| G⊢s -e≡>v -n→ s' ≡ G⊢s -In3 e>-n→ (In3 v , s')

```

— propagation of abrupt completion

```
| Abrupt: G⊢(Some xc,s) -t>-n→ (arbitrary3 t,(Some xc,s))
```

— evaluation of variables

```

| LVar: G⊢Norm s -LVar vn=>lvar vn s-n→ Norm s
| FVar: [[G⊢Norm s0 -Init statDeclC-n→ s1; G⊢s1 -e->a-n→ s2;
  (v,s2') = fvar statDeclC stat fn a s2;
  s3 = check-field-access G accC statDeclC fn stat a s2]] =>
  G⊢Norm s0 -{accC,statDeclC,stat}e..fn=>v-n→ s3
| AVar: [[G⊢ Norm s0 -e1->a-n→ s1 ; G⊢s1 -e2->i-n→ s2;
  (v,s2') = avar G i a s2]] =>
  G⊢Norm s0 -e1.[e2]=>v-n→ s2'

```

— evaluation of expressions

```
| NewC: [[G⊢Norm s0 -Init C-n→ s1;
```


— evaluation of expression lists

| *Nil*:
$$G \vdash \text{Norm } s0 \text{ -} [\dot{=} \succ] \text{-} n \rightarrow \text{Norm } s0$$

| *Cons*:
$$\begin{aligned} & \llbracket G \vdash \text{Norm } s0 \text{ -} e \text{ -} \succ v \text{ -} n \rightarrow s1; \\ & \quad G \vdash \quad s1 \text{ -} es \dot{=} \succ vs \text{ -} n \rightarrow s2 \rrbracket \implies \\ & \quad G \vdash \text{Norm } s0 \text{ -} e \# es \dot{=} \succ v \# vs \text{ -} n \rightarrow s2 \end{aligned}$$

— execution of statements

| *Skip*:
$$G \vdash \text{Norm } s \text{ -} \text{Skip} \text{-} n \rightarrow \text{Norm } s$$

| *Expr*:
$$\begin{aligned} & \llbracket G \vdash \text{Norm } s0 \text{ -} e \text{ -} \succ v \text{ -} n \rightarrow s1 \rrbracket \implies \\ & \quad G \vdash \text{Norm } s0 \text{ -} \text{Expr } e \text{-} n \rightarrow s1 \end{aligned}$$

| *Lab*:
$$\begin{aligned} & \llbracket G \vdash \text{Norm } s0 \text{ -} c \text{ -} n \rightarrow s1 \rrbracket \implies \\ & \quad G \vdash \text{Norm } s0 \text{ -} l \cdot c \text{-} n \rightarrow \text{abupd } (\text{absorb } l) s1 \end{aligned}$$

| *Comp*:
$$\begin{aligned} & \llbracket G \vdash \text{Norm } s0 \text{ -} c1 \text{ -} n \rightarrow s1; \\ & \quad G \vdash \quad s1 \text{ -} c2 \text{ -} n \rightarrow s2 \rrbracket \implies \\ & \quad G \vdash \text{Norm } s0 \text{ -} c1;; c2 \text{-} n \rightarrow s2 \end{aligned}$$

| *If*:
$$\begin{aligned} & \llbracket G \vdash \text{Norm } s0 \text{ -} e \text{ -} \succ b \text{-} n \rightarrow s1; \\ & \quad G \vdash \quad s1 \text{ -} (\text{if the-Bool } b \text{ then } c1 \text{ else } c2) \text{-} n \rightarrow s2 \rrbracket \implies \\ & \quad G \vdash \text{Norm } s0 \text{ -} \text{If}(e) c1 \text{ Else } c2 \text{ -} n \rightarrow s2 \end{aligned}$$

| *Loop*:
$$\begin{aligned} & \llbracket G \vdash \text{Norm } s0 \text{ -} e \text{ -} \succ b \text{-} n \rightarrow s1; \\ & \quad \text{if the-Bool } b \\ & \quad \quad \text{then } (G \vdash s1 \text{ -} c \text{-} n \rightarrow s2 \wedge \\ & \quad \quad \quad G \vdash (\text{abupd } (\text{absorb } (\text{Cont } l)) s2) \text{ -} l \cdot \text{While}(e) c \text{-} n \rightarrow s3) \\ & \quad \quad \text{else } s3 = s1 \rrbracket \implies \\ & \quad G \vdash \text{Norm } s0 \text{ -} l \cdot \text{While}(e) c \text{-} n \rightarrow s3 \end{aligned}$$

| *Jmp*:
$$G \vdash \text{Norm } s \text{ -} \text{Jmp } j \text{-} n \rightarrow (\text{Some } (\text{Jump } j), s)$$

| *Throw*:
$$\begin{aligned} & \llbracket G \vdash \text{Norm } s0 \text{ -} e \text{ -} \succ a' \text{-} n \rightarrow s1 \rrbracket \implies \\ & \quad G \vdash \text{Norm } s0 \text{ -} \text{Throw } e \text{-} n \rightarrow \text{abupd } (\text{throw } a') s1 \end{aligned}$$

| *Try*:
$$\begin{aligned} & \llbracket G \vdash \text{Norm } s0 \text{ -} c1 \text{-} n \rightarrow s1; G \vdash s1 \text{ -} \text{salloc} \rightarrow s2; \\ & \quad \text{if } G, s2 \vdash \text{catch } tn \text{ then } G \vdash \text{new-xcpt-var } vn \text{ } s2 \text{ -} c2 \text{-} n \rightarrow s3 \text{ else } s3 = s2 \rrbracket \\ & \implies \\ & \quad G \vdash \text{Norm } s0 \text{ -} \text{Try } c1 \text{ Catch}(tn \text{ } vn) \text{ } c2 \text{-} n \rightarrow s3 \end{aligned}$$

| *Fin*:
$$\begin{aligned} & \llbracket G \vdash \text{Norm } s0 \text{ -} c1 \text{-} n \rightarrow (x1, s1); \\ & \quad G \vdash \text{Norm } s1 \text{ -} c2 \text{-} n \rightarrow s2; \\ & \quad s3 = (\text{if } (\exists \text{ err. } x1 = \text{Some } (\text{Error } \text{err})) \\ & \quad \quad \text{then } (x1, s1) \\ & \quad \quad \text{else } \text{abupd } (\text{abrupt-if } (x1 \neq \text{None}) x1) s2) \rrbracket \implies \\ & \quad G \vdash \text{Norm } s0 \text{ -} c1 \text{ Finally } c2 \text{-} n \rightarrow s3 \end{aligned}$$

| *Init*:
$$\begin{aligned} & \llbracket \text{the } (\text{class } G \text{ } C) = c; \\ & \quad \text{if } \text{inited } C \text{ (globs } s0) \text{ then } s3 = \text{Norm } s0 \\ & \quad \text{else } (G \vdash \text{Norm } (\text{init-class-obj } G \text{ } C \text{ } s0) \\ & \quad \quad \text{-(if } C = \text{Object then Skip else Init (super } c)) \text{-} n \rightarrow s1 \wedge \\ & \quad \quad G \vdash \text{set-lvars empty } s1 \text{ -} \text{init } c \text{-} n \rightarrow s2 \wedge \\ & \quad \quad s3 = \text{restore-lvars } s1 \text{ } s2) \rrbracket \\ & \implies \end{aligned}$$

$$G \vdash \text{Norm } s0 \text{ --Init } C \text{ --}n \rightarrow s3$$

monos

if-bool-eq-conj

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*: $G \vdash s \text{ --}t \succ \text{--}n \rightarrow (v, s')$

inductive-cases *evaln-elim-cases*:

$G \vdash (\text{Some } xc, s) \text{ --}t$	$\succ \text{--}n \rightarrow (v, s')$
$G \vdash \text{Norm } s \text{ --In1r } \text{Skip}$	$\succ \text{--}n \rightarrow (x, s')$
$G \vdash \text{Norm } s \text{ --In1r } (\text{Jmp } j)$	$\succ \text{--}n \rightarrow (x, s')$
$G \vdash \text{Norm } s \text{ --In1r } (l \cdot c)$	$\succ \text{--}n \rightarrow (x, s')$
$G \vdash \text{Norm } s \text{ --In3 } (\llbracket \rrbracket)$	$\succ \text{--}n \rightarrow (v, s')$
$G \vdash \text{Norm } s \text{ --In3 } (e \# es)$	$\succ \text{--}n \rightarrow (v, s')$
$G \vdash \text{Norm } s \text{ --In1l } (\text{Lit } w)$	$\succ \text{--}n \rightarrow (v, s')$
$G \vdash \text{Norm } s \text{ --In1l } (\text{UnOp } unop \ e)$	$\succ \text{--}n \rightarrow (v, s')$
$G \vdash \text{Norm } s \text{ --In1l } (\text{BinOp } binop \ e1 \ e2)$	$\succ \text{--}n \rightarrow (v, s')$
$G \vdash \text{Norm } s \text{ --In2 } (\text{LVar } vn)$	$\succ \text{--}n \rightarrow (v, s')$
$G \vdash \text{Norm } s \text{ --In1l } (\text{Cast } T \ e)$	$\succ \text{--}n \rightarrow (v, s')$
$G \vdash \text{Norm } s \text{ --In1l } (e \ \text{InstOf } T)$	$\succ \text{--}n \rightarrow (v, s')$
$G \vdash \text{Norm } s \text{ --In1l } (\text{Super})$	$\succ \text{--}n \rightarrow (v, s')$
$G \vdash \text{Norm } s \text{ --In1l } (\text{Acc } va)$	$\succ \text{--}n \rightarrow (v, s')$
$G \vdash \text{Norm } s \text{ --In1r } (\text{Expr } e)$	$\succ \text{--}n \rightarrow (x, s')$
$G \vdash \text{Norm } s \text{ --In1r } (c1 ;; c2)$	$\succ \text{--}n \rightarrow (x, s')$
$G \vdash \text{Norm } s \text{ --In1l } (\text{Methd } C \ \text{sig})$	$\succ \text{--}n \rightarrow (x, s')$
$G \vdash \text{Norm } s \text{ --In1l } (\text{Body } D \ c)$	$\succ \text{--}n \rightarrow (x, s')$
$G \vdash \text{Norm } s \text{ --In1l } (e0 \ ? \ e1 : e2)$	$\succ \text{--}n \rightarrow (v, s')$
$G \vdash \text{Norm } s \text{ --In1r } (\text{If } (e) \ c1 \ \text{Else } c2)$	$\succ \text{--}n \rightarrow (x, s')$
$G \vdash \text{Norm } s \text{ --In1r } (l \cdot \text{While}(e) \ c)$	$\succ \text{--}n \rightarrow (x, s')$
$G \vdash \text{Norm } s \text{ --In1r } (c1 \ \text{Finally } c2)$	$\succ \text{--}n \rightarrow (x, s')$
$G \vdash \text{Norm } s \text{ --In1r } (\text{Throw } e)$	$\succ \text{--}n \rightarrow (x, s')$
$G \vdash \text{Norm } s \text{ --In1l } (\text{NewC } C)$	$\succ \text{--}n \rightarrow (v, s')$
$G \vdash \text{Norm } s \text{ --In1l } (\text{New } T[e])$	$\succ \text{--}n \rightarrow (v, s')$
$G \vdash \text{Norm } s \text{ --In1l } (\text{Ass } va \ e)$	$\succ \text{--}n \rightarrow (v, s')$
$G \vdash \text{Norm } s \text{ --In1r } (\text{Try } c1 \ \text{Catch}(tn \ vn) \ c2)$	$\succ \text{--}n \rightarrow (x, s')$
$G \vdash \text{Norm } s \text{ --In2 } (\{\text{accC}, \text{statDeclC}, \text{stat}\} e..fn)$	$\succ \text{--}n \rightarrow (v, s')$
$G \vdash \text{Norm } s \text{ --In2 } (e1.[e2])$	$\succ \text{--}n \rightarrow (v, s')$
$G \vdash \text{Norm } s \text{ --In1l } (\{\text{accC}, \text{statT}, \text{mode}\} e.mn(\{pT\}p))$	$\succ \text{--}n \rightarrow (v, s')$
$G \vdash \text{Norm } s \text{ --In1r } (\text{Init } C)$	$\succ \text{--}n \rightarrow (x, s')$
$G \vdash \text{Norm } s \text{ --In1r } (\text{Init } C)$	$\succ \text{--}n \rightarrow (x, s')$

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*]

$\langle ML \rangle$

lemma *evaln-Inj-elim*: $G \vdash s \text{ --}t \succ \text{--}n \rightarrow (w, s') \implies \text{case } t \text{ of In1 } ec \implies$

$(\text{case } ec \text{ of In1 } e \implies (\exists v. w = \text{In1 } v) \mid \text{Inr } c \implies w = \diamond)$

$\mid \text{In2 } e \implies (\exists v. w = \text{In2 } v) \mid \text{In3 } e \implies (\exists v. w = \text{In3 } v)$

$\langle \text{proof} \rangle$

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*).

lemma *evaln-expr-eq*: $G\vdash s -In1l\ t \succ -n \rightarrow (w, s') = (\exists v. w=In1\ v \wedge G\vdash s -t \succ v -n \rightarrow s')$
 ⟨proof⟩

lemma *evaln-var-eq*: $G\vdash s -In2\ t \succ -n \rightarrow (w, s') = (\exists vf. w=In2\ vf \wedge G\vdash s -t \succ vf -n \rightarrow s')$
 ⟨proof⟩

lemma *evaln-exprs-eq*: $G\vdash s -In3\ t \succ -n \rightarrow (w, s') = (\exists vs. w=In3\ vs \wedge G\vdash s -t \succ vs -n \rightarrow s')$
 ⟨proof⟩

lemma *evaln-stmt-eq*: $G\vdash s -In1r\ t \succ -n \rightarrow (w, s') = (w=\diamond \wedge G\vdash s -t -n \rightarrow s')$
 ⟨proof⟩

⟨ML⟩

declare *evaln-AbruptIs* [intro!]

lemma *evaln-Callee*: $G\vdash Norm\ s -In1l\ (Callee\ l\ e) \succ -n \rightarrow (v, s') = False$
 ⟨proof⟩

lemma *evaln-InsInitE*: $G\vdash Norm\ s -In1l\ (InsInitE\ c\ e) \succ -n \rightarrow (v, s') = False$
 ⟨proof⟩

lemma *evaln-InsInitV*: $G\vdash Norm\ s -In2\ (InsInitV\ c\ w) \succ -n \rightarrow (v, s') = False$
 ⟨proof⟩

lemma *evaln-FinA*: $G\vdash Norm\ s -In1r\ (FinA\ a\ c) \succ -n \rightarrow (v, s') = False$
 ⟨proof⟩

lemma *evaln-abrupt-lemma*: $G\vdash s -e \succ -n \rightarrow (v, s') \implies$
 $fst\ s = Some\ xc \implies s' = s \wedge v = arbitrary3\ e$
 ⟨proof⟩

lemma *evaln-abrupt*:

$\bigwedge s'. G\vdash (Some\ xc, s) -e \succ -n \rightarrow (w, s') = (s' = (Some\ xc, s) \wedge$
 $w = arbitrary3\ e \wedge G\vdash (Some\ xc, s) -e \succ -n \rightarrow (arbitrary3\ e, (Some\ xc, s)))$
 ⟨proof⟩

⟨ML⟩

lemma *evaln-LitI*: $G\vdash s -Lit\ v \succ (if\ normal\ s\ then\ v\ else\ arbitrary) -n \rightarrow s$
 ⟨proof⟩

lemma *CondI*:

$\bigwedge s1. \llbracket G\vdash s -e \succ b -n \rightarrow s1; G\vdash s1 - (if\ the\ Bool\ b\ then\ e1\ else\ e2) \succ v -n \rightarrow s2 \rrbracket \implies$
 $G\vdash s -e\ ?\ e1 : e2 \succ (if\ normal\ s1\ then\ v\ else\ arbitrary) -n \rightarrow s2$

$\langle \text{proof} \rangle$

lemma *evaln-SkipI* [*intro!*]: $G \vdash s \text{ -Skip-} n \rightarrow s$

$\langle \text{proof} \rangle$

lemma *evaln-ExprI*: $G \vdash s \text{ -e-} \succ v \text{ -n} \rightarrow s' \implies G \vdash s \text{ -Expr } e \text{ -n} \rightarrow s'$

$\langle \text{proof} \rangle$

lemma *evaln-CompI*: $\llbracket G \vdash s \text{ -c1-} n \rightarrow s1; G \vdash s1 \text{ -c2-} n \rightarrow s2 \rrbracket \implies G \vdash s \text{ -c1;; c2-} n \rightarrow s2$

$\langle \text{proof} \rangle$

lemma *evaln-IfI*:

$\llbracket G \vdash s \text{ -e-} \succ v \text{ -n} \rightarrow s1; G \vdash s1 \text{ -(if the-Bool } v \text{ then } c1 \text{ else } c2)\text{-n} \rightarrow s2 \rrbracket \implies$
 $G \vdash s \text{ -If}(e) \text{ } c1 \text{ Else } c2 \text{ -n} \rightarrow s2$

$\langle \text{proof} \rangle$

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*: $G \vdash s0 \text{ -t-} \succ \text{ -n} \rightarrow (v, s1)$

shows $G \vdash s0 \text{ -t-} \succ \rightarrow (v, s1)$

$\langle \text{proof} \rangle$

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*]:

$G \vdash s \text{ -t-} \succ \text{ -n} \rightarrow (w, s') \implies \forall m. n \leq m \longrightarrow G \vdash s \text{ -t-} \succ \text{ -m} \rightarrow (w, s')$

$\langle \text{proof} \rangle$

lemmas *evaln-nonstrict-Suc = evaln-nonstrict* [*OF - le-refl [THEN le-SucI]*]

lemma *evaln-max2*: $\llbracket G \vdash s1 \text{ -t1-} \succ \text{ -n1} \rightarrow (w1, s1'); G \vdash s2 \text{ -t2-} \succ \text{ -n2} \rightarrow (w2, s2') \rrbracket \implies$

$G \vdash s1 \text{ -t1-} \succ \text{ -max } n1 \text{ } n2 \rightarrow (w1, s1') \wedge G \vdash s2 \text{ -t2-} \succ \text{ -max } n1 \text{ } n2 \rightarrow (w2, s2')$

$\langle \text{proof} \rangle$

corollary *evaln-max2E* [*consumes 2*]:

$\llbracket G \vdash s1 \text{ -t1-} \succ \text{ -n1} \rightarrow (w1, s1'); G \vdash s2 \text{ -t2-} \succ \text{ -n2} \rightarrow (w2, s2') \rrbracket$

$\llbracket G \vdash s1 \text{ -t1-} \succ \text{ -max } n1 \text{ } n2 \rightarrow (w1, s1'); G \vdash s2 \text{ -t2-} \succ \text{ -max } n1 \text{ } n2 \rightarrow (w2, s2') \rrbracket \implies P \rrbracket \implies P$

$\langle \text{proof} \rangle$

lemma *evaln-max3*:

$$\begin{aligned} & \llbracket G \vdash s1 -t1 \succ -n1 \rightarrow (w1, s1'); G \vdash s2 -t2 \succ -n2 \rightarrow (w2, s2'); G \vdash s3 -t3 \succ -n3 \rightarrow (w3, s3') \rrbracket \implies \\ & G \vdash s1 -t1 \succ -\max (\max n1 n2) n3 \rightarrow (w1, s1') \wedge \\ & G \vdash s2 -t2 \succ -\max (\max n1 n2) n3 \rightarrow (w2, s2') \wedge \\ & G \vdash s3 -t3 \succ -\max (\max n1 n2) n3 \rightarrow (w3, s3') \\ & \langle \text{proof} \rangle \end{aligned}$$

corollary *evaln-max3E*:

$$\begin{aligned} & \llbracket G \vdash s1 -t1 \succ -n1 \rightarrow (w1, s1'); G \vdash s2 -t2 \succ -n2 \rightarrow (w2, s2'); G \vdash s3 -t3 \succ -n3 \rightarrow (w3, s3'); \\ & \quad \llbracket G \vdash s1 -t1 \succ -\max (\max n1 n2) n3 \rightarrow (w1, s1'); \\ & \quad G \vdash s2 -t2 \succ -\max (\max n1 n2) n3 \rightarrow (w2, s2'); \\ & \quad G \vdash s3 -t3 \succ -\max (\max n1 n2) n3 \rightarrow (w3, s3') \\ & \quad \rrbracket \implies P \\ & \quad \rrbracket \implies P \\ & \langle \text{proof} \rangle \end{aligned}$$

lemma *le-max3I1*: $(n2::nat) \leq \max n1 (\max n2 n3)$
 $\langle \text{proof} \rangle$

lemma *le-max3I2*: $(n3::nat) \leq \max n1 (\max n2 n3)$
 $\langle \text{proof} \rangle$

declare $[[\text{simproc del: wt-expr wt-var wt-exprs wt-stmt}]]$

eval implies evaln

lemma *eval-evaln*:

assumes *eval*: $G \vdash s0 -t \succ \rightarrow (v, s1)$
shows $\exists n. G \vdash s0 -t \succ -n \rightarrow (v, s1)$
 $\langle \text{proof} \rangle$

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*. $\llbracket v = (Lit\ a)..*i* \rrbracket \Longrightarrow P$

shows *P*

<proof>

constdefs *groundExprs*:: *expr* *list* \Rightarrow *bool*
groundExprs *es* \equiv *list-all* (λ *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*

<proof>

declare *the-var-FVar-def* [*simp* *del*]

lemma *the-var-AVar-simp*:

the-var $G\ s\ ((Lit\ a).[Lit\ i]) = avar\ G\ i\ a\ s$
 ⟨proof⟩

declare *the-var-AVar-def* [*simp del*]

syntax (*xsymbols*)

Ref :: *loc* ⇒ *expr*

SKIP :: *expr*

translations

Ref a == *Lit (Addr a)*

SKIP == *Lit Unit*

inductive

step :: [*prog, term* × *state, term* × *state*] ⇒ *bool* (|-| → 1 -[61,82,82] 81)

for *G* :: *prog*

where

Abrupt:
 ⌊⌊∀ *v. t* ≠ ⟨*Lit v*⟩;
 ∀ *t. t* ≠ ⟨*l* · *Skip*⟩;
 ∀ *C vn c. t* ≠ ⟨*Try Skip Catch (C vn) c*⟩;
 ∀ *x c. t* ≠ ⟨*Skip Finally c*⟩ ∧ *xc* ≠ *Xcpt x*;
 ∀ *a c. t* ≠ ⟨*FinA a c*⟩⌋
 ⇒
 $G \vdash (t, \text{Some } xc, s) \mapsto 1 \langle \langle \text{Lit arbitrary} \rangle, \text{Some } xc, s \rangle$

| *InsInitE*: ⌊ $G \vdash (\langle c \rangle, \text{Norm } s) \mapsto 1 \langle \langle c \rangle, s \rangle$ ⌋
 ⇒
 $G \vdash (\langle \text{InsInitE } c\ e \rangle, \text{Norm } s) \mapsto 1 \langle \langle \text{InsInitE } c'\ e \rangle, s \rangle$

| *NewC*: $G \vdash (\langle \text{NewC } C \rangle, \text{Norm } s) \mapsto 1 \langle \langle \text{InsInitE (Init } C \rangle (\text{NewC } C) \rangle, \text{Norm } s \rangle$
 | *NewCInitE*: ⌊ $G \vdash \text{Norm } s \text{ -halloc } (\text{CInst } C) \succ a \rightarrow s \uparrow$ ⌋
 ⇒
 $G \vdash (\langle \text{InsInitE Skip (NewC } C) \rangle, \text{Norm } s) \mapsto 1 \langle \langle \text{Ref } a \rangle, s \rangle$

| *NewA*:
 $G \vdash (\langle \text{New } T[e] \rangle, \text{Norm } s) \mapsto 1 \langle \langle \text{InsInitE (init-comp-ty } T \rangle (\text{New } T[e]) \rangle, \text{Norm } s \rangle$
 | *InsInitNewAIdx*:
 ⌊ $G \vdash (\langle e \rangle, \text{Norm } s) \mapsto 1 \langle \langle e \rangle, s \rangle$ ⌋
 ⇒
 $G \vdash (\langle \text{InsInitE Skip (New } T[e]) \rangle, \text{Norm } s) \mapsto 1 \langle \langle \text{InsInitE Skip (New } T[e]) \rangle, s \rangle$
 | *InsInitNewA*:
 ⌊ $G \vdash \text{abupd (check-neg } i) (\text{Norm } s) \text{ -halloc } (\text{Arr } T (\text{the-Intg } i)) \succ a \rightarrow s \uparrow$ ⌋
 ⇒
 $G \vdash (\langle \text{InsInitE Skip (New } T[\text{Lit } i]) \rangle, \text{Norm } s) \mapsto 1 \langle \langle \text{Ref } a \rangle, s \rangle$

| *CastE*:
 ⌊ $G \vdash (\langle e \rangle, \text{Norm } s) \mapsto 1 \langle \langle e \rangle, s \rangle$ ⌋
 ⇒

- $$G\vdash(\langle\text{Cast } T \ e\rangle, \text{None}, s) \mapsto 1 (\langle\text{Cast } T \ e'\rangle, s')$$
- | *Cast*: $\llbracket s' = \text{abupd } (\text{raise-if } (\neg G, s\vdash v \text{ fits } T) \ \text{ClassCast}) \ (\text{Norm } s) \rrbracket$
 \implies
 $G\vdash(\langle\text{Cast } T \ (\text{Lit } v)\rangle, \text{Norm } s) \mapsto 1 (\langle\text{Lit } v\rangle, s')$
- | *InstE*: $\llbracket G\vdash(\langle e\rangle, \text{Norm } s) \mapsto 1 (\langle e'::\text{expr}\rangle, s') \rrbracket$
 \implies
 $G\vdash(\langle e \ \text{InstOf } T\rangle, \text{Norm } s) \mapsto 1 (\langle e'\rangle, s')$
- | *Inst*: $\llbracket b = (v \neq \text{Null} \wedge G, s\vdash v \text{ fits RefT } T) \rrbracket$
 \implies
 $G\vdash(\langle(\text{Lit } v) \ \text{InstOf } T\rangle, \text{Norm } s) \mapsto 1 (\langle\text{Lit } (\text{Bool } b)\rangle, s')$
- | *UnOpE*: $\llbracket G\vdash(\langle e\rangle, \text{Norm } s) \mapsto 1 (\langle e'\rangle, s') \rrbracket$
 \implies
 $G\vdash(\langle\text{UnOp } \text{unop } e\rangle, \text{Norm } s) \mapsto 1 (\langle\text{UnOp } \text{unop } e'\rangle, s')$
- | *UnOp*: $G\vdash(\langle\text{UnOp } \text{unop } (\text{Lit } v)\rangle, \text{Norm } s) \mapsto 1 (\langle\text{Lit } (\text{eval-unop } \text{unop } v)\rangle, \text{Norm } s)$
- | *BinOpE1*: $\llbracket G\vdash(\langle e1\rangle, \text{Norm } s) \mapsto 1 (\langle e1'\rangle, s') \rrbracket$
 \implies
 $G\vdash(\langle\text{BinOp } \text{binop } e1 \ e2\rangle, \text{Norm } s) \mapsto 1 (\langle\text{BinOp } \text{binop } e1' \ e2\rangle, s')$
- | *BinOpE2*: $\llbracket \text{need-second-arg } \text{binop } v1; G\vdash(\langle e2\rangle, \text{Norm } s) \mapsto 1 (\langle e2'\rangle, s') \rrbracket$
 \implies
 $G\vdash(\langle\text{BinOp } \text{binop } (\text{Lit } v1) \ e2\rangle, \text{Norm } s) \mapsto 1 (\langle\text{BinOp } \text{binop } (\text{Lit } v1) \ e2'\rangle, s')$
- | *BinOpTerm*: $\llbracket \neg \text{need-second-arg } \text{binop } v1 \rrbracket$
 \implies
 $G\vdash(\langle\text{BinOp } \text{binop } (\text{Lit } v1) \ e2\rangle, \text{Norm } s) \mapsto 1 (\langle\text{Lit } v1\rangle, \text{Norm } s)$
- | *BinOp*: $G\vdash(\langle\text{BinOp } \text{binop } (\text{Lit } v1) \ (\text{Lit } v2)\rangle, \text{Norm } s) \mapsto 1 (\langle\text{Lit } (\text{eval-binop } \text{binop } v1 \ v2)\rangle, \text{Norm } s)$
- | *Super*: $G\vdash(\langle\text{Super}\rangle, \text{Norm } s) \mapsto 1 (\langle\text{Lit } (\text{val-this } s)\rangle, \text{Norm } s)$
- | *AccVA*: $\llbracket G\vdash(\langle va\rangle, \text{Norm } s) \mapsto 1 (\langle va'\rangle, s') \rrbracket$
 \implies
 $G\vdash(\langle\text{Acc } va\rangle, \text{Norm } s) \mapsto 1 (\langle\text{Acc } va'\rangle, s')$
- | *Acc*: $\llbracket \text{groundVar } va; ((v, vf), s') = \text{the-var } G \ (\text{Norm } s) \ va \rrbracket$
 \implies
 $G\vdash(\langle\text{Acc } va\rangle, \text{Norm } s) \mapsto 1 (\langle\text{Lit } v\rangle, s')$
- | *AssVA*: $\llbracket G\vdash(\langle va\rangle, \text{Norm } s) \mapsto 1 (\langle va'\rangle, s') \rrbracket$
 \implies
 $G\vdash(\langle va:=e\rangle, \text{Norm } s) \mapsto 1 (\langle va':=e\rangle, s')$
- | *AssE*: $\llbracket \text{groundVar } va; G\vdash(\langle e\rangle, \text{Norm } s) \mapsto 1 (\langle e'\rangle, s') \rrbracket$
 \implies
 $G\vdash(\langle va:=e\rangle, \text{Norm } s) \mapsto 1 (\langle va:=e'\rangle, s')$
- | *Ass*: $\llbracket \text{groundVar } va; ((w, f), s') = \text{the-var } G \ (\text{Norm } s) \ va \rrbracket$
 \implies
 $G\vdash(\langle va:=(\text{Lit } v)\rangle, \text{Norm } s) \mapsto 1 (\langle\text{Lit } v, \text{assign } f \ v \ s'\rangle)$
- | *CondC*: $\llbracket G\vdash(\langle e0\rangle, \text{Norm } s) \mapsto 1 (\langle e0'\rangle, s') \rrbracket$

$$\begin{array}{l}
\implies \\
G\vdash(\langle e0? e1:e2 \rangle, Norm\ s) \mapsto 1 (\langle e0'? e1:e2 \rangle, s') \\
| \text{Cond: } G\vdash(\langle Lit\ b? e1:e2 \rangle, Norm\ s) \mapsto 1 (\langle if\ the\text{-}Bool\ b\ then\ e1\ else\ e2 \rangle, Norm\ s) \\
\\
| \text{CallTarget: } \llbracket G\vdash(\langle e \rangle, Norm\ s) \mapsto 1 (\langle e' \rangle, s') \rrbracket \\
\implies \\
G\vdash(\langle \{accC, statT, mode\} e \cdot mn(\{pTs\} args) \rangle, Norm\ s) \\
\mapsto 1 (\langle \{accC, statT, mode\} e' \cdot mn(\{pTs\} args) \rangle, s') \\
| \text{CallArgs: } \llbracket G\vdash(\langle args \rangle, Norm\ s) \mapsto 1 (\langle args' \rangle, s') \rrbracket \\
\implies \\
G\vdash(\langle \{accC, statT, mode\} Lit\ a \cdot mn(\{pTs\} args) \rangle, Norm\ s) \\
\mapsto 1 (\langle \{accC, statT, mode\} Lit\ a \cdot mn(\{pTs\} args') \rangle, s') \\
| \text{Call: } \llbracket groundExprs\ args; vs = map\ the\text{-}val\ args; \\
D = invocation\text{-}declclass\ G\ mode\ s\ a\ statT\ (\llbracket name=mn, parTs=pTs \rrbracket); \\
s' = init\text{-}lvars\ G\ D\ (\llbracket name=mn, parTs=pTs \rrbracket)\ mode\ a'\ vs\ (Norm\ s) \rrbracket \\
\implies \\
G\vdash(\langle \{accC, statT, mode\} Lit\ a \cdot mn(\{pTs\} args) \rangle, Norm\ s) \\
\mapsto 1 (\langle Callee\ (locals\ s)\ (Methd\ D\ (\llbracket name=mn, parTs=pTs \rrbracket)) \rangle, s') \\
\\
| \text{Callee: } \llbracket G\vdash(\langle e \rangle, Norm\ s) \mapsto 1 (\langle e'::expr \rangle, s') \rrbracket \\
\implies \\
G\vdash(\langle Callee\ lcls\text{-}caller\ e \rangle, Norm\ s) \mapsto 1 (\langle e' \rangle, s') \\
\\
| \text{CalleeRet: } G\vdash(\langle Callee\ lcls\text{-}caller\ (Lit\ v) \rangle, Norm\ s) \\
\mapsto 1 (\langle Lit\ v \rangle, (set\text{-}lvars\ lcls\text{-}caller\ (Norm\ s))) \\
\\
| \text{Methd: } G\vdash(\langle Methd\ D\ sig \rangle, Norm\ s) \mapsto 1 (\langle body\ G\ D\ sig \rangle, Norm\ s) \\
\\
| \text{Body: } G\vdash(\langle Body\ D\ c \rangle, Norm\ s) \mapsto 1 (\langle InsInitE\ (Init\ D)\ (Body\ D\ c) \rangle, Norm\ s) \\
\\
| \text{InsInitBody:} \\
\llbracket G\vdash(\langle c \rangle, Norm\ s) \mapsto 1 (\langle c' \rangle, s') \rrbracket \\
\implies \\
G\vdash(\langle InsInitE\ Skip\ (Body\ D\ c) \rangle, Norm\ s) \mapsto 1 (\langle InsInitE\ Skip\ (Body\ D\ c') \rangle, s') \\
| \text{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)) \\
\\
| \text{FVar: } \llbracket \neg\ initied\ statDeclC\ (globs\ s) \rrbracket \\
\implies \\
G\vdash(\langle \{accC, statDeclC, stat\} e \cdot fn \rangle, Norm\ s) \\
\mapsto 1 (\langle InsInitV\ (Init\ statDeclC)\ (\{accC, statDeclC, stat\} e \cdot fn) \rangle, Norm\ s) \\
| \text{InsInitFVarE:} \\
\llbracket G\vdash(\langle e \rangle, Norm\ s) \mapsto 1 (\langle e' \rangle, s') \rrbracket \\
\implies \\
G\vdash(\langle InsInitV\ Skip\ (\{accC, statDeclC, stat\} e \cdot fn) \rangle, Norm\ s) \\
\mapsto 1 (\langle InsInitV\ Skip\ (\{accC, statDeclC, stat\} e' \cdot fn) \rangle, s') \\
| \text{InsInitFVar:} \\
G\vdash(\langle InsInitV\ Skip\ (\{accC, statDeclC, stat\} Lit\ a \cdot fn) \rangle, Norm\ s) \\
\mapsto 1 (\langle \{accC, statDeclC, stat\} Lit\ a \cdot fn \rangle, Norm\ s)
\end{array}$$

— 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*.

$$\begin{aligned} | \text{AVarE1}: & \llbracket G\vdash(\langle e1 \rangle, \text{Norm } s) \mapsto 1 (\langle e1' \rangle, s') \rrbracket \\ & \implies \\ & G\vdash(\langle e1.[e2] \rangle, \text{Norm } s) \mapsto 1 (\langle e1'.[e2] \rangle, s') \end{aligned}$$

$$\begin{aligned} | \text{AVarE2}: & G\vdash(\langle e2 \rangle, \text{Norm } s) \mapsto 1 (\langle e2' \rangle, s') \\ & \implies \\ & G\vdash(\langle \text{Lit } a.[e2] \rangle, \text{Norm } s) \mapsto 1 (\langle \text{Lit } a.[e2'] \rangle, s') \end{aligned}$$

— *Nil* is fully evaluated

$$\begin{aligned} | \text{ConsHd}: & \llbracket G\vdash(\langle e::\text{expr} \rangle, \text{Norm } s) \mapsto 1 (\langle e'::\text{expr} \rangle, s') \rrbracket \\ & \implies \\ & G\vdash(\langle e\#es \rangle, \text{Norm } s) \mapsto 1 (\langle e'\#es \rangle, s') \end{aligned}$$

$$\begin{aligned} | \text{ConsTl}: & \llbracket G\vdash(\langle es \rangle, \text{Norm } s) \mapsto 1 (\langle es' \rangle, s') \rrbracket \\ & \implies \\ & G\vdash(\langle (\text{Lit } v)\#es \rangle, \text{Norm } s) \mapsto 1 (\langle (\text{Lit } v)\#es' \rangle, s') \end{aligned}$$

$$| \text{Skip}: G\vdash(\langle \text{Skip} \rangle, \text{Norm } s) \mapsto 1 (\langle \text{SKIP} \rangle, \text{Norm } s)$$

$$\begin{aligned} | \text{ExprE}: & \llbracket G\vdash(\langle e \rangle, \text{Norm } s) \mapsto 1 (\langle e' \rangle, s') \rrbracket \\ & \implies \\ & G\vdash(\langle \text{Expr } e \rangle, \text{Norm } s) \mapsto 1 (\langle \text{Expr } e' \rangle, s') \\ | \text{Expr}: & G\vdash(\langle \text{Expr } (\text{Lit } v) \rangle, \text{Norm } s) \mapsto 1 (\langle \text{Skip} \rangle, \text{Norm } s) \end{aligned}$$

$$\begin{aligned} | \text{LabC}: & \llbracket G\vdash(\langle c \rangle, \text{Norm } s) \mapsto 1 (\langle c' \rangle, s') \rrbracket \\ & \implies \\ & G\vdash(\langle l \cdot c \rangle, \text{Norm } s) \mapsto 1 (\langle l \cdot c' \rangle, s') \\ | \text{Lab}: & G\vdash(\langle l \cdot \text{Skip} \rangle, s) \mapsto 1 (\langle \text{Skip} \rangle, \text{abupd } (\text{absorb } l) s) \end{aligned}$$

$$\begin{aligned} | \text{CompC1}: & \llbracket G\vdash(\langle c1 \rangle, \text{Norm } s) \mapsto 1 (\langle c1' \rangle, s') \rrbracket \\ & \implies \\ & G\vdash(\langle c1;; c2 \rangle, \text{Norm } s) \mapsto 1 (\langle c1';; c2 \rangle, s') \end{aligned}$$

$$| \text{Comp}: G\vdash(\langle \text{Skip};; c2 \rangle, \text{Norm } s) \mapsto 1 (\langle c2 \rangle, \text{Norm } s)$$

$$\begin{aligned} | \text{IfE}: & \llbracket G\vdash(\langle e \rangle, \text{Norm } s) \mapsto 1 (\langle e' \rangle, s') \rrbracket \\ & \implies \\ & G\vdash(\langle \text{If } (e) \text{ } s1 \text{ Else } s2 \rangle, \text{Norm } s) \mapsto 1 (\langle \text{If } (e') \text{ } s1 \text{ Else } s2 \rangle, s') \\ | \text{If}: & G\vdash(\langle \text{If } (\text{Lit } v) \text{ } s1 \text{ Else } s2 \rangle, \text{Norm } s) \\ & \mapsto 1 (\langle \text{if the-Bool } v \text{ then } s1 \text{ else } s2 \rangle, \text{Norm } s) \end{aligned}$$

$$\begin{aligned} | \text{Loop}: & G\vdash(\langle l \cdot \text{While}(e) \text{ } c \rangle, \text{Norm } s) \\ & \mapsto 1 (\langle \text{If } (e) \text{ } (\text{Cont } l \cdot c;; l \cdot \text{While}(e) \text{ } c) \text{ Else Skip} \rangle, \text{Norm } s) \end{aligned}$$

$$| \text{Jmp}: G\vdash(\langle \text{Jmp } j \rangle, \text{Norm } s) \mapsto 1 (\langle \text{Skip} \rangle, (\text{Some } (\text{Jump } j), s))$$

$$| \text{ThrowE}: \llbracket G\vdash(\langle e \rangle, \text{Norm } s) \mapsto 1 (\langle e' \rangle, s') \rrbracket$$

$$\begin{aligned} & \implies \\ & G\vdash(\langle \text{Throw } e \rangle, \text{Norm } s) \mapsto 1 (\langle \text{Throw } e' \rangle, s') \\ | \text{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)) \\ | \text{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 \text{ vn}) c2 \rangle, \text{Norm } s) \mapsto 1 (\langle \text{Try } c1' \text{ Catch}(C \text{ vn}) c2 \rangle, s') \\ | \text{Try: } & \llbracket G\vdash s \text{ -salloc} \rightarrow s' \rrbracket \\ & \implies \\ & G\vdash(\langle \text{Try Skip Catch}(C \text{ vn}) c2 \rangle, s) \\ & \mapsto 1 (\text{if } G, s \vdash \text{catch } C \text{ then } (\langle c2 \rangle, \text{new-xcpt-var } \text{vn } s') \\ & \quad \text{else } (\langle \text{Skip} \rangle, s')) \\ | \text{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') \\ | \text{Fin: } & G\vdash(\langle \text{Skip Finally } c2 \rangle, (a, s)) \mapsto 1 (\langle \text{FinA } a \text{ } c2 \rangle, \text{Norm } s) \\ | \text{FinAC: } & \llbracket G\vdash(\langle c \rangle, s) \mapsto 1 (\langle c' \rangle, s') \rrbracket \\ & \implies \\ & G\vdash(\langle \text{FinA } a \text{ } c \rangle, s) \mapsto 1 (\langle \text{FinA } a \text{ } c' \rangle, s') \\ | \text{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) \\ \\ | \text{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) \\ | \text{Init: } & \llbracket \text{the } (\text{class } G \text{ } 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 then Skip else } (\text{Init } (\text{super } c))) \rangle; \\ & \quad \text{Expr } (\text{Callee } (\text{locals } s) (\text{InsInitE } (\text{init } c) \text{ SKIP})) \rangle \\ & \quad , \text{Norm } (\text{init-class-obj } G \text{ } C \text{ } s)) \\ - \text{InsInitE is just used as trick to embed the statement } \text{init } c \text{ into an expression} \\ | \text{InsInitESKIP:} & \\ & G\vdash(\langle \text{InsInitE Skip SKIP} \rangle, \text{Norm } s) \mapsto 1 (\langle \text{SKIP} \rangle, \text{Norm } s) \end{aligned}$$
abbreviation

$$\begin{aligned} \text{stepn: } & [\text{prog}, \text{term} \times \text{state}, \text{nat}, \text{term} \times \text{state}] \Rightarrow \text{bool } (\vdash \mapsto - [61, 82, 82] \text{ } 81) \\ \text{where } & G\vdash p \mapsto n \text{ } p' \equiv (p, p') \in \{(x, y). \text{step } G \text{ } x \text{ } y\}^n \end{aligned}$$
abbreviation

$$\begin{aligned} \text{steptr: } & [\text{prog}, \text{term} \times \text{state}, \text{term} \times \text{state}] \Rightarrow \text{bool } (\vdash \mapsto * - [61, 82, 82] \text{ } 81) \\ \text{where } & G\vdash p \mapsto * \text{ } p' \equiv (p, p') \in \{(x, y). \text{step } G \text{ } x \text{ } y\}^* \end{aligned}$$

lemma *rtrancel-imp-rel-pow*: $p \in R^{\wedge *} \implies \exists n. p \in R^{\wedge n}$

<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* $::$ [*ptrn*] \Rightarrow *ptrn* (*Val*:- [951] 950)

-*Var* $::$ [*ptrn*] \Rightarrow *ptrn* (*Var*:- [951] 950)

-*Vals* $::$ [*ptrn*] \Rightarrow *ptrn* (*Vals*:- [951] 950)

translations

λ *Val*:*v* . *b* $==$ (λ *v* . *b*) \circ *the-In1*

λ *Var*:*v* . *b* $==$ (λ *v* . *b*) \circ *the-In2*

λ *Vals*:*v* . *b* $==$ (λ *v* . *b*) \circ *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*]: $peek\text{-}and\ P\ p\ Y\ s = (\lambda Z. (P\ Y\ s\ Z \wedge p\ s))$
 ⟨*proof*⟩

lemma *peek-and-Not* [*simp*]: $(P \wedge. (\lambda s. \neg f s)) = (P \wedge. Not \circ f)$
 ⟨*proof*⟩

lemma *peek-and-and* [*simp*]: $peek\text{-}and\ (peek\text{-}and\ P\ p)\ p = peek\text{-}and\ P\ p$
 ⟨*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*]: $peek\text{-}and\ (Normal\ P)\ p = Normal\ (peek\text{-}and\ P\ p)$
 ⟨*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*]: $assn\text{-}supd\ P\ f\ Y\ s'\ Z = (\exists s. P Y s Z \wedge s' = f s)$
 ⟨*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*]: $((f ;. Q) ;. f) Y s Z \Longrightarrow Q Y s Z$

<proof>

lemma *supd-assn-supdI* [*elim*]: $Q\ Y\ s\ Z \implies (f\ .; (Q\ ;. f))\ Y\ s\ Z$

<proof>

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*]: $(P \leftarrow w)\ Y = P\ w$

<proof>

lemma *subst-subst-res* [*simp*]: $P \leftarrow w \leftarrow v = P \leftarrow w$

<proof>

lemma *peek-and-subst-res* [*simp*]: $(P \wedge. p) \leftarrow w = (P \leftarrow w \wedge. p)$

<proof>

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*]:

$(P \leftarrow = b)\ Y\ s\ Z = (\exists v. P\ (Val\ v)\ s\ Z \wedge (normal\ s \longrightarrow the-Bool\ v=b))$

<proof>

lemma *subst-Bool-the-BoolI*: $P\ (Val\ b)\ s\ Z \implies (P \leftarrow = the-Bool\ b)\ Y\ s\ Z$

<proof>

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* $:: pptrn \Rightarrow 'a\ assn \Rightarrow 'a\ assn$ (λ :-. - [0,3] 3)

translations

$\lambda w. P == peek-res\ (\lambda w. P)$

lemma *peek-res-def2* [*simp*]: $peek-res\ P\ Y = P\ Y\ Y$

<proof>

lemma *peek-res-subst-res* [*simp*]: $peek-res\ P \leftarrow w = P\ w \leftarrow w$

<proof>

lemma *peek-subst-res-allI*:

$(\bigwedge a. T a (P (f a) \leftarrow f a)) \implies \forall a. T a (peek\text{-res } P \leftarrow f a)$
 ⟨proof⟩

57 ign-res

constdefs

ign-res :: 'a assn \Rightarrow 'a assn (-↓ [1000] 1000)
 $P \downarrow \equiv \lambda Y s Z. \exists Y. P Y s Z$

lemma *ign-res-def2* [simp]: $P \downarrow Y s Z = (\exists Y. P Y s Z)$
 ⟨proof⟩

lemma *ign-ign-res* [simp]: $P \downarrow \downarrow = P \downarrow$
 ⟨proof⟩

lemma *ign-subst-res* [simp]: $P \downarrow \leftarrow w = P \downarrow$
 ⟨proof⟩

lemma *peek-and-ign-res* [simp]: $(P \wedge p) \downarrow = (P \downarrow \wedge p)$
 ⟨proof⟩

58 peek-st

constdefs

peek-st :: (st \Rightarrow 'a assn) \Rightarrow 'a assn
 $peek\text{-st } P \equiv \lambda Y s. P (store s) Y s$

syntax

@*peek-st* :: pptrn \Rightarrow 'a assn \Rightarrow 'a assn ($\lambda \dots - [0,3] 3$)

translations

$\lambda s.. P == peek\text{-st } (\lambda s. P)$

lemma *peek-st-def2* [simp]: $(\lambda s.. P f s) Y s = P f (store s) Y s$
 ⟨proof⟩

lemma *peek-st-triv* [simp]: $(\lambda s.. P) = P$
 ⟨proof⟩

lemma *peek-st-st* [simp]: $(\lambda s.. \lambda s'. P s s') = (\lambda s.. P s s)$
 ⟨proof⟩

lemma *peek-st-split* [simp]: $(\lambda s.. \lambda Y s'. P s Y s') = (\lambda Y s. P (store s) Y s)$
 ⟨proof⟩

lemma *peek-st-subst-res* [simp]: $(\lambda s.. P s) \leftarrow w = (\lambda s.. P s \leftarrow w)$
 ⟨proof⟩

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⟩

$ax\text{-valids} :: prog \Rightarrow 'b\ triples \Rightarrow 'a\ triples \Rightarrow bool$
 $(-,|-|- [61,58,58] 57)$

syntax

$triples\text{-valid} :: prog \Rightarrow nat \Rightarrow 'a\ triples \Rightarrow bool$
 $(-|-|- [61,0, 58] 57)$
 $ax\text{-valid} :: prog \Rightarrow 'b\ triples \Rightarrow 'a\ triple \Rightarrow bool$
 $(-,|-|- [61,58,58] 57)$

syntax (*xsymbols*)

$triples\text{-valid} :: prog \Rightarrow nat \Rightarrow 'a\ triples \Rightarrow bool$
 $(-|-|- [61,0, 58] 57)$
 $ax\text{-valid} :: prog \Rightarrow 'b\ triples \Rightarrow 'a\ triple \Rightarrow bool$
 $(-,|-|- [61,58,58] 57)$

defs $triple\text{-valid}\text{-def}: G \models n:t \equiv \text{case } t \text{ of } \{P\} t \triangleright \{Q\} \Rightarrow$
 $\forall Y\ s\ Z. P\ Y\ s\ Z \longrightarrow \text{type-ok } G\ t\ s \longrightarrow$
 $(\forall Y'\ s'. G \vdash s -t \triangleright -n \longrightarrow (Y',s') \longrightarrow Q\ Y'\ s'\ Z)$

translations $G \models n:ts \equiv \text{Ball } ts\ (\text{triple-valid } G\ n)$
defs $ax\text{-valids}\text{-def}: G, A \models ts \equiv \forall n. G \models n:A \longrightarrow G \models n:ts$
translations $G, A \models t \equiv G, A \models \{t\}$

lemma $triple\text{-valid}\text{-def}2: G \models n:\{P\} t \triangleright \{Q\} =$
 $(\forall Y\ s\ Z. P\ Y\ s\ Z$
 $\longrightarrow (\exists L. (\text{normal } s \longrightarrow (\exists C\ T\ A. (\text{prg}=G, \text{cls}=C, \text{lcl}=L) \vdash t :: T \wedge$
 $(\text{prg}=G, \text{cls}=C, \text{lcl}=L) \vdash_{\text{dom}} (\text{locals } (\text{store } s)) \triangleright t \triangleright A)) \wedge$
 $s :: \preceq(G, L))$
 $\longrightarrow (\forall Y'\ s'. G \vdash s -t \triangleright -n \longrightarrow (Y',s') \longrightarrow Q\ Y'\ s'\ Z))$
 $\langle \text{proof} \rangle$

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*]
 $\langle \text{ML} \rangle$

inductive

$ax\text{-derivs} :: prog \Rightarrow 'a\ triples \Rightarrow 'a\ triples \Rightarrow bool$ ($-,|-|- [61,58,58] 57$)
and $ax\text{-deriv} :: prog \Rightarrow 'a\ triples \Rightarrow 'a\ triple \Rightarrow bool$ ($-,|-|- [61,58,58] 57$)
for $G :: prog$
where

$G, A \vdash t \equiv G, A \models \{t\}$

| $empty: G, A \models \{\}$
| $insert: [G, A \vdash t; G, A \models ts] \Longrightarrow$
 $G, A \models insert\ t\ ts$

| $asm: ts \subseteq A \Longrightarrow G, A \models ts$

| $weaken: [G, A \models ts'; ts \subseteq ts'] \Longrightarrow G, A \models ts$

| $conseq: \forall Y\ s\ Z. P\ Y\ s\ Z \longrightarrow (\exists P'\ Q'. G, A \models \{P'\} t \triangleright \{Q'\} \wedge (\forall Y'\ s'\ Z'$
 $(\forall Y'\ Z'. P'\ Y'\ s'\ Z' \longrightarrow Q'\ Y'\ s'\ Z')) \longrightarrow$
 $Q\ Y'\ s'\ Z))$

$$\Longrightarrow G, A \vdash \{P\} t \triangleright \{Q\}$$

$$| \text{hazard}: G, A \vdash \{P \wedge. \text{Not} \circ \text{type-ok } G \ t\} t \triangleright \{Q\}$$

$$| \text{Abrupt}: G, A \vdash \{P \leftarrow (\text{arbitrary3 } t) \wedge. \text{Not} \circ \text{normal}\} t \triangleright \{P\}$$

— variables

$$| \text{LVar}: G, A \vdash \{\text{Normal } (\lambda s.. P \leftarrow \text{Var } (\text{lvar } vn \ s))\} \text{LVar } vn \triangleright \{P\}$$

$$| \text{FVar}: \llbracket G, A \vdash \{\text{Normal } P\} . \text{Init } C. \{Q\}; \\ G, A \vdash \{Q\} e \triangleright \{\lambda \text{Val}:a.. \text{fvar } C \ \text{stat } \text{fn } a \ \dots; R\} \rrbracket \Longrightarrow \\ G, A \vdash \{\text{Normal } P\} \{\text{acc } C, C, \text{stat}\} e.. \text{fn} \triangleright \{R\}$$

$$| \text{AVar}: \llbracket G, A \vdash \{\text{Normal } P\} e1 \triangleright \{Q\}; \\ \forall a. G, A \vdash \{Q \leftarrow \text{Val } a\} e2 \triangleright \{\lambda \text{Val}:i.. \text{avar } G \ i \ a \ \dots; R\} \rrbracket \Longrightarrow \\ G, A \vdash \{\text{Normal } P\} e1.[e2] \triangleright \{R\}$$

— expressions

$$| \text{NewC}: \llbracket G, A \vdash \{\text{Normal } P\} . \text{Init } C. \{\text{Alloc } G \ (C \text{Inst } C) \ Q\} \rrbracket \Longrightarrow \\ G, A \vdash \{\text{Normal } P\} \text{NewC } C \triangleright \{Q\}$$

$$| \text{NewA}: \llbracket G, A \vdash \{\text{Normal } P\} . \text{init-comp-ty } T. \{Q\}; G, A \vdash \{Q\} e \triangleright \\ \{\lambda \text{Val}:i.. \text{abupd } (\text{check-neg } i) \ .; \text{Alloc } G \ (\text{Arr } T \ (\text{the-Intg } i)) \ R\} \rrbracket \Longrightarrow \\ G, A \vdash \{\text{Normal } P\} \text{New } T[e] \triangleright \{R\}$$

$$| \text{Cast}: \llbracket G, A \vdash \{\text{Normal } P\} e \triangleright \{\lambda \text{Val}:v.. \lambda s.. \\ \text{abupd } (\text{raise-if } (\neg G, s \vdash v \ \text{fits } T) \ \text{ClassCast}) \ .; Q \leftarrow \text{Val } v\} \rrbracket \Longrightarrow \\ G, A \vdash \{\text{Normal } P\} \text{Cast } T \ e \triangleright \{Q\}$$

$$| \text{Inst}: \llbracket G, A \vdash \{\text{Normal } P\} e \triangleright \{\lambda \text{Val}:v.. \lambda s.. \\ Q \leftarrow \text{Val } (\text{Bool } (v \neq \text{Null} \wedge G, s \vdash v \ \text{fits } \text{RefT } T))\} \rrbracket \Longrightarrow \\ G, A \vdash \{\text{Normal } P\} e \ \text{InstOf } T \triangleright \{Q\}$$

$$| \text{Lit}: G, A \vdash \{\text{Normal } (P \leftarrow \text{Val } v)\} \text{Lit } v \triangleright \{P\}$$

$$| \text{UnOp}: \llbracket G, A \vdash \{\text{Normal } P\} e \triangleright \{\lambda \text{Val}:v.. Q \leftarrow \text{Val } (\text{eval-unop } \text{unop } v)\} \rrbracket \\ \Longrightarrow \\ G, A \vdash \{\text{Normal } P\} \text{UnOp } \text{unop } e \triangleright \{Q\}$$

$$| \text{BinOp}: \\ \llbracket G, A \vdash \{\text{Normal } P\} e1 \triangleright \{Q\}; \\ \forall v1. G, A \vdash \{Q \leftarrow \text{Val } v1\} \\ (\text{if need-second-arg binop } v1 \ \text{then } (\text{In1l } e2) \ \text{else } (\text{In1r } \text{Skip})) \triangleright \\ \{\lambda \text{Val}:v2.. R \leftarrow \text{Val } (\text{eval-binop } \text{binop } v1 \ v2)\} \rrbracket \\ \Longrightarrow \\ G, A \vdash \{\text{Normal } P\} \text{BinOp } \text{binop } e1 \ e2 \triangleright \{R\}$$

$$| \text{Super}: G, A \vdash \{\text{Normal } (\lambda s.. P \leftarrow \text{Val } (\text{val-this } s))\} \text{Super} \triangleright \{P\}$$

$$| \text{Acc}: \llbracket G, A \vdash \{\text{Normal } P\} va \triangleright \{\lambda \text{Var}:(v,f).. Q \leftarrow \text{Val } v\} \rrbracket \Longrightarrow \\ G, A \vdash \{\text{Normal } P\} \text{Acc } va \triangleright \{Q\}$$

$$| \text{Ass}: \llbracket G, A \vdash \{\text{Normal } P\} va \triangleright \{Q\}; \\ \forall vf. G, A \vdash \{Q \leftarrow \text{Var } vf\} e \triangleright \{\lambda \text{Val}:v.. \text{assign } (\text{snd } vf) \ v \ .; R\} \rrbracket \Longrightarrow \\ G, A \vdash \{\text{Normal } P\} va := e \triangleright \{R\}$$

$$| \text{Cond}: \llbracket G, A \vdash \{\text{Normal } P\} e0 \triangleright \{P\}; \\ \forall b. G, A \vdash \{P' \leftarrow b\} (\text{if } b \ \text{then } e1 \ \text{else } e2) \triangleright \{Q\} \rrbracket \Longrightarrow \\ G, A \vdash \{\text{Normal } P\} e0 \ ? \ e1 \ : \ e2 \triangleright \{Q\}$$

- | *Call*:

$$\begin{aligned} & \llbracket G, A \vdash \{ \text{Normal } P \} e \multimap \{ Q \}; \forall a. G, A \vdash \{ Q \leftarrow \text{Val } a \} \text{ args} \doteq \{ R \ a \}; \\ & \forall a \text{ vs } \text{invC declC } l. G, A \vdash \{ (R \ a \leftarrow \text{Vals } \text{vs} \wedge. \\ & (\lambda s. \text{declC} = \text{invocation-declclass } G \text{ mode } (\text{store } s) \ a \ \text{statT } (\text{name} = \text{mn}, \text{parTs} = \text{pTs}) \wedge \\ & \text{invC} = \text{invocation-class mode } (\text{store } s) \ a \ \text{statT } \wedge \\ & l = \text{locals } (\text{store } s)) \}; \\ & \text{init-lvars } G \ \text{declC } (\text{name} = \text{mn}, \text{parTs} = \text{pTs}) \ \text{mode } a \ \text{vs} \wedge. \\ & (\lambda s. \text{normal } s \longrightarrow G \vdash \text{mode} \rightarrow \text{invC} \preceq \text{statT}) \} \\ & \text{Methd declC } (\text{name} = \text{mn}, \text{parTs} = \text{pTs}) \multimap \{ \text{set-lvars } l \ .; S \} \rrbracket \Longrightarrow \\ & G, A \vdash \{ \text{Normal } P \} \{ \text{accC}, \text{statT}, \text{mode} \} e \cdot \text{mn}(\{ \text{pTs} \} \text{args}) \multimap \{ S \} \end{aligned}$$
- | *Methd*:
$$\llbracket G, A \cup \{ \{ P \} \text{Methd} \multimap \{ Q \} \mid \text{ms} \} \vdash \{ \{ P \} \text{body } G \multimap \{ Q \} \mid \text{ms} \} \rrbracket \Longrightarrow G, A \vdash \{ \{ P \} \text{Methd} \multimap \{ Q \} \mid \text{ms} \}$$
- | *Body*:
$$\begin{aligned} & \llbracket G, A \vdash \{ \text{Normal } P \} . \text{Init } D. \{ Q \}; \\ & G, A \vdash \{ Q \} .c. \{ \lambda s. . \text{abupd } (\text{absorb } \text{Ret}) \ .; R \leftarrow (\text{In1 } (\text{the } (\text{locals } s \ \text{Result}))) \} \rrbracket \\ & \Longrightarrow \\ & G, A \vdash \{ \text{Normal } P \} \text{Body } D \ c \multimap \{ R \} \end{aligned}$$
- expression lists
- | *Nil*:
$$G, A \vdash \{ \text{Normal } (P \leftarrow \text{Vals } []) \} [] \doteq \{ P \}$$
- | *Cons*:
$$\begin{aligned} & \llbracket G, A \vdash \{ \text{Normal } P \} e \multimap \{ Q \}; \\ & \forall v. G, A \vdash \{ Q \leftarrow \text{Val } v \} \text{es} \doteq \{ \lambda \text{Vals} : \text{vs}. R \leftarrow \text{Vals } (v \# \text{vs}) \} \rrbracket \Longrightarrow \\ & G, A \vdash \{ \text{Normal } P \} e \# \text{es} \doteq \{ R \} \end{aligned}$$
- statements
- | *Skip*:
$$G, A \vdash \{ \text{Normal } (P \leftarrow \diamond) \} . \text{Skip}. \{ P \}$$
- | *Expr*:
$$\begin{aligned} & \llbracket G, A \vdash \{ \text{Normal } P \} e \multimap \{ Q \leftarrow \diamond \} \rrbracket \Longrightarrow \\ & G, A \vdash \{ \text{Normal } P \} . \text{Expr } e. \{ Q \} \end{aligned}$$
- | *Lab*:
$$\begin{aligned} & \llbracket G, A \vdash \{ \text{Normal } P \} .c. \{ \text{abupd } (\text{absorb } l) \ .; Q \} \rrbracket \Longrightarrow \\ & G, A \vdash \{ \text{Normal } P \} .l. c. \{ Q \} \end{aligned}$$
- | *Comp*:
$$\begin{aligned} & \llbracket G, A \vdash \{ \text{Normal } P \} .c1. \{ Q \}; \\ & G, A \vdash \{ Q \} .c2. \{ R \} \rrbracket \Longrightarrow \\ & G, A \vdash \{ \text{Normal } P \} .c1;;c2. \{ R \} \end{aligned}$$
- | *If*:
$$\begin{aligned} & \llbracket G, A \vdash \{ \text{Normal } P \} e \multimap \{ P' \}; \\ & \forall b. G, A \vdash \{ P' \leftarrow = b \} .(\text{if } b \text{ then } c1 \text{ else } c2). \{ Q \} \rrbracket \Longrightarrow \\ & G, A \vdash \{ \text{Normal } P \} . \text{If}(e) \ c1 \ \text{Else } c2. \{ Q \} \end{aligned}$$
- | *Loop*:
$$\begin{aligned} & \llbracket G, A \vdash \{ P \} e \multimap \{ P' \}; \\ & G, A \vdash \{ \text{Normal } (P' \leftarrow = \text{True}) \} .c. \{ \text{abupd } (\text{absorb } (\text{Cont } l)) \ .; P \} \rrbracket \Longrightarrow \\ & G, A \vdash \{ P \} .l. \text{While}(e) \ c. \{ (P' \leftarrow = \text{False}) \downarrow = \diamond \} \end{aligned}$$
- | *Jmp*:
$$G, A \vdash \{ \text{Normal } (\text{abupd } (\lambda a. (\text{Some } (\text{Jump } j))) \ .; P \leftarrow \diamond) \} . \text{Jmp } j. \{ P \}$$
- | *Throw*:
$$\begin{aligned} & \llbracket G, A \vdash \{ \text{Normal } P \} e \multimap \{ \lambda \text{Val} : a. \text{abupd } (\text{throw } a) \ .; Q \leftarrow \diamond \} \rrbracket \Longrightarrow \\ & G, A \vdash \{ \text{Normal } P \} . \text{Throw } e. \{ Q \} \end{aligned}$$
- | *Try*:
$$\begin{aligned} & \llbracket G, A \vdash \{ \text{Normal } P \} .c1. \{ \text{SXAlloc } G \ Q \}; \\ & G, A \vdash \{ Q \wedge. (\lambda s. G, s \vdash \text{catch } C) \ .; \text{new-xcpt-var } \text{vn} \} .c2. \{ R \}; \\ & (Q \wedge. (\lambda s. \neg G, s \vdash \text{catch } C)) \Rightarrow R \rrbracket \Longrightarrow \\ & G, A \vdash \{ \text{Normal } P \} . \text{Try } c1 \ \text{Catch}(C \ \text{vn}) \ c2. \{ R \} \end{aligned}$$

| *Fin*: $\llbracket G, A \vdash \{ \text{Normal } P \} .c1. \{ Q \};$
 $\forall x. G, A \vdash \{ Q \wedge (\lambda s. x = \text{fst } s) ; . \text{abupd } (\lambda x. \text{None}) \}$
 $.c2. \{ \text{abupd } (\text{abrupt-if } (x \neq \text{None}) x) .; R \} \rrbracket \implies$
 $G, A \vdash \{ \text{Normal } P \} .c1 \text{ Finally } c2. \{ R \}$

| *Done*: $G, A \vdash \{ \text{Normal } (P \leftarrow \diamond \wedge \text{initd } C) \} .\text{Init } C. \{ P \}$

| *Init*: $\llbracket \text{the } (\text{class } G \ C) = c;$
 $G, A \vdash \{ \text{Normal } ((P \wedge \text{Not } \circ \text{initd } C) ; . \text{supd } (\text{init-class-obj } G \ C)) \}$
 $.(\text{if } C = \text{Object then Skip else Init } (\text{super } c)). \{ Q \};$
 $\forall l. G, A \vdash \{ Q \wedge (\lambda s. l = \text{locals } (\text{store } s)) ; . \text{set-lvars empty} \}$
 $.\text{init } c. \{ \text{set-lvars } l .; R \} \rrbracket \implies$
 $G, A \vdash \{ \text{Normal } (P \wedge \text{Not } \circ \text{initd } C) \} .\text{Init } C. \{ R \}$

— 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 \implies 'a assn \implies 'a assn \implies '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*: $\llbracket G, A' \Vdash ts; G, A \Vdash A' \rrbracket \implies G, A \Vdash ts$
<proof>

lemma *ax-thin* [*rule-format* (*no-asm*)]:

$G, (A :: 'a \text{ triple set}) \Vdash (ts :: 'a \text{ triple set}) \implies \forall A. A' \subseteq A \longrightarrow G, A \Vdash ts$
<proof>

lemma *ax-thin-insert*: $G, (A :: 'a \text{ triple set}) \Vdash (t :: 'a \text{ triple}) \implies G, \text{insert } x \ A \Vdash t$
<proof>

lemma *subset-mtriples-iff*:

$ts \subseteq \{ \{ P \} \text{ mb-} \multimap \{ Q \} \mid ms \} = (\exists ms'. ms' \subseteq ms \wedge ts = \{ \{ P \} \text{ mb-} \multimap \{ Q \} \mid ms' \})$
<proof>

lemma *weaken*:

$G, (A :: 'a \text{ triple set}) \Vdash (ts' :: 'a \text{ triple set}) \implies !ts. ts \subseteq ts' \longrightarrow G, A \Vdash ts$
<proof>

rules derived from conseq

In the following rules we often have to give some type annotations like: $G, A \vdash \{ P \} \ t \multimap \{ Q \}$. 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.Method* 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*: $\llbracket G, (A::'a \text{ triple set}) \vdash \{P'::'a \text{ assn}\} t \succ \{Q'\};$
 $\forall Y s Z. P Y s Z \longrightarrow (\forall Y' s'. (\forall Y Z'. P' Y s Z' \longrightarrow Q' Y' s' Z') \longrightarrow$
 $Q Y' s' Z) \rrbracket$
 $\implies G, A \vdash \{P::'a \text{ assn}\} t \succ \{Q\}$
 ⟨proof⟩

lemma *conseq12'*: $\llbracket G, (A::'a \text{ triple set}) \vdash \{P'::'a \text{ assn}\} t \succ \{Q'\}; \forall s Y' s'.$
 $(\forall Y Z. P' Y s Z \longrightarrow Q' Y' s' Z) \longrightarrow$
 $(\forall Y Z. P Y s Z \longrightarrow Q Y' s' Z) \rrbracket$
 $\implies G, A \vdash \{P::'a \text{ assn}\} t \succ \{Q\}$
 ⟨proof⟩

lemma *conseq12-from-conseq12'*: $\llbracket G, (A::'a \text{ triple set}) \vdash \{P'::'a \text{ assn}\} t \succ \{Q'\};$
 $\forall Y s Z. P Y s Z \longrightarrow (\forall Y' s'. (\forall Y Z'. P' Y s Z' \longrightarrow Q' Y' s' Z') \longrightarrow$
 $Q Y' s' Z) \rrbracket$
 $\implies G, A \vdash \{P::'a \text{ assn}\} t \succ \{Q\}$
 ⟨proof⟩

lemma *conseq1*: $\llbracket G, (A::'a \text{ triple set}) \vdash \{P'::'a \text{ assn}\} t \succ \{Q\}; P \Rightarrow P' \rrbracket$
 $\implies G, A \vdash \{P::'a \text{ assn}\} t \succ \{Q\}$
 ⟨proof⟩

lemma *conseq2*: $\llbracket G, (A::'a \text{ triple set}) \vdash \{P::'a \text{ assn}\} t \succ \{Q'\}; Q' \Rightarrow Q \rrbracket$
 $\implies G, A \vdash \{P::'a \text{ assn}\} t \succ \{Q\}$
 ⟨proof⟩

lemma *ax-escape*:
 $\llbracket \forall Y s Z. P Y s Z$
 $\longrightarrow G, (A::'a \text{ triple set}) \vdash \{\lambda Y' s' (Z'::'a). (Y', s') = (Y, s)\}$
 $t \succ$
 $\{\lambda Y s Z'. Q Y s Z\}$
 $\rrbracket \implies G, A \vdash \{P::'a \text{ assn}\} t \succ \{Q::'a \text{ assn}\}$
 ⟨proof⟩

lemma *ax-constant*: $\llbracket C \implies G, (A::'a \text{ triple set}) \vdash \{P::'a \text{ assn}\} t \succ \{Q\} \rrbracket$
 $\implies G, A \vdash \{\lambda Y s Z. C \wedge P Y s Z\} t \succ \{Q\}$
 ⟨proof⟩

lemma *ax-impossible* [intro]:
 $G, (A::'a \text{ triple set}) \vdash \{\lambda Y s Z. \text{False}\} t \succ \{Q::'a \text{ assn}\}$
 ⟨proof⟩

lemma *ax-nochange-lemma*: $\llbracket P Y s; \text{All } (op = w) \rrbracket \implies P w s$
 ⟨proof⟩

lemma *ax-nochange*:

$$G, (A::(res \times state) \text{ triple set}) \vdash \{\lambda Y s Z. (Y,s)=Z\} t> \{\lambda Y s Z. (Y,s)=Z\} \\ \implies G, A \vdash \{P::(res \times state) \text{ assn}\} t> \{P\} \\ \langle \text{proof} \rangle$$

lemma *ax-trivial*: $G, (A::'a \text{ triple set}) \vdash \{P::'a \text{ assn}\} t> \{\lambda Y s Z. \text{True}\}$

$\langle \text{proof} \rangle$

lemma *ax-disj*:

$$\llbracket G, (A::'a \text{ triple set}) \vdash \{P1::'a \text{ assn}\} t> \{Q1\}; G, A \vdash \{P2::'a \text{ assn}\} t> \{Q2\} \rrbracket \\ \implies G, A \vdash \{\lambda Y s Z. P1 Y s Z \vee P2 Y s Z\} t> \{\lambda Y s Z. Q1 Y s Z \vee Q2 Y s Z\} \\ \langle \text{proof} \rangle$$

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*:

$$\llbracket G, (A::'a \text{ triple set}) \vdash \{P \wedge. C\} t> \{Q::'a \text{ assn}\}; \\ G, A \vdash \{P \wedge. \text{Not} \circ C\} t> \{Q\} \rrbracket \implies G, A \vdash \{P\} t> \{Q\} \\ \langle \text{proof} \rangle$$

lemma *ax-adapt*: $G, (A::'a \text{ triple set}) \vdash \{P::'a \text{ assn}\} t> \{Q\}$

$$\implies G, A \vdash \{\text{adapt-pre } P \ Q \ Q'\} t> \{Q'\} \\ \langle \text{proof} \rangle$$

lemma *adapt-pre-adapts*: $G, (A::'a \text{ triple set}) \models \{P::'a \text{ assn}\} t> \{Q\}$

$$\longrightarrow G, A \models \{\text{adapt-pre } P \ Q \ Q'\} t> \{Q'\} \\ \langle \text{proof} \rangle$$

lemma *adapt-pre-weakest*:

$$\forall G (A::'a \text{ triple set}) t. G, A \models \{P\} t> \{Q\} \longrightarrow G, A \models \{P'\} t> \{Q'\} \implies \\ P' \Rightarrow \text{adapt-pre } P \ Q \ (Q'::'a \text{ assn}) \\ \langle \text{proof} \rangle$$

lemma *peek-and-forget1-Normal*:

$$G, (A::'a \text{ triple set}) \vdash \{\text{Normal } P\} t> \{Q::'a \text{ assn}\} \\ \implies G, A \vdash \{\text{Normal } (P \wedge. p)\} t> \{Q\} \\ \langle \text{proof} \rangle$$

lemma *peek-and-forget1*:

$$G, (A::'a \text{ triple set}) \vdash \{P::'a \text{ assn}\} t> \{Q\} \\ \implies G, A \vdash \{P \wedge. p\} t> \{Q\} \\ \langle \text{proof} \rangle$$

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*:

$$G, (A :: 'a \text{ triple set}) \vdash \{\text{Normal } P :: 'a \text{ assn}\} \text{Lit } v \succ \{\text{Normal } (P \downarrow = \text{Val } v)\}$$

<proof>

lemma *ax-Lit2-test-complete*:

$$G, (A :: 'a \text{ triple set}) \vdash \{\text{Normal } (P \leftarrow \text{Val } v) :: 'a \text{ assn}\} \text{Lit } v \succ \{P\}$$

<proof>

lemma *ax-LVar2*: $G, (A :: 'a \text{ triple set}) \vdash \{\text{Normal } P :: 'a \text{ assn}\} \text{LVar } vn \succ \{\text{Normal } (\lambda s. P \downarrow = \text{Var } (\text{lvar } vn \ s))\}$

<proof>

lemma *ax-Super2*: $G, (A :: 'a \text{ triple set}) \vdash$

$$\{\text{Normal } P :: 'a \text{ assn}\} \text{Super} \succ \{\text{Normal } (\lambda s. P \downarrow = \text{Val } (\text{val-this } s))\}$$

<proof>

lemma *ax-Nil2*:

$$G, (A :: 'a \text{ triple set}) \vdash \{\text{Normal } P :: 'a \text{ assn}\} [] \succ \{\text{Normal } (P \downarrow = \text{Vals } [])\}$$

<proof>

misc derived structural rules

lemma *ax-finite-mtriples-lemma*: $\llbracket F \subseteq ms; \text{finite } ms; \forall (C, sig) \in ms.$

$$G, (A :: 'a \text{ triple set}) \vdash \{\text{Normal } (P \ C \ sig) :: 'a \text{ assn}\} \text{mb } C \ sig \succ \{Q \ C \ sig\} \implies$$

$$G, A \vdash \{\{P\} \text{mb} \succ \{Q\} \mid F\}$$

<proof>

lemmas *ax-finite-mtriples* = *ax-finite-mtriples-lemma* [OF subset-refl]

lemma *ax-derivs-insertD*:

$G, (A::'a \text{ triple set}) \vdash \text{insert } (t::'a \text{ triple}) \text{ } ts \implies G, A \vdash t \wedge G, A \vdash ts$
 ⟨proof⟩

lemma *ax-methods-spec*:

$\llbracket G, (A::'a \text{ triple set}) \vdash \text{split } f \text{ ' } ms; (C, sig) \in ms \rrbracket \implies G, A \vdash ((f \ C \ sig)::'a \text{ triple})$
 ⟨proof⟩

lemma *ax-finite-pointwise-lemma* [rule-format]: $\llbracket F \subseteq ms; \text{finite } ms \rrbracket \implies$

$(\forall (C, sig) \in F. G, (A::'a \text{ triple set}) \vdash (f \ C \ sig)::'a \text{ triple})) \longrightarrow (\forall (C, sig) \in ms. G, A \vdash (g \ C \ sig)::'a \text{ triple})) \longrightarrow$
 $G, A \vdash \text{split } f \text{ ' } F \longrightarrow G, A \vdash \text{split } g \text{ ' } F$

⟨proof⟩

lemmas *ax-finite-pointwise* = *ax-finite-pointwise-lemma* [OF subset-refl]

lemma *ax-no-hazard*:

$G, (A::'a \text{ triple set}) \vdash \{P \ \wedge. \ \text{type-ok } G \ t\} \ t \succ \{Q::'a \text{ assn}\} \implies G, A \vdash \{P\} \ t \succ \{Q\}$
 ⟨proof⟩

lemma *ax-free-wt*:

$(\exists T \ L \ C. (\text{prg} = G, \text{cls} = C, \text{lcl} = L) \vdash t::T)$
 $\longrightarrow G, (A::'a \text{ triple set}) \vdash \{\text{Normal } P\} \ t \succ \{Q::'a \text{ assn}\} \implies$
 $G, A \vdash \{\text{Normal } P\} \ t \succ \{Q\}$

⟨proof⟩

⟨ML⟩

declare *ax-Abrupts* [intro!]

lemmas *ax-Normal-cases* = *ax-cases* [of - - - normal]

lemma *ax-Skip* [intro!]: $G, (A::'a \text{ triple set}) \vdash \{P \leftarrow \diamond\} \ .\text{Skip}. \{P::'a \text{ assn}\}$

⟨proof⟩

lemmas *ax-SkipI* = *ax-Skip* [THEN conseq1, standard]

derived rules for methd call

lemma *ax-Call-known-DynT*:

$\llbracket G \vdash \text{IntVir} \rightarrow C \preceq \text{statT};$
 $\forall a \ \text{vs } l. G, A \vdash \{(R \ a \leftarrow \text{Vals } \text{vs} \ \wedge. \ (\lambda s. \ l = \text{locals } (store \ s))) \};$
 $\text{init-lvars } G \ C \ (\text{name} = mn, \text{parTs} = pTs) \ \text{IntVir } a \ \text{vs}\}$
 $\text{Methd } C \ (\text{name} = mn, \text{parTs} = pTs) \dashv \succ \{\text{set-lvars } l \ .; \ S\};$
 $\forall a. G, A \vdash \{Q \leftarrow \text{Val } a\} \ \text{args} \dashv \succ$
 $\{R \ a \ \wedge. \ (\lambda s. \ C = \text{obj-class } (the \ (heap \ (store \ s) \ (the-Addr \ a)))) \ \wedge$
 $C = \text{invocation-declclass}$
 $G \ \text{IntVir } (store \ s) \ a \ \text{statT } (\text{name} = mn, \text{parTs} = pTs) \ \};$
 $G, (A::'a \text{ triple set}) \vdash \{\text{Normal } P\} \ e \dashv \succ \{Q::'a \text{ assn}\}$
 $\implies G, A \vdash \{\text{Normal } P\} \ \{\text{acc } C, \text{statT}, \text{IntVir}\} e \ .mn \ (\{pTs\} \ \text{args}) \dashv \succ \{S\}$

⟨proof⟩

lemma *ax-Call-Static*:

$$\begin{aligned}
& \llbracket \forall a \text{ vs } l. G, A \vdash \{R \leftarrow \text{Vals } vs \wedge. (\lambda s. l = \text{locals } (store \ s)) \}; \\
& \quad \text{init-lvars } G \ C \ (\!| \text{name=mn, parTs=pTs} \!) \ \text{Static any-Addr } vs \} \\
& \quad \text{Methd } C \ (\!| \text{name=mn, parTs=pTs} \!) \text{-} \succ \ \{ \text{set-lvars } l \ .; S \}; \\
& \quad G, A \vdash \{ \text{Normal } P \} \ e \text{-} \succ \ \{ Q \}; \\
& \quad \forall a. G, (A :: 'a \ \text{triple set}) \vdash \{ Q \leftarrow \text{Val } a \} \ \text{args} \dot{\succ} \ \{ (R :: \text{val} \Rightarrow 'a \ \text{assn}) \ a \\
& \quad \wedge. (\lambda s. C = \text{invocation-declclass} \\
& \quad \quad G \ \text{Static } (store \ s) \ a \ \text{statT } (\!| \text{name=mn, parTs=pTs} \!)) \} \\
\llbracket & \implies G, A \vdash \{ \text{Normal } P \} \ \{ \text{accC, statT, Static} \} \cdot \text{mn}(\{pTs\} \ \text{args}) \text{-} \succ \ \{ S \} \\
& \langle \text{proof} \rangle
\end{aligned}$$

lemma *ax-Method1*:

$$\begin{aligned}
& \llbracket G, A \cup \{ \{ P \} \ \text{Methd} \text{-} \succ \ \{ Q \} \mid ms \} \vdash \{ \{ P \} \ \text{body } G \text{-} \succ \ \{ Q \} \mid ms \}; (C, sig) \in ms \rrbracket \implies \\
& \quad G, A \vdash \{ \text{Normal } (P \ C \ sig) \} \ \text{Methd } C \ sig \text{-} \succ \ \{ Q \ C \ sig \} \\
& \langle \text{proof} \rangle
\end{aligned}$$

lemma *ax-MethodN*:

$$\begin{aligned}
& G, \text{insert}(\{ \text{Normal } P \} \ \text{Methd } C \ sig \text{-} \succ \ \{ Q \}) \ A \vdash \\
& \quad \{ \text{Normal } P \} \ \text{body } G \ C \ sig \text{-} \succ \ \{ Q \} \implies \\
& \quad G, A \vdash \{ \text{Normal } P \} \ \text{Methd } C \ sig \text{-} \succ \ \{ Q \} \\
& \langle \text{proof} \rangle
\end{aligned}$$

lemma *ax-StatRef*:

$$\begin{aligned}
& G, (A :: 'a \ \text{triple set}) \vdash \{ \text{Normal } (P \leftarrow \text{Val } \text{Null}) \} \ \text{StatRef } rt \text{-} \succ \ \{ P :: 'a \ \text{assn} \} \\
& \langle \text{proof} \rangle
\end{aligned}$$

rules derived from Init and Done

lemma *ax-InitS*: $\llbracket \text{the } (class \ G \ C) = c; C \neq \text{Object};$

$$\begin{aligned}
& \forall l. G, A \vdash \{ Q \wedge. (\lambda s. l = \text{locals } (store \ s)) \}; \ \text{set-lvars empty} \} \\
& \quad \text{init } c. \ \{ \text{set-lvars } l \ .; R \}; \\
& \quad G, A \vdash \{ \text{Normal } ((P \wedge. \text{Not} \circ \text{initd } C) \ .; \ \text{supd } (\text{init-class-obj } G \ C)) \} \\
& \quad \text{.Init } (\text{super } c). \ \{ Q \} \rrbracket \implies \\
& \quad G, (A :: 'a \ \text{triple set}) \vdash \{ \text{Normal } (P \wedge. \text{Not} \circ \text{initd } C) \} \ \text{.Init } C. \ \{ R :: 'a \ \text{assn} \} \\
& \langle \text{proof} \rangle
\end{aligned}$$

lemma *ax-Init-Skip-lemma*:

$$\begin{aligned}
& \forall l. G, (A :: 'a \ \text{triple set}) \vdash \{ P \leftarrow \diamond \wedge. (\lambda s. l = \text{locals } (store \ s)) \}; \ \text{set-lvars } l' \} \\
& \quad \text{.Skip}. \ \{ (\text{set-lvars } l \ .; P) :: 'a \ \text{assn} \} \\
& \langle \text{proof} \rangle
\end{aligned}$$

lemma *ax-triv-InitS*: $\llbracket \text{the } (class \ G \ C) = c; \text{init } c = \text{Skip}; C \neq \text{Object};$

$$\begin{aligned}
& P \leftarrow \diamond \Rightarrow (\text{supd } (\text{init-class-obj } G \ C) \ .; P); \\
& \quad G, A \vdash \{ \text{Normal } (P \wedge. \text{initd } C) \} \ \text{.Init } (\text{super } c). \ \{ (P \wedge. \text{initd } C) \leftarrow \diamond \} \rrbracket \implies \\
& \quad G, (A :: 'a \ \text{triple set}) \vdash \{ \text{Normal } P \leftarrow \diamond \} \ \text{.Init } C. \ \{ (P \wedge. \text{initd } C) :: 'a \ \text{assn} \} \\
& \langle \text{proof} \rangle
\end{aligned}$$

lemma *ax-Init-Object*: $wf\text{-prog } G \implies G, (A :: 'a \ \text{triple set}) \vdash$

$$\begin{aligned}
& \{ \text{Normal } ((\text{supd } (\text{init-class-obj } G \ \text{Object}) \ .; P \leftarrow \diamond) \wedge. \text{Not} \circ \text{initd } \text{Object}) \} \\
& \quad \text{.Init } \text{Object}. \ \{ (P \wedge. \text{initd } \text{Object}) :: 'a \ \text{assn} \} \\
& \langle \text{proof} \rangle
\end{aligned}$$

lemma *ax-triv-Init-Object*: $\llbracket \text{wf-prog } G; (P::'a \text{ assn}) \Rightarrow (\text{supd } (\text{init-class-obj } G \text{ Object}) .; P) \rrbracket \Longrightarrow$
 $G, (A::'a \text{ triple set}) \vdash \{ \text{Normal } P \leftarrow \diamond \} . \text{Init Object} . \{ P \wedge . \text{initd Object} \}$
 $\langle \text{proof} \rangle$

introduction rules for Alloc and SXAlloc

lemma *ax-SXAlloc-Normal*:
 $G, (A::'a \text{ triple set}) \vdash \{ P::'a \text{ assn} \} .c. \{ \text{Normal } Q \}$
 $\Longrightarrow G, A \vdash \{ P \} .c. \{ \text{SXAlloc } G \ Q \}$
 $\langle \text{proof} \rangle$

lemma *ax-Alloc*:
 $G, (A::'a \text{ triple set}) \vdash \{ P::'a \text{ assn} \} t \succ$
 $\{ \text{Normal } (\lambda Y (x, s) Z. (\forall a. \text{new-Addr } (\text{heap } s) = \text{Some } a \longrightarrow$
 $Q (\text{Val } (\text{Addr } a)) (\text{Norm}(\text{init-obj } G (\text{CInst } C) (\text{Heap } a) s)) Z)) \wedge .$
 $\text{heap-free } (\text{Suc } (\text{Suc } 0)) \}$
 $\Longrightarrow G, A \vdash \{ P \} t \succ \{ \text{Alloc } G (\text{CInst } C) \ Q \}$
 $\langle \text{proof} \rangle$

lemma *ax-Alloc-Arr*:
 $G, (A::'a \text{ triple set}) \vdash \{ P::'a \text{ assn} \} t \succ$
 $\{ \lambda \text{Val}:i. \text{Normal } (\lambda Y (x, s) Z. \neg \text{the-Intg } i < 0 \wedge$
 $(\forall a. \text{new-Addr } (\text{heap } s) = \text{Some } a \longrightarrow$
 $Q (\text{Val } (\text{Addr } a)) (\text{Norm } (\text{init-obj } G (\text{Arr } T (\text{the-Intg } i)) (\text{Heap } a) s)) Z)) \wedge .$
 $\text{heap-free } (\text{Suc } (\text{Suc } 0)) \}$
 \Longrightarrow
 $G, A \vdash \{ P \} t \succ \{ \lambda \text{Val}:i. \text{abupd } (\text{check-neg } i) .; \text{Alloc } G (\text{Arr } T(\text{the-Intg } i)) \ Q \}$
 $\langle \text{proof} \rangle$

lemma *ax-SXAlloc-catch-SXcpt*:
 $\llbracket G, (A::'a \text{ triple set}) \vdash \{ P::'a \text{ assn} \} t \succ$
 $\{ (\lambda Y (x, s) Z. x = \text{Some } (\text{Xcpt } (\text{Std } xn)) \wedge$
 $(\forall a. \text{new-Addr } (\text{heap } s) = \text{Some } a \longrightarrow$
 $Q Y (\text{Some } (\text{Xcpt } (\text{Loc } a)), \text{init-obj } G (\text{CInst } (\text{SXcpt } xn)) (\text{Heap } a) s) Z))$
 $\wedge . \text{heap-free } (\text{Suc } (\text{Suc } 0)) \}$
 \Longrightarrow
 $G, A \vdash \{ P \} t \succ \{ \text{SXAlloc } G (\lambda Y s Z. Q Y s Z \wedge G, s \vdash \text{catch } \text{SXcpt } xn) \}$
 $\langle \text{proof} \rangle$

end

Chapter 23

AxSound

62 Soundness proof for Axiomatic semantics of Java expressions and statements

theory *AxSound* imports *AxSem* begin

validity

consts

$$\begin{aligned} \text{triple-valid2} &:: \text{prog} \Rightarrow \text{nat} \Rightarrow \quad 'a \text{ triple} \Rightarrow \text{bool} \\ &\quad (_ \models _ :: - [61, 0, 58] 57) \\ \text{ax-valids2} &:: \text{prog} \Rightarrow 'a \text{ triples} \Rightarrow 'a \text{ triples} \Rightarrow \text{bool} \\ &\quad (_ \models _ :: - [61, 58, 58] 57) \end{aligned}$$

defs *triple-valid2-def*: $G \models n :: t \equiv \text{case } t \text{ of } \{P\} t \triangleright \{Q\} \Rightarrow$
 $\forall Y s Z. P Y s Z \longrightarrow (\forall L. s :: \preceq(G, L)$
 $\longrightarrow (\forall T C A. (\text{normal } s \longrightarrow ((\text{prg} = G, \text{cls} = C, \text{lcl} = L) \vdash t :: T \wedge$
 $\quad (\text{prg} = G, \text{cls} = C, \text{lcl} = L) \vdash \text{dom } (\text{locals } (\text{store } s)) \triangleright t \triangleright A)) \longrightarrow$
 $(\forall Y' s'. G \vdash s - t \triangleright - n \rightarrow (Y', s') \longrightarrow Q Y' s' Z \wedge s' :: \preceq(G, L))))$

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*: $G \models n :: \{P\} t \triangleright \{Q\} =$
 $(\forall Y s Z. P Y s Z \longrightarrow (\forall Y' s'. G \vdash s - t \triangleright - n \rightarrow (Y', s') \longrightarrow$
 $(\forall L. s :: \preceq(G, L) \longrightarrow (\forall T C A. (\text{normal } s \longrightarrow ((\text{prg} = G, \text{cls} = C, \text{lcl} = L) \vdash t :: T \wedge$
 $\quad (\text{prg} = G, \text{cls} = C, \text{lcl} = L) \vdash \text{dom } (\text{locals } (\text{store } s)) \triangleright t \triangleright A)) \longrightarrow$
 $\quad Q Y' s' Z \wedge s' :: \preceq(G, L))))$
 $\langle \text{proof} \rangle$

lemma *triple-valid2-eq* [*rule-format* (*no-asm*)]:
 $\text{wf-prog } G \implies \text{triple-valid2 } G = \text{triple-valid } G$
 $\langle \text{proof} \rangle$

lemma *ax-valids2-eq*: $\text{wf-prog } G \implies G, A \models :: ts = G, A \models ts$
 $\langle \text{proof} \rangle$

lemma *triple-valid2-Suc* [*rule-format* (*no-asm*)]: $G \models \text{Suc } n :: t \longrightarrow G \models n :: t$
 $\langle \text{proof} \rangle$

lemma *Methd-triple-valid2-0*: $G \models 0 :: \{\text{Normal } P\} \text{Methd } C \text{ sig} \triangleright \{Q\}$
 $\langle \text{proof} \rangle$

lemma *Methd-triple-valid2-SucI*:
 $[[G \models n :: \{\text{Normal } P\} \text{body } G C \text{ sig} \triangleright \{Q\}]$
 $\implies G \models \text{Suc } n :: \{\text{Normal } P\} \text{Methd } C \text{ sig} \triangleright \{Q\}]$
 $\langle \text{proof} \rangle$

lemma *triples-valid2-Suc*:

$G \vdash s0 \text{ -c-n} \rightarrow s1; P \ Y \ s0 \ Z \rceil \Longrightarrow Q \diamond s1 \ Z \wedge s1::\preceq(G,L)$

shows $G, A \models::\{ \{P\} \langle c \rangle_s \succ \{Q\} \}$

<proof>

lemma *valid-stmt-NormalI*:

assumes $I: \bigwedge n \ s0 \ L \ accC \ C \ s1 \ Y \ Z.$

$\llbracket \forall t \in A. G \models n::t; s0::\preceq(G,L); \text{normal } s0; (\text{prg}=G, \text{cls}=\text{acc}C, \text{lcl}=L) \vdash c::\surd;$

$(\text{prg}=G, \text{cls}=\text{acc}C, \text{lcl}=L) \vdash \text{dom } (\text{locals } (\text{store } s0)) \rangle \langle c \rangle_s \rangle C;$

$G \vdash s0 \text{ -c-n} \rightarrow s1; (\text{Normal } P) \ Y \ s0 \ Z \rceil \Longrightarrow Q \diamond s1 \ Z \wedge s1::\preceq(G,L)$

shows $G, A \models::\{ \{ \text{Normal } P \} \langle c \rangle_s \succ \{Q\} \}$

<proof>

lemma *valid-var-NormalI*:

assumes $I: \bigwedge n \ s0 \ L \ accC \ T \ C \ \text{vf} \ s1 \ Y \ Z.$

$\llbracket \forall t \in A. G \models n::t; s0::\preceq(G,L); \text{normal } s0;$

$(\text{prg}=G, \text{cls}=\text{acc}C, \text{lcl}=L) \vdash t::=T;$

$(\text{prg}=G, \text{cls}=\text{acc}C, \text{lcl}=L) \vdash \text{dom } (\text{locals } (\text{store } s0)) \rangle \langle t \rangle_v \rangle C;$

$G \vdash s0 \text{ -t-} \succ \text{vf-n} \rightarrow s1; (\text{Normal } P) \ Y \ s0 \ Z \rceil$

$\Longrightarrow Q \ (\text{In2 } \text{vf}) \ s1 \ Z \wedge s1::\preceq(G,L)$

shows $G, A \models::\{ \{ \text{Normal } P \} \langle t \rangle_v \succ \{Q\} \}$

<proof>

lemma *valid-expr-NormalI*:

assumes $I: \bigwedge n \ s0 \ L \ accC \ T \ C \ v \ s1 \ Y \ Z.$

$\llbracket \forall t \in A. G \models n::t; s0::\preceq(G,L); \text{normal } s0;$

$(\text{prg}=G, \text{cls}=\text{acc}C, \text{lcl}=L) \vdash t::-T;$

$(\text{prg}=G, \text{cls}=\text{acc}C, \text{lcl}=L) \vdash \text{dom } (\text{locals } (\text{store } s0)) \rangle \langle t \rangle_e \rangle C;$

$G \vdash s0 \text{ -t-} \succ v-n \rightarrow s1; (\text{Normal } P) \ Y \ s0 \ Z \rceil$

$\Longrightarrow Q \ (\text{In1 } v) \ s1 \ Z \wedge s1::\preceq(G,L)$

shows $G, A \models::\{ \{ \text{Normal } P \} \langle t \rangle_e \succ \{Q\} \}$

<proof>

lemma *valid-expr-list-NormalI*:

assumes $I: \bigwedge n \ s0 \ L \ accC \ T \ C \ \text{vs} \ s1 \ Y \ Z.$

$\llbracket \forall t \in A. G \models n::t; s0::\preceq(G,L); \text{normal } s0;$

$(\text{prg}=G, \text{cls}=\text{acc}C, \text{lcl}=L) \vdash t::\doteq T;$

$(\text{prg}=G, \text{cls}=\text{acc}C, \text{lcl}=L) \vdash \text{dom } (\text{locals } (\text{store } s0)) \rangle \langle t \rangle_l \rangle C;$

$G \vdash s0 \text{ -t-} \succ \text{vs-n} \rightarrow s1; (\text{Normal } P) \ Y \ s0 \ Z \rceil$

$\Longrightarrow Q \ (\text{In3 } \text{vs}) \ s1 \ Z \wedge s1::\preceq(G,L)$

shows $G, A \models::\{ \{ \text{Normal } P \} \langle t \rangle_l \succ \{Q\} \}$

<proof>

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*: $G \vdash s0 \text{ -t-} \succ -n \rightarrow (v, s1)$

and *wt*: $\text{normal } s0 \Longrightarrow (\text{prg}=G, \text{cls}=\text{acc}C, \text{lcl}=L) \vdash t::T$

and *da*: $\text{normal } s0 \Longrightarrow (\text{prg}=G, \text{cls}=\text{acc}C, \text{lcl}=L) \vdash \text{dom } (\text{locals } (\text{store } s0)) \rangle t \rangle C$

and *elim*: $\llbracket Q \ v \ s1 \ Z; s1::\preceq(G,L) \rceil \Longrightarrow \text{concl}$

shows *concl*

<proof>

lemma *all-empty*: $(!x. P) = P$

\langle proof \rangle

corollary *evaln-type-sound*:

assumes *evaln*: $G \vdash s0 \text{ -t>-n} \rightarrow (v, s1)$ **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 $s1 :: \preceq (G, L) \wedge (\text{normal } s1 \rightarrow G, L, \text{store } s1 \vdash t \gg v :: \preceq T) \wedge$
 $(\text{error-free } s0 = \text{error-free } s1)$

\langle proof \rangle

corollary *dom-locals-evaln-mono-elim* [*consumes 1*]:

assumes
evaln: $G \vdash s0 \text{ -t>-n} \rightarrow (v, s1)$ **and**
hyps: $\llbracket \text{dom} (\text{locals} (\text{store } s0)) \subseteq \text{dom} (\text{locals} (\text{store } s1)) \rrbracket$
 $\wedge \llbracket v = \text{In2 } vv; \text{normal } s1 \rrbracket$
 $\implies \text{dom} (\text{locals} (\text{store } s))$
 $\subseteq \text{dom} (\text{locals} (\text{store} ((\text{snd } vv) \text{ val } s))) \rrbracket \implies P$

shows P

\langle proof \rangle

lemma *evaln-no-abrupt*:

$\llbracket s \text{ s}'. \llbracket G \vdash s \text{ -t>-n} \rightarrow (w, s') \rrbracket; \text{normal } s' \rrbracket \implies \text{normal } s$

\langle proof \rangle

declare *inj-term-simps* [*simp*]

lemma *ax-sound2*:

assumes *wf*: *wf-prog* G
and *deriv*: $G, A \vdash ts$
shows $G, A \Vdash ts$

\langle proof \rangle

declare *inj-term-simps* [*simp del*]

theorem *ax-sound*:

wf-prog $G \implies G, (A :: 'a \text{ triple set}) \vdash (ts :: 'a \text{ triple set}) \implies G, A \Vdash ts$

\langle proof \rangle

lemma *sound-valid2-lemma*:

$\llbracket \forall v n. \text{Ball } A (\text{triple-valid2 } G n) \rightarrow P v n; \text{Ball } A (\text{triple-valid2 } G n) \rrbracket$
 $\implies P v n$

\langle 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 (-j, Thomas Kleymann)

set of not yet initialized classes

constdefs

nyinitcls :: *prog* \Rightarrow *state* \Rightarrow *qname set*
nyinitcls *G s* \equiv $\{C. \text{is-class } G \ C \wedge \neg \text{initd } C \ s\}$

lemma *nyinitcls-subset-class*: *nyinitcls* *G s* \subseteq $\{C. \text{is-class } G \ C\}$
 \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* $\{C. \text{is-class } G \ C\}$
 \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*: $\llbracket \text{nyinitcls } G \ s = \{\}; \text{is-class } G \ C \rrbracket \Longrightarrow \text{initd } C \ s$
 \langle *proof* \rangle

lemma *card-Suc-lemma*:

$\llbracket \text{card } (\text{insert } a \ A) \leq \text{Suc } n; a \notin A; \text{finite } A \rrbracket \implies \text{card } A \leq n$
 <proof>

lemma *nyinitcls-le-SucD*:

$\llbracket \text{card } (\text{nyinitcls } G \ (x,s)) \leq \text{Suc } n; \neg \text{inited } C \ (\text{globs } s); \text{class } G \ C = \text{Some } y \rrbracket \implies$
 $\text{card } (\text{nyinitcls } G \ (x, \text{init-class-obj } G \ C \ s)) \leq n$
 <proof>

lemma *inited-gext'*: $\llbracket s \leq |s'; \text{inited } C \ (\text{globs } s) \rrbracket \implies \text{inited } C \ (\text{globs } s')$

<proof>

lemma *nyinitcls-gext*: $\text{snd } s \leq | \text{snd } s' \implies \text{nyinitcls } G \ s' \subseteq \text{nyinitcls } G \ s$

<proof>

lemma *card-nyinitcls-gext*:

$\llbracket \text{snd } s \leq | \text{snd } s'; \text{card } (\text{nyinitcls } G \ s) \leq n \rrbracket \implies \text{card } (\text{nyinitcls } G \ s') \leq n$
 <proof>

init-le

constdefs

init-le :: *prog* \Rightarrow *nat* \Rightarrow *state* \Rightarrow *bool* ($\vdash \text{init} \leq$ - [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*]: $(G \vdash \text{init} \leq n) \ s = (\text{card } (\text{nyinitcls } G \ s) \leq n)$

<proof>

lemma *All-init-leD*:

$\forall n::\text{nat}. G, (A::'a \ \text{triple set}) \vdash \{P \ \wedge. \ G \vdash \text{init} \leq n\} \ t \succ \ \{Q::'a \ \text{assn}\}$
 $\implies G, A \vdash \{P\} \ t \succ \ \{Q\}$

<proof>

Most General Triples and Formulas

constdefs

remember-init-state :: *state assn* (\doteq)
 $\doteq \equiv \lambda Y \ s \ Z. \ s = Z$

lemma *remember-init-state-def2* [*simp*]: $\doteq \ Y = \text{op} =$

<proof>

consts

MGF :: [*state assn*, *term*, *prog*] \Rightarrow *state triple* ($\{-\}$ \dashv $\{-\rightarrow\}$ [3,65,3] 62)
MGFn:: [*nat* , *term*, *prog*] \Rightarrow *state triple* ($\{=\!-\}$ \dashv $\{-\rightarrow\}$ [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* $G \implies G, \{\dot{=}\} \models \{\dot{=}\} t \succ \{G \rightarrow\}$

<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:*

$$G, A \vdash \{=:n\} t \succ \{G \rightarrow\} = G, A \vdash \{\dot{=} \wedge. G \vdash \text{init} \leq n\} t \succ \{\lambda Y s' s. G \vdash s - t \rightarrow (Y, s')\}$$

<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:*

$$G, (A::\text{state triple set}) \vdash \{=:n\} t \succ \{G \rightarrow\} \implies G, A \vdash \{(\lambda Y' s' s. s' = s \wedge P s) \wedge. G \vdash \text{init} \leq n\} t \succ \{(\lambda Y' s' s. G \vdash s - t \rightarrow (Y', s') \wedge P s) \wedge. G \vdash \text{init} \leq n\}$$

<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:* $G, A \vdash \{\text{Normal } \dot{=}\} t \succ \{G \rightarrow\} \implies$

$$G, (A::\text{state triple set}) \vdash \{\dot{=}::\text{state assn}\} t \succ \{G \rightarrow\}$$

<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:*

$$G, (A::\text{state triple set}) \vdash \{\text{Normal}((\lambda Y' s' s. s' = s \wedge \text{normal } s) \wedge. G \vdash \text{init} \leq n)\} t \succ \{\lambda Y s' s. G \vdash s - t \rightarrow (Y, s')\} \implies G, A \vdash \{=:n\} t \succ \{G \rightarrow\}$$

<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:*

$$(\exists T L C. (\text{prg} = G, \text{cls} = C, \text{lcl} = L) \vdash t::T) \longrightarrow G, (A::\text{state triple set}) \vdash \{=:n\} t \succ \{G \rightarrow\}$$

$$\implies G, A \vdash \{=:n\} t \succ \{G \rightarrow\}$$

<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 $G, (A :: \text{state triple set}) \vdash \{=:n\} \langle \text{Init } C \rangle_s \succ \{G \rightarrow\}$

<proof>

lemmas *MGFn-InitD = MGFn-Init [THEN MGFnD, THEN 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 $G, A \vdash \{=:n\} \langle \{ \text{acc } C, \text{stat } T, \text{mode} \} e \cdot \text{mn}(\{ pTs \wedge ps \}) \rangle_e \succ \{G \rightarrow\}$

<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

$unroll:: prog \Rightarrow label \Rightarrow expr \Rightarrow stmt \Rightarrow (state \times state) set$

$unroll\ G\ l\ e\ c \equiv \{(s,t). \exists v\ s1\ s2. \\ G \vdash s -e-\> v \rightarrow s1 \wedge the-Bool\ v \wedge normal\ s1 \wedge \\ G \vdash s1 -c\rightarrow s2 \wedge t=(abupd\ (absorb\ (Cont\ l))\ s2)\}$

lemma unroll-while:

assumes $unroll: (s, t) \in (unroll\ G\ l\ e\ c)^*$
and $eval-e: G \vdash t -e-\> v \rightarrow s'$
and $normal-termination: normal\ s' \longrightarrow \neg the-Bool\ v$
and $wt: (\prg=G, cls=C, lcl=L) \vdash e:: -T$
and $wf: wf-prog\ G$
shows $G \vdash s -l\cdot While(e)\ c \rightarrow s'$
 $\langle proof \rangle$

lemma MGFn-Loop:

assumes $mfg-e: G, (A::state\ triple\ set) \vdash \{=:n\} \langle e \rangle_e \succ \{G \rightarrow\}$
and $mfg-c: G, A \vdash \{=:n\} \langle c \rangle_s \succ \{G \rightarrow\}$
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 $mfg-init: G, A \vdash \{=:n\} \langle Init\ statDeclC \rangle_s \succ \{G \rightarrow\}$
and $mfg-e: G, A \vdash \{=:n\} \langle e \rangle_e \succ \{G \rightarrow\}$
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 $mfg-c1: G, A \vdash \{=:n\} \langle c1 \rangle_s \succ \{G \rightarrow\}$
and $mfg-c2: G, A \vdash \{=:n\} \langle c2 \rangle_s \succ \{G \rightarrow\}$
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 $wt-c: (\prg=G, cls=C, lcl=L) \vdash c:: \surd$
and $clsD: class\ G\ D = Some\ d$
and $wf: wf-prog\ G$
shows $\forall l. abrupt\ s2 \neq Some\ (Jump\ (Break\ l)) \wedge \\ abrupt\ s2 \neq Some\ (Jump\ (Cont\ l))$
 $\langle proof \rangle$

lemma *MGFn-Body*:

assumes *wf*: *wf-prog* *G*
and *mgf-init*: $G, A \vdash \{=:n\} \langle \text{Init } D \rangle_s \succ \{G \rightarrow\}$
and *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 *mgf-methods*:
 $\bigwedge n. \forall C \text{ sig}. G, (A :: \text{state triple set}) \vdash \{=:n\} \langle \text{Methd } C \ \text{sig} \rangle_e \succ \{G \rightarrow\}$
and *wf*: *wf-prog* *G*
shows $\bigwedge t. G, A \vdash \{=:n\} t \succ \{G \rightarrow\}$
 $\langle \text{proof} \rangle$

lemma *MGF-asm*:

$\llbracket \forall C \text{ sig}. \text{is-methd } G \ C \ \text{sig} \longrightarrow G, A \vdash \{\doteq\} \text{In1l } (\text{Methd } C \ \text{sig}) \succ \{G \rightarrow\}; \text{wf-prog } G \rrbracket$
 $\implies G, (A :: \text{state triple set}) \vdash \{\doteq\} t \succ \{G \rightarrow\}$
 $\langle \text{proof} \rangle$

nested version

lemma *nesting-lemma'* [*rule-format* (*no-asm*)]:

assumes *ax-derivs-asm*: $\bigwedge A \ ts. ts \subseteq A \implies P \ A \ ts$
and *MGF-nested-Methd*: $\bigwedge A \ pn. \forall b \in \text{bdy } pn. P \ (\text{insert } (\text{mgf-call } pn) \ A) \ \{\text{mgf } b\}$
 $\implies P \ A \ \{\text{mgf-call } pn\}$
and *MGF-asm*: $\bigwedge A \ t. \forall pn \in U. P \ A \ \{\text{mgf-call } pn\} \implies P \ A \ \{\text{mgf } t\}$
and *finU*: *finite* *U*
and *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 *ax-derivs-asm*: $\bigwedge A \ ts. ts \subseteq A \implies P \ A \ ts$
and *MGF-nested-Methd*: $\bigwedge A \ pn. \forall b \in \text{bdy } pn. P \ (\text{insert } (\text{mgf } (f \ pn)) \ A) \ \{\text{mgf } b\}$
 $\implies P \ A \ \{\text{mgf } (f \ pn)\}$
and *MGF-asm*: $\bigwedge A \ t. \forall pn \in U. P \ A \ \{\text{mgf } (f \ pn)\} \implies P \ A \ \{\text{mgf } t\}$

and *finU*: *finite* *U*

shows $P \ \{\} \ \{\text{mgf } t\}$

$\langle \text{proof} \rangle$

lemma *MGF-nested-Methd*: \llbracket

$G, \text{insert } (\{\text{Normal } \doteq\} \langle \text{Methd } C \ \text{sig} \rangle_e \succ \{G \rightarrow\}) \ A$
 $\vdash \{\text{Normal } \doteq\} \langle \text{body } G \ C \ \text{sig} \rangle_e \succ \{G \rightarrow\}$

$\rrbracket \implies G, A \vdash \{\text{Normal } \doteq\} \langle \text{Methd } C \ \text{sig} \rangle_e \succ \{G \rightarrow\}$

$\langle \text{proof} \rangle$

lemma *MGF-deriv*: *wf-prog* *G* $\implies G, (\{\} :: \text{state triple set}) \vdash \{\doteq\} t \succ \{G \rightarrow\}$

$\langle \text{proof} \rangle$

simultaneous version

lemma *MGF-simult-Method-lemma: finite ms* \implies
 $G, A \cup (\lambda(C, sig). \{Normal \doteq\} \langle Method\ C\ sig \rangle_e \succ \{G \rightarrow\}) \text{ 'ms}$
 $\vdash (\lambda(C, sig). \{Normal \doteq\} \langle body\ G\ C\ sig \rangle_e \succ \{G \rightarrow\}) \text{ 'ms} \implies$
 $G, A \vdash (\lambda(C, sig). \{Normal \doteq\} \langle Method\ C\ sig \rangle_e \succ \{G \rightarrow\}) \text{ 'ms}$
<proof>

lemma *MGF-simult-Method: wf-prog G* \implies
 $G, (\{\} :: state\ triple\ set) \vdash (\lambda(C, sig). \{Normal \doteq\} \langle Method\ C\ sig \rangle_e \succ \{G \rightarrow\})$
 $\text{ 'Collect (split (is-method G))}$
<proof>

corollaries

lemma *eval-to-evaln*: $\llbracket G \vdash s \text{ -} t \succ \rightarrow (Y', s'); type-ok\ G\ t\ s; wf-prog\ G \rrbracket$
 $\implies \exists n. G \vdash s \text{ -} t \succ \text{-} n \rightarrow (Y', s')$
<proof>

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 $G, (\{\} :: state\ triple\ set) \vdash \{P :: state\ assn\} t \succ \{Q\}$
<proof>

theorem *ax-complete*:
assumes *wf*: *wf-prog G*
and *valid*: $G, \{\} \models \{P :: state\ assn\} t \succ \{Q\}$
shows $G, (\{\} :: state\ triple\ set) \vdash \{P\} t \succ \{Q\}$
<proof>

end

Chapter 25

Ax**E**xample

64 Example of a proof based on the Bali axiomatic semantics

theory *AxExample* **imports** *AxSem Example* **begin**

constdefs

```

arr-inv :: st ⇒ bool
arr-inv ≡ λs. ∃ obj a T el. globs s (Stat Base) = Some obj ∧
  values obj (Inl (arr, Base)) = Some (Addr a) ∧
  heap s a = Some (|tag=Arr T 2,values=el)

```

lemma *arr-inv-new-obj*:

```

∧ a. [ arr-inv s; new-Addr (heap s) = Some a ] ⇒ arr-inv (gupd (Inl a ↦ x) s)
⟨proof⟩

```

lemma *arr-inv-set-locals* [*simp*]: *arr-inv (set-locals l s) = arr-inv s*

⟨*proof*⟩

lemma *arr-inv-gupd-Stat* [*simp*]:

```

Base ≠ C ⇒ arr-inv (gupd (Stat C ↦ obj) s) = arr-inv s
⟨proof⟩

```

lemma *ax-inv-lupd* [*simp*]: *arr-inv (lupd (x ↦ y) s) = arr-inv s*

⟨*proof*⟩

declare *split-if-asm* [*split del*]

declare *lvar-def* [*simp*]

⟨*ML*⟩

theorem *ax-test*: *tprg*,({::'a triple set}) ⊢

```

{ Normal (λY s Z::'a. heap-free four s ∧ ¬initd Base s ∧ ¬ initd Ext s) }

```

```

.test [Class Base].

```

```

{ λY s Z. abrupt s = Some (Xcpt (Std IndOutBound)) }

```

⟨*proof*⟩

lemma *Loop-Xcpt-benchmark*:

```

Q = (λY (x,s) Z. x ≠ None → the-Bool (the (locals s i))) ⇒

```

```

G,({::'a triple set}) ⊢ { Normal (λY s Z::'a. True) }

```

```

.lab1 • While (Lit (Bool True)) (If (Acc (LVar i)) (Throw (Acc (LVar xcpt))) Else
  (Expr (Ass (LVar i) (Acc (LVar j))))). { Q }

```

⟨*proof*⟩

end