

Java Source and Bytecode Formalizations in Isabelle: Bali

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

December 17, 2025

Contents

1	Overview	5
2	Basis	9
1	Definitions extending HOL as logical basis of Bali	9
3	Table	13
1	Abstract tables and their implementation as lists	13
4	Name	21
1	Java names	21
5	Value	23
1	Java values	23
6	Type	25
1	Java types	25
7	Term	27
1	Java expressions and statements	27
8	Decl	35
1	Field, method, interface, and class declarations, whole Java programs	35
2	Modifier	35
3	Declaration (base "class" for member,interface and class declarations	37
4	Member (field or method)	37
5	Field	37
6	Method	37
7	Interface	39
8	Class	40
9	TypeRel	47
1	The relations between Java types	47
10	DeclConcepts	55
1	Advanced concepts on Java declarations like overriding, inheritance, dynamic method lookup	55
2	accessibility of types (cf. 6.6.1)	55
3	accessibility of members	56
4	imethds	75
5	accimethd	75
6	methd	76
7	accmethd	77
8	dynmethd	77

9	dynlookup	79
10	fields	79
11	accfield	80
12	is methd	80
11	WellType	83
1	Well-typedness of Java programs	83
12	DefiniteAssignment	95
1	Definite Assignment	95
2	Very restricted calculation fallback calculation	97
3	Analysis of constant expressions	98
4	Main analysis for boolean expressions	99
5	Lifting set operations to range of tables (map to a set)	100
13	WellForm	109
1	Well-formedness of Java programs	109
2	accessibility concerns	125
14	State	129
1	State for evaluation of Java expressions and statements	129
2	access	132
3	memory allocation	133
4	initialization	133
5	update	133
6	update	137
15	Eval	141
1	Operational evaluation (big-step) semantics of Java expressions and statements	141
16	Example	157
1	Example Bali program	157
17	Conform	171
1	Conformance notions for the type soundness proof for Java	171
18	DefiniteAssignmentCorrect	179
1	Correctness of Definite Assignment	179
19	TypeSafe	187
1	The type soundness proof for Java	187
2	accessibility	194
3	Ideas for the future	199
20	Evaln	201
1	Operational evaluation (big-step) semantics of Java expressions and statements	201
21	Trans	209
22	AxSem	215
1	Axiomatic semantics of Java expressions and statements (see also Eval.thy) .	215
2	peek-and	216
3	assn-supd	216
4	supd-assn	216

5	subst-res	217
6	subst-Bool	217
7	peek-res	217
8	ign-res	218
9	peek-st	218
10	ign-res-eq	218
11	RefVar	219
12	allocation	219

23 AxSound 231

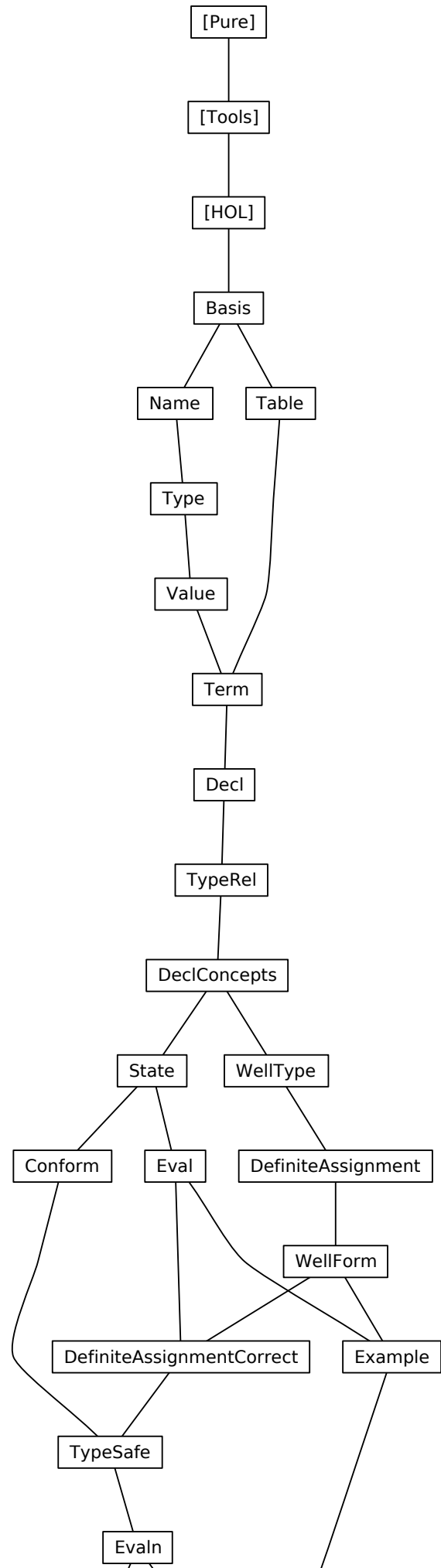
1	Soundness proof for Axiomatic semantics of Java expressions and statements	231
---	--	-----

24 AxCompl 235

1	Completeness proof for Axiomatic semantics of Java expressions and statements	235
---	---	-----

25 AxExample 243

1	Example of a proof based on the Bali axiomatic semantics	243
---	--	-----



Chapter 1

Overview

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

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

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

The following features are missing in Bali wrt. JavaCard:

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

- Inner Classes

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

Overview of the theories:

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

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

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

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

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

Term JavaCard terms. Variables, expressions and statements.

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

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

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

WellType Typesystem on the JavaCard term level.

DefiniteAssignment The definite assignment analysis on the JavaCard term level.

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

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

Eval Operational (big step) semantics for JavaCard.

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

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

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

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

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

Trans A smallstep operational semantics for JavaCard.

AxSem An axiomatic semantics (Hoare logic) for JavaCard.

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

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

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

Chapter 2

Basis

1 Definitions extending HOL as logical basis of Bali

```
theory Basis
imports Main
begin
```

misc

$\langle ML \rangle$

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

$\langle ML \rangle$

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

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

```
lemma Collect-split-eq:  $\{p. P (case-prod f p)\} = \{(a,b). P (f a b)\}$ 
   $\langle proof \rangle$ 
```

```
lemma subset-insertD:  $A \subseteq insert\ x\ B \implies A \subseteq B \wedge x \notin A \vee (\exists B'. A = insert\ x\ B' \wedge B' \subseteq B)$ 
   $\langle proof \rangle$ 
```

```
abbreviation nat3 :: nat  $\langle \text{3} \rangle$  where 3  $\equiv$  Suc 2
```

```
abbreviation nat4 :: nat  $\langle \text{4} \rangle$  where 4  $\equiv$  Suc 3
```

```
lemma irrefl-tranclI':  $r^{-1} \cap r^+ = \{\}$   $\implies \forall x. (x, x) \notin r^+$ 
   $\langle proof \rangle$ 
```

```
lemma trancl-rtrancl-trancl:  $\llbracket (x, y) \in r^+; (y, z) \in r^* \rrbracket \implies (x, z) \in r^+$ 
   $\langle proof \rangle$ 
```

```
lemma rtrancl-into-trancl3:  $\llbracket (a, b) \in r^*; a \neq b \rrbracket \implies (a, b) \in r^+$ 
   $\langle proof \rangle$ 
```

```
lemma rtrancl-into-rtrancl2:  $\llbracket (a, b) \in r; (b, c) \in r^* \rrbracket \implies (a, c) \in r^*$ 
   $\langle proof \rangle$ 
```

lemma triangle-lemma:

assumes unique: $\bigwedge a\ b\ c. \llbracket (a,b) \in r; (a,c) \in r \rrbracket \implies b = c$

and ax: $(a,x) \in r^*$ **and** ay: $(a,y) \in r^*$

shows $(x,y) \in r^* \vee (y,x) \in r^*$

$\langle proof \rangle$

lemma *rtranc1-cases*:

assumes $(a,b) \in r^*$
obtains $(\text{Refl})\ a = b$
 $\mid (\text{Tranc1})\ (a,b) \in r^+$
 $\langle \text{proof} \rangle$

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

lemma *finite-SetCompr2*:

finite $\{f\ y\ x \mid x\ y. P\ y\}$ **if** *finite* $(\text{Collect } P)$
 $\forall y. P\ y \longrightarrow \text{finite } (\text{range } (f\ y))$
 $\langle \text{proof} \rangle$

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

pairs

lemma *surjective-pairing5*:

$p = (\text{fst } p, \text{fst } (\text{snd } p), \text{fst } (\text{snd } (\text{snd } p)), \text{fst } (\text{snd } (\text{snd } (\text{snd } p))),$
 $\text{snd } (\text{snd } (\text{snd } (\text{snd } p))))$
 $\langle \text{proof} \rangle$

lemma *fst-splitE* $[\text{elim!}]$:

assumes $\text{fst } s' = x'$
obtains $x\ s$ **where** $s' = (x,s)$ **and** $x = x'$
 $\langle \text{proof} \rangle$

lemma *fst-in-set-lemma*: $(x, y) \in \text{set } l \Longrightarrow x \in \text{fst } ' \text{set } l$
 $\langle \text{proof} \rangle$

quantifiers

lemma *All-Ex-refl-eq2* $[\text{simp}]$: $(\forall x. (\exists b. x = f\ b \wedge Q\ b) \longrightarrow P\ x) = (\forall b. Q\ b \longrightarrow P\ (f\ b))$
 $\langle \text{proof} \rangle$

lemma *ex-ex-miniscope1* $[\text{simp}]$: $(\exists w\ v. P\ w\ v \wedge Q\ v) = (\exists v. (\exists w. P\ w\ v) \wedge Q\ v)$
 $\langle \text{proof} \rangle$

lemma *ex-miniscope2* $[\text{simp}]$: $(\exists v. P\ v \wedge Q \wedge R\ v) = (Q \wedge (\exists v. P\ v \wedge R\ v))$
 $\langle \text{proof} \rangle$

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* $[\text{simp}]$: $(\forall x. (\exists b. x = f\ b) \longrightarrow P\ x) = (\forall b. P\ (f\ b))$
 $\langle \text{proof} \rangle$

sums

notation *case-sum* $(\text{infixr } ' (+)' 80)$

primrec *the-Inl* $:: 'a + 'b \Rightarrow 'a$
where *the-Inl* $(\text{Inl } a) = a$

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

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

datatype (*'a*, *'b*, *'c*) *sum3* = *In1* *'a* | *In2* *'b* | *In3* *'c*

primrec *the-In1* :: (*'a*, *'b*, *'c*) *sum3* \Rightarrow *'a*
where *the-In1* (*In1* *a*) = *a*

primrec *the-In2* :: (*'a*, *'b*, *'c*) *sum3* \Rightarrow *'b*
where *the-In2* (*In2* *b*) = *b*

primrec *the-In3* :: (*'a*, *'b*, *'c*) *sum3* \Rightarrow *'c*
where *the-In3* (*In3* *c*) = *c*

abbreviation *In1l* :: *'al* \Rightarrow (*'al* + *'ar*, *'b*, *'c*) *sum3*
where *In1l* *e* \equiv *In1* (*Inl* *e*)

abbreviation *In1r* :: *'ar* \Rightarrow (*'al* + *'ar*, *'b*, *'c*) *sum3*
where *In1r* *c* \equiv *In1* (*Inr* *c*)

abbreviation *the-In1l* :: (*'al* + *'ar*, *'b*, *'c*) *sum3* \Rightarrow *'al*
where *the-In1l* \equiv *the-Inl* \circ *the-In1*

abbreviation *the-In1r* :: (*'al* + *'ar*, *'b*, *'c*) *sum3* \Rightarrow *'ar*
where *the-In1r* \equiv *the-Inr* \circ *the-In1*

$\langle ML \rangle$

quantifiers for option type

syntax

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

syntax (symbols)

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

syntax-consts

-*Oall* \Leftarrow *Ball* **and**
 -*Oex* \Leftarrow *Bex*

translations

$\forall x \in A: P \Leftrightarrow \forall x \in \text{CONST set-option } A. P$
 $\exists x \in A: P \Leftrightarrow \exists x \in \text{CONST set-option } A. P$

Special map update

Deemed too special for theory Map.

definition *chg-map* :: (*'b* \Rightarrow *'b*) \Rightarrow *'a* \Rightarrow (*'a* \rightarrow *'b*) \Rightarrow (*'a* \rightarrow *'b*)
where *chg-map* *f* *a* *m* = (*case* *m* *a* of *None* \Rightarrow *m* | *Some* *b* \Rightarrow *m*(*a* \mapsto *f* *b*))

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

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

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

unique association lists

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

lemma $uniqueD$: $unique\ l \Longrightarrow (x, y) \in set\ l \Longrightarrow (x', y') \in set\ l \Longrightarrow x = x' \Longrightarrow y = y'$
 $\langle proof \rangle$

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

lemma $unique-Cons$ [simp]: $unique\ ((x,y)\#l) = (unique\ l \wedge (\forall y. (x,y) \notin set\ l))$
 $\langle proof \rangle$

lemma $unique-ConsD$: $unique\ (x\#xs) \Longrightarrow unique\ xs$
 $\langle proof \rangle$

lemma $unique-single$ [simp]: $\bigwedge p. unique\ [p]$
 $\langle proof \rangle$

lemma $unique-append$ [rule-format (no-asm)]: $unique\ l' \Longrightarrow unique\ l \Longrightarrow$
 $(\forall (x,y) \in set\ l. \forall (x',y') \in set\ l'. x' \neq x \longrightarrow unique\ (l @ l'))$
 $\langle proof \rangle$

lemma $unique-map-inj$: $unique\ l \Longrightarrow inj\ f \Longrightarrow unique\ (map\ (\lambda(k,x). (f\ k, g\ k\ x))\ l)$
 $\langle proof \rangle$

lemma $map-of-SomeI$: $unique\ l \Longrightarrow (k, x) \in set\ l \Longrightarrow map-of\ l\ k = Some\ x$
 $\langle proof \rangle$

list patterns

definition $lsplit :: [['a, 'a\ list] \Rightarrow 'b, 'a\ list] \Rightarrow 'b$
where $lsplit = (\lambda f\ l. f\ (hd\ l)\ (tl\ l))$

list patterns – extends pre-defined type "pttrn" used in abstractions

syntax

$-lpttrn :: [pttrn, pttrn] \Rightarrow pttrn \quad (\langle \# / \rangle [901, 900]\ 900)$

syntax-consts

$-lpttrn \rightleftharpoons lsplit$

translations

$\lambda y\ \# x\ \# xs. b \rightleftharpoons CONST\ lsplit\ (\lambda y\ x\ \# xs. b)$
 $\lambda x\ \# xs. b \rightleftharpoons CONST\ lsplit\ (\lambda x\ xs. b)$

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

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

end

Chapter 3

Table

1 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'

type-synonym (*'a*, *'b*) *table* — table with key type *'a* and contents type *'b*
= *'a* \rightarrow *'b*

type-synonym (*'a*, *'b*) *tables* — non-unique table with key *'a* and contents *'b*
= *'a* \Rightarrow *'b set*

map of / table of

abbreviation

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

translations

(*type*) (*'a*, *'b*) *table* \leq (*type*) *'a* \rightarrow *'b*

lemma *map-add-find-left[simp]*: $n\ k = \text{None} \implies (m\ ++\ n)\ k = m\ k$
<proof>

Conditional Override

definition *cond-override* :: (*'b* \Rightarrow *'b* \Rightarrow *bool*) \Rightarrow (*'a*, *'b*) *table* \Rightarrow (*'a*, *'b*) *table* \Rightarrow (*'a*, *'b*) *table* **where**

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

```

cond-override cond old new =
  (λk.
    (case new k of
      None      ⇒ old k
    | Some new-val ⇒ (case old k of
        None      ⇒ Some new-val
      | Some old-val ⇒ (if cond new-val old-val
          then Some new-val
        else Some old-val))))

```

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

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

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

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

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

lemma *dom-cond-override*: *dom (cond-override C s t) ⊆ dom s ∪ dom t*
 ⟨proof⟩

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

Filter on Tables

definition *filter-tab* :: (*'a* ⇒ *'b* ⇒ bool) ⇒ (*'a*, *'b*) table ⇒ (*'a*, *'b*) table
 where
filter-tab c t = (λk. (case t k of
 None ⇒ None
 | Some x ⇒ if c k x then Some x else None))

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

lemma *filter-tab-True*[simp]: *filter-tab (λx y. True) t = t*
 ⟨proof⟩

lemma *filter-tab-False*[simp]: *filter-tab (λx y. False) t = Map.empty*
 ⟨proof⟩

lemma *filter-tab-ran-subset*: *ran (filter-tab c t) ⊆ ran t*
 ⟨proof⟩

lemma *filter-tab-range-subset*: *range (filter-tab c t) ⊆ range t ∪ {None}*
 ⟨proof⟩

lemma *finite-range-filter-tab*:

$\text{finite } (\text{range } t) \implies \text{finite } (\text{range } (\text{filter-tab } c \ t))$
 $\langle \text{proof} \rangle$

lemma *filter-tab-SomeD[dest!]*:

$\text{filter-tab } c \ t \ k = \text{Some } x \implies (t \ k = \text{Some } x) \wedge c \ k \ x$
 $\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

lemma *filter-tab-all-True*:

$\forall k \ y. t \ k = \text{Some } y \longrightarrow p \ k \ y \implies \text{filter-tab } p \ t = t$
 $\langle \text{proof} \rangle$

lemma *filter-tab-all-True-Some*:

$\llbracket \forall k \ y. t \ k = \text{Some } y \longrightarrow p \ k \ y; t \ k = \text{Some } v \rrbracket \implies \text{filter-tab } p \ t \ k = \text{Some } v$
 $\langle \text{proof} \rangle$

lemma *filter-tab-all-False*:

$\forall k \ y. t \ k = \text{Some } y \longrightarrow \neg p \ k \ y \implies \text{filter-tab } p \ t = \text{Map.empty}$
 $\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

lemma *finite-dom-filter-tab*:

$\text{finite } (\text{dom } t) \implies \text{finite } (\text{dom } (\text{filter-tab } C \ t))$
 $\langle \text{proof} \rangle$

lemma *filter-tab-weaken*:

$\llbracket \forall a \in t \ k. \exists b \in s \ k. P \ a \ b; \bigwedge k \ x \ y. \llbracket t \ k = \text{Some } x; s \ k = \text{Some } y \rrbracket \implies \text{cond } k \ x \longrightarrow \text{cond } k \ y \rrbracket \implies \forall a \in \text{filter-tab } \text{cond } t \ k. \exists b \in \text{filter-tab } \text{cond } s \ k. P \ a \ b$
 $\langle \text{proof} \rangle$

lemma *cond-override-filter*:

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

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*:

$\llbracket (\forall (x,y) \in \text{set } l. P \ x \ y); x \in \text{set-option } (\text{table-of } l \ x) \rrbracket \implies P \ x \ x$
 <proof>

declare *map-of-SomeD* [elim]

lemma *table-of-Some-in-set*:
 $\text{table-of } l \ k = \text{Some } x \implies (k,x) \in \text{set } l$
 <proof>

lemma *set-get-eq*:
 $\text{unique } l \implies (k, \text{the } (\text{table-of } l \ k)) \in \text{set } l = (\text{table-of } l \ k \neq \text{None})$
 <proof>

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

lemma *table-of-mapconst-SomeI*:
 $\llbracket \text{table-of } t \ k = \text{Some } y'; \text{snd } y=y'; \text{fst } y=c \rrbracket \implies$
 $\text{table-of } (\text{map } (\lambda(k,x). (k,c,x)) \ t) \ k = \text{Some } y$
 <proof>

lemma *table-of-mapconst-NoneI*:
 $\llbracket \text{table-of } t \ k = \text{None} \rrbracket \implies$
 $\text{table-of } (\text{map } (\lambda(k,x). (k,c,x)) \ t) \ k = \text{None}$
 <proof>

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

lemma *table-of-map-SomeI*: $\text{table-of } t \ k = \text{Some } x \implies$
 $\text{table-of } (\text{map } (\lambda(k,x). (k, f \ x)) \ t) \ k = \text{Some } (f \ x)$
 <proof>

lemma *table-of-remap-SomeD*:
 $\text{table-of } (\text{map } (\lambda((k,k'),x). (k,(k',x))) \ t) \ k = \text{Some } (k',x) \implies$
 $\text{table-of } t \ (k, k') = \text{Some } x$
 <proof>

lemma *table-of-mapf-Some*:
 $\forall x \ y. f \ x = f \ y \longrightarrow x = y \implies$
 $\text{table-of } (\text{map } (\lambda(k,x). (k, f \ x)) \ t) \ k = \text{Some } (f \ x) \implies \text{table-of } t \ k = \text{Some } x$
 <proof>

lemma *table-of-mapf-SomeD* [dest!]:
 $\text{table-of } (\text{map } (\lambda(k,x). (k, f \ x)) \ t) \ k = \text{Some } z \implies (\exists y \in \text{table-of } t \ k. z=f \ y)$
 <proof>

lemma *table-of-mapf-NoneD* [dest!]:
 $\text{table-of } (\text{map } (\lambda(k,x). (k, f \ x)) \ t) \ k = \text{None} \implies (\text{table-of } t \ k = \text{None})$
 <proof>

lemma *table-of-mapkey-SomeD* [dest!]:
 $\text{table-of } (\text{map } (\lambda(k,x). ((k,C),x)) \ t) \ (k,D) = \text{Some } x \implies C = D \wedge \text{table-of } t \ k = \text{Some } x$
 <proof>

lemma *table-of-mapkey-SomeD2* [dest!]:
 $\text{table-of } (\text{map } (\lambda(k,x). ((k,C),x)) \ t) \ ek = \text{Some } x \implies$
 $C = \text{snd } ek \wedge \text{table-of } t \ (\text{fst } ek) = \text{Some } x$
 <proof>

lemma *table-append-Some-iff*: $\text{table-of } (xs@ys) \ k = \text{Some } z =$
 $(\text{table-of } xs \ k = \text{Some } z \vee (\text{table-of } xs \ k = \text{None} \wedge \text{table-of } ys \ k = \text{Some } z))$
 $\langle \text{proof} \rangle$

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

definition *Un-tables* :: $('a, 'b) \text{ tables set} \Rightarrow ('a, 'b) \text{ tables}$
where *Un-tables* $ts = (\lambda k. \bigcup_{t \in ts.} t \ k)$

definition *overrides-t* :: $('a, 'b) \text{ tables} \Rightarrow ('a, 'b) \text{ tables} \Rightarrow ('a, 'b) \text{ tables}$
 $(\text{infixl } \langle \oplus \oplus \rangle \ 100)$
where $s \oplus \oplus t = (\lambda k. \text{if } t \ k = \{\} \text{ then } s \ k \text{ else } t \ k)$

definition
hidings-entails :: $('a, 'b) \text{ tables} \Rightarrow ('a, 'c) \text{ tables} \Rightarrow ('b \Rightarrow 'c \Rightarrow \text{bool}) \Rightarrow \text{bool}$
 $(\langle \text{hidings} - \text{entails} \rangle \ 20)$
where $(t \ \text{hidings } s \ \text{entails } R) = (\forall k. \forall x \in t \ k. \forall y \in s \ k. R \ x \ y)$

definition
— variant for unique table:
hiding-entails :: $('a, 'b) \text{ table} \Rightarrow ('a, 'c) \text{ table} \Rightarrow ('b \Rightarrow 'c \Rightarrow \text{bool}) \Rightarrow \text{bool}$
 $(\langle \text{hiding} - \text{entails} \rangle \ 20)$
where $(t \ \text{hiding } s \ \text{entails } R) = (\forall k. \forall x \in t \ k: \forall y \in s \ k: R \ x \ y)$

definition
— variant for a unique table and conditional overriding:
cond-hiding-entails :: $('a, 'b) \text{ table} \Rightarrow ('a, 'c) \text{ table}$
 $\Rightarrow ('b \Rightarrow 'c \Rightarrow \text{bool}) \Rightarrow ('b \Rightarrow 'c \Rightarrow \text{bool}) \Rightarrow \text{bool}$
 $(\langle \text{hiding} - \text{under} - \text{entails} \rangle \ 20)$
where $(t \ \text{hiding } s \ \text{under } C \ \text{entails } R) = (\forall k. \forall x \in t \ k: \forall y \in s \ k: C \ x \ y \longrightarrow R \ x \ y)$

Untables

lemma *Un-tablesI* [intro]: $t \in ts \implies x \in t \ k \implies x \in \text{Un-tables } ts \ k$
 $\langle \text{proof} \rangle$

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

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

overrides

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

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

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

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*:
 $t \text{ hiding } s \text{ entails } R \implies t \ k = \text{Some } x \implies s \ k = \text{Some } y \implies R \ x \ y$
 ⟨proof⟩

lemma *empty-hiding-entails* [simp]: $\text{Map.empty hiding } s \text{ entails } R$
 ⟨proof⟩

lemma *hiding-empty-entails* [simp]: $t \text{ hiding } \text{Map.empty} \text{ entails } R$
 ⟨proof⟩

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]: $\text{Map.empty hiding } s \text{ under } C \text{ entails } R$
 ⟨proof⟩

lemma *cond-hiding-empty-entails*[simp]: $t \text{ hiding } \text{Map.empty} \text{ under } C \text{ entails } R$
 ⟨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* [intro!]: $t \text{ hidings } (\lambda k. \{\}) \text{ entails } R$
 ⟨proof⟩

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

primrec *atleast-free* :: $('a \multimap 'b) \Rightarrow \text{nat} \Rightarrow \text{bool}$

where

$\text{atleast-free } m \ 0 = \text{True}$
 $\mid \text{atleast-free-Suc: atleast-free } m \ (\text{Suc } n) = (\exists a. m \ a = \text{None} \wedge (\forall b. \text{atleast-free } (m(a \mapsto b)) \ n))$

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

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

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

lemma *atleast-free-SucD-lemma* [rule-format (no-asm)]:
 $\forall m \ a. m \ a = \text{None} \longrightarrow (\forall c. \text{atleast-free } (m(a \mapsto c)) \ n) \longrightarrow$
 $(\forall b \ d. a \neq b \longrightarrow \text{atleast-free } (m(b \mapsto d)) \ n)$
 ⟨proof⟩

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

lemma *atleast-free-SucD*: *atleast-free* *h* (*Suc* *n*) ==> *atleast-free* (*h*(*a*|->*b*)) *n*
 <*proof*>

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

end

Chapter 4

Name

1 Java names

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

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

typeddecl *pname* — package name

typeddecl *mname* — method name

typeddecl *vname* — variable or field name

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

datatype *ename* — expression name

= *VName vname*

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

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

= *ENAME ename*

| *This*

abbreviation *VName* :: *vname* \Rightarrow *lname*

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

abbreviation *Result* :: *lname*

where *Result* == *ENAME* *Res*

datatype *xname* — names of standard exceptions

= *Throwable*

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

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

lemma *xn-cases*:

$xn = \text{Throwable} \vee xn = \text{NullPointerException} \vee$

$xn = \text{OutOfMemory} \vee xn = \text{ClassCast} \vee$

$xn = \text{NegativeArraySize} \vee xn = \text{IndexOutOfBoundsException} \vee xn = \text{ArrayStore}$

$\langle \text{proof} \rangle$

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

= *Object'*

| *SXcpt'* *xname*

| *TName* *tname*

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

pid :: *pname*

tid :: *tname*

```

class has-pname =
  fixes pname :: 'a  $\Rightarrow$  pname

instantiation pname :: has-pname
begin

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

instance  $\langle$ proof $\rangle$ 

end

class has-tname =
  fixes tname :: 'a  $\Rightarrow$  tname

instantiation tname :: has-tname
begin

definition
  tname-tname-def: tname (t::tname) = t

instance  $\langle$ proof $\rangle$ 

end

definition
  qname-qname-def: qname (q::'a qname-scheme) = q

translations
  (type) qname <= (type) ( $\lfloor$ pid::pname,tid::tname $\rfloor$ )
  (type) 'a qname-scheme <= (type) ( $\lfloor$ pid::pname,tid::tname,...::'a $\rfloor$ )

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

definition
  Object :: qname
  where Object = ( $\lfloor$ pid = java-lang, tid = Object $\rfloor$ )

definition SXcpt :: xname  $\Rightarrow$  qname
  where SXcpt = ( $\lambda x.$  ( $\lfloor$ 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

1 Java values

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

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

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

primrec *the-Bool* :: *val* \Rightarrow *bool*
 where *the-Bool* (*Bool b*) = *b*

primrec *the-Intg* :: *val* \Rightarrow *int*
 where *the-Intg* (*Intg i*) = *i*

primrec *the-Addr* :: *val* \Rightarrow *loc*
 where *the-Addr* (*Addr a*) = *a*

type-synonym *dyn-ty* = *loc* \Rightarrow *ty option*

primrec *typeof* :: *dyn-ty* \Rightarrow *val* \Rightarrow *ty option*
where
 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* :: *prim-ty* \Rightarrow *val* — default value for primitive types
where
 defpval Void = *Unit*
 | *defpval Boolean* = *Bool False*
 | *defpval Integer* = *Intg 0*

primrec *default-val* :: *ty* \Rightarrow *val* — default value for all types
where
 default-val (PrimT pt) = *defpval pt*
 | *default-val (RefT r)* = *Null*

end

Chapter 6

Type

1 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

abbreviation *NT* == *RefT NullT*
abbreviation *Iface I* == *RefT (IfaceT I)*
abbreviation *Class C* == *RefT (ClassT C)*
abbreviation *Array* :: *ty* \Rightarrow *ty* ($\langle \cdot \rangle$ [90] 90)
where *T.[]* == *RefT (ArrayT T)*

definition
the-Class :: *ty* \Rightarrow *qname*
where *the-Class T* = (*SOME C. T* = *Class C*)

lemma *the-Class-eq [simp]*: *the-Class (Class C)* = *C*
 $\langle proof \rangle$

end

Chapter 7

Term

1 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

type-synonym *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 execption 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 programms

type-synonym
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
 (*type*) *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
 (*type*) *sig* <= (*type*) (*name*::*mname*,*parTs*::*ty list*)
 (*type*) *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

and <i>stmt</i>	
= <i>Skip</i>	— empty statement
<i>Expr expr</i>	— expression statement
<i>Lab jump stmt</i>	($\leftarrow \rightarrow$ [99,66]66) — labeled statement; handles break
<i>Comp stmt stmt</i>	($\leftarrow ; \rightarrow$ [66,65]65)
<i>If' expr stmt stmt</i>	($\leftarrow \text{If}'(-) - \text{Else} \rightarrow$ [80,79,79]70)
<i>Loop label expr stmt</i>	($\leftarrow \cdot \text{While}'(-) \rightarrow$ [99,80,79]70)
<i>Jmp jump</i>	— break, continue, return
<i>Throw expr</i>	
<i>TryC stmt qname vname stmt</i>	($\leftarrow \text{Try} - \text{Catch}'(-) \rightarrow$ [79,99,80,79]70)
— <i>Try c1 Catch(C vn) c2</i>	
— <i>c1</i> :	block where exception may be thrown
— <i>C</i> :	exception class to catch
— <i>vn</i> :	local name for exception used in <i>c2</i>
— <i>c2</i> :	block to execute when exception is caught
<i>Fin stmt stmt</i>	($\leftarrow \text{Finally} \rightarrow$ [79,79]70)
<i>FinA abrupt stmt</i>	— Save abrupt of first statement
	— technical term for smallstep sem.)
<i>Init qname</i>	— class initialization

datatype-compat *var expr stmt*

The expressions `Method` 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.

type-synonym *term* = (*expr+stmt,var,expr list*) *sum3*
translations

(type) sig <= (type) mname × ty list
 (type) term <= (type) (expr+stmt,var,expr list) sum3

abbreviation this :: expr
 where this == Acc (LVar This)

abbreviation LAcc :: vname ⇒ expr (⟨!!⟩)
 where !!v == Acc (LVar (ENam (VNam v)))

abbreviation
 LAss :: vname ⇒ expr ⇒ stmt (⟨-:=⟩ [90,85] 85)
 where v:=e == Expr (Ass (LVar (ENam (VNam v))) e)

abbreviation
 Return :: expr ⇒ stmt
 where Return e == Expr (Ass (LVar (ENam Res)) e);; Jmp Ret — Res := e;; Jmp Ret

abbreviation
 StatRef :: ref-ty ⇒ expr
 where StatRef rt == Cast (RefT rt) (Lit Null)

definition
 is-stmt :: term ⇒ bool
 where 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.

abbreviation (input)
 expr-inj-term :: expr ⇒ term (⟨⟨-⟩_e⟩ 1000)
 where ⟨e⟩_e == In1l e

abbreviation (input)
 stmt-inj-term :: stmt ⇒ term (⟨⟨-⟩_s⟩ 1000)
 where ⟨c⟩_s == In1r c

abbreviation (input)
 var-inj-term :: var ⇒ term (⟨⟨-⟩_v⟩ 1000)
 where ⟨v⟩_v == In2 v

abbreviation (input)
 lst-inj-term :: expr list ⇒ term (⟨⟨-⟩_l⟩ 1000)
 where ⟨es⟩_l == In3 es

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

class inj-term =
 fixes inj-term:: 'a ⇒ 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 abbreviations above are used as bridge between the different conventions.

instantiation stmt :: inj-term

begin

definition

stmt-inj-term-def: $\langle c::stmt \rangle = In1r\ c$

instance $\langle proof \rangle$

end

lemma *stmt-inj-term-simp*: $\langle c::stmt \rangle = In1r\ c$
 $\langle proof \rangle$

lemma *stmt-inj-term [iff]*: $\langle x::stmt \rangle = \langle y \rangle \equiv x = y$
 $\langle proof \rangle$

instantiation *expr :: inj-term*

begin

definition

expr-inj-term-def: $\langle e::expr \rangle = In1l\ e$

instance $\langle proof \rangle$

end

lemma *expr-inj-term-simp*: $\langle e::expr \rangle = In1l\ e$
 $\langle proof \rangle$

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

instantiation *var :: inj-term*

begin

definition

var-inj-term-def: $\langle v::var \rangle = In2\ v$

instance $\langle proof \rangle$

end

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

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

class *expr-of* =
fixes *expr-of* :: $'a \Rightarrow expr$

instantiation *expr :: expr-of*
begin

definition

expr-of = $(\lambda(e::expr). e)$

instance $\langle proof \rangle$

end

instantiation *list* :: (*expr-of*) *inj-term*
begin

definition

$\langle es::'a\ list \rangle = In3\ (map\ expr-of\ es)$

instance $\langle proof \rangle$

end

lemma *expr-list-inj-term-def*:

$\langle es::expr\ list \rangle \equiv In3\ es$
 $\langle proof \rangle$

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

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

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]: $\langle t::stmt \rangle \neq \langle w::expr \rangle$
 $\langle proof \rangle$

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

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

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

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

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

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

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

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

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

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

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

lemma *term-cases*:

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

Evaluation of unary operations

primrec *eval-unop* :: *unop* \Rightarrow *val* \Rightarrow *val*

where

| *eval-unop* *UPlus* *v* = *Intg* (*the-Intg* *v*)
 | *eval-unop* *UMinus* *v* = *Intg* (\neg (*the-Intg* *v*))
 | *eval-unop* *UBitNot* *v* = *Intg* 42 — FIXME: Not yet implemented
 | *eval-unop* *UNot* *v* = *Bool* (\neg *the-Bool* *v*)

Evaluation of binary operations

primrec *eval-binop* :: *binop* \Rightarrow *val* \Rightarrow *val* \Rightarrow *val*

where

| *eval-binop* *Mul* *v1* *v2* = *Intg* ((*the-Intg* *v1*) * (*the-Intg* *v2*))
 | *eval-binop* *Div* *v1* *v2* = *Intg* ((*the-Intg* *v1*) div (*the-Intg* *v2*))
 | *eval-binop* *Mod* *v1* *v2* = *Intg* ((*the-Intg* *v1*) mod (*the-Intg* *v2*))
 | *eval-binop* *Plus* *v1* *v2* = *Intg* ((*the-Intg* *v1*) + (*the-Intg* *v2*))
 | *eval-binop* *Minus* *v1* *v2* = *Intg* ((*the-Intg* *v1*) - (*the-Intg* *v2*))
 — Be aware of the explicit coercion of the shift distance to nat
 | *eval-binop* *LShift* *v1* *v2* = *Intg* ((*the-Intg* *v1*) * ($2^{\neg(\text{nat } (\text{the-Intg } v2))}$))
 | *eval-binop* *RShift* *v1* *v2* = *Intg* ((*the-Intg* *v1*) div ($2^{\neg(\text{nat } (\text{the-Intg } v2))}$))
 | *eval-binop* *RShiftU* *v1* *v2* = *Intg* 42 — FIXME: Not yet implemented
 | *eval-binop* *Less* *v1* *v2* = *Bool* ((*the-Intg* *v1*) < (*the-Intg* *v2*))
 | *eval-binop* *Le* *v1* *v2* = *Bool* ((*the-Intg* *v1*) \leq (*the-Intg* *v2*))
 | *eval-binop* *Greater* *v1* *v2* = *Bool* ((*the-Intg* *v2*) < (*the-Intg* *v1*))
 | *eval-binop* *Ge* *v1* *v2* = *Bool* ((*the-Intg* *v2*) \leq (*the-Intg* *v1*))
 | *eval-binop* *Eq* *v1* *v2* = *Bool* (*v1*=*v2*)
 | *eval-binop* *Neq* *v1* *v2* = *Bool* (*v1* \neq *v2*)
 | *eval-binop* *BitAnd* *v1* *v2* = *Intg* 42 — FIXME: Not yet implemented
 | *eval-binop* *And* *v1* *v2* = *Bool* ((*the-Bool* *v1*) \wedge (*the-Bool* *v2*))
 | *eval-binop* *BitXor* *v1* *v2* = *Intg* 42 — FIXME: Not yet implemented
 | *eval-binop* *Xor* *v1* *v2* = *Bool* ((*the-Bool* *v1*) \neq (*the-Bool* *v2*))
 | *eval-binop* *BitOr* *v1* *v2* = *Intg* 42 — FIXME: Not yet implemented
 | *eval-binop* *Or* *v1* *v2* = *Bool* ((*the-Bool* *v1*) \vee (*the-Bool* *v2*))
 | *eval-binop* *CondAnd* *v1* *v2* = *Bool* ((*the-Bool* *v1*) \wedge (*the-Bool* *v2*))
 | *eval-binop* *CondOr* *v1* *v2* = *Bool* ((*the-Bool* *v1*) \vee (*the-Bool* *v2*))

definition

need-second-arg :: *binop* \Rightarrow *val* \Rightarrow *bool* **where**

need-second-arg *binop* *v1* = (\neg ((*binop*=*CondAnd* \wedge \neg *the-Bool* *v1*) \vee
 (*binop*=*CondOr* \wedge *the-Bool* *v1*)))

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

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

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

lemma *need-second-arg-strict*[*simp*]:

$\llbracket \text{binop} \neq \text{CondAnd}; \text{binop} \neq \text{CondOr} \rrbracket \Longrightarrow \text{need-second-arg } \text{binop } b$
 <proof>

end

Chapter 8

Decl

1 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/developer/bugParade/index.jshtml>)

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

2 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

instantiation *acc-modi* :: *linorder*

begin

definition

less-acc-def: $a < b$
 \longleftrightarrow (case a of
 | *Private* $\Rightarrow (b = \text{Package} \vee b = \text{Protected} \vee b = \text{Public})$
 | *Package* $\Rightarrow (b = \text{Protected} \vee b = \text{Public})$
 | *Protected* $\Rightarrow (b = \text{Public})$
 | *Public* $\Rightarrow \text{False}$)

definition

le-acc-def: $(a :: \text{acc-modi}) \leq b \longleftrightarrow a < b \vee a = b$

instance

$\langle \text{proof} \rangle$

end

lemma *acc-modi-top* [*simp*]: $\text{Public} \leq a \Longrightarrow a = \text{Public}$

$\langle \text{proof} \rangle$

lemma *acc-modi-top1* [*simp, intro!*]: $a \leq \text{Public}$

$\langle \text{proof} \rangle$

lemma *acc-modi-le-Public*:

$a \leq \text{Public} \Longrightarrow a = \text{Private} \vee a = \text{Package} \vee a = \text{Protected} \vee a = \text{Public}$

$\langle \text{proof} \rangle$

lemma *acc-modi-bottom*: $a \leq \text{Private} \Longrightarrow a = \text{Private}$

$\langle \text{proof} \rangle$

lemma *acc-modi-Private-le*:

$\text{Private} \leq a \Longrightarrow a = \text{Private} \vee a = \text{Package} \vee a = \text{Protected} \vee a = \text{Public}$

$\langle \text{proof} \rangle$

lemma *acc-modi-Package-le*:

$\text{Package} \leq a \Longrightarrow a = \text{Package} \vee a = \text{Protected} \vee a = \text{Public}$

$\langle \text{proof} \rangle$

lemma *acc-modi-le-Package*:

$a \leq \text{Package} \Longrightarrow a = \text{Private} \vee a = \text{Package}$

$\langle \text{proof} \rangle$

lemma *acc-modi-Protected-le*:

$\text{Protected} \leq a \Longrightarrow a = \text{Protected} \vee a = \text{Public}$

$\langle \text{proof} \rangle$

lemma *acc-modi-le-Protected*:

$a \leq \text{Protected} \Longrightarrow a = \text{Private} \vee a = \text{Package} \vee a = \text{Protected}$

$\langle \text{proof} \rangle$

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*:

assumes $\text{Package} \leq m$

obtains $(\text{Package})\ m = \text{Package}$

$\mid (Protected) \ m = Protected$
 $\mid (Public) \ m = Public$
 $\langle proof \rangle$

Static Modifier

type-synonym *stat-modi* = *bool*

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

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

translations

$(type) \ decl \leq (type) \ (\downarrow access::acc-modi)$
 $(type) \ decl \leq (type) \ (\downarrow access::acc-modi, \dots::'a)$

4 Member (field or method)

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

translations

$(type) \ member \leq (type) \ (\downarrow access::acc-modi, static::bool)$
 $(type) \ member \leq (type) \ (\downarrow access::acc-modi, static::bool, \dots::'a)$

5 Field

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

translations

$(type) \ field \leq (type) \ (\downarrow access::acc-modi, static::bool, type::ty)$
 $(type) \ field \leq (type) \ (\downarrow access::acc-modi, static::bool, type::ty, \dots::'a)$

type-synonym *fdecl*
 = *vname* \times *field*

translations

$(type) \ fdecl \leq (type) \ vname \times field$

6 Method

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

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

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

type-synonym *mdecl* = *sig* \times *methd*

translations

$(type) \ mhead \leq (type) \ (\downarrow access::acc-modi, static::bool,$
 $\quad \quad \quad pars::vname \ list, resT::ty)$

```

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

```

definition

```

mhead :: methd ⇒ mhead
where mhead m = (|access=access m, static=static m, pars=pars m, resT=resT m|)

```

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

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

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

lemma *resT-mhead* [simp]: *resT (mhead m) = 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*

```

class has-memberid =
  fixes memberid :: 'a ⇒ memberid

```

instantiation *memberdecl* :: *has-memberid*
begin

definition

```

memberdecl-memberid-def:
  memberid m = (case m of
                fdecl (vn,f) ⇒ fid vn
                | mdecl (sig,m) ⇒ mid sig)

```

instance ⟨proof⟩

end

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

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

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

lemma *memberid-mdecl-simp1*: *memberid (mdecl m) = mid (fst m)*

$\langle \text{proof} \rangle$

instantiation $\text{prod} :: (\text{type}, \text{has-memberid}) \text{ has-memberid}$
begin

definition

$\text{pair-memberid-def}:$

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

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

end

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

$\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

definition

$\text{is-field} :: \text{qname} \times \text{memberdecl} \Rightarrow \text{bool}$

where $\text{is-field } m = (\exists \text{ declC } f. m = (\text{declC}, f \text{decl } f))$

lemma $\text{is-fieldD}: \text{is-field } m \Longrightarrow \exists \text{ declC } f. m = (\text{declC}, f \text{decl } f)$

$\langle \text{proof} \rangle$

lemma $\text{is-fieldI}: \text{is-field } (C, f \text{decl } f)$

$\langle \text{proof} \rangle$

definition

$\text{is-method} :: \text{qname} \times \text{memberdecl} \Rightarrow \text{bool}$

where $\text{is-method } \text{membr} = (\exists \text{ declC } m. \text{membr} = (\text{declC}, m \text{decl } m))$

lemma $\text{is-methodD}: \text{is-method } \text{membr} \Longrightarrow \exists \text{ declC } m. \text{membr} = (\text{declC}, m \text{decl } m)$

$\langle \text{proof} \rangle$

lemma $\text{is-methodI}: \text{is-method } (C, m \text{decl } m)$

$\langle \text{proof} \rangle$

7 Interface

record $\text{ibody} = \text{decl} + \text{— interface body}$

$\text{imethods} :: (\text{sig} \times \text{mhead}) \text{ list} \text{— method heads}$

record $\text{iface} = \text{ibody} + \text{— interface}$

$\text{isuperIfs} :: \text{qname list} \text{— superinterface list}$

type-synonym

$\text{idecl} \text{— interface declaration, cf. 9.1}$

$= \text{qname} \times \text{iface}$

translations

$(\text{type}) \text{ ibody} \leq (\text{type}) (\lambda \text{ access} :: \text{acc-modi}, \text{imethods} :: (\text{sig} \times \text{mhead}) \text{ list})$

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

$(\text{type}) \text{ iface} \leq (\text{type}) (\lambda \text{ access} :: \text{acc-modi}, \text{imethods} :: (\text{sig} \times \text{mhead}) \text{ list},$
 $\text{isuperIfs} :: \text{qname list})$

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

$(\text{type}) \text{ idecl} \leq (\text{type}) \text{ qname} \times \text{iface}$

definition

$ibody :: iface \Rightarrow ibody$
where $ibody\ i = (\backslash access = access\ i, imethods = imethods\ i)$

lemma $access\text{-}ibody\ [simp]: (access\ (ibody\ i)) = access\ i$
 $\langle proof \rangle$

lemma $imethods\text{-}ibody\ [simp]: (imethods\ (ibody\ i)) = imethods\ i$
 $\langle proof \rangle$

8 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

type-synonym

$cdecl$ — class declaration, cf. 8.1
 $= qtname \times class$

translations

$(type)\ cbody \leq (type)\ (\backslash access :: acc\ modi, cfields :: fdecl\ list,$
 $methods :: mdecl\ list, init :: stmt)$
 $(type)\ cbody \leq (type)\ (\backslash access :: acc\ modi, cfields :: fdecl\ list,$
 $methods :: mdecl\ list, init :: stmt, \dots 'a)$
 $(type)\ class \leq (type)\ (\backslash access :: acc\ modi, cfields :: fdecl\ list,$
 $methods :: mdecl\ list, init :: stmt,$
 $super :: qtname, superIfs :: qtname\ list)$
 $(type)\ class \leq (type)\ (\backslash access :: acc\ modi, cfields :: fdecl\ list,$
 $methods :: mdecl\ list, init :: stmt,$
 $super :: qtname, superIfs :: qtname\ list, \dots 'a)$
 $(type)\ cdecl \leq (type)\ qtname \times class$

definition

$cbody :: class \Rightarrow cbody$
where $cbody\ c = (\backslash access = access\ c, cfields = cfields\ c, methods = methods\ c, init = init\ c)$

lemma $access\text{-}cbody\ [simp]: access\ (cbody\ c) = access\ c$
 $\langle proof \rangle$

lemma $cfields\text{-}cbody\ [simp]: cfields\ (cbody\ c) = cfields\ c$
 $\langle proof \rangle$

lemma $methods\text{-}cbody\ [simp]: methods\ (cbody\ c) = methods\ c$
 $\langle proof \rangle$

lemma $init\text{-}cbody\ [simp]: init\ (cbody\ c) = init\ c$
 $\langle proof \rangle$

standard classes**consts**

$Object\text{-}mdecls :: mdecl\ list$ — methods of Object
 $SXcpt\text{-}mdecls :: mdecl\ list$ — methods of SXcpts

definition

ObjectC :: *cdecl* — declaration of root class **where**
ObjectC = (*Object*, (*access*=*Public*, *cfields*=[], *methods*=*Object-mdecls*,
init=*Skip*, *super*=*undefined*, *superIfs*=[]))

definition

SXcptC :: *xname* ⇒ *cdecl* — declarations of throwable classes **where**
SXcptC *xn* = (*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* ≠ *SXcptC* *xn*

⟨proof⟩

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

⟨proof⟩

definition

standard-classes :: *cdecl* list **where**
standard-classes = [*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

(*type*) *prog* <= (*type*) (*ifaces*::*idecl* list, *classes*::*cdecl* list)
(*type*) *prog* <= (*type*) (*ifaces*::*idecl* list, *classes*::*cdecl* list, ...:'a)

abbreviation

iface :: *prog* ⇒ (*qname*, *iface*) table
where *iface* *G* *I* == table-of (*ifaces* *G*) *I*

abbreviation

class :: *prog* ⇒ (*qname*, *class*) table
where *class* *G* *C* == table-of (*classes* *G*) *C*

abbreviation

is-iface :: *prog* ⇒ *qname* ⇒ bool
where *is-iface* *G* *I* == *iface* *G* *I* ≠ None

abbreviation

is-class :: *prog* ⇒ *qname* ⇒ bool
where *is-class* *G* *C* == *class* *G* *C* ≠ None

is type

primrec *is-type* :: *prog* ⇒ *ty* ⇒ bool
and *isrtype* :: *prog* ⇒ *ref-ty* ⇒ bool

where

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

```

| isrtype G (IfaceT tn) = is-iface G tn
| isrtype G (ClassT tn) = is-class G tn
| isrtype G (ArrayT T) = is-type G T

```

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

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

subinterface and subclass relation, in anticipation of TypeRel.thy

definition

```

subint1 :: prog => (qname × qname) set — direct subinterface
where subint1 G = {(I,J). ∃ i∈iface G I: J∈set (isuperIfs i)}

```

definition

```

subcls1 :: prog => (qname × qname) set — direct subclass
where subcls1 G = {(C,D). C≠Object ∧ (∃ c∈class G C: super c = D)}

```

abbreviation

```

subcls1-syntax :: prog => [qname, qname] => bool (⋈+--<_C 1-> [71,71,71] 70)
where G⊢ C <_C 1 D == (C,D) ∈ subcls1 G

```

abbreviation

```

subclseq-syntax :: prog => [qname, qname] => bool (⋈+--≤_C -> [71,71,71] 70)
where G⊢ C ≤_C D == (C,D) ∈ (subcls1 G)*

```

abbreviation

```

subcls-syntax :: prog => [qname, qname] => bool (⋈+--<_C -> [71,71,71] 70)
where G⊢ C <_C D == (C,D) ∈ (subcls1 G)+

```

notation (ASCII)

```

subcls1-syntax (⋈|-<:C1-> [71,71,71] 70) and
subclseq-syntax (⋈|-<=:C->[71,71,71] 70) and
subcls-syntax (⋈|-<:C->[71,71,71] 70)

```

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:

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

lemma subcls1-def2:

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

$\langle \text{proof} \rangle$

lemma *subcls-is-class*:

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

$\langle \text{proof} \rangle$

lemma *no-subcls1-Object*: $G \vdash \text{Object} \prec_C 1 \ D \implies P$

$\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

well-structured programs

definition

$\text{ws-idecl} :: \text{prog} \Rightarrow \text{qname} \Rightarrow \text{qname list} \Rightarrow \text{bool}$

where $\text{ws-idecl } G \ I \ si = (\forall J \in \text{set } si. \text{is-iface } G \ J \ \wedge \ (J, I) \notin (\text{subint1 } G)^+)$

definition

$\text{ws-cdecl} :: \text{prog} \Rightarrow \text{qname} \Rightarrow \text{qname} \Rightarrow \text{bool}$

where $\text{ws-cdecl } G \ C \ sc = (C \neq \text{Object} \longrightarrow \text{is-class } G \ sc \wedge (sc, C) \notin (\text{subcls1 } G)^+)$

definition

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

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

lemma *ws-progI*:

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

$\langle \text{proof} \rangle$

lemma *ws-prog-ideclD*:

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

$\langle \text{proof} \rangle$

lemma *ws-prog-cdeclD*:

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

$\langle \text{proof} \rangle$

well-foundedness

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

$\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

lemma *subint1-irrefl-lemma1*:

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

lemma *subcls1-irrefl-lemma1*:

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

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

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

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

$\langle proof \rangle$

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

$\langle proof \rangle$

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

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

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

$\langle proof \rangle$

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

$\langle proof \rangle$

lemma *subint1-induct*:

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

lemma *subcls1-induct* [consumes 1]:

$\llbracket ws\text{-}prog\ G; \bigwedge x. \forall y. (x, y) \in subcls1\ G \longrightarrow P\ y \implies P\ x \rrbracket \implies P\ a$
 $\langle proof \rangle$

lemma *ws-subint1-induct*:

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

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

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

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

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

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

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

lemma *ws-class-induct''* [consumes 2, case-names *Object Subcls*]:
 $\llbracket \text{class } G \ C = \text{Some } c; \text{ws-prog } G; \\
\bigwedge \text{co. class } G \ \text{Object} = \text{Some } \text{co} \implies P \ \text{Object } \text{co}; \\
\bigwedge \ C \ c \ \text{sc.} \llbracket \text{class } G \ C = \text{Some } c; \text{class } G \ (\text{super } c) = \text{Some } \text{sc}; \\
C \neq \text{Object}; P \ (\text{super } c) \ \text{sc} \rrbracket \implies P \ C \ c \\
\rrbracket \implies P \ C \ c$
 <proof>

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*
 <proof>

general recursion operators for the interface and class hierarchies

function *iface-rec* :: *prog* \Rightarrow *qname* \Rightarrow (*qname* \Rightarrow *iface* \Rightarrow 'a *set* \Rightarrow 'a) \Rightarrow 'a
where
 [simp del]: *iface-rec* *G I f* =
 (case *iface* *G I* of
 None \Rightarrow undefined
 | Some *i* \Rightarrow if *ws-prog* *G*
 then *f I i*
 (($\lambda J. \text{iface-rec } G \ J \ f$) 'set (*isuperIfs* *i*))
 else undefined)
 <proof>
termination
 <proof>

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

function
class-rec :: *prog* \Rightarrow *qname* \Rightarrow 'a \Rightarrow (*qname* \Rightarrow *class* \Rightarrow 'a \Rightarrow 'a) \Rightarrow 'a
where
 [simp del]: *class-rec* *G C t f* =
 (case *class* *G C* of
 None \Rightarrow undefined
 | Some *c* \Rightarrow if *ws-prog* *G*
 then *f C c*
 (if *C* = *Object* then *t*
 else *class-rec* *G* (*super* *c*) *t f*)
 else undefined)
 <proof>
termination
 <proof>

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

definition

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

end

Chapter 9

TypeRel

1 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

definition

implmt1 :: *prog* \Rightarrow (*qname* \times *qname*) *set* — direct implementation
— direct implementation, cf. 8.1.3
where *implmt1* *G* = {(*C*,*I*). *C* \neq *Object* \wedge ($\exists c \in \text{class } G \ C: I \in \text{set } (\text{superIfs } c)$)}

abbreviation

subint1-syntax :: *prog* \Rightarrow [*qname*, *qname*] \Rightarrow *bool* ($\langle \vdash \neg I1 \rangle$ [*71*,*71*,*71*] *70*)
where *G* $\vdash I \neg I1 J$ == (*I*,*J*) \in *subint1* *G*

abbreviation

subint-syntax :: *prog* \Rightarrow [*qname*, *qname*] \Rightarrow *bool* ($\langle \vdash \neg I \rangle$ [*71*,*71*,*71*] *70*)
where *G* $\vdash I \neg I J$ == (*I*,*J*) \in (*subint1* *G*)* — cf. 9.1.3

abbreviation

implmt1-syntax :: *prog* \Rightarrow [*qname*, *qname*] \Rightarrow *bool* ($\langle \vdash \neg \sim 1 \rangle$ [*71*,*71*,*71*] *70*)
where *G* $\vdash C \sim 1 I$ == (*C*,*I*) \in *implmt1* *G*

notation (*ASCII*)

subint1-syntax ($\langle \vdash \neg \neg I1 \rangle$ [*71*,*71*,*71*] *70*) **and**
subint-syntax ($\langle \vdash \neg \neg \neg I \rangle$ [*71*,*71*,*71*] *70*) **and**
implmt1-syntax ($\langle \vdash \neg \neg \sim 1 \rangle$ [*71*,*71*,*71*] *70*)

subclass and subinterface relations

lemmas *subcls-direct* = *subcls1I* [*THEN* *r-into-rtrancI*]

lemma *subcls-direct1*:

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

lemma *subcls1I1*:

$\llbracket \text{class } G \ C = \text{Some } c; C \neq \text{Object}; D = \text{super } c \rrbracket \implies G \vdash C \prec_C 1 \ D$
 $\langle \text{proof} \rangle$

lemma *subcls-direct2*:

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

lemma *subclseq-trans*: $\llbracket G \vdash A \preceq_C B; G \vdash B \preceq_C C \rrbracket \implies G \vdash A \preceq_C C$
 $\langle \text{proof} \rangle$

lemma *subcls-trans*: $\llbracket G \vdash A \prec_C B; G \vdash B \prec_C C \rrbracket \implies G \vdash A \prec_C C$
 $\langle \text{proof} \rangle$

lemma *SXcpt-subcls-Throwable-lemma*:

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

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

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

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

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

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

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

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

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

lemma *single-inheritance*:

$\llbracket G \vdash A \prec_C 1 \ B; G \vdash A \prec_C 1 \ C \rrbracket \implies B = C$
 $\langle \text{proof} \rangle$

lemma *subcls-compareable*:

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

lemma *subcls1-irrefl*: $\llbracket G \vdash C \prec_C 1 \ D; \text{ws-prog } G \rrbracket$
 $\implies C \neq D$

$\langle \text{proof} \rangle$

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*:

assumes $G \vdash C \preceq_C D$

obtains $(Eq) \ C = D \mid (Subcls) \ G \vdash C \prec_C D$

$\langle \text{proof} \rangle$

lemma *subclseq-acyclic*:

$\llbracket G \vdash C \preceq_C D; G \vdash D \preceq_C C; \text{ws-prog } G \rrbracket \implies C = D$

$\langle \text{proof} \rangle$

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

$\implies C \neq D$

$\langle \text{proof} \rangle$

lemma *invert-subclseq*:

$\llbracket G \vdash C \preceq_C D; \text{ws-prog } G \rrbracket$

$\implies \neg G \vdash D \prec_C C$

$\langle \text{proof} \rangle$

lemma *invert-subcls*:

$\llbracket G \vdash C \prec_C D; \text{ws-prog } G \rrbracket$

$\implies \neg G \vdash D \preceq_C C$

$\langle \text{proof} \rangle$

lemma *subcls-superD*:

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

$\langle \text{proof} \rangle$

lemma *subclseq-superD*:

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

$\langle \text{proof} \rangle$

implementation relation

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

$\langle \text{proof} \rangle$

inductive — implementation, cf. 8.1.4

implmt :: *prog* \Rightarrow *qtname* \Rightarrow *qtname* \Rightarrow *bool* ($\langle \vdash \rightsquigarrow \rangle$ [71,71,71] 70)

for *G* :: *prog*

where

direct: $G \vdash C \rightsquigarrow 1J \implies G \vdash C \rightsquigarrow J$

| *subint*: $G \vdash C \rightsquigarrow 1I \implies G \vdash I \preceq I \ J \implies G \vdash C \rightsquigarrow J$

| *subcls1*: $G \vdash C \prec_C 1D \implies G \vdash D \rightsquigarrow J \implies G \vdash C \rightsquigarrow J$

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

$\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

lemma *subcls-implmt* [rule-format (no-asm)]: $G \vdash A \preceq_C B \implies G \vdash B \rightsquigarrow K \longrightarrow G \vdash A \rightsquigarrow K$
 ⟨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* (\hookrightarrow \vdash \preceq) [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*: $G \vdash T.\boxed{} \preceq \text{Class } \text{Object}$

| *array*: $G \vdash \text{RefT } S \preceq \text{RefT } T \implies G \vdash \text{RefT } S.\boxed{} \preceq \text{RefT } T.\boxed{}$

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*: $G \vdash \text{Iface } I \preceq T \implies T = \text{Class } \text{Object} \vee (\exists J. T = \text{Iface } J)$
 $\langle \text{proof} \rangle$

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

lemma *widen-NT2*: $G \vdash S \preceq NT \implies S = NT$
 $\langle \text{proof} \rangle$

lemma *widen-Object*:

assumes *isrtype* G T **and** *ws-prog* G
shows $G \vdash \text{RefT } T \preceq \text{Class } \text{Object}$
 $\langle \text{proof} \rangle$

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

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

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

lemmas *subint-antisym* =
subint1-acyclic [*THEN* *acyclic-impl-antisym-rtranc1*]

lemmas *subcls-antisym* =
subcls1-acyclic [*THEN* *acyclic-impl-antisym-rtranc1*]

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

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

definition

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

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

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

narrowing relation

inductive — narrowing reference conversion, cf. 5.1.5

narrow :: *prog* \Rightarrow *ty* \Rightarrow *ty* \Rightarrow *bool* ($\hookrightarrow \vdash \neg \succ \neg \rightarrow$) [*71, 71, 71*] *70*)

for $G :: \text{prog}$

where

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

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

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

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

cast :: $\text{prog} \Rightarrow \text{ty} \Rightarrow \text{ty} \Rightarrow \text{bool} \ (\hookrightarrow \vdash \preceq^? \rightarrow [71, 71, 71] \ 70)$

for $G :: \text{prog}$

where

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

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

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

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

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

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

lemma *cast-Boolean*:

assumes *bool-cast*: $G \vdash \text{PrimT } \text{Boolean} \preceq^? T$

shows $T = \text{PrimT } \text{Boolean}$

⟨proof⟩

lemma *cast-Boolean2*:

assumes *bool-cast*: $G \vdash S \preceq^? \text{PrimT } \text{Boolean}$

shows $S = \text{PrimT } \text{Boolean}$

⟨proof⟩

end

Chapter 10

DeclConcepts

1 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)

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

2 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

primrec

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

where

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

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

definition

is-acc-class :: *prog* \Rightarrow *pname* \Rightarrow *qname* \Rightarrow *bool*
where *is-acc-class* *G* *P* *C* = (*is-class G C* \wedge *G* \vdash (*Class C*) *accessible-in P*)

definition

is-acc-iface :: *prog* \Rightarrow *pname* \Rightarrow *qname* \Rightarrow *bool*
where *is-acc-iface* *G* *P* *I* = (*is-iface G I* \wedge *G* \vdash (*Iface I*) *accessible-in P*)

definition

is-acc-type :: *prog* \Rightarrow *pname* \Rightarrow *ty* \Rightarrow *bool*
where *is-acc-type* *G* *P* *T* = (*is-type G T* \wedge *G* \vdash *T* *accessible-in P*)

definition

is-acc-reftype :: *prog* \Rightarrow *pname* \Rightarrow *ref-ty* \Rightarrow *bool*
where *is-acc-reftype* *G P T* = (*isrtype* *G T* \wedge *G* \vdash *T accessible-in' P*)

lemma *is-acc-classD*:

is-acc-class *G P C* \Longrightarrow *is-class* *G C* \wedge *G* \vdash (*Class C*) *accessible-in P*
 \langle *proof* \rangle

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

\langle *proof* \rangle

lemma *is-acc-ifaceD*:

is-acc-iface *G P I* \Longrightarrow *is-iface* *G I* \wedge *G* \vdash (*Iface I*) *accessible-in P*
 \langle *proof* \rangle

lemma *is-acc-typeD*:

is-acc-type *G P T* \Longrightarrow *is-type* *G T* \wedge *G* \vdash *T accessible-in P*
 \langle *proof* \rangle

lemma *is-acc-reftypeD*:

is-acc-reftype *G P T* \Longrightarrow *isrtype* *G T* \wedge *G* \vdash *T accessible-in' P*
 \langle *proof* \rangle

3 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

class *has-accmodi* =

fixes *accmodi*:: 'a \Rightarrow *acc-modi*

instantiation *acc-modi* :: *has-accmodi*

begin

definition

acc-modi-accmodi-def: *accmodi* (*a*::*acc-modi*) = *a*

instance \langle *proof* \rangle

end

lemma *acc-modi-accmodi-simp*[*simp*]: *accmodi* (*a*::*acc-modi*) = *a*

\langle *proof* \rangle

instantiation *decl-ext* :: (*type*) *has-accmodi*

begin

definition

decl-acc-modi-def: *accmodi* (*d*::('a:: *type*) *decl-scheme*) = *access d*

instance \langle *proof* \rangle

end

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

instantiation *prod* :: (type, *has-accmodi*) *has-accmodi*
begin

definition

pair-acc-modi-def: *accmodi p* = *accmodi (snd p)*

instance ⟨*proof*⟩

end

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

instantiation *memberdecl* :: *has-accmodi*
begin

definition

memberdecl-acc-modi-def: *accmodi m* = (case *m* of
 fdecl f ⇒ *accmodi f*
 | *mdecl m* ⇒ *accmodi m*)

instance ⟨*proof*⟩

end

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

class *has-declclass* =
fixes *declclass*:: 'a ⇒ *qname*

instantiation *qname-ext* :: (type) *has-declclass*
begin

definition

declclass q = (| *pid* = *pid q*, *tid* = *tid q* |)

instance ⟨*proof*⟩

end

lemma *qname-declclass-def*:
declclass q ≡ (*q*::*qname*)
 ⟨*proof*⟩

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

instantiation *prod* :: (*has-declclass*, *type*) *has-declclass*
begin

definition

pair-declclass-def: *declclass* *p* = *declclass* (*fst* *p*)

instance $\langle proof \rangle$

end

lemma *pair-declclass-simp*[*simp*]: *declclass* (*c*,*x*) = *declclass* *c*
 $\langle proof \rangle$

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

class *has-static* =
fixes *is-static* :: 'a \Rightarrow bool

instantiation *decl-ext* :: (*has-static*) *has-static*
begin

instance $\langle proof \rangle$

end

instantiation *member-ext* :: (*type*) *has-static*
begin

instance $\langle proof \rangle$

end

axiomatization where

static-field-type-is-static-def: *is-static* (*m*::('a *member-scheme*)) \equiv *static* *m*

lemma *member-is-static-simp*: *is-static* (*m*::('a *member-scheme*)) = *static* *m*
 $\langle proof \rangle$

instantiation *prod* :: (*type*, *has-static*) *has-static*
begin

definition

pair-is-static-def: *is-static* *p* = *is-static* (*snd* *p*)

instance $\langle proof \rangle$

end

lemma *pair-is-static-simp* [*simp*]: *is-static* (*x*,*s*) = *is-static* *s*
 $\langle proof \rangle$

lemma *pair-is-static-simp1*: *is-static* *p* = *is-static* (*snd* *p*)
 $\langle proof \rangle$

instantiation *memberdecl* :: *has-static*
begin

definition

memberdecl-is-static-def:
is-static *m* = (case *m* of

$$\begin{array}{l} fdecl\ f \Rightarrow is-static\ f \\ | mdecl\ m \Rightarrow is-static\ m \end{array}$$

instance $\langle proof \rangle$

end

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

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

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

definition

$decliface :: qname \times 'a\ decl-scheme \Rightarrow qname$ **where**
 $decliface = fst$ — get the interface component

definition

$mbr :: qname \times memberdecl \Rightarrow memberdecl$ **where**
 $mbr = snd$ — get the memberdecl component

definition

$mthd :: 'b \times 'a \Rightarrow 'a$ **where**
 $mthd = snd$ — get the method component
— also used for mdecl, mhead

definition

$fld :: 'b \times 'a\ decl-scheme \Rightarrow 'a\ decl-scheme$ **where**
 $fld = snd$ — get the field component
— also used for $((vname \times qname) \times field)$

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

definition

$fname :: vname \times 'a \Rightarrow vname$
where $fname = fst$
— also used for fdecl

definition

$declclassf :: (vname \times qname) \Rightarrow qname$
where $declclassf = snd$

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

definition

$fldname :: vname \times qname \Rightarrow vname$
where $fldname = fst$

definition

$fldclass :: vname \times qname \Rightarrow qname$
where $fldclass = snd$

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

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

lemma *ext-fldname-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*

definition

$methdMembr :: qname \times mdecl \Rightarrow qname \times memberdecl$
where $methdMembr\ m = (fst\ m, mdecl\ (snd\ m))$

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

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

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

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

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

definition

method :: *sig* ⇒ (*qname* × *methd*) ⇒ (*qname* × *memberdecl*)
where *method sig m* = (*declclass m*, *mdecl (sig, mthd m)*)

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

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

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

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

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

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

definition

fieldm :: *vname* ⇒ (*qname* × *field*) ⇒ (*qname* × *memberdecl*)
where *fieldm n f* = (*declclass f*, *fdecl (n, fld f)*)

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

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

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

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

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

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

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

definition

msig :: (*qtname* × *mdecl*) ⇒ *sig*
where *msig* *m* = *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*

definition

qmdecl :: *sig* ⇒ (*qtname* × *methd*) ⇒ (*qtname* × *mdecl*)
where *qmdecl* *sig* *m* = (*declclass* *m*, (*sig*,*methd* *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 (*methd* (*qmdecl* *sig* *new*)) = *mdecl* (*sig*, *methd* *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

class *has-resTy* =
fixes *resTy*:: 'a ⇒ *ty*

instantiation *decl-ext* :: (*has-resTy*) *has-resTy*
begin

instance ⟨*proof*⟩

end

instantiation *member-ext* :: (*has-resTy*) *has-resTy*
begin

instance ⟨*proof*⟩

end

instantiation *mhead-ext* :: (*type*) *has-resTy*
begin

instance $\langle proof \rangle$

end

axiomatization where

$mhead-ext-type-resTy-def: resTy (m::('b mhead-scheme)) \equiv resT m$

lemma $mhead-resTy-simp: resTy (m::'a mhead-scheme) = resT m$
 $\langle proof \rangle$

lemma $resTy-mhead [simp]: resTy (mhead m) = resTy m$
 $\langle proof \rangle$

instantiation $prod :: (type, has-resTy) has-resTy$
begin

definition

$pair-resTy-def: resTy p = resTy (snd p)$

instance $\langle proof \rangle$

end

lemma $pair-resTy-simp[simp]: resTy (x,m) = resTy m$
 $\langle proof \rangle$

lemma $qmdecl-resTy-simp [simp]: resTy (qmdecl sig m) = resTy m$
 $\langle proof \rangle$

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

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.

definition

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

where

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

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

abbreviation

Method-inheritable-in-syntax::

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

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

abbreviation

Method-inheritable-in::

$prog \Rightarrow sig \Rightarrow (qname \times methd) \Rightarrow pname \Rightarrow bool$
 $(\vdash \vdash Method - - inheritable'-in \rightarrow [61,61,61,61] 60)$
where $G \vdash Method\ s\ m\ inheritable-in\ p == G \vdash (method\ s\ m)\ inheritable-in\ p$

declared-in/undeclared-in

definition

$cdeclaredmethd :: prog \Rightarrow qname \Rightarrow (sig, methd)\ table\ \mathbf{where}$
 $cdeclaredmethd\ G\ C =$
 $(case\ class\ G\ C\ of$
 $\quad None \Rightarrow \lambda\ sig.\ None$
 $\quad | Some\ c \Rightarrow table-of\ (methods\ c))$

definition

$cdeclaredfield :: prog \Rightarrow qname \Rightarrow (vname, field)\ table\ \mathbf{where}$
 $cdeclaredfield\ G\ C =$
 $(case\ class\ G\ C\ of$
 $\quad None \Rightarrow \lambda\ sig.\ None$
 $\quad | Some\ c \Rightarrow table-of\ (cfields\ c))$

definition

$declared-in :: prog \Rightarrow memberdecl \Rightarrow qname \Rightarrow bool\ (\vdash \vdash - declared'-in \rightarrow [61,61,61] 60)$
where
 $G \vdash m\ declared-in\ C = (case\ m\ of$
 $\quad fdecl\ (fn, f) \Rightarrow cdeclaredfield\ G\ C\ fn = Some\ f$
 $\quad | mdecl\ (sig, m) \Rightarrow cdeclaredmethd\ G\ C\ sig = Some\ m)$

abbreviation

$method-declared-in :: prog \Rightarrow (qname \times mdecl) \Rightarrow qname \Rightarrow bool$
 $(\vdash \vdash Method - - declared'-in \rightarrow [61,61,61] 60)$
where $G \vdash Method\ m\ declared-in\ C == G \vdash mdecl\ (methd\ m)\ declared-in\ C$

abbreviation

$methd-declared-in :: prog \Rightarrow sig \Rightarrow (qname \times methd) \Rightarrow qname \Rightarrow bool$
 $(\vdash \vdash Method - - declared'-in \rightarrow [61,61,61,61] 60)$
where $G \vdash Method\ s\ m\ declared-in\ C == G \vdash mdecl\ (s, methd\ m)\ declared-in\ C$

lemma *declared-in-classD*:

$G \vdash m\ declared-in\ C \implies is-class\ G\ C$
 $\langle proof \rangle$

definition

$undeclared-in :: prog \Rightarrow memberid \Rightarrow qname \Rightarrow bool\ (\vdash \vdash - undeclared'-in \rightarrow [61,61,61] 60)$
where
 $G \vdash m\ undeclared-in\ C = (case\ m\ of$
 $\quad fid\ fn \Rightarrow cdeclaredfield\ G\ C\ fn = None$
 $\quad | mid\ sig \Rightarrow cdeclaredmethd\ G\ C\ sig = None)$

members

inductive

$members :: prog \Rightarrow (qname \times memberdecl) \Rightarrow qname \Rightarrow bool$
 $(\vdash \vdash - member'-of \rightarrow [61,61,61] 60)$
for $G :: prog$
where

Immediate: $\llbracket G \vdash mbr\ m\ declared-in\ C; declclass\ m = C \rrbracket \implies G \vdash m\ member-of\ C$
Inherited: $\llbracket G \vdash m\ inheritable-in\ (pid\ C); G \vdash memberid\ m\ undeclared-in\ C; \rrbracket$

$$G \vdash C \prec_C 1 S; G \vdash (\text{Class } S) \text{ accessible-in } (\text{pid } C); G \vdash m \text{ member-of } S \\ \Downarrow \implies G \vdash m \text{ member-of } C$$

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

abbreviation

method-member-of:: $\text{prog} \Rightarrow (\text{qname} \times \text{mdecl}) \Rightarrow \text{qname} \Rightarrow \text{bool}$
 $(\hookleftarrow \vdash \text{Method} - \text{member'-of} \rightarrow [61,61,61] \ 60)$
where $G \vdash \text{Method } m \text{ member-of } C == G \vdash (\text{methdMembr } m) \text{ member-of } C$

abbreviation

methd-member-of:: $\text{prog} \Rightarrow \text{sig} \Rightarrow (\text{qname} \times \text{methd}) \Rightarrow \text{qname} \Rightarrow \text{bool}$
 $(\hookleftarrow \vdash \text{Methd} - \text{member'-of} \rightarrow [61,61,61,61] \ 60)$
where $G \vdash \text{Methd } s \text{ member-of } C == G \vdash (\text{method } s \ m) \text{ member-of } C$

abbreviation

fieldm-member-of:: $\text{prog} \Rightarrow \text{vname} \Rightarrow (\text{qname} \times \text{field}) \Rightarrow \text{qname} \Rightarrow \text{bool}$
 $(\hookleftarrow \vdash \text{Field} - \text{member'-of} \rightarrow [61,61,61] \ 60)$
where $G \vdash \text{Field } n \ f \text{ member-of } C == G \vdash \text{fieldm } n \ f \text{ member-of } C$

definition

inherits:: $\text{prog} \Rightarrow \text{qname} \Rightarrow (\text{qname} \times \text{memberdecl}) \Rightarrow \text{bool} (\hookleftarrow \vdash - \text{inherits} \rightarrow [61,61,61] \ 60)$
where
 $G \vdash C \text{ inherits } m =$
 $(G \vdash m \text{ inheritable-in } (\text{pid } C) \wedge G \vdash \text{memberid } m \text{ undeclared-in } C \wedge$
 $(\exists S. G \vdash C \prec_C 1 S \wedge G \vdash (\text{Class } S) \text{ accessible-in } (\text{pid } C) \wedge G \vdash m \text{ member-of } S))$

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

definition

member-in:: $\text{prog} \Rightarrow (\text{qname} \times \text{memberdecl}) \Rightarrow \text{qname} \Rightarrow \text{bool} (\hookleftarrow \vdash - \text{member'-in} \rightarrow [61,61,61] \ 60)$
where $G \vdash m \text{ member-in } C = (\exists \text{ provC}. G \vdash C \preceq_C \text{ provC} \wedge G \vdash m \text{ member-of } \text{ provC})$

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

abbreviation

method-member-in:: $\text{prog} \Rightarrow (\text{qname} \times \text{mdecl}) \Rightarrow \text{qname} \Rightarrow \text{bool}$
 $(\hookleftarrow \vdash \text{Method} - \text{member'-in} \rightarrow [61,61,61] \ 60)$
where $G \vdash \text{Method } m \text{ member-in } C == G \vdash (\text{methdMembr } m) \text{ member-in } C$

abbreviation

methd-member-in:: $\text{prog} \Rightarrow \text{sig} \Rightarrow (\text{qname} \times \text{methd}) \Rightarrow \text{qname} \Rightarrow \text{bool}$
 $(\hookleftarrow \vdash \text{Methd} - \text{member'-in} \rightarrow [61,61,61,61] \ 60)$
where $G \vdash \text{Methd } s \text{ member-in } C == G \vdash (\text{method } s \ m) \text{ member-in } C$

lemma *member-inD*: $G \vdash m \text{ member-in } C$
 $\implies \exists \text{ provC}. G \vdash C \preceq_C \text{ provC} \wedge G \vdash m \text{ member-of } \text{ provC}$
 $\langle \text{proof} \rangle$

lemma *member-inI*: $\llbracket G \vdash m \text{ member-of } \text{ provC}; G \vdash C \preceq_C \text{ provC} \rrbracket \implies G \vdash m \text{ member-in } C$
 $\langle \text{proof} \rangle$

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 :: $prog \Rightarrow (qname \times mdecl) \Rightarrow (qname \times mdecl) \Rightarrow bool$
 $(\langle -, \vdash - \rangle \text{ overrides}_S \rightarrow [61, 61, 61] \ 60)$
for $G :: prog$

where

Direct: $\llbracket \neg \text{ is-static new}; msig \text{ new} = msig \text{ old};$
 $G \vdash \text{Method new declared-in (declclass new)};$
 $G \vdash \text{Method old declared-in (declclass old)};$
 $G \vdash \text{Method old inheritable-in pid (declclass new)};$
 $G \vdash (\text{declclass new}) \prec_C 1 \text{ superNew};$
 $G \vdash \text{Method old member-of superNew}$
 $\rrbracket \implies G \vdash \text{new overrides}_S \text{ old}$

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

Dynamic overriding (used during the typecheck of the compiler)

inductive

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

where

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

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

abbreviation (input)

sig-stat-overrides::

$prog \Rightarrow sig \Rightarrow (qname \times methd) \Rightarrow (qname \times methd) \Rightarrow bool$
 $(\langle -, \vdash - \rangle \text{ overrides}_S \rightarrow [61, 61, 61, 61] \ 60)$

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

abbreviation (input)

sig-overrides:: $prog \Rightarrow sig \Rightarrow (qname \times methd) \Rightarrow (qname \times methd) \Rightarrow bool$
 $(\langle -, \vdash - \rangle \text{ overrides} \rightarrow [61, 61, 61, 61] \ 60)$

where $G, s \vdash \text{new overrides} \text{ old} == G \vdash (qmdecl \ s \ \text{new}) \text{ overrides} (qmdecl \ s \ \text{old})$

Hiding

definition

$hides :: prog \Rightarrow (qname \times mdecl) \Rightarrow (qname \times mdecl) \Rightarrow bool \ (\hookrightarrow - \text{ hides } \rightarrow [61,61,61] \ 60)$

where

$G \vdash_{new} hides \ old =$
 $(is-static \ new \wedge msig \ new = msig \ old \wedge$
 $G \vdash (declclass \ new) \prec_C (declclass \ old) \wedge$
 $G \vdash Method \ new \text{ declared-in } (declclass \ new) \wedge$
 $G \vdash Method \ old \text{ declared-in } (declclass \ old) \wedge$
 $G \vdash Method \ old \text{ inheritable-in pid } (declclass \ new))$

abbreviation

$sig-hides :: prog \Rightarrow sig \Rightarrow (qname \times methd) \Rightarrow (qname \times methd) \Rightarrow bool$
 $(\hookrightarrow -, \vdash - \text{ hides } \rightarrow [61,61,61,61] \ 60)$

where $G, s \vdash_{new} hides \ old == G \vdash (qmdecl \ s \ new) \text{ hides } (qmdecl \ s \ old)$

lemma *hidesI*:

$\llbracket is-static \ new; msig \ new = msig \ old;$
 $G \vdash (declclass \ new) \prec_C (declclass \ old);$
 $G \vdash Method \ new \text{ declared-in } (declclass \ new);$
 $G \vdash Method \ old \text{ declared-in } (declclass \ old);$
 $G \vdash Method \ old \text{ inheritable-in pid } (declclass \ new)$
 $\rrbracket \implies G \vdash_{new} hides \ old$
 $\langle proof \rangle$

lemma *hidesD*:

$\llbracket G \vdash_{new} hides \ old \rrbracket \implies$
 $declclass \ new \neq Object \wedge is-static \ new \wedge msig \ new = msig \ old \wedge$
 $G \vdash (declclass \ new) \prec_C (declclass \ old) \wedge$
 $G \vdash Method \ new \text{ declared-in } (declclass \ new) \wedge$
 $G \vdash Method \ old \text{ declared-in } (declclass \ old)$
 $\langle proof \rangle$

lemma *overrides-commonD*:

$\llbracket G \vdash_{new} overrides \ old \rrbracket \implies$
 $declclass \ new \neq Object \wedge \neg is-static \ new \wedge \neg is-static \ old \wedge$
 $accmodi \ new \neq Private \wedge$
 $msig \ new = msig \ old \wedge$
 $G \vdash (declclass \ new) \prec_C (declclass \ old) \wedge$
 $G \vdash Method \ new \text{ declared-in } (declclass \ new) \wedge$
 $G \vdash Method \ old \text{ declared-in } (declclass \ old)$
 $\langle proof \rangle$

lemma *ws-overrides-commonD*:

$\llbracket G \vdash_{new} overrides \ old; ws-prog \ G \rrbracket \implies$
 $declclass \ new \neq Object \wedge \neg is-static \ new \wedge \neg is-static \ old \wedge$
 $accmodi \ new \neq Private \wedge G \vdash_{resTy} new \preceq resTy \ old \wedge$
 $msig \ new = msig \ old \wedge$
 $G \vdash (declclass \ new) \prec_C (declclass \ old) \wedge$
 $G \vdash Method \ new \text{ declared-in } (declclass \ new) \wedge$
 $G \vdash Method \ old \text{ declared-in } (declclass \ old)$
 $\langle proof \rangle$

lemma *overrides-eq-sigD*:

$\llbracket G \vdash_{new} overrides \ old \rrbracket \implies msig \ old = msig \ new$
 $\langle proof \rangle$

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

definition

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

where

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

The subcondition of the *Protected* case: $G \vdash \text{accclass} \prec_C \text{declclass membr}$ could also be relaxed to: $G \vdash \text{accclass} \preceq_C \text{declclass membr}$ since in case both classes are the same the other condition $\text{pid } (\text{declclass membr}) = \text{pid accclass}$ holds anyway.

Like in case of overriding, the static and dynamic accessibility of members is not uniform.

- Statically the class/interface of the member must be accessible for the member to be accessible. During runtime this is not necessary. For Example, if a class is accessible and we are allowed to access a member of this class (statically) we expect that we can access this member in an arbitrary subclass (during runtime). It's not intended to restrict the access to accessible subclasses during runtime.
- Statically the member we want to access must be "member of" the class. Dynamically it must only be "member in" the class.

inductive

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

where

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

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

$| \text{Immediate: } !!\text{membr class.}$

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

$| \text{Overriding: } !!\text{membr class } C \text{ new old supr.}$

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

$\mathbb{J} \Rightarrow G \vdash \text{membr of class accessible-from accclass}$

abbreviation

methd-accessible-from::

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

where

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

abbreviation

field-accessible-from::

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

where

$G \vdash \text{Field fn f of C accessible-from accclass} ==$
 $G \vdash (\text{fieldm fn f}) \text{ of C accessible-from accclass}$

inductive

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

where

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

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

| *Immediate:* $\text{!!class. } \mathbb{J} G \vdash \text{membr member-in class};$
 $G \vdash \text{membr in class permits-acc-from accclass}$
 $\mathbb{J} \Rightarrow G \vdash \text{membr in class dyn-accessible-from accclass}$

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

abbreviation

methd-dyn-accessible-from::

$\text{prog} \Rightarrow \text{sig} \Rightarrow (\text{qname} \times \text{methd}) \Rightarrow \text{qname} \Rightarrow \text{qname} \Rightarrow \text{bool}$
 $(\langle \vdash \vdash \text{Methd} \text{ - - in - dyn'-accessible'-from} \rightarrow [61,61,61,61,61] \ 60)$

where

$G \vdash \text{Methd } s \text{ m in C dyn-accessible-from accC} ==$
 $G \vdash (\text{method } s \text{ m}) \text{ in C dyn-accessible-from accC}$

abbreviation

field-dyn-accessible-from::

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

where

$G \vdash \text{Field fn f in dynC dyn-accessible-from accC} ==$
 $G \vdash (\text{fieldm fn f}) \text{ in dynC dyn-accessible-from accC}$

lemma *accessible-from-commonD*: $G \vdash m \text{ of C accessible-from S}$

$\implies G \vdash m \text{ member-of } C \wedge G \vdash (\text{Class } C) \text{ accessible-in } (\text{pid } S)$
 $\langle \text{proof} \rangle$

lemma *unique-declaration:*

$\llbracket G \vdash m \text{ declared-in } C; G \vdash n \text{ declared-in } C; \text{memberid } m = \text{memberid } n \rrbracket$
 $\implies m = n$
 $\langle \text{proof} \rangle$

lemma *declared-not-undeclared:*

$G \vdash m \text{ declared-in } C \implies \neg G \vdash \text{memberid } m \text{ undeclared-in } C$
 $\langle \text{proof} \rangle$

lemma *undeclared-not-declared:*

$G \vdash \text{memberid } m \text{ undeclared-in } C \implies \neg G \vdash m \text{ declared-in } C$
 $\langle \text{proof} \rangle$

lemma *not-undeclared-declared:*

$\neg G \vdash \text{membr-id undeclared-in } C \implies (\exists m. G \vdash m \text{ declared-in } C \wedge$
 $\text{membr-id} = \text{memberid } m)$
 $\langle \text{proof} \rangle$

lemma *unique-declared-in:*

$\llbracket G \vdash m \text{ declared-in } C; G \vdash n \text{ declared-in } C; \text{memberid } m = \text{memberid } n \rrbracket$
 $\implies m = n$
 $\langle \text{proof} \rangle$

lemma *unique-member-of:*

assumes $n: G \vdash n \text{ member-of } C$ **and**
 $m: G \vdash m \text{ member-of } C$ **and**
 $\text{eqid: memberid } n = \text{memberid } m$
shows $n=m$
 $\langle \text{proof} \rangle$

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

lemma *member-of-declC:*

$G \vdash m \text{ member-of } C$
 $\implies G \vdash \text{mbr } m \text{ declared-in } (\text{declclass } m)$
 $\langle \text{proof} \rangle$

lemma *member-of-member-of-declC:*

$G \vdash m \text{ member-of } C$
 $\implies G \vdash m \text{ member-of } (\text{declclass } m)$
 $\langle \text{proof} \rangle$

lemma *member-of-class-relation:*

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

lemma *member-in-class-relation:*

$G \vdash m \text{ member-in } C \implies G \vdash C \preceq_C \text{declclass } m$
 $\langle \text{proof} \rangle$

lemma *stat-override-declclasses-relation:*

$\llbracket G \vdash (\text{declclass new}) \prec_C 1 \text{ superNew}; G \vdash \text{Method old member-of superNew} \rrbracket$
 $\implies G \vdash (\text{declclass new}) \prec_C (\text{declclass old})$
 $\langle \text{proof} \rangle$

lemma *stat-overrides-commonD*:

$\llbracket G \vdash \text{new overrides}_S \text{ old} \rrbracket \implies$
 $\text{declclass new} \neq \text{Object} \wedge \neg \text{is-static new} \wedge \text{msig new} = \text{msig old} \wedge$
 $G \vdash (\text{declclass new}) \prec_C (\text{declclass old}) \wedge$
 $G \vdash \text{Method new declared-in } (\text{declclass new}) \wedge$
 $G \vdash \text{Method old declared-in } (\text{declclass old})$
 $\langle \text{proof} \rangle$

lemma *member-of-Package*:

assumes $G \vdash m \text{ member-of } C$
and $\text{accmodi } m = \text{Package}$
shows $\text{pid } (\text{declclass } m) = \text{pid } C$
 $\langle \text{proof} \rangle$

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

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

lemma *no-Private-stat-override*:

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

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

lemma *permits-acc-inheritance*:

$\llbracket G \vdash m \text{ in statC permits-acc-from accC}; G \vdash \text{dynC} \preceq_C \text{statC} \rrbracket$
 $\implies G \vdash m \text{ in dynC permits-acc-from accC}$
 $\langle \text{proof} \rangle$

lemma *permits-acc-static-declC*:

$\llbracket G \vdash m \text{ in } C \text{ permits-acc-from accC}; G \vdash m \text{ member-in } C; \text{is-static } m \rrbracket$
 $\implies G \vdash m \text{ in } (\text{declclass } m) \text{ permits-acc-from accC}$
 $\langle \text{proof} \rangle$

lemma *dyn-accessible-from-static-declC*:

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

lemma *field-accessible-fromD*:

$\llbracket G \vdash \text{membr of } C \text{ accessible-from accC}; \text{is-field membr} \rrbracket$
 $\implies G \vdash \text{membr member-of } C \wedge$
 $G \vdash (\text{Class } C) \text{ accessible-in } (\text{pid accC}) \wedge$
 $G \vdash \text{membr in } C \text{ permits-acc-from accC}$
 $\langle \text{proof} \rangle$

lemma *field-accessible-from-permits-acc-inheritance*:

$\llbracket G \vdash \text{membr of statC accessible-from accC}; \text{is-field membr}; G \vdash \text{dynC} \preceq_C \text{statC} \rrbracket$
 $\implies G \vdash \text{membr in dynC permits-acc-from accC}$
 $\langle \text{proof} \rangle$

lemma *accessible-fieldD*:

$\llbracket G \vdash \text{membr of } C \text{ accessible-from } \text{acc}C; \text{is-field } \text{membr} \rrbracket$
 $\implies G \vdash \text{membr member-of } C \wedge$
 $G \vdash (\text{Class } C) \text{ accessible-in } (\text{pid } \text{acc}C) \wedge$
 $G \vdash \text{membr in } C \text{ permits-acc-from } \text{acc}C$
 $\langle \text{proof} \rangle$

lemma *member-of-Private*:

$\llbracket G \vdash m \text{ member-of } C; \text{accmodi } m = \text{Private} \rrbracket \implies \text{declclass } m = C$
 $\langle \text{proof} \rangle$

lemma *member-of-subclseq-declC*:

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

lemma *member-of-inheritance*:

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

lemma *member-of-subcls*:

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

corollary *member-of-overrides-subcls*:

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

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

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

lemma *inherited-field-access*:

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

lemma *accessible-inheritance*:

assumes $\text{stat-acc}: G \vdash m \text{ of stat}C \text{ accessible-from } \text{acc}C$ **and**
 $\text{subclseq}: G \vdash \text{dyn}C \preceq_C \text{stat}C$ **and**

$member_dynC: \vdash m \text{ member-of } dynC$ **and**
 $dynC_acc: \vdash (Class \ dynC) \text{ accessible-in } (pid \ accC)$
shows $\vdash m \text{ of } dynC \text{ accessible-from } accC$
 $\langle proof \rangle$

fields and methods

type-synonym

$f_{spec} = vname \times qname$

translations

$(type) \ f_{spec} \leq (type) \ vname \times qname$

definition

$imethds :: prog \Rightarrow qname \Rightarrow (sig, qname \times mhead) \text{ tables}$ **where**
 $imethds \ G \ I =$
 $iface_rec \ G \ I \ (\lambda I \ i \ ts. (Un_tables \ ts) \oplus \oplus$
 $\quad (set_option \circ table_of \ (map \ (\lambda(s,m). (s,I,m)) \ (imethds \ i))))$

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

definition

$accimethds :: prog \Rightarrow pname \Rightarrow qname \Rightarrow (sig, qname \times mhead) \text{ tables}$ **where**
 $accimethds \ G \ pack \ I =$
 $(if \ \vdash I \text{ face } I \text{ accessible-in } pack$
 $\quad \text{then } imethds \ G \ I$
 $\quad \text{else } (\lambda k. \{\}))$

only returns imethds if the interface is accessible

definition

$methd :: prog \Rightarrow qname \Rightarrow (sig, qname \times methd) \text{ table}$ **where**
 $methd \ G \ C =$
 $class_rec \ G \ C \ Map.empty$
 $\quad (\lambda C \ c \ subcls_methds.$
 $\quad \quad filter_tab \ (\lambda sig \ m. \vdash C \text{ inherits method } sig \ m)$
 $\quad \quad \quad subcls_methds$
 $\quad ++$
 $\quad \quad table_of \ (map \ (\lambda(s,m). (s,C,m)) \ (methods \ c)))$

$methd \ G \ C$: 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.

definition

$accmethd :: prog \Rightarrow qname \Rightarrow qname \Rightarrow (sig, qname \times methd) \text{ table}$ **where**
 $accmethd \ G \ S \ C =$
 $filter_tab \ (\lambda sig \ m. \vdash method \ sig \ m \text{ of } C \text{ accessible-from } S) \ (methd \ G \ C)$

$accmethd \ G \ S \ C$: 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*)

definition

$dynmethd :: prog \Rightarrow qname \Rightarrow qname \Rightarrow (sig, qname \times methd) \text{ table}$ **where**
 $dynmethd \ G \ statC \ dynC =$
 $(\lambda sig.$
 $\quad (if \ \vdash dynC \preceq_C \ statC$
 $\quad \quad \text{then } (case \ methd \ G \ statC \ sig \text{ of}$
 $\quad \quad \quad None \Rightarrow None$

```

    | Some statM
      ⇒ (class-rec G dynC Map.empty
        (λC c subcls-mthds.
          subcls-mthds
          ++
          (filter-tab
            (λ - dynM. G, sig ⊢ dynM overrides statM ∨ dynM = statM)
            (methd G C) ))
        ) sig
    )
  else None))

```

dynmethd *G statC dynC*: dynamic method lookup of a reference with dynamic class *dynC* and static class *statC*

Note some kind of duality between *methd* and *dynmethd* in the *class-rec* arguments. Whereas *methd* filters the subclass methods (to get only the inherited ones), *dynmethd* filters the new methods (to get only those methods which actually override the methods of the static class)

definition

```

dynimethd :: prog ⇒ qname ⇒ qname ⇒ (sig, qname × methd) table where
dynimethd G I dynC =
  (λsig. if imethds G I sig ≠ {}
    then methd G dynC sig
    else dynmethd G Object dynC sig)

```

dynimethd *G I dynC*: dynamic method lookup of a reference with dynamic class *dynC* and static interface type *I*

When calling an interface method, we must distinguish if the method signature was defined in the interface or if it must be an Object method in the other case. If it was an interface method we search the class hierarchy starting at the dynamic class of the object up to Object to find the first matching method (*methd*). Since all interface methods have public access the method can't be coalesced due to some odd visibility effects like in case of *dynmethd*. The method will be inherited or overridden in all classes from the first class implementing the interface down to the actual dynamic class.

definition

```

dynlookup :: prog ⇒ ref-ty ⇒ qname ⇒ (sig, qname × methd) table where
dynlookup G statT dynC =
  (case statT of
    NullT      ⇒ Map.empty
    | IfaceT I  ⇒ dynimethd G I      dynC
    | ClassT statC ⇒ dynmethd G statC dynC
    | ArrayT ty  ⇒ dynmethd G Object dynC)

```

dynlookup *G statT dynC*: dynamic lookup of a method within the static reference type *statT* and the dynamic class *dynC*. In a wellformd context *statT* will not be *NullT* and in case *statT* is an array type, *dynC*=Object

definition

```

fields :: prog ⇒ qname ⇒ ((vname × qname) × field) list where
fields G C =
  class-rec G C [] (λC c ts. map (λ(n,t). ((n,C),t)) (cfields c) @ ts)

```

DeclConcepts.fields *G C* list of fields of a class, including all the fields of the superclasses (private, inherited and hidden ones) not only the accessible ones (an instance of a object allocates all these fields)

definition

```

accfield :: prog ⇒ qname ⇒ qname ⇒ (vname, qname × field) table where
accfield G S C =

```

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

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)

definition

is-methd $:: \text{prog} \Rightarrow \text{qname} \Rightarrow \text{sig} \Rightarrow \text{bool}$
where *is-methd* $G = (\lambda C \ \text{sig}. \text{is-class } G \ C \wedge \text{methd } G \ C \ \text{sig} \neq \text{None})$

definition

efname $:: ((\text{vname} \times \text{qname}) \times \text{field}) \Rightarrow (\text{vname} \times \text{qname})$
where *efname* $= \text{fst}$

lemma *efname-simp*[*simp*]: *efname* $(n,f) = n$

$\langle \text{proof} \rangle$

4 imethds

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

$\langle \text{proof} \rangle$

lemma *imethds-norec*:

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

$\langle \text{proof} \rangle$

lemma *imethds-declI*: $\llbracket m \in \text{imethds } G \ I \ \text{sig}; \text{ws-prog } G; \text{is-iface } G \ I \rrbracket \Longrightarrow$

$(\exists i. \text{iface } G \ (\text{decliface } m) = \text{Some } i \wedge$
 $\text{table-of } (\text{imethds } i) \ \text{sig} = \text{Some } (\text{methd } m)) \wedge$
 $(I, \text{decliface } m) \in (\text{subint1 } G)^* \wedge m \in \text{imethds } G \ (\text{decliface } m) \ \text{sig}$

$\langle \text{proof} \rangle$

lemma *imethds-cases*:

assumes *im*: $im \in \text{imethds } G \ I \ \text{sig}$
and *ifI*: $\text{iface } G \ I = \text{Some } i$
and *ws*: $\text{ws-prog } G$
obtains $(\text{NewMethod}) \ \text{table-of } (\text{map } (\lambda(s, mh). (s, I, mh)) (\text{imethds } i)) \ \text{sig} = \text{Some } im$
 $| (\text{InheritedMethod}) \ J \text{ where } J \in \text{set } (\text{isuperIfs } i) \text{ and } im \in \text{imethds } G \ J \ \text{sig}$

$\langle \text{proof} \rangle$

5 accimethd

lemma *accimethds-simp* [*simp*]:

$G \vdash \text{Iface } I \text{ accessible-in pack} \Longrightarrow \text{accimethds } G \ \text{pack } I = \text{imethds } G \ I$

$\langle \text{proof} \rangle$

lemma *accimethdsD*:

$im \in \text{accimethds } G \ \text{pack } I \ \text{sig}$
 $\Longrightarrow im \in \text{imethds } G \ I \ \text{sig} \wedge G \vdash \text{Iface } I \text{ accessible-in pack}$

$\langle \text{proof} \rangle$

lemma *accimethdsI*:

$\llbracket im \in imethds\ G\ I\ sig; G \vdash I \text{face } I \text{ accessible-in pack} \rrbracket$
 $\implies im \in accimethds\ G\ pack\ I\ sig$
 $\langle proof \rangle$

6 methd

lemma *methd-rec*: $\llbracket class\ G\ C = Some\ c; ws-prog\ G \rrbracket \implies$
 $methd\ G\ C$

$= (if\ C = Object$
 $\quad then\ Map.empty$
 $\quad else\ filter-tab\ (\lambda sig\ m.\ G \vdash C\ inherits\ method\ sig\ m)$
 $\quad \quad (methd\ G\ (super\ c)))$
 $++\ table-of\ (map\ (\lambda(s,m).\ (s,C,m))\ (methods\ c))$

$\langle proof \rangle$

lemma *methd-norec*:

$\llbracket class\ G\ declC = Some\ c; ws-prog\ G; table-of\ (methods\ c)\ sig = Some\ m \rrbracket$
 $\implies methd\ G\ declC\ sig = Some\ (declC,\ m)$
 $\langle proof \rangle$

lemma *methd-declC*:

$\llbracket methd\ G\ C\ sig = Some\ m; ws-prog\ G; is-class\ G\ C \rrbracket \implies$
 $(\exists d.\ class\ G\ (declclass\ m) = Some\ d \wedge table-of\ (methods\ d)\ sig = Some\ (methd\ m)) \wedge$
 $G \vdash C \preceq_C (declclass\ m) \wedge methd\ G\ (declclass\ m)\ sig = Some\ m$
 $\langle proof \rangle$

lemma *methd-inheritedD*:

$\llbracket class\ G\ C = Some\ c; ws-prog\ G; methd\ G\ C\ sig = Some\ m \rrbracket$
 $\implies (declclass\ m \neq C \longrightarrow G \vdash C\ inherits\ method\ sig\ m)$
 $\langle proof \rangle$

lemma *methd-diff-cls*:

$\llbracket ws-prog\ G; is-class\ G\ C; is-class\ G\ D;$
 $methd\ G\ C\ sig = m; methd\ G\ D\ sig = n; m \neq n$
 $\rrbracket \implies C \neq D$
 $\langle proof \rangle$

lemma *method-declared-inI*:

$\llbracket table-of\ (methods\ c)\ sig = Some\ m; class\ G\ C = Some\ c \rrbracket$
 $\implies G \vdash mdecl\ (sig,m)\ declared-in\ C$
 $\langle proof \rangle$

lemma *methd-declared-in-declclass*:

$\llbracket methd\ G\ C\ sig = Some\ m; ws-prog\ G; is-class\ G\ C \rrbracket$
 $\implies G \vdash Methd\ sig\ m\ declared-in\ (declclass\ m)$
 $\langle proof \rangle$

lemma *member-methd*:

assumes *member-of*: $G \vdash Methd\ sig\ m\ member-of\ C$ **and**
 $ws: ws-prog\ G$
shows $methd\ G\ C\ sig = Some\ m$
 $\langle proof \rangle$

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

$\langle \text{proof} \rangle$

lemma *finite-dom-methd*:

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

$\langle \text{proof} \rangle$

7 accmethd

lemma *accmethd-SomeD*:

$\text{accmethd } G \ S \ C \ \text{sig} = \text{Some } m$

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

$\langle \text{proof} \rangle$

lemma *accmethd-SomeI*:

$\llbracket \text{methd } G \ C \ \text{sig} = \text{Some } m; G \vdash \text{method sig } m \text{ of } C \text{ accessible-from } S \rrbracket$

$\implies \text{accmethd } G \ S \ C \ \text{sig} = \text{Some } m$

$\langle \text{proof} \rangle$

lemma *accmethd-declC*:

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

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

$G \vdash C \preceq_C (\text{declclass } m) \wedge \text{methd } G \ (\text{declclass } m) \ \text{sig} = \text{Some } m \wedge$
 $G \vdash \text{method sig } m \text{ of } C \text{ accessible-from } S$

$\langle \text{proof} \rangle$

lemma *finite-dom-accmethd*:

$\llbracket \text{ws-prog } G; \text{ is-class } G \ C \rrbracket \implies \text{finite } (\text{dom } (\text{accmethd } G \ S \ C))$

$\langle \text{proof} \rangle$

8 dynmethd

lemma *dynmethd-rec*:

$\llbracket \text{class } G \ \text{dynC} = \text{Some } c; \text{ws-prog } G \rrbracket \implies$

$\text{dynmethd } G \ \text{statC} \ \text{dynC} \ \text{sig}$

$= (\text{if } G \vdash \text{dynC} \preceq_C \text{statC}$

then (case methd $G \ \text{statC} \ \text{sig}$ of

None \Rightarrow None

| Some statM

\Rightarrow (case methd $G \ \text{dynC} \ \text{sig}$ of

None $\Rightarrow \text{dynmethd } G \ \text{statC} \ (\text{super } c) \ \text{sig}$

| Some dynM \Rightarrow

(if $G, \text{sig} \vdash \text{dynM}$ overrides statM $\vee \text{dynM} = \text{statM}$

then Some dynM

else (dynmethd $G \ \text{statC} \ (\text{super } c) \ \text{sig}$)

)))

else None)

(is - \implies - \implies ?Dynmethd-def dynC sig = ?Dynmethd-rec dynC c sig)

$\langle \text{proof} \rangle$

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

$\implies \text{dynmethd } G \ C \ C \ \text{sig} = \text{methd } G \ C \ \text{sig}$

$\langle \text{proof} \rangle$

lemma *dynmethdSomeD*:

$\llbracket \text{dynmethd } G \ \text{statC} \ \text{dynC} \ \text{sig} = \text{Some } \text{dynM}; \text{is-class } G \ \text{dynC}; \text{ws-prog } G \rrbracket$

$\implies G \vdash \text{dynC} \preceq_C \text{statC} \wedge (\exists \text{statM}. \text{methd } G \ \text{statC} \ \text{sig} = \text{Some } \text{statM})$

$\langle \text{proof} \rangle$

lemma *dynmethd-Some-cases:*

assumes $\text{dynM}: \text{dynmethd } G \text{ statC dynC sig} = \text{Some dynM}$
and $\text{is-clc-dynC}: \text{is-class } G \text{ dynC}$
and $\text{ws}: \text{ws-prog } G$
obtains $(\text{Static}) \text{ methd } G \text{ statC sig} = \text{Some dynM}$
 $| (\text{Overrides}) \text{ statM}$
where $\text{methd } G \text{ statC sig} = \text{Some statM}$
and $\text{dynM} \neq \text{statM}$
and $G, \text{sig} \vdash \text{dynM overrides statM}$

$\langle \text{proof} \rangle$

lemma *no-override-in-Object:*

assumes $\text{dynM}: \text{dynmethd } G \text{ statC dynC sig} = \text{Some dynM}$ **and**
 $\text{is-clc-dynC}: \text{is-class } G \text{ dynC}$ **and**
 $\text{ws}: \text{ws-prog } G$ **and**
 $\text{statM}: \text{methd } G \text{ statC sig} = \text{Some statM}$ **and**
 $\text{neq-dynM-statM}: \text{dynM} \neq \text{statM}$
shows $\text{dynC} \neq \text{Object}$

$\langle \text{proof} \rangle$

lemma *dynmethd-Some-rec-cases:*

assumes $\text{dynM}: \text{dynmethd } G \text{ statC dynC sig} = \text{Some dynM}$
and $\text{clsDynC}: \text{class } G \text{ dynC} = \text{Some } c$
and $\text{ws}: \text{ws-prog } G$
obtains $(\text{Static}) \text{ methd } G \text{ statC sig} = \text{Some dynM}$
 $| (\text{Override}) \text{ statM}$ **where** $\text{methd } G \text{ statC sig} = \text{Some statM}$
and $\text{methd } G \text{ dynC sig} = \text{Some dynM}$ **and** $\text{statM} \neq \text{dynM}$
and $G, \text{sig} \vdash \text{dynM overrides statM}$
 $| (\text{Recursion}) \text{ dynC} \neq \text{Object}$ **and** $\text{dynmethd } G \text{ statC (super } c) \text{ sig} = \text{Some dynM}$

$\langle \text{proof} \rangle$

lemma *dynmethd-declC:*

$\llbracket \text{dynmethd } G \text{ statC dynC sig} = \text{Some } m; \text{is-class } G \text{ statC}; \text{ws-prog } G \rrbracket \implies$
 $(\exists d. \text{class } G (\text{declclass } m) = \text{Some } d \wedge \text{table-of (methods } d) \text{ sig} = \text{Some (methd } m)) \wedge$
 $G \vdash \text{dynC} \preceq_C (\text{declclass } m) \wedge \text{methd } G (\text{declclass } m) \text{ sig} = \text{Some } m$

$\langle \text{proof} \rangle$

lemma *methd-Some-dynmethd-Some:*

assumes $\text{statM}: \text{methd } G \text{ statC sig} = \text{Some statM}$ **and**
 $\text{subclseq}: G \vdash \text{dynC} \preceq_C \text{statC}$ **and**
 $\text{is-clc-statC}: \text{is-class } G \text{ statC}$ **and**
 $\text{ws}: \text{ws-prog } G$
shows $\exists \text{ dynM}. \text{dynmethd } G \text{ statC dynC sig} = \text{Some dynM}$
(is ?P dynC)

$\langle \text{proof} \rangle$

lemma *dynmethd-cases:*

assumes $\text{statM}: \text{methd } G \text{ statC sig} = \text{Some statM}$
and $\text{subclseq}: G \vdash \text{dynC} \preceq_C \text{statC}$
and $\text{is-clc-statC}: \text{is-class } G \text{ statC}$
and $\text{ws}: \text{ws-prog } G$
obtains $(\text{Static}) \text{ dynmethd } G \text{ statC dynC sig} = \text{Some statM}$
 $| (\text{Overrides}) \text{ dynM}$ **where** $\text{dynmethd } G \text{ statC dynC sig} = \text{Some dynM}$
and $\text{dynM} \neq \text{statM}$ **and** $G, \text{sig} \vdash \text{dynM overrides statM}$

$\langle \text{proof} \rangle$

lemma *ws-dynmethd*:

assumes *statM*: *methd G statC sig = Some statM* **and**

subclseq: $G \vdash \text{dyn}C \preceq_C \text{stat}C$ **and**

is-cla-statC: *is-class G statC* **and**

ws: *ws-prog G*

shows

$\exists \text{ dyn}M. \text{dynmethd } G \text{ stat}C \text{ dyn}C \text{ sig} = \text{Some dyn}M \wedge$

$\text{is-static dyn}M = \text{is-static stat}M \wedge G \vdash \text{resTy dyn}M \preceq_{\text{resTy}} \text{stat}M$

$\langle \text{proof} \rangle$

9 dynlookup

lemma *dynlookup-cases*:

assumes *dynlookup G statT dynC sig = x*

obtains $(\text{Null}T) \text{ stat}T = \text{Null}T$ **and** $\text{Map.empty sig} = x$

| $(\text{Iface}T) I$ **where** $\text{stat}T = \text{Iface}T I$ **and** $\text{dynmethd } G I \text{ dyn}C \text{ sig} = x$

| $(\text{Class}T) \text{ stat}C$ **where** $\text{stat}T = \text{Class}T \text{ stat}C$ **and** $\text{dynmethd } G \text{ stat}C \text{ dyn}C \text{ sig} = x$

| $(\text{Array}T) ty$ **where** $\text{stat}T = \text{Array}T ty$ **and** $\text{dynmethd } G \text{ Object dyn}C \text{ sig} = x$

$\langle \text{proof} \rangle$

10 fields

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

$\text{fields } G C = \text{map } (\lambda(fn, ft). ((fn, C), ft)) (\text{cfields } c) @$

$(\text{if } C = \text{Object} \text{ then } [] \text{ else } \text{fields } G (\text{super } c))$

$\langle \text{proof} \rangle$

lemma *fields-norec*:

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

$\implies \text{table-of } (\text{fields } G fd) (fn, fd) = \text{Some } f$

$\langle \text{proof} \rangle$

lemma *table-of-fieldsD*:

$\text{table-of } (\text{map } (\lambda(fn, ft). ((fn, C), ft)) (\text{cfields } c)) \text{efn} = \text{Some } f$

$\implies (\text{declclassf efn}) = C \wedge \text{table-of } (\text{cfields } c) (fname \text{efn}) = \text{Some } f$

$\langle \text{proof} \rangle$

lemma *fields-declC*:

$\llbracket \text{table-of } (\text{fields } G C) \text{efn} = \text{Some } f; \text{ws-prog } G; \text{is-class } G C \rrbracket \implies$

$(\exists d. \text{class } G (\text{declclassf efn}) = \text{Some } d \wedge$

$\text{table-of } (\text{cfields } d) (fname \text{efn}) = \text{Some } f) \wedge$

$G \vdash C \preceq_C (\text{declclassf efn}) \wedge \text{table-of } (\text{fields } G (\text{declclassf efn})) \text{efn} = \text{Some } f$

$\langle \text{proof} \rangle$

lemma *fields-emptyI*: $\bigwedge y. \llbracket \text{ws-prog } G; \text{class } G C = \text{Some } c; \text{cfields } c = [];$

$C \neq \text{Object} \longrightarrow \text{class } G (\text{super } c) = \text{Some } y \wedge \text{fields } G (\text{super } c) = [] \rrbracket \implies$

$\text{fields } G C = []$

$\langle \text{proof} \rangle$

lemma *fields-mono-lemma*:

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

$\implies x \in \text{set } (\text{fields } G D)$

$\langle \text{proof} \rangle$

lemma *ws-unique-fields-lemma*:

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

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

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

11 accfield

lemma *accfield-fields*:

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

lemma *accfield-declC-is-class*:

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

lemma *accfield-accessibleD*:

$\text{accfield } G \ S \ C \ fn = \text{Some } f \implies G \vdash \text{Field } fn \ f \text{ of } C \text{ accessible-from } S$
 $\langle \text{proof} \rangle$

12 is methd

lemma *is-methdI*:

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

lemma *is-methdD*:

$\text{is-methd } G \ C \ sig \implies \text{class } G \ C \neq \text{None} \wedge \text{methd } G \ C \ sig \neq \text{None}$
 $\langle \text{proof} \rangle$

lemma *finite-is-methd*:

$\text{ws-prog } G \implies \text{finite } (\text{Collect } (\text{case-prod } (\text{is-methd } G)))$
 $\langle \text{proof} \rangle$

calculation of the superclasses of a class

definition

$\text{superclasses} :: \text{prog} \Rightarrow \text{qtname} \Rightarrow \text{qtname set}$ **where**
 $\text{superclasses } G \ C = \text{class-rec } G \ C \ \{\}$
 $\quad (\lambda \ C \ c \ \text{superclss}. \text{if } C = \text{Object}$
 $\quad \quad \text{then } \{\}$
 $\quad \quad \text{else insert } (\text{super } c) \ \text{superclss}))$

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

$\text{superclasses } G \ C$
 $= (\text{if } (C = \text{Object})$
 $\quad \text{then } \{\}$
 $\quad \text{else insert } (\text{super } c) \ (\text{superclasses } G \ (\text{super } c)))$
 $\langle \text{proof} \rangle$

lemma *superclasses-mono*:

assumes *clsrel*: $G \vdash C \prec_C D$
and *ws*: *ws-prog* G
and *cls-C*: *class* G $C = \text{Some } c$
and *wf*: $\bigwedge C \ c. \llbracket \text{class } G \ C = \text{Some } c; C \neq \text{Object} \rrbracket$
 $\implies \exists \text{sc. } \text{class } G (\text{super } c) = \text{Some } \text{sc}$
and *x*: $x \in \text{superclasses } G \ D$
shows $x \in \text{superclasses } G \ C$ *<proof>*

lemma *subclsEval*:

assumes *clsrel*: $G \vdash C \prec_C D$
and *ws*: *ws-prog* G
and *cls-C*: *class* G $C = \text{Some } c$
and *wf*: $\bigwedge C \ c. \llbracket \text{class } G \ C = \text{Some } c; C \neq \text{Object} \rrbracket$
 $\implies \exists \text{sc. } \text{class } G (\text{super } c) = \text{Some } \text{sc}$
shows $D \in \text{superclasses } G \ C$ *<proof>*

end

Chapter 11

WellType

1 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.

type-synonym *lenv*
= (*lname*, *ty*) *table* — local variables, including This and Result

record *env* =
 prg:: *prog* — program
 cls:: *qtname* — current package and class name
 lcl:: *lenv* — local environment

translations
(*type*) *lenv* <= (*type*) (*lname*, *ty*) *table*
(*type*) *lenv* <= (*type*) *lname* \Rightarrow *ty option*
(*type*) *env* <= (*type*) (\langle *prg*::*prog*,*cls*::*qtname*,*lcl*::*lenv* \rangle)
(*type*) *env* <= (*type*) (\langle *prg*::*prog*,*cls*::*qtname*,*lcl*::*lenv*,...::'*a*' \rangle)

abbreviation
pkg :: *env* \Rightarrow *pname* — select the current package from an environment
where *pkg e* == *pid (cls e)*

Static overloading: maximally specific methods

type-synonym

$emhead = ref\text{-}ty \times mhead$

— Some mnemonic selectors for `emhead`

definition

$declrefT :: emhead \Rightarrow ref\text{-}ty$

where $declrefT = fst$

definition

$mhd :: emhead \Rightarrow mhead$

where $mhd \equiv snd$

lemma $declrefT\text{-}simp[simp]: declrefT (r, m) = r$

$\langle proof \rangle$

lemma $mhd\text{-}simp[simp]: mhd (r, m) = m$

$\langle proof \rangle$

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

$\langle proof \rangle$

lemma $mhd\text{-}resTy\text{-}simp [simp]: resTy (mhd m) = resTy m$

$\langle proof \rangle$

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

$\langle proof \rangle$

lemma $mhd\text{-}accmodi\text{-}simp [simp]: accmodi (mhd m) = accmodi m$

$\langle proof \rangle$

definition

$cmheads :: prog \Rightarrow qname \Rightarrow qname \Rightarrow sig \Rightarrow emhead \text{ set}$

where $cmheads G S C = (\lambda sig. (\lambda (Cls, mthd). (ClassT Cls, (mhead mthd)))) \text{ ‘ set-option (accmethd } G S C \text{ sig) }$

definition

$Objectmheads :: prog \Rightarrow qname \Rightarrow sig \Rightarrow emhead \text{ set}$ **where**

$Objectmheads G S =$

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

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

definition

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

where

$accObjectmheads G S T =$

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

$\text{ then } Objectmheads G S$

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

primrec $mheads :: prog \Rightarrow qname \Rightarrow ref\text{-}ty \Rightarrow sig \Rightarrow emhead \text{ set}$

where

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

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

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

$accObjectmheads G S (IfaceT I) sig)$

$| mheads G S (ClassT C) = cmheads G S C$

$| mheads G S (ArrayT T) = accObjectmheads G S (ArrayT T)$

definition

— applicable methods, cf. 15.11.2.1

$appl\text{-}methds :: prog \Rightarrow qname \Rightarrow ref\text{-}ty \Rightarrow sig \Rightarrow (emhead \times ty\ list) \text{ set } \mathbf{where}$
 $appl\text{-}methds\ G\ S\ rt = (\lambda\ sig.$
 $\{(mh, pTs') \mid mh\ pTs'.\ mh \in mheads\ G\ S\ rt\ (\downarrow name = name\ sig, parTs = pTs') \wedge$
 $G \vdash (parTs\ sig) [\preceq] pTs'\})$

definition

— more specific methods, cf. 15.11.2.2

$more\text{-}spec :: prog \Rightarrow emhead \times ty\ list \Rightarrow emhead \times ty\ list \Rightarrow bool\ \mathbf{where}$
 $more\text{-}spec\ G = (\lambda(mh, pTs). \lambda(mh', pTs').\ G \vdash pTs [\preceq] pTs')$

definition

— maximally specific methods, cf. 15.11.2.2

$max\text{-}spec :: prog \Rightarrow qname \Rightarrow ref\text{-}ty \Rightarrow sig \Rightarrow (emhead \times ty\ list) \text{ set } \mathbf{where}$
 $max\text{-}spec\ G\ S\ rt\ sig = \{m.\ m \in appl\text{-}methds\ G\ S\ rt\ sig \wedge$
 $(\forall m' \in appl\text{-}methds\ G\ S\ rt\ sig.\ more\text{-}spec\ G\ m'\ m \longrightarrow m' = m)\}$

lemma $max\text{-}spec2appl\text{-}methds$:

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

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

$mh \in mheads\ G\ S\ T\ (\downarrow name = mn, parTs = pTs') \wedge G \vdash pTs [\preceq] pTs'$
 $\langle proof \rangle$

lemma $max\text{-}spec2mheads$:

$max\text{-}spec\ G\ S\ rt\ (\downarrow name = mn, parTs = pTs') = insert\ (mh, pTs')\ A$
 $\implies mh \in mheads\ G\ S\ rt\ (\downarrow name = mn, parTs = pTs') \wedge G \vdash pTs [\preceq] pTs'$
 $\langle proof \rangle$

definition

$empty\text{-}dt :: dyn\text{-}ty$
 $\mathbf{where}\ empty\text{-}dt = (\lambda a.\ None)$

definition

$invmode :: ('a :: type) member\text{-}scheme \Rightarrow expr \Rightarrow inv\text{-}mode\ \mathbf{where}$
 $invmode\ m\ e = (\text{if } is\text{-}static\ m$
 $\text{then } Static$
 $\text{else if } e = Super \text{ then } SuperM \text{ else } IntVir)$

lemma $invmode\text{-}nonstatic\ [simp]$:

$invmode\ (\downarrow access = a, static = False, \dots = x)\ (Acc\ (LVar\ e)) = IntVir$
 $\langle proof \rangle$

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

$\langle proof \rangle$

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

$\langle proof \rangle$

lemma $Null\text{-}staticD$:

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

$\langle \text{proof} \rangle$

Typing for unary operations

primrec *unop-type* :: *unop* \Rightarrow *prim-ty*

where

unop-type *UPlus* = *Integer*
 | *unop-type* *UMinus* = *Integer*
 | *unop-type* *UBitNot* = *Integer*
 | *unop-type* *UNot* = *Boolean*

primrec *wt-unop* :: *unop* \Rightarrow *ty* \Rightarrow *bool*

where

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

primrec *binop-type* :: *binop* \Rightarrow *prim-ty*

where

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*

primrec *wt-binop* :: *prog* \Rightarrow *binop* \Rightarrow *ty* \Rightarrow *ty* \Rightarrow *bool*

where

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*))

<i>wt-binop</i> <i>G Eq</i>	$t1\ t2 = (G \vdash t1 \preceq t2 \vee G \vdash t2 \preceq t1)$
<i>wt-binop</i> <i>G Neq</i>	$t1\ t2 = (G \vdash t1 \preceq t2 \vee G \vdash t2 \preceq t1)$
<i>wt-binop</i> <i>G BitAnd</i>	$t1\ t2 = ((t1 = \text{PrimT Integer}) \wedge (t2 = \text{PrimT Integer}))$
<i>wt-binop</i> <i>G And</i>	$t1\ t2 = ((t1 = \text{PrimT Boolean}) \wedge (t2 = \text{PrimT Boolean}))$
<i>wt-binop</i> <i>G BitXor</i>	$t1\ t2 = ((t1 = \text{PrimT Integer}) \wedge (t2 = \text{PrimT Integer}))$
<i>wt-binop</i> <i>G Xor</i>	$t1\ t2 = ((t1 = \text{PrimT Boolean}) \wedge (t2 = \text{PrimT Boolean}))$
<i>wt-binop</i> <i>G BitOr</i>	$t1\ t2 = ((t1 = \text{PrimT Integer}) \wedge (t2 = \text{PrimT Integer}))$
<i>wt-binop</i> <i>G Or</i>	$t1\ t2 = ((t1 = \text{PrimT Boolean}) \wedge (t2 = \text{PrimT Boolean}))$
<i>wt-binop</i> <i>G CondAnd</i>	$t1\ t2 = ((t1 = \text{PrimT Boolean}) \wedge (t2 = \text{PrimT Boolean}))$
<i>wt-binop</i> <i>G CondOr</i>	$t1\ t2 = ((t1 = \text{PrimT Boolean}) \wedge (t2 = \text{PrimT Boolean}))$

Typing for terms

type-synonym $tys = ty + ty\ list$

translations

$$(type) \ tys \leq (type) \ ty + ty \ list$$
$$\begin{aligned}
\textbf{inductive } wt :: env \Rightarrow dyn\text{-}ty \Rightarrow [term, tys] \Rightarrow bool \quad (<-,\models\text{-}::> [51,51,51,51] 50) \\
\textbf{and } wt\text{-}stmt :: env \Rightarrow dyn\text{-}ty \Rightarrow stmt \Rightarrow bool \quad (<-,\models\text{-}::\surd> [51,51,51] 50) \\
\textbf{and } ty\text{-}expr :: env \Rightarrow dyn\text{-}ty \Rightarrow [expr, ty] \Rightarrow bool \quad (<-,\models\text{-}::\text{--}> [51,51,51,51] 50) \\
\textbf{and } ty\text{-}var :: env \Rightarrow dyn\text{-}ty \Rightarrow [var, ty] \Rightarrow bool \quad (<-,\models\text{-}::\text{==}> [51,51,51,51] 50) \\
\textbf{and } ty\text{-}exprs :: env \Rightarrow dyn\text{-}ty \Rightarrow [expr\ list, ty\ list] \Rightarrow bool \\
\quad (<-,\models\text{-}::\dot{=}> [51,51,51,51] 50)
\end{aligned}$$
$$\begin{array}{l} E, dt \models s::\sqrt{} \equiv E, dt \models \text{In}1r \ s::\text{In}l \ (\text{Prim}T \ \text{Void}) \\ E, dt \models e::-T \equiv E, dt \models \text{In}1l \ e::\text{In}l \ T \\ E, dt \models e::=T \equiv E, dt \models \text{In}2 \ e::\text{In}l \ T \\ E, dt \models e::\dot{=}T \equiv E, dt \models \text{In}3 \ e::\text{In}r \ T \end{array}$$

- well-typed statements

$$| \textit{Skip}: E, dt | = \textit{Skip} :: \sqrt{}$$
$$\mid Expr: \llbracket E, dt \mid e :: -T \rrbracket \Longrightarrow E, dt \mid Expr \ e :: \surd$$

— cf. 14.6

$$| \text{Lab: } E, dt \models c::\sqrt{} \implies E, dt \models l. \ c::\sqrt{}$$
$$\mid \textit{Comp}: \llbracket E, dt \rrbracket = c1 :: \sqrt{}; \\ E, dt \rrbracket = c2 :: \sqrt{} \Longrightarrow E, dt \rrbracket = c1;; c2 :: \sqrt{}$$

— cf. 14.8

$$\begin{array}{l} | \text{ If: } \llbracket E, dt \models e :: -\text{PrimT Boolean}; \\ \quad E, dt \models c1 :: \checkmark; \\ \quad E, dt \models c2 :: \checkmark \rrbracket \implies \\ \quad E, dt \models \text{If}(e) \ c1 \ \text{Else} \ c2 :: \checkmark \end{array}$$

— cf. 14.10

$$\begin{array}{l} | \text{Loop: } \llbracket E, dt \rrbracket = e :: -\text{PrimT Boolean}; \\ \quad E, dt \rrbracket = c :: \sqrt{} \rrbracket \implies \\ \hspace{15em} E, dt \rrbracket = l. \text{ While}(e) \ c :: \sqrt{} \end{array}$$

— cf. 14.13, 14.15, 14.16

$$| \text{Imp} : E, dt \models \text{Imp} \text{ jump} :: \sqrt{}$$

— cf. 14.16

- | *Throw*: $\llbracket E, dt \models e :: - \text{Class } tn; \text{prg } E \vdash tn \preceq_C \text{ SXcpt Throwable} \rrbracket \Longrightarrow$

$$E, dt \models \text{Throw } e :: \checkmark$$
- cf. 14.18
- | *Try*: $\llbracket E, dt \models c1 :: \checkmark; \text{prg } E \vdash tn \preceq_C \text{ SXcpt Throwable}; \text{lcl } E (VName \text{ } vn) = \text{None}; E \llbracket \text{lcl} := (\text{lcl } E)(VName \text{ } vn \mapsto \text{Class } tn) \rrbracket, dt \models c2 :: \checkmark \rrbracket$

$$\Longrightarrow$$

$$E, dt \models \text{Try } c1 \text{ Catch}(tn \text{ } vn) \text{ } c2 :: \checkmark$$
- cf. 14.18
- | *Fin*: $\llbracket E, dt \models c1 :: \checkmark; E, dt \models c2 :: \checkmark \rrbracket \Longrightarrow$

$$E, dt \models c1 \text{ Finally } c2 :: \checkmark$$
- | *Init*: $\llbracket \text{is-class } (\text{prg } E) \text{ } C \rrbracket \Longrightarrow$

$$E, dt \models \text{Init } C :: \checkmark$$
 - *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
- | *NewC*: $\llbracket \text{is-acc-class } (\text{prg } E) (\text{pkg } E) \text{ } C \rrbracket \Longrightarrow$

$$E, dt \models \text{NewC } C :: - \text{Class } C$$
- cf. 15.9
- | *NewA*: $\llbracket \text{is-acc-type } (\text{prg } E) (\text{pkg } E) \text{ } T; E, dt \models i :: - \text{PrimT Integer} \rrbracket \Longrightarrow$

$$E, dt \models \text{New } T[i] :: - T.$$
- cf. 15.15
- | *Cast*: $\llbracket E, dt \models e :: - T; \text{is-acc-type } (\text{prg } E) (\text{pkg } E) \text{ } T'; \text{prg } E \vdash T \preceq^? T' \rrbracket \Longrightarrow$

$$E, dt \models \text{Cast } T' \text{ } e :: - T'$$
- cf. 15.19.2
- | *Inst*: $\llbracket E, dt \models e :: - \text{RefT } T; \text{is-acc-type } (\text{prg } E) (\text{pkg } E) (\text{RefT } T'); \text{prg } E \vdash \text{RefT } T \preceq^? \text{RefT } T' \rrbracket \Longrightarrow$

$$E, dt \models e \text{ InstOf } T' :: - \text{PrimT Boolean}$$
- cf. 15.7.1
- | *Lit*: $\llbracket \text{typeof } dt \text{ } x = \text{Some } T \rrbracket \Longrightarrow$

$$E, dt \models \text{Lit } x :: - T$$
- | *UnOp*: $\llbracket E, dt \models e :: - T_e; \text{wt-unop unop } T_e; T = \text{PrimT } (\text{unop-type unop}) \rrbracket$

$$\Longrightarrow$$

$$E, dt \models \text{UnOp unop } e :: - T$$
- | *BinOp*: $\llbracket E, dt \models e1 :: - T1; E, dt \models e2 :: - T2; \text{wt-binop } (\text{prg } E) \text{ binop } T1 \text{ } T2; T = \text{PrimT } (\text{binop-type binop}) \rrbracket$

$$\Longrightarrow$$

$$E, dt \models \text{BinOp binop } e1 \text{ } e2 :: - T$$
- cf. 15.10.2, 15.11.1
- | *Super*: $\llbracket \text{lcl } E \text{ This} = \text{Some } (\text{Class } C); C \neq \text{Object}; \text{class } (\text{prg } E) \text{ } C = \text{Some } c \rrbracket \Longrightarrow$

$$E, dt \models \text{Super} :: - \text{Class } (\text{super } c)$$
- cf. 15.13.1, 15.10.1, 15.12
- | *Acc*: $\llbracket E, dt \models va :: T \rrbracket \Longrightarrow$

$$E, dt \models \text{Acc } va :: - T$$

— cf. 15.25, 15.25.1

| *Ass*: $\llbracket E, dt \models va ::= T; va \neq LVar \text{ This};$
 $E, dt \models v ::= T';$
 $prg \vdash T' \preceq T \rrbracket \implies$
 $E, dt \models va ::= v ::= T'$

— cf. 15.24

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

— cf. 15.11.1, 15.11.2, 15.11.3

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

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

— The class C is the dynamic class of the method call (cf. Eval.thy). It hasn't got to be directly accessible from the current package $pkg \ E$. Only the static class must be accessible (ensured indirectly by *Call*). Note that l is just a dummy value. It is only used in the smallstep semantics. To proof typesafety directly for the smallstep semantics we would have to assume conformance of l here!

| *Body*: $\llbracket \text{is-class } (prg \ E) \ D;$
 $E, dt \models blk ::= \checkmark;$
 $(lcl \ E) \ \text{Result} = \text{Some } T;$
 $\text{is-type } (prg \ E) \ T \rrbracket \implies$
 $E, dt \models \text{Body } D \ \text{blk} ::= T$

— The class D implementing the method must not directly be accessible from the current package $pkg \ E$, but can also be indirectly accessible due to inheritance (ensured in *Call*) The result type hasn't got to be accessible in Java! (If it is not accessible you can only assign it to Object). For dummy value l see rule *Methd*.

— well-typed variables

— cf. 15.13.1

| *LVar*: $\llbracket lcl \ E \ vn = \text{Some } T; \text{is-acc-type } (prg \ E) \ (pkg \ E) \ T \rrbracket \implies$
 $E, dt \models LVar \ vn ::= T$

— cf. 15.10.1

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

— cf. 15.12

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

— well-typed expression lists

— cf. 15.11.???

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

— cf. 15.11.???

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

$E, dt \models e \# es :: \doteq T \# Ts$

abbreviation

$wt\text{-}syntax :: env \Rightarrow [term, tys] \Rightarrow bool \ (\langle \vdash :: - \rangle \ [51, 51, 51] \ 50)$
where $E \vdash t :: T == E, empty\text{-}dt \models t :: T$

abbreviation

$wt\text{-}stmt\text{-}syntax :: env \Rightarrow stmt \Rightarrow bool \ (\langle \vdash :: - \rangle \ [51, 51] \ 50)$
where $E \vdash s :: \surd == E \vdash In1r \ s :: In1 \ (PrimT \ Void)$

abbreviation

$ty\text{-}expr\text{-}syntax :: env \Rightarrow [expr, ty] \Rightarrow bool \ (\langle \vdash :: - \rangle \ [51, 51, 51] \ 50)$
where $E \vdash e :: - T == E \vdash In1l \ e :: In1 \ T$

abbreviation

$ty\text{-}var\text{-}syntax :: env \Rightarrow [var, ty] \Rightarrow bool \ (\langle \vdash :: - \rangle \ [51, 51, 51] \ 50)$
where $E \vdash e :: T == E \vdash In2 \ e :: In1 \ T$

abbreviation

$ty\text{-}exprs\text{-}syntax :: env \Rightarrow [expr \ list, ty \ list] \Rightarrow bool \ (\langle \vdash :: - \rangle \ [51, 51, 51] \ 50)$
where $E \vdash e :: \doteq T == E \vdash In3 \ e :: Inr \ T$

notation (ASCII)

$wt\text{-}syntax \ (\langle \vdash :: - \rangle \ [51, 51, 51] \ 50)$ **and**
 $wt\text{-}stmt\text{-}syntax \ (\langle \vdash :: - \rangle \ [51, 51] \ 50)$ **and**
 $ty\text{-}expr\text{-}syntax \ (\langle \vdash :: - \rangle \ [51, 51, 51] \ 50)$ **and**
 $ty\text{-}var\text{-}syntax \ (\langle \vdash :: - \rangle \ [51, 51, 51] \ 50)$ **and**
 $ty\text{-}exprs\text{-}syntax \ (\langle \vdash :: - \rangle \ [51, 51, 51] \ 50)$

declare $not\text{-}None\text{-}eq \ [simp \ del]$

declare $if\text{-}split \ [split \ del] \ if\text{-}split\text{-}asm \ [split \ del]$

declare $split\text{-}paired\text{-}All \ [simp \ del] \ split\text{-}paired\text{-}Ex \ [simp \ del]$

$\langle ML \rangle$

inductive-cases $wt\text{-}elim\text{-}cases \ [cases \ set]:$

$E, dt \models In2 \ (LVar \ vn) \quad \quad \quad :: T$
 $E, dt \models In2 \ (\{accC, statDeclC, s\} e..fn) :: T$
 $E, dt \models In2 \ (e.[i]) \quad \quad \quad \quad \quad :: T$
 $E, dt \models In1l \ (NewC \ C) \quad \quad \quad \quad \quad :: T$
 $E, dt \models In1l \ (New \ T'[i]) \quad \quad \quad \quad \quad :: T$
 $E, dt \models In1l \ (Cast \ T' \ e) \quad \quad \quad \quad \quad :: T$
 $E, dt \models In1l \ (e \ InstOf \ T') \quad \quad \quad \quad \quad :: T$
 $E, dt \models In1l \ (Lit \ x) \quad \quad \quad \quad \quad \quad :: T$
 $E, dt \models In1l \ (UnOp \ unop \ e) \quad \quad \quad \quad \quad :: T$
 $E, dt \models In1l \ (BinOp \ binop \ e1 \ e2) \quad \quad \quad :: T$
 $E, dt \models In1l \ (Super) \quad \quad \quad \quad \quad \quad \quad :: T$
 $E, dt \models In1l \ (Acc \ va) \quad \quad \quad \quad \quad \quad \quad :: T$
 $E, dt \models In1l \ (Ass \ va \ v) \quad \quad \quad \quad \quad \quad \quad :: T$
 $E, dt \models In1l \ (e0 \ ? \ e1 : e2) \quad \quad \quad \quad \quad :: T$
 $E, dt \models In1l \ (\{accC, statT, mode\} e.mn(\{pT'\}p)) :: T$
 $E, dt \models In1l \ (Methd \ C \ sig) \quad \quad \quad \quad \quad :: T$
 $E, dt \models In1l \ (Body \ D \ blk) \quad \quad \quad \quad \quad \quad :: T$
 $E, dt \models In3 \ (\[]) \quad \quad \quad \quad \quad \quad \quad \quad \quad :: Ts$
 $E, dt \models In3 \ (e \# es) \quad \quad \quad \quad \quad \quad \quad \quad :: Ts$

$$\begin{array}{ll}
E, dt \models \text{In1r } \text{Skip} & ::x \\
E, dt \models \text{In1r } (\text{Expr } e) & ::x \\
E, dt \models \text{In1r } (c1;; c2) & ::x \\
E, dt \models \text{In1r } (l \bullet c) & ::x \\
E, dt \models \text{In1r } (\text{If}(e) \ c1 \ \text{Else } c2) & ::x \\
E, dt \models \text{In1r } (l \bullet \text{While}(e) \ c) & ::x \\
E, dt \models \text{In1r } (\text{Jump } \text{jump}) & ::x \\
E, dt \models \text{In1r } (\text{Throw } e) & ::x \\
E, dt \models \text{In1r } (\text{Try } c1 \ \text{Catch}(tn \ vn) \ c2) & ::x \\
E, dt \models \text{In1r } (c1 \ \text{Finally } c2) & ::x \\
E, dt \models \text{In1r } (\text{Init } C) & ::x
\end{array}$$

declare *not-None-eq* [simp]
declare *if-split* [split] *if-split-asm* [split]
declare *split-paired-All* [simp] *split-paired-Ex* [simp]
 $\langle ML \rangle$

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

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*:

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

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

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]:
 $(\forall T. (\exists T'. T = \text{Inj } T' \wedge P T') \longrightarrow Q T) = (\forall T'. P T' \longrightarrow Q (\text{Inj } T'))$
 $\langle \text{proof} \rangle$

lemma *single-valued-tys-lemma* [rule-format (no-asm)]:
 $\forall S T. G \vdash S \preceq T \longrightarrow G \vdash T \preceq S \longrightarrow S = T \implies E, dt \models t :: T \implies$
 $G = \text{prg } E \longrightarrow (\forall T'. E, dt \models t :: T' \longrightarrow T = T')$
 $\langle \text{proof} \rangle$

lemma *single-valued-tys*:

ws-prog (prg E) \implies *single-valued* $\{(t, T). E, dt \models t :: T\}$

\langle *proof* \rangle

lemma *typeof-empty-is-type*: *typeof* $(\lambda a. \text{None})\ v = \text{Some } T \implies$ *is-type* $G\ T$

\langle *proof* \rangle

lemma *typeof-is-type*: $(\forall a. v \neq \text{Addr } a) \implies \exists T. \text{typeof } dt\ v = \text{Some } T \wedge$ *is-type* $G\ T$

\langle *proof* \rangle

end

Chapter 12

DefiniteAssignment

1 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.

primrec *jumpNestingOkS* :: *jump set* \Rightarrow *stmt* \Rightarrow *bool*

where

$\text{jumpNestingOkS } \text{jmps } (\text{Skip}) = \text{True}$
 $\text{jumpNestingOkS } \text{jmps } (\text{Expr } e) = \text{True}$
 $\text{jumpNestingOkS } \text{jmps } (j \bullet s) = \text{jumpNestingOkS } (\{j\} \cup \text{jmps}) s$
 $\text{jumpNestingOkS } \text{jmps } (c1;;c2) = (\text{jumpNestingOkS } \text{jmps } c1 \wedge \text{jumpNestingOkS } \text{jmps } c2)$
 $\text{jumpNestingOkS } \text{jmps } (\text{If}(e) c1 \text{ Else } c2) = (\text{jumpNestingOkS } \text{jmps } c1 \wedge \text{jumpNestingOkS } \text{jmps } c2)$
 $\text{jumpNestingOkS } \text{jmps } (l \bullet \text{While}(e) c) = \text{jumpNestingOkS } (\{ \text{Cont } l \} \cup \text{jmps}) c$
 — The label of the while loop only handles continue jumps. Breaks are only handled by *Lab*
 $\text{jumpNestingOkS } \text{jmps } (\text{Jmp } j) = (j \in \text{jmps})$
 $\text{jumpNestingOkS } \text{jmps } (\text{Throw } e) = \text{True}$
 $\text{jumpNestingOkS } \text{jmps } (\text{Try } c1 \text{ Catch}(C \text{ vn}) c2) = (\text{jumpNestingOkS } \text{jmps } c1 \wedge \text{jumpNestingOkS } \text{jmps } c2)$
 $\text{jumpNestingOkS } \text{jmps } (c1 \text{ Finally } c2) = (\text{jumpNestingOkS } \text{jmps } c1 \wedge \text{jumpNestingOkS } \text{jmps } c2)$
 $\text{jumpNestingOkS } \text{jmps } (\text{Init } C) = \text{True}$
 — wellformedness of the program must enshure that for all initializers jumpNestingOkS holds
 — Dummy analysis for intermediate smallstep term *FinA*
 $\text{jumpNestingOkS } \text{jmps } (\text{FinA } a \ c) = \text{False}$

definition $\text{jumpNestingOk} :: \text{jump set} \Rightarrow \text{term} \Rightarrow \text{bool}$ where

$\text{jumpNestingOk } \text{jmps } t = (\text{case } t \text{ of}$
 $\quad \text{In1 } se \Rightarrow (\text{case } se \text{ of}$
 $\quad \quad \text{Inl } e \Rightarrow \text{True}$
 $\quad \quad | \text{Inr } s \Rightarrow \text{jumpNestingOkS } \text{jmps } s)$
 $\quad | \text{In2 } v \Rightarrow \text{True}$
 $\quad | \text{In3 } es \Rightarrow \text{True})$

lemma $\text{jumpNestingOk-expr-simp } [\text{simp}]: \text{jumpNestingOk } \text{jmps } (\text{In1l } e) = \text{True}$
 $\langle \text{proof} \rangle$

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

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

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

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

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

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

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

Calculation of assigned variables for boolean expressions

2 Very restricted calculation fallback calculation

primrec *the-LVar-name* :: *var* \Rightarrow *lname*
where *the-LVar-name* (*LVar* *n*) = *n*

primrec *assignsE* :: *expr* \Rightarrow *lname set*
and *assignsV* :: *var* \Rightarrow *lname set*
and *assignsEs* :: *expr list* \Rightarrow *lname set*

where

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*

definition *assigns* :: *term* \Rightarrow *lname set* **where**

assigns *t* = (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*
<proof>

lemma *assigns-expr-simp1* [*simp*]: *assigns* (<*e*>) = *assignsE* *e*
<proof>

lemma *assigns-stmt-simp* [*simp*]: *assigns* (*In1r* *s*) = {}
<proof>

lemma *assigns-stmt-simp1* [*simp*]: *assigns* (<*s*::*stmt*>) = {}

$\langle \text{proof} \rangle$

lemma *assigns-var-simp* [simp]: *assigns* (*In2* *v*) = *assignsV* *v*
 $\langle \text{proof} \rangle$

lemma *assigns-var-simp1* [simp]: *assigns* ($\langle v \rangle$) = *assignsV* *v*
 $\langle \text{proof} \rangle$

lemma *assigns-expr-list-simp* [simp]: *assigns* (*In3* *es*) = *assignsEs* *es*
 $\langle \text{proof} \rangle$

lemma *assigns-expr-list-simp1* [simp]: *assigns* ($\langle es \rangle$) = *assignsEs* *es*
 $\langle \text{proof} \rangle$

3 Analysis of constant expressions

primrec *constVal* :: *expr* \Rightarrow *val option*

where

```

constVal (NewC c)      = None
| constVal (NewA t e)   = None
| constVal (Cast t e)   = None
| constVal (Inst e r)   = None
| constVal (Lit val)    = Some val
| constVal (UnOp unop e) = (case (constVal e) of
    None  $\Rightarrow$  None
  | Some v  $\Rightarrow$  Some (eval-unop unop v))
| constVal (BinOp binop e1 e2) = (case (constVal e1) of
    None  $\Rightarrow$  None
  | Some v1  $\Rightarrow$  (case (constVal e2) of
    None  $\Rightarrow$  None
  | Some v2  $\Rightarrow$  Some (eval-binop binop v1 v2)))
| constVal (Super)      = None
| constVal (Acc v)       = None
| constVal (Ass v e)     = None
| constVal (Cond b e1 e2) = (case (constVal b) of
    None  $\Rightarrow$  None
  | Some bv  $\Rightarrow$  (case the-Bool bv of
    True  $\Rightarrow$  (case (constVal e2) of
    None  $\Rightarrow$  None
  | Some v  $\Rightarrow$  constVal e1)
  | False  $\Rightarrow$  (case (constVal e1) of
    None  $\Rightarrow$  None
  | Some v  $\Rightarrow$  constVal e2)))

```

— Note that *constVal* (*Cond* *b* *e1* *e2*) is stricter as it could be. It requires that all tree expressions are constant even if we can decide which branch to choose, provided the constant value of *b*

```

constVal (Call accC statT mode objRef mn pTs args) = None
| constVal (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: $\bigwedge v. P (\text{Lit } v)$ **and**

hyp-UnOp: $\bigwedge \text{unop } e'. P e' \Longrightarrow P (\text{UnOp unop } e')$ **and**

hyp-BinOp: $\bigwedge \text{binop } e1 e2. \llbracket P e1; P e2 \rrbracket \Longrightarrow P (\text{BinOp binop } e1 e2)$ **and**

hyp-CondL: $\bigwedge b bv e1 e2. \llbracket \text{constVal } b = \text{Some } bv; \text{the-Bool } bv; P b; P e1 \rrbracket$
 $\Longrightarrow P (b? e1 : e2)$ **and**

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

shows $P \ e$

<proof>

lemma *assignsE-const-simp*: $\text{constVal } e = \text{Some } v \implies \text{assignsE } e = \{\}$

<proof>

4 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.

primrec *assigns-if* :: $\text{bool} \Rightarrow \text{expr} \Rightarrow \text{lname set}$

where

<i>assigns-if</i> b (<i>NewC</i> c)	= UNIV — can never evaluate to Boolean
<i>assigns-if</i> b (<i>NewA</i> $t \ e$)	= UNIV — can never evaluate to Boolean
<i>assigns-if</i> b (<i>Cast</i> $t \ e$)	= <i>assigns-if</i> $b \ e$
<i>assigns-if</i> b (<i>Inst</i> $e \ r$)	= <i>assignsE</i> e — Inst has type Boolean but e is a reference type
<i>assigns-if</i> b (<i>Lit</i> val)	= (if $val = \text{Bool } b$ then $\{\}$ else UNIV)
<i>assigns-if</i> b (<i>UnOp</i> $unop \ e$)	= (case <i>constVal</i> (<i>UnOp</i> $unop \ e$) of
	<i>None</i> \Rightarrow (if $unop = \text{UNot}$
	then <i>assigns-if</i> ($\neg b$) e
	else UNIV)
	<i>Some</i> $v \Rightarrow$ (if $v = \text{Bool } b$
	then $\{\}$
	else UNIV))
<i>assigns-if</i> b (<i>BinOp</i> $binop \ e1 \ e2$)	
	= (case <i>constVal</i> (<i>BinOp</i> $binop \ e1 \ e2$) of
	<i>None</i> \Rightarrow (if $binop = \text{CondAnd}$ then
	(case b of
	<i>True</i> \Rightarrow <i>assigns-if</i> <i>True</i> $e1 \cup$ <i>assigns-if</i> <i>True</i> $e2$
	<i>False</i> \Rightarrow <i>assigns-if</i> <i>False</i> $e1 \cap$
	(<i>assigns-if</i> <i>True</i> $e1 \cup$ <i>assigns-if</i> <i>False</i> $e2$))
	else
	(if $binop = \text{CondOr}$ then
	(case b of
	<i>True</i> \Rightarrow <i>assigns-if</i> <i>True</i> $e1 \cap$
	(<i>assigns-if</i> <i>False</i> $e1 \cup$ <i>assigns-if</i> <i>True</i> $e2$)
	<i>False</i> \Rightarrow <i>assigns-if</i> <i>False</i> $e1 \cup$ <i>assigns-if</i> <i>False</i> $e2$)
	else <i>assignsE</i> $e1 \cup$ <i>assignsE</i> $e2$))
	<i>Some</i> $v \Rightarrow$ (if $v = \text{Bool } b$ then $\{\}$ else UNIV))
<i>assigns-if</i> b (<i>Super</i>)	= UNIV — can never evaluate to Boolean
<i>assigns-if</i> b (<i>Acc</i> v)	= (<i>assignsV</i> v)
<i>assigns-if</i> b ($v := e$)	= (<i>assignsE</i> (<i>Ass</i> $v \ e$))
<i>assigns-if</i> b ($c? \ e1 : e2$)	= (<i>assignsE</i> c) \cup
	(case (<i>constVal</i> c) of
	<i>None</i> \Rightarrow (<i>assigns-if</i> $b \ e1$) \cap
	(<i>assigns-if</i> $b \ e2$)
	<i>Some</i> $bv \Rightarrow$ (case <i>the-Bool</i> bv of
	<i>True</i> \Rightarrow <i>assigns-if</i> $b \ e1$
	<i>False</i> \Rightarrow <i>assigns-if</i> $b \ e2$))
<i>assigns-if</i> b ($\{\text{accC}, \text{statT}, \text{mode}\} \text{objRef} \cdot \text{mn}(\{pTs\} \text{args})$)	
	= <i>assignsE</i> ($\{\text{accC}, \text{statT}, \text{mode}\} \text{objRef} \cdot \text{mn}(\{pTs\} \text{args})$)

— Only dummy analysis for intermediate expressions *Method*, *Body*, *InsInitE* and *Callee*

<i>assigns-if</i> b (<i>Method</i> $C \ sig$)	= $\{\}$
<i>assigns-if</i> b (<i>Body</i> $C \ s$)	= $\{\}$

| *assigns-if* b (*InsInitE* s e) = $\{\}$
 | *assigns-if* b (*Callee* l e) = $\{\}$

lemma *assigns-if-const-b-simp*:
assumes *boolConst*: *constVal* e = *Some* (*Bool* b) (**is** ?*Const* b e)
shows *assigns-if* b e = $\{\}$ (**is** ?*Ass* b e)
 <proof>

lemma *assigns-if-const-not-b-simp*:
assumes *boolConst*: *constVal* e = *Some* (*Bool* b) (**is** ?*Const* b e)
shows *assigns-if* ($\neg b$) e = *UNIV* (**is** ?*Ass* b e)
 <proof>

5 Lifting set operations to range of tables (map to a set)

definition

union-ts :: ($'a, 'b$) *tables* \Rightarrow ($'a, 'b$) *tables* \Rightarrow ($'a, 'b$) *tables* ($\langle - \Rightarrow \cup - \rangle$ [67,67] 65)
where $A \Rightarrow \cup B = (\lambda k. A\ k \cup B\ k)$

definition

intersect-ts :: ($'a, 'b$) *tables* \Rightarrow ($'a, 'b$) *tables* \Rightarrow ($'a, 'b$) *tables* ($\langle - \Rightarrow \cap - \rangle$ [72,72] 71)
where $A \Rightarrow \cap B = (\lambda k. A\ k \cap B\ k)$

definition

all-union-ts :: ($'a, 'b$) *tables* \Rightarrow $'b$ *set* \Rightarrow ($'a, 'b$) *tables* (**infixl** $\langle \Rightarrow \cup_{\forall} \rangle$ 40)
where $(A \Rightarrow \cup_{\forall} B) = (\lambda k. A\ k \cup B)$

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!]:

$\llbracket c \in (A \Rightarrow \cap B) \ k; \llbracket c \in A \ k; c \in B \ k \rrbracket \Longrightarrow P \rrbracket \Longrightarrow P$
 $\langle proof \rangle$

All-Union of tables and set

lemma *all-union-ts-iff* [simp]: $(c \in (A \Rightarrow \cup_{\forall} B) \ k) = (c \in A \ k \vee c \in B)$

$\langle proof \rangle$

lemma *all-union-tsI1* [elim?]: $c \in A \ k \Longrightarrow c \in (A \Rightarrow \cup_{\forall} B) \ k$

$\langle proof \rangle$

lemma *all-union-tsI2* [elim?]: $c \in B \Longrightarrow c \in (A \Rightarrow \cup_{\forall} B) \ k$

$\langle proof \rangle$

lemma *all-union-tsCI* [intro!]: $(c \notin B \Longrightarrow c \in A \ k) \Longrightarrow c \in (A \Rightarrow \cup_{\forall} B) \ k$

$\langle proof \rangle$

lemma *all-union-tsE* [elim!]:

$\llbracket c \in (A \Rightarrow \cup_{\forall} B) \ k; (c \in A \ k \Longrightarrow P); (c \in B \Longrightarrow P) \rrbracket \Longrightarrow P$
 $\langle proof \rangle$

The rules of definite assignment

type-synonym *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

definition

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

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

definition

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

where $\Rightarrow \cap A = \{x \mid x. \forall k. x \in A \ k\}$

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 ($\langle \vdash - \gg - \rangle \rightarrow$ [65,65,65,65] 71)

where

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

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

\Longrightarrow

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

| *Lab*: $\llbracket Env \vdash B \gg \langle c \rangle \gg C; \text{nrm } A = \text{nrm } C \cap (\text{brk } C) \ l; \text{brk } A = \text{rmlab } l (\text{brk } C) \rrbracket$

\Longrightarrow

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

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

$$\begin{aligned}
& nrm\ A = nrm\ C2; \text{brk}\ A = (\text{brk}\ C1) \Rightarrow \cap (\text{brk}\ C2) \\
& \implies \\
& Env \vdash B \gg \langle c1;; c2 \rangle \gg A
\end{aligned}$$

$$\begin{aligned}
| \text{ If: } & \llbracket Env \vdash B \gg \langle e \rangle \gg E; \\
& Env \vdash (B \cup \text{assigns-if True } e) \gg \langle c1 \rangle \gg C1; \\
& Env \vdash (B \cup \text{assigns-if False } e) \gg \langle c2 \rangle \gg C2; \\
& nrm\ A = nrm\ C1 \cap nrm\ C2; \\
& \text{brk}\ A = \text{brk}\ C1 \Rightarrow \cap \text{brk}\ C2 \rrbracket \\
& \implies \\
& Env \vdash B \gg \langle \text{If}(e)\ c1\ \text{Else}\ c2 \rangle \gg A
\end{aligned}$$

— 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 $nrm\ C2 = UNIV$. So finally $nrm\ A = nrm\ C1$. For the break maps this trick workd too, because the trivial break map will map all labels to *UNIV*. In the example, if no break occurs in $c2$ the break maps will trivially map to *UNIV* and if a break occurs it will map to *UNIV* too, because *assigns-if False* $e = UNIV$. So in the intersection of the break maps the path $c2$ will have no contribution.

$$\begin{aligned}
| \text{ Loop: } & \llbracket Env \vdash B \gg \langle e \rangle \gg E; \\
& Env \vdash (B \cup \text{assigns-if True } e) \gg \langle c \rangle \gg C; \\
& nrm\ A = nrm\ C \cap (B \cup \text{assigns-if False } e); \\
& \text{brk}\ A = \text{brk}\ C \rrbracket \\
& \implies \\
& Env \vdash B \gg \langle l \cdot \text{While}(e)\ c \rangle \gg A
\end{aligned}$$

— The *Loop* rule resembles some of the ideas of the *If* rule. For the $nrm\ A$ 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 ($nrm\ C$) or with a break. But in this model, the label l of the loop only handles continue labels, not break labels. The break label will be handled by an enclosing *Lab* statement. So we don't have to handle the breaks specially.

$$\begin{aligned}
| \text{ Jmp: } & \llbracket \text{jump} = \text{Ret} \longrightarrow \text{Result} \in B; \\
& nrm\ A = UNIV; \\
& \text{brk}\ A = (\text{case jump of} \\
& \quad \text{Break } l \Rightarrow \lambda k. \text{ if } k=l \text{ then } B \text{ else } UNIV \\
& \quad | \text{Cont } l \Rightarrow \lambda k. UNIV \\
& \quad | \text{Ret} \Rightarrow \lambda k. UNIV) \rrbracket \\
& \implies \\
& Env \vdash B \gg \langle \text{Jmp jump} \rangle \gg A
\end{aligned}$$

— 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 *Jmp* 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 enshure that the result value is assigned.

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

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

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

$$\implies$$

$$Env \vdash B \gg \langle c1 \text{ Finally } c2 \rangle \gg A$$

— 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 $nrm\ C2$ ($\Rightarrow \cup$). 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 a expression. So for all expressions the break sets could be set to the trivial one: $\lambda l. UNIV$. But we can't trivially proof, 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 proof 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*)

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

— 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.

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

$$| \text{NewA: } Env \vdash B \gg \langle e \rangle \gg A$$

$$\implies$$

$$Env \vdash B \gg \langle \text{New } T[e] \rangle \gg A$$

$$| \text{Cast: } Env \vdash B \gg \langle e \rangle \gg A$$

$$\implies$$

$$Env \vdash B \gg \langle \text{Cast } T\ e \rangle \gg A$$

$$| \text{Inst: } Env \vdash B \gg \langle e \rangle \gg A$$

$$\implies$$

$$Env \vdash B \gg \langle e\ \text{InstOf } T \rangle \gg A$$

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

$$| \text{UnOp: } Env \vdash B \gg \langle e \rangle \gg A$$

$$\implies$$

$$Env \vdash B \gg \langle \text{UnOp } \text{unop } e \rangle \gg A$$

$$| \text{CondAnd: } \llbracket Env \vdash B \gg \langle e1 \rangle \gg E1; Env \vdash (B \cup \text{assigns-if True } e1) \gg \langle e2 \rangle \gg E2;$$

$$\text{nrm } A = B \cup (\text{assigns-if True } (\text{BinOp CondAnd } e1\ e2) \cap$$

$$\text{assigns-if False } (\text{BinOp CondAnd } e1\ e2));$$

$$\text{brk } A = (\lambda l. UNIV) \rrbracket$$

$$\implies$$

$$Env \vdash B \gg \langle \text{BinOp CondAnd } e1\ e2 \rangle \gg A$$

$$| \text{CondOr: } \llbracket Env \vdash B \gg \langle e1 \rangle \gg E1; Env \vdash (B \cup \text{assigns-if False } e1) \gg \langle e2 \rangle \gg E2;$$

$$\text{nrm } A = B \cup (\text{assigns-if True } (\text{BinOp CondOr } e1\ e2) \cap$$

$$\text{assigns-if False } (\text{BinOp CondOr } e1\ e2));$$

- $$\begin{array}{l}
\text{brk } A = (\lambda l. \text{ UNIV}) \text{]} \\
\implies \\
\text{Env} \vdash B \gg \langle \text{BinOp CondOr } e1 \ e2 \rangle \gg A \\
\\
| \text{ BinOp: } \llbracket \text{Env} \vdash B \gg \langle e1 \rangle \gg E1; \text{Env} \vdash \text{nrm } E1 \gg \langle e2 \rangle \gg A; \\
\text{binop} \neq \text{CondAnd}; \text{binop} \neq \text{CondOr} \rrbracket \\
\implies \\
\text{Env} \vdash B \gg \langle \text{BinOp binop } e1 \ e2 \rangle \gg A \\
\\
| \text{ Super: } \text{This} \in B \\
\implies \\
\text{Env} \vdash B \gg \langle \text{Super} \rangle \gg (\text{nrm} = B, \text{brk} = \lambda l. \text{ UNIV}) \\
\\
| \text{ AccLVar: } \llbracket vn \in B; \\
\text{nrm } A = B; \text{brk } A = (\lambda k. \text{ UNIV}) \rrbracket \\
\implies \\
\text{Env} \vdash B \gg \langle \text{Acc (LVar } vn) \rangle \gg A \\
\\
— \text{ To properly access a local variable we have to test the definite assignment here. The variable must occur} \\
\text{in the set } B \\
\\
| \text{ Acc: } \llbracket \forall vn. v \neq \text{LVar } vn; \\
\text{Env} \vdash B \gg \langle v \rangle \gg A \rrbracket \\
\implies \\
\text{Env} \vdash B \gg \langle \text{Acc } v \rangle \gg A \\
\\
| \text{ AssLVar: } \llbracket \text{Env} \vdash B \gg \langle e \rangle \gg E; \text{nrm } A = \text{nrm } E \cup \{vn\}; \text{brk } A = \text{brk } E \rrbracket \\
\implies \\
\text{Env} \vdash B \gg \langle (\text{LVar } vn) := e \rangle \gg A \\
\\
| \text{ Ass: } \llbracket \forall vn. v \neq \text{LVar } vn; \text{Env} \vdash B \gg \langle v \rangle \gg V; \text{Env} \vdash \text{nrm } V \gg \langle e \rangle \gg A \rrbracket \\
\implies \\
\text{Env} \vdash B \gg \langle v := e \rangle \gg A \\
\\
| \text{ CondBool: } \llbracket \text{Env} \vdash (c ? 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 = B \cup (\text{assigns-if True } (c ? e1 : e2) \cap \\
\text{assigns-if False } (c ? e1 : e2)); \\
\text{brk } A = (\lambda l. \text{ UNIV}) \rrbracket \\
\implies \\
\text{Env} \vdash B \gg \langle c ? e1 : e2 \rangle \gg A \\
\\
| \text{ Cond: } \llbracket \neg \text{Env} \vdash (c ? 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 ? e1 : e2 \rangle \gg A \\
\\
| \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 \{accC, statT, mode\} e \cdot mn(\{pTs\} args) \rangle \gg A
\end{array}$$

— 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 *Methd* and then further to *Body* during evaluation to establish the definite assignment of *Methd* during evaluation of *Call*, and of *Body* during evaluation of *Methd*. This is necessary since definite assignment will be a precondition for each induction hypothesis coming out of the evaluation rules, and therefor we have to establish the definite assignment of the sub-evaluation during the type-safety proof. Note that well-typedness is also a precondition for type-safety and so we can omit some assertion that are already ensured by well-typedness.

| *Methd*: $\llbracket \text{methd } (prg \text{ Env}) \ D \ \text{sig} = \text{Some } m; \\ \text{Env} \vdash B \gg \langle \text{Body } (\text{declclass } m) \ (\text{stmt } (\text{mbody } (\text{methd } m))) \rangle \gg A \\ \rrbracket \\ \implies \\ \text{Env} \vdash B \gg \langle \text{Methd } D \ \text{sig} \rangle \gg A$

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

— 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.

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

| *FVar*: $\text{Env} \vdash B \gg \langle e \rangle \gg A \\ \implies \\ \text{Env} \vdash B \gg \langle \{\text{accC}, \text{statDeclC}, \text{stat}\} e..fn \rangle \gg A$

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

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

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

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*]:

$\text{Env} \vdash B \gg \langle \text{Skip} \rangle \gg A$
 $\text{Env} \vdash B \gg \text{In1r } \text{Skip} \gg A$
 $\text{Env} \vdash B \gg \langle \text{Expr } e \rangle \gg A$
 $\text{Env} \vdash B \gg \text{In1r } (\text{Expr } e) \gg A$
 $\text{Env} \vdash B \gg \langle l \cdot c \rangle \gg A$
 $\text{Env} \vdash B \gg \text{In1r } (l \cdot c) \gg A$
 $\text{Env} \vdash B \gg \langle c1;; c2 \rangle \gg A$
 $\text{Env} \vdash B \gg \text{In1r } (c1;; c2) \gg A$
 $\text{Env} \vdash B \gg \langle \text{If}(e) \ c1 \ \text{Else } c2 \rangle \gg A$
 $\text{Env} \vdash B \gg \text{In1r } (\text{If}(e) \ c1 \ \text{Else } c2) \gg A$
 $\text{Env} \vdash B \gg \langle l \cdot \text{While}(e) \ c \rangle \gg A$
 $\text{Env} \vdash B \gg \text{In1r } (l \cdot \text{While}(e) \ c) \gg A$

$Env \vdash B \gg \langle \text{Jump } jump \rangle \gg A$
 $Env \vdash B \gg \text{In1r } (\text{Jump } jump) \gg A$
 $Env \vdash B \gg \langle \text{Throw } e \rangle \gg A$
 $Env \vdash B \gg \text{In1r } (\text{Throw } e) \gg A$
 $Env \vdash B \gg \langle \text{Try } c1 \text{ Catch } (C \text{ } vn) \text{ } c2 \rangle \gg A$
 $Env \vdash B \gg \text{In1r } (\text{Try } c1 \text{ Catch } (C \text{ } vn) \text{ } c2) \gg A$
 $Env \vdash B \gg \langle c1 \text{ Finally } c2 \rangle \gg A$
 $Env \vdash B \gg \text{In1r } (c1 \text{ Finally } c2) \gg A$
 $Env \vdash B \gg \langle \text{Init } C \rangle \gg A$
 $Env \vdash B \gg \text{In1r } (\text{Init } C) \gg A$
 $Env \vdash B \gg \langle \text{NewC } C \rangle \gg A$
 $Env \vdash B \gg \text{In1l } (\text{NewC } C) \gg A$
 $Env \vdash B \gg \langle \text{New } T[e] \rangle \gg A$
 $Env \vdash B \gg \text{In1l } (\text{New } T[e]) \gg A$
 $Env \vdash B \gg \langle \text{Cast } T \text{ } e \rangle \gg A$
 $Env \vdash B \gg \text{In1l } (\text{Cast } T \text{ } e) \gg A$
 $Env \vdash B \gg \langle e \text{ InstOf } T \rangle \gg A$
 $Env \vdash B \gg \text{In1l } (e \text{ InstOf } T) \gg A$
 $Env \vdash B \gg \langle \text{Lit } v \rangle \gg A$
 $Env \vdash B \gg \text{In1l } (\text{Lit } v) \gg A$
 $Env \vdash B \gg \langle \text{UnOp } unop \text{ } e \rangle \gg A$
 $Env \vdash B \gg \text{In1l } (\text{UnOp } unop \text{ } e) \gg A$
 $Env \vdash B \gg \langle \text{BinOp } binop \text{ } e1 \text{ } e2 \rangle \gg A$
 $Env \vdash B \gg \text{In1l } (\text{BinOp } binop \text{ } e1 \text{ } e2) \gg A$
 $Env \vdash B \gg \langle \text{Super} \rangle \gg A$
 $Env \vdash B \gg \text{In1l } (\text{Super}) \gg A$
 $Env \vdash B \gg \langle \text{Acc } v \rangle \gg A$
 $Env \vdash B \gg \text{In1l } (\text{Acc } v) \gg A$
 $Env \vdash B \gg \langle v := e \rangle \gg A$
 $Env \vdash B \gg \text{In1l } (v := e) \gg A$
 $Env \vdash B \gg \langle c \text{ ? } e1 : e2 \rangle \gg A$
 $Env \vdash B \gg \text{In1l } (c \text{ ? } e1 : e2) \gg A$
 $Env \vdash B \gg \langle \{accC, statT, mode\} e.mn(\{pTs\} args) \rangle \gg A$
 $Env \vdash B \gg \text{In1l } (\{accC, statT, mode\} e.mn(\{pTs\} args)) \gg A$
 $Env \vdash B \gg \langle \text{Methd } C \text{ } sig \rangle \gg A$
 $Env \vdash B \gg \text{In1l } (\text{Methd } C \text{ } sig) \gg A$
 $Env \vdash B \gg \langle \text{Body } D \text{ } c \rangle \gg A$
 $Env \vdash B \gg \text{In1l } (\text{Body } D \text{ } c) \gg A$
 $Env \vdash B \gg \langle \text{LVar } vn \rangle \gg A$
 $Env \vdash B \gg \text{In2 } (\text{LVar } vn) \gg A$
 $Env \vdash B \gg \langle \{accC, statDeclC, stat\} e..fn \rangle \gg A$
 $Env \vdash B \gg \text{In2 } (\{accC, statDeclC, stat\} e..fn) \gg A$
 $Env \vdash B \gg \langle e1.[e2] \rangle \gg A$
 $Env \vdash B \gg \text{In2 } (e1.[e2]) \gg A$
 $Env \vdash B \gg \langle [] :: \text{expr list} \rangle \gg A$
 $Env \vdash B \gg \text{In3 } ([] :: \text{expr list}) \gg A$
 $Env \vdash B \gg \langle e \# es \rangle \gg A$
 $Env \vdash B \gg \text{In3 } (e \# es) \gg A$

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

lemma *da-NewC*: $A = (\text{norm} = B, \text{brk} = \lambda l. \text{UNIV}) \implies Env \vdash B \gg \langle \text{NewC } C \rangle \gg A$

$\langle proof \rangle$

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*: $E \vdash e :: \neg PrimT\ Boolean$ (**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*]: $\forall k. A\ k \subseteq B\ k \implies \Rightarrow \bigcap A \subseteq \Rightarrow \bigcap B$
 $\langle proof \rangle$

lemma *range-inter-ts-subseteq'*: $\forall k. A\ k \subseteq B\ k \implies x \in \Rightarrow \bigcap A \implies x \in \Rightarrow \bigcap B$
 $\langle proof \rangle$

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 $(nrm\ A \subseteq nrm\ A') \wedge (\forall l. (brk\ A\ l \subseteq brk\ A'\ l))$
 $\langle proof \rangle$

lemma *da-weaken*:

assumes *da*: $Env \vdash B \gg t \gg A$ **and** $B \subseteq B'$
shows $\exists A'. Env \vdash B' \gg t \gg A'$
 $\langle proof \rangle$

corollary *da-weakenE* [*consumes 2*]:

assumes da : $Env \vdash B \gg t \gg A$ **and**
 $B': B \subseteq B'$ **and**
 $ex\text{-}mono$: $\bigwedge A'. \llbracket Env \vdash B' \gg t \gg A'; nrm\ A \subseteq nrm\ A';$
 $\bigwedge l. brk\ A\ l \subseteq brk\ A'\ l \rrbracket \implies P$
shows P
 $\langle proof \rangle$

end

Chapter 13

WellForm

1 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)

definition

$wf_fdecl :: prog \Rightarrow pname \Rightarrow fdecl \Rightarrow bool$
where $wf_fdecl\ G\ P = (\lambda(fn,f). is_acc_type\ G\ P\ (type\ f))$

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

definition

$wf_mhead :: prog \Rightarrow pname \Rightarrow sig \Rightarrow mhead \Rightarrow bool$ **where**
 $wf_mhead\ G\ P = (\lambda\ sig\ mh. length\ (parTs\ sig) = length\ (pars\ mh) \wedge$
 $(\forall\ T \in set\ (parTs\ sig). is_acc_type\ G\ P\ T) \wedge$
 $is_acc_type\ G\ P\ (resTy\ mh) \wedge$
 $distinct\ (pars\ mh))$

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.

definition

$callee_lcl :: qname \Rightarrow sig \Rightarrow methd \Rightarrow lenv$ **where**
 $callee_lcl\ C\ sig\ m =$
 $(\lambda k. (case\ k\ of$
 $\quad EName\ e$
 $\quad \Rightarrow (case\ e\ of$
 $\quad \quad VName\ v$
 $\quad \quad \Rightarrow ((table_of\ (lcls\ (mbody\ m)))(pars\ m\ [\mapsto]\ parTs\ sig))\ v$
 $\quad \quad | Res \Rightarrow Some\ (resTy\ m))$
 $\quad | This \Rightarrow if\ is_static\ m\ then\ None\ else\ Some\ (Class\ C)))$

definition

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

definition

$wf_mdecl :: prog \Rightarrow qname \Rightarrow mdecl \Rightarrow bool$ **where**
 $wf_mdecl\ G\ C =$
 $(\lambda(sig,m).$
 $\quad wf_mhead\ G\ (pid\ C)\ sig\ (mhead\ m) \wedge$
 $\quad unique\ (lcls\ (mbody\ m)) \wedge$
 $\quad (\forall\ (vn,T) \in set\ (lcls\ (mbody\ m)). is_acc_type\ G\ (pid\ C)\ T) \wedge$
 $\quad (\forall\ pn \in set\ (pars\ m). table_of\ (lcls\ (mbody\ m))\ pn = None) \wedge$
 $\quad jumpNestingOkS\ \{Ret\}\ (stmt\ (mbody\ m)) \wedge$
 $\quad is_class\ G\ C \wedge$
 $\quad (\llbracket prg=G,cls=C,lcl=callee_lcl\ C\ sig\ m \rrbracket \vdash (stmt\ (mbody\ m)) :: \surd \wedge$
 $\quad (\exists\ A. (\llbracket prg=G,cls=C,lcl=callee_lcl\ C\ sig\ m \rrbracket$
 $\quad \quad \vdash parameters\ m \gg stmt\ (mbody\ m) \gg A$
 $\quad \quad \wedge Result \in nrm\ A)))$

lemma $callee_lcl_VName_simp\ [simp]:$

$callee_lcl\ C\ sig\ m\ (EName\ (VName\ v))$
 $= ((table_of\ (lcls\ (mbody\ m)))(pars\ m\ [\mapsto]\ parTs\ sig))\ v$
 $\langle proof \rangle$

lemma $callee_lcl_Res_simp\ [simp]:$

callee-lcl C *sig* m (*EName* Res) = *Some* (*resTy* m)
 ⟨proof⟩

lemma *callee-lcl-This-simp* [*simp*]:
callee-lcl C *sig* m (*This*) = (if *is-static* m then *None* else *Some* (*Class* C))
 ⟨proof⟩

lemma *callee-lcl-This-static-simp*:
is-static $m \implies$ *callee-lcl* C *sig* m (*This*) = *None*
 ⟨proof⟩

lemma *callee-lcl-This-not-static-simp*:
 \neg *is-static* $m \implies$ *callee-lcl* C *sig* m (*This*) = *Some* (*Class* C)
 ⟨proof⟩

lemma *wf-mheadI*:
 $\llbracket \text{length } (parTs \text{ sig}) = \text{length } (pars \ m); \forall T \in \text{set } (parTs \text{ sig}). \text{is-acc-type } G \ P \ T;$
 $\text{is-acc-type } G \ P \ (\text{resTy } m); \text{distinct } (pars \ m) \rrbracket \implies$
 $\text{wf-mhead } G \ P \ \text{sig } m$
 ⟨proof⟩

lemma *wf-mdeclI*: \llbracket
 $\text{wf-mhead } G \ (\text{pid } C) \ \text{sig } (\text{mhead } m); \text{unique } (\text{lcls } (\text{mbody } m));$
 $(\forall pn \in \text{set } (pars \ m). \text{table-of } (\text{lcls } (\text{mbody } m)) \ pn = \text{None});$
 $\forall (vn, T) \in \text{set } (\text{lcls } (\text{mbody } m)). \text{is-acc-type } G \ (\text{pid } C) \ T;$
 $\text{jumpNestingOkS } \{Ret\} \ (\text{stmt } (\text{mbody } m));$
 $\text{is-class } G \ C;$
 $(\llbracket \text{prg} = G, \text{cls} = C, \text{lcl} = \text{callee-lcl } C \ \text{sig } m \rrbracket \vdash (\text{stmt } (\text{mbody } m)) :: \checkmark;$
 $(\exists A. (\llbracket \text{prg} = G, \text{cls} = C, \text{lcl} = \text{callee-lcl } C \ \text{sig } m \rrbracket \vdash \text{parameters } m \gg (\text{stmt } (\text{mbody } m)) \gg A$
 $\wedge \text{Result} \in \text{nrm } A)$
 $\rrbracket \implies$
 $\text{wf-mdecl } G \ C \ (\text{sig}, m)$
 ⟨proof⟩

lemma *wf-mdeclE* [*consumes 1*]:
 $\llbracket \text{wf-mdecl } G \ C \ (\text{sig}, m);$
 $\llbracket \text{wf-mhead } G \ (\text{pid } C) \ \text{sig } (\text{mhead } m); \text{unique } (\text{lcls } (\text{mbody } m));$
 $\forall pn \in \text{set } (pars \ m). \text{table-of } (\text{lcls } (\text{mbody } m)) \ pn = \text{None};$
 $\forall (vn, T) \in \text{set } (\text{lcls } (\text{mbody } m)). \text{is-acc-type } G \ (\text{pid } C) \ T;$
 $\text{jumpNestingOkS } \{Ret\} \ (\text{stmt } (\text{mbody } m));$
 $\text{is-class } G \ C;$
 $(\llbracket \text{prg} = G, \text{cls} = C, \text{lcl} = \text{callee-lcl } C \ \text{sig } m \rrbracket \vdash (\text{stmt } (\text{mbody } m)) :: \checkmark;$
 $(\exists A. (\llbracket \text{prg} = G, \text{cls} = C, \text{lcl} = \text{callee-lcl } C \ \text{sig } m \rrbracket \vdash \text{parameters } m \gg (\text{stmt } (\text{mbody } m)) \gg A$
 $\wedge \text{Result} \in \text{nrm } A)$
 $\rrbracket \implies P$
 $\rrbracket \implies P$
 ⟨proof⟩

lemma *wf-mdeclD1*:
 $\text{wf-mdecl } G \ C \ (\text{sig}, m) \implies$
 $\text{wf-mhead } G \ (\text{pid } C) \ \text{sig } (\text{mhead } m) \wedge \text{unique } (\text{lcls } (\text{mbody } m)) \wedge$
 $(\forall pn \in \text{set } (pars \ m). \text{table-of } (\text{lcls } (\text{mbody } m)) \ pn = \text{None}) \wedge$
 $(\forall (vn, T) \in \text{set } (\text{lcls } (\text{mbody } m)). \text{is-acc-type } G \ (\text{pid } C) \ T)$
 ⟨proof⟩

lemma *wf-mdecl-bodyD*:
 $\text{wf-mdecl } G \ C \ (\text{sig}, m) \implies$
 $(\exists T. (\llbracket \text{prg} = G, \text{cls} = C, \text{lcl} = \text{callee-lcl } C \ \text{sig } m \rrbracket \vdash \text{Body } C \ (\text{stmt } (\text{mbody } m)) :: - T \wedge$

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

lemma *rT-is-acc-type*:

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

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

definition

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

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

lemma *wf-idecl-hidings*:

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

$\langle proof \rangle$

lemma *wf-idecl-hiding*:

$wf-idecl\ G\ (I, i) \implies$
 $(table-of\ (imethods\ i)$
 $\quad hiding\ (methd\ G\ Object)$
 $\quad under\ (\lambda\ new\ old.\ accmodi\ old \neq Private)$
 $\quad entails\ (\lambda new\ old.\ G \vdash_{resTy} new \preceq_{resTy} old \wedge$
 $\quad \quad is-static\ new = is-static\ old))$

$\langle proof \rangle$

lemma *wf-idecl-supD*:

$\llbracket wf-idecl\ G\ (I, i); J \in set\ (isuperIfs\ i) \rrbracket$
 $\implies is-acc-iface\ G\ (pid\ I)\ J \wedge (J, I) \notin (subint1\ G)^+$
 $\langle proof \rangle$

well-formed class declarations

A class declaration is wellformed if:

- there is no interface with the same name
- all superinterfaces are accessible and for all methods implementing an interface method the result type widens to the result type of the interface method, the method is not static and offers at least as much access (this actually means that the method has Public access, since all interface methods have public access)
- all field declarations are wellformed and the field names are unique
- all method declarations are wellformed and the method names are unique
- the initialization statement is welltyped
- the classhierarchy is wellstructured
- Unless the class is Object:
 - the superclass is accessible
 - for all methods overriding another method (of a superclass) the result type widens to the result type of the overridden method, the access modifier of the new method provides at least as much access as the overwritten one.
 - for all methods hiding a method (of a superclass) the hidden method must be static and offer at least as much access rights. Remark: In contrast to the Java Language Specification we don't restrict the result types of the method (as in case of overriding), because there seems to be no reason, since there is no dynamic binding of static methods. (cf. 8.4.6.3 vs. 15.12.1). Stricly speaking the restrictions on the access rights aren't necessary to, since the static type and the access rights together determine which method is to be called statically. But if a class gains more then one static method with the same signature due to inheritance, it is confusing when the method selection depends on the access rights only: e.g. Class C declares static public method foo(). Class D is subclass of C and declares static method foo() with default package access. D.foo() ? if this call is in the same package as D then foo of class D is called, otherwise foo of class C.

definition

$entails :: ('a, 'b)\ table \Rightarrow ('b \Rightarrow bool) \Rightarrow bool\ (\hookleftarrow entails \rightarrow 20)$
where $(t\ entails\ P) = (\forall k.\ \forall\ x \in t\ k:\ P\ x)$

lemma *entailsD*:

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

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

$\langle \text{proof} \rangle$

definition

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

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

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

$$\begin{array}{l} (G, \text{sigl} \vdash \text{new hides old} \\ \longrightarrow (\text{accmodi old} \leq \text{accmodi new} \wedge \\ \text{is-static old}))) \\ \text{)]} \implies P \\ \langle \text{proof} \rangle \end{array}$$

lemma *wf-cdecl-unique*:
 $wf\text{-}cdecl\ G\ (C, c) \implies unique\ (cfields\ c) \wedge unique\ (methods\ c)$
<proof>

lemma *wf-cdecl-fdecl*:

$$\llbracket wf\text{-}cdecl\ G\ (C,c); f \in set\ (cfields\ c) \rrbracket \implies wf\text{-}fdecl\ G\ (pid\ C)\ f$$
<proof>

lemma *wf-cdecl-mdecl*:

$$\llbracket wf\text{-}cdecl\ G\ (C,c); m \in set\ (methods\ c) \rrbracket \implies wf\text{-}mdecl\ G\ C\ m$$
<proof>

$$\begin{aligned} & \textbf{lemma } wf\text{-}cdecl\text{-}impD: \\ & \llbracket wf\text{-}cdecl\ G\ (C, c); I \in set\ (superIfs\ c) \rrbracket \\ & \implies is\text{-}acc\text{-}iface\ G\ (pid\ C)\ I \wedge \\ & \quad (\forall s. \forall im \in imethds\ G\ I\ s. \\ & \quad \quad (\exists cm \in methd\ G\ C\ s: \vdash_{resTy}\ cm \preceq_{resTy}\ im \wedge \neg is\text{-}static\ cm \wedge \\ & \quad \quad \quad accmodi\ im \leq accmodi\ cm)) \\ & \langle proof \rangle \end{aligned}$$
$$\begin{aligned}
& \textbf{lemma } wf\text{-}cdecl\text{-}supD: \\
& \llbracket wf\text{-}cdecl\ G\ (C,c); C \neq Object \rrbracket \implies \\
& \textit{is-acc-class } G\ (\textit{pid } C)\ (\textit{super } c) \wedge (\textit{super } c, C) \notin (\textit{subcls1 } G)^+ \wedge \\
& (\textit{table-of } (\textit{map } (\lambda\ (s,m). (s, C, m))\ (\textit{methods } c))) \\
& \textit{entails } (\lambda\ \textit{new}. \forall\ \textit{old}\ \textit{sig}. \\
& \quad (G, \textit{sig} \vdash \textit{new overrides}_S\ \textit{old} \\
& \quad \longrightarrow (G \vdash \textit{resTy } \textit{new} \preceq \textit{resTy } \textit{old} \wedge \\
& \quad \quad \textit{accmodi } \textit{old} \leq \textit{accmodi } \textit{new} \wedge \\
& \quad \quad \neg \textit{is-static } \textit{old})) \wedge \\
& \quad (G, \textit{sig} \vdash \textit{new hides } \textit{old} \\
& \quad \longrightarrow (\textit{accmodi } \textit{old} \leq \textit{accmodi } \textit{new} \wedge \\
& \quad \quad \textit{is-static } \textit{old}))) \\
& \langle \textit{proof} \rangle
\end{aligned}$$

lemma *wf-cdecl-overrides-SomeD*:

$$\begin{aligned} & \llbracket wf\text{-}cdecl\ G\ (C, c); C \neq Object; table\text{-}of\ (methods\ c)\ sig = Some\ newM; \\ & \quad G, sig \vdash (C, newM)\ overrides_S\ old \\ & \rrbracket \implies G \vdash resTy\ newM \preceq_{resTy}\ old \wedge \\ & \quad accmodi\ old \leq accmodi\ newM \wedge \\ & \quad \neg is\text{-}static\ old \\ & \langle proof \rangle \end{aligned}$$

lemma *wf-cdecl-hides-SomeD*:

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

lemma *wf-cdecl-wt-init*:
 $wf\text{-}cdecl\ G\ (C,\ c) \implies (\text{prg}=G, \text{cls}=C, \text{lcl}=\text{Map.empty}) \vdash \text{init}\ c :: \sqrt{}$

$\langle proof \rangle$

well-formed programs

A program declaration is wellformed if:

- the class `ObjectC` of `Object` is defined
- every method of `Object` has an access modifier distinct from `Package`. This is necessary since every interface automatically inherits from `Object`. We must know, that every time a `Object` method is "overridden" by an interface method this is also overridden by the class implementing the the interface (see *implement-dynmethd* and *class-mheadsD*)
- all standard Exceptions are defined
- all defined interfaces are wellformed
- all defined classes are wellformed

definition

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

lemma $wf\text{-}prog\text{-}idecl: \llbracket iface\ G\ I = Some\ i; wf\text{-}prog\ G \rrbracket \Longrightarrow wf\text{-}idecl\ G\ (I,i)$
 $\langle proof \rangle$

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

lemma $wf\text{-}prog\text{-}Object\text{-}mdecls:$
 $wf\text{-}prog\ G \Longrightarrow (\forall\ m \in set\ Object\text{-}mdecls. accmodi\ m \neq Package)$
 $\langle proof \rangle$

lemma $wf\text{-}prog\text{-}acc\text{-}superD:$
 $\llbracket wf\text{-}prog\ G; class\ G\ C = Some\ c; C \neq Object \rrbracket$
 $\Longrightarrow is\text{-}acc\text{-}class\ G\ (pid\ C)\ (super\ c)$
 $\langle proof \rangle$

lemma $wf\text{-}ws\text{-}prog\ [elim!,simp]: wf\text{-}prog\ G \Longrightarrow ws\text{-}prog\ G$
 $\langle proof \rangle$

lemma $class\text{-}Object\ [simp]:$
 $wf\text{-}prog\ G \Longrightarrow$
 $\quad class\ G\ Object = Some\ (\llbracket access=Public, cfields=[], methods=Object\text{-}mdecls,$
 $\quad \quad init=Skip, super=undefined, superIfs=[] \rrbracket)$
 $\langle proof \rangle$

lemma $methd\text{-}Object[simp]: wf\text{-}prog\ G \Longrightarrow methd\ G\ Object =$
 $\quad table\text{-}of\ (map\ (\lambda(s,m). (s, Object, m))\ Object\text{-}mdecls)$
 $\langle proof \rangle$

lemma $wf\text{-}prog\text{-}Object\text{-}methd:$
 $\llbracket wf\text{-}prog\ G; methd\ G\ Object\ sig = Some\ m \rrbracket \Longrightarrow accmodi\ m \neq Package$
 $\langle proof \rangle$

lemma *wf-prog-Object-is-public*[intro]:
 $wf\text{-prog } G \implies is\text{-public } G \text{ Object}$
 ⟨proof⟩

lemma *class-SXcpt* [simp]:
 $wf\text{-prog } G \implies$
 $class\ G\ (SXcpt\ xn) = Some\ (\lambda access=Public, cfields=[], methods=SXcpt\ mdecls,$
 $\quad\quad\quad init=Skip,$
 $\quad\quad\quad super=if\ xn = Throwable\ then\ Object$
 $\quad\quad\quad\quad\quad\quad else\ SXcpt\ Throwable,$
 $\quad\quad\quad superIfs=[])$
 ⟨proof⟩

lemma *wf-ObjectC* [simp]:
 $wf\text{-cdecl } G\ ObjectC = (\neg is\text{-iface } G\ Object \wedge Ball\ (set\ Object\ mdecls)$
 $\quad (wf\text{-mdecl } G\ Object) \wedge unique\ Object\ mdecls)$
 ⟨proof⟩

lemma *Object-is-class* [simp, elim!]: $wf\text{-prog } G \implies is\text{-class } G\ Object$
 ⟨proof⟩

lemma *Object-is-acc-class* [simp, elim!]: $wf\text{-prog } G \implies is\text{-acc-class } G\ S\ Object$
 ⟨proof⟩

lemma *SXcpt-is-class* [simp, elim!]: $wf\text{-prog } G \implies is\text{-class } G\ (SXcpt\ xn)$
 ⟨proof⟩

lemma *SXcpt-is-acc-class* [simp, elim!]:
 $wf\text{-prog } G \implies is\text{-acc-class } G\ S\ (SXcpt\ xn)$
 ⟨proof⟩

lemma *fields-Object* [simp]: $wf\text{-prog } G \implies DeclConcepts.fields\ G\ Object = []$
 ⟨proof⟩

lemma *accfield-Object* [simp]:
 $wf\text{-prog } G \implies accfield\ G\ S\ Object = Map.empty$
 ⟨proof⟩

lemma *fields-Throwable* [simp]:
 $wf\text{-prog } G \implies DeclConcepts.fields\ G\ (SXcpt\ Throwable) = []$
 ⟨proof⟩

lemma *fields-SXcpt* [simp]: $wf\text{-prog } G \implies DeclConcepts.fields\ G\ (SXcpt\ xn) = []$
 ⟨proof⟩

lemmas *widen-trans* = *ws-widen-trans* [OF - - *wf-ws-prog*, *elim*]
lemma *widen-trans2* [elim]: $\llbracket G \vdash U \preceq T; G \vdash S \preceq U; wf\text{-prog } G \rrbracket \implies G \vdash S \preceq T$
 ⟨proof⟩

lemma *Xcpt-subcls-Throwable* [simp]:
 $wf\text{-prog } G \implies G \vdash SXcpt\ xn \preceq_C SXcpt\ Throwable$
 ⟨proof⟩

lemma *unique-fields*:
 $\llbracket is\text{-class } G\ C; wf\text{-prog } G \rrbracket \implies unique\ (DeclConcepts.fields\ G\ C)$
 ⟨proof⟩

lemma *fields-mono*:

$\llbracket \text{table-of } (\text{DeclConcepts.fields } G \ C) \ fn = \text{Some } f; G \vdash D \preceq_C C; \\ \text{is-class } G \ D; \text{wf-prog } G \rrbracket \\ \implies \text{table-of } (\text{DeclConcepts.fields } G \ D) \ fn = \text{Some } f \\ \langle \text{proof} \rangle$

lemma *fields-is-type* [elim]:
 $\llbracket \text{table-of } (\text{DeclConcepts.fields } G \ C) \ m = \text{Some } f; \text{wf-prog } G; \text{is-class } G \ C \rrbracket \implies \\ \text{is-type } G \ (\text{type } f) \\ \langle \text{proof} \rangle$

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*:
 $\llbracket \text{wf-prog } G; \text{methd } G \ C \ \text{sig} = \text{Some } m; \\ \text{class } G \ C = \text{Some } y \rrbracket \\ \implies \text{is-type } G \ (\text{resTy } m) \\ \langle \text{proof} \rangle$

lemma *accmethd-rT-is-type*:
 $\llbracket \text{wf-prog } G; \text{accmethd } G \ S \ C \ \text{sig} = \text{Some } m; \\ \text{class } G \ C = \text{Some } y \rrbracket \\ \implies \text{is-type } G \ (\text{resTy } m) \\ \langle \text{proof} \rangle$

lemma *methd-Object-SomeD*:
 $\llbracket \text{wf-prog } G; \text{methd } G \ \text{Object} \ \text{sig} = \text{Some } m \rrbracket \\ \implies \text{declclass } m = \text{Object} \\ \langle \text{proof} \rangle$

lemmas *iface-rec-induct'* = *iface-rec.induct* [of %x y z. P x y] for P

lemma *wf-imethdsD*:
 $\llbracket im \in \text{imethds } G \ I \ \text{sig}; \text{wf-prog } G; \text{is-iface } G \ I \rrbracket \\ \implies \neg \text{is-static } im \wedge \text{accmodi } im = \text{Public} \\ \langle \text{proof} \rangle$

lemma *wf-prog-hidesD*:
assumes *hides*: $G \vdash \text{new hides old}$ **and** *wf*: $\text{wf-prog } G$
shows
 $\text{accmodi } old \leq \text{accmodi } new \wedge \\ \text{is-static } old \\ \langle \text{proof} \rangle$

Compare this lemma about static overriding $G \vdash \text{new overrides}_S \text{old}$ with the definition of dynamic overriding $G \vdash \text{new overrides old}$. 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*: $G \vdash \text{new overrides}_S \text{ old}$ **and** *wf*: *wf-prog* G

shows

$G \vdash \text{resTy new} \leq \text{resTy old} \wedge$
 $\text{accmodi old} \leq \text{accmodi new} \wedge$
 $\neg \text{is-static old}$

<proof>

lemma *static-to-dynamic-overriding*:

assumes *stat-override*: $G \vdash \text{new overrides}_S \text{ old}$ **and** *wf* : *wf-prog* G

shows $G \vdash \text{new overrides old}$

<proof>

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: *wf-prog* G

shows $G \vdash \text{Method old member-of } C \vee$

$(\exists \text{ new. } G \vdash \text{new overrides}_S \text{ old} \wedge G \vdash \text{Method new member-of } C)$

<proof>

lemma *non-Package-instance-method-inheritance-cases*:

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: *wf-prog* G

obtains (*Inheritance*) $G \vdash \text{Method old member-of } C$

| (*Overriding*) *new* **where** $G \vdash \text{new overrides}_S \text{ old}$ **and** $G \vdash \text{Method new member-of } C$

<proof>

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: *wf-prog* G

shows $G \vdash \text{new overrides}_S \text{ old}$

<proof>

lemma *wf-prog-dyn-override-prop*:

assumes *dyn-override*: $G \vdash \text{new overrides old}$ **and**

wf: *wf-prog* G

shows $\text{accmodi old} \leq \text{accmodi new}$

<proof>

lemma *overrides-Package-old*:

assumes *dyn-override*: $G \vdash \text{new overrides old}$ **and**

accmodi-new: $\text{accmodi new} = \text{Package}$ **and**

wf: *wf-prog* G

shows $\text{accmodi old} = \text{Package}$

<proof>

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: *wf-prog* G

shows $\text{pid}(\text{declclass old}) = \text{pid}(\text{declclass new})$

<proof>

lemma *dyn-override-Package-escape*:

assumes *dyn-override*: $G \vdash \text{new overrides old}$ **and**
accmodi-old: $\text{accmodi old} = \text{Package}$ **and**
outside-pack: $\text{pid}(\text{declclass old}) \neq \text{pid}(\text{declclass new})$ **and**
wf: *wf-prog* G

shows $\exists \text{inter}. G \vdash \text{new overrides inter} \wedge G \vdash \text{inter overrides old} \wedge$
 $\text{pid}(\text{declclass old}) = \text{pid}(\text{declclass inter}) \wedge$
 $\text{Protected} \leq \text{accmodi inter}$

<proof>

lemmas *class-rec-induct'* = *class-rec.induct* [of $\%x y z w. P x y$] **for** P

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*:

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: $C \neq \text{Object}$

obtains (*NewMethod*) *table-of* ($\text{map } (\lambda(s, m). (s, C, m)) (\text{methods } c)$) $\text{sig} = \text{Some } m$

| (*InheritedMethod*) $G \vdash C \text{ inherits } (\text{method sig } m)$ **and** $\text{methd } G (\text{super } c) \text{ sig} = \text{Some } m$

<proof>

lemma *methd-member-of*:

assumes *wf*: *wf-prog* G

shows

$\llbracket \text{is-class } G C; \text{methd } G C \text{ sig} = \text{Some } m \rrbracket \Longrightarrow G \vdash \text{Methd sig } m \text{ member-of } C$

(**is** ?*Class* $C \Longrightarrow$?*Method* $C \Longrightarrow$?*MemberOf* C)

<proof>

lemma *current-methd*:

$\llbracket \text{table-of } (\text{methods } c) \text{ sig} = \text{Some new};$

ws-prog $G; \text{class } G C = \text{Some } c; C \neq \text{Object};$

$\text{methd } G (\text{super } c) \text{ sig} = \text{Some old} \rrbracket$

$\Longrightarrow \text{methd } G C \text{ sig} = \text{Some } (C, \text{new})$

<proof>

lemma *wf-prog-staticD*:

assumes *wf*: wf-prog *G* **and**
clsC: class *G* *C* = Some *c* **and**
neg-C-Obj: *C* ≠ Object **and**
old: methd *G* (super *c*) sig = Some *old* **and**
accmodi-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**
accmodi-old: Protected ≤ accmodi *old* **and**
not-static-old: ¬ is-static *old*
shows
 $\exists \text{new. methd } G \ C \ \text{sig} = \text{Some new} \wedge$
 $(\text{new} = \text{old} \vee G, \text{sig} \vdash \text{new overrides}_S \text{old})$
 (is ($\exists \text{new. } (? \text{Constraint } C \ \text{new } \text{old}))$)
 ⟨proof⟩

lemma *inheritable-instance-methd-cases*:

assumes *subclseq-C-D*: $G \vdash C \preceq_C D$ **and**
is-cls-D: is-class *G* *D* **and**
wf: wf-prog *G* **and**
old: methd *G* *D* sig = Some *old* **and**
accmodi-old: Protected ≤ accmodi *old* **and**
not-static-old: ¬ is-static *old*
obtains (Inheritance) methd *G* *C* sig = Some *old*
 | (Overriding) new **where** methd *G* *C* sig = Some *new* **and** $G, \text{sig} \vdash \text{new overrides}_S \text{old}$
 ⟨proof⟩

lemma *inheritable-instance-methd-props*:

assumes *subclseq-C-D*: $G \vdash C \preceq_C D$ **and**
is-cls-D: is-class *G* *D* **and**
wf: wf-prog *G* **and**
old: methd *G* *D* sig = Some *old* **and**
accmodi-old: Protected ≤ accmodi *old* **and**
not-static-old: ¬ is-static *old*
shows
 $\exists \text{new. methd } G \ C \ \text{sig} = \text{Some new} \wedge$
 $\neg \text{is-static new} \wedge G \vdash \text{resTy new} \preceq \text{resTy old} \wedge \text{accmodi old} \leq \text{accmodi new}$
 (is ($\exists \text{new. } (? \text{Constraint } C \ \text{new } \text{old}))$)
 ⟨proof⟩

lemma *boxI'*: $x \in A \implies P \ x \implies \exists x \in A. P \ x$ ⟨proof⟩

lemma *ballE'*: $\forall x \in A. P \ x \implies (x \notin A \implies Q) \implies (P \ x \implies Q) \implies Q$ ⟨proof⟩

lemma *subint-widen-imethds*:

assumes *irel*: $G \vdash I \preceq I \ J$
and *wf*: wf-prog *G*
and *is-iface*: is-iface *G* *J*
and *jm*: *jm* ∈ imethds *G* *J* sig
shows $\exists \text{im} \in \text{imethds } G \ I \ \text{sig. is-static im} = \text{is-static jm} \wedge$
 $\text{accmodi im} = \text{accmodi jm} \wedge$

$G \vdash \text{resTy } im \leq_{\text{resTy}} jm$

$\langle \text{proof} \rangle$

lemma *implmt1-methd*:

$\bigwedge \text{sig. } \llbracket G \vdash C \rightsquigarrow 1I; \text{wf-prog } G; im \in \text{imethds } G \ I \ \text{sig} \rrbracket \implies$
 $\exists cm \in \text{methd } G \ C \ \text{sig: } \neg \text{is-static } cm \wedge \neg \text{is-static } im \wedge$
 $G \vdash \text{resTy } cm \leq_{\text{resTy}} im \wedge$
 $\text{accmodi } im = \text{Public} \wedge \text{accmodi } cm = \text{Public}$

$\langle \text{proof} \rangle$

lemma *implmt-methd* [rule-format (no-asm)]:

$\llbracket \text{wf-prog } G; G \vdash C \rightsquigarrow I \rrbracket \implies \text{is-iface } G \ I \longrightarrow$
 $(\forall im \in \text{imethds } G \ I \ \text{sig.}$
 $\exists cm \in \text{methd } G \ C \ \text{sig: } \neg \text{is-static } cm \wedge \neg \text{is-static } im \wedge$
 $G \vdash \text{resTy } cm \leq_{\text{resTy}} im \wedge$
 $\text{accmodi } im = \text{Public} \wedge \text{accmodi } cm = \text{Public})$

$\langle \text{proof} \rangle$

lemma *mheadsD* [rule-format (no-asm)]:

$emh \in \text{mheads } G \ S \ t \ \text{sig} \longrightarrow \text{wf-prog } G \longrightarrow$
 $(\exists C \ D \ m. t = \text{ClassT } C \wedge \text{declrefT } emh = \text{ClassT } D \wedge$
 $\text{accmethd } G \ S \ C \ \text{sig} = \text{Some } m \wedge$
 $(\text{declclass } m = D) \wedge \text{mhead } (mthd \ m) = (mhd \ emh)) \vee$
 $(\exists I. t = \text{IfaceT } I \wedge ((\exists im. im \in \text{accimethds } G \ (pid \ S) \ I \ \text{sig} \wedge$
 $mthd \ im = mhd \ emh) \vee$
 $(\exists m. G \vdash \text{Iface } I \text{ accessible-in } (pid \ S) \wedge \text{accmethd } G \ S \ \text{Object } \text{sig} = \text{Some } m \wedge$
 $\text{accmodi } m \neq \text{Private} \wedge$
 $\text{declrefT } emh = \text{ClassT } \text{Object} \wedge \text{mhead } (mthd \ m) = mhd \ emh))) \vee$
 $(\exists T \ m. t = \text{ArrayT } T \wedge G \vdash \text{Array } T \text{ accessible-in } (pid \ S) \wedge$
 $\text{accmethd } G \ S \ \text{Object } \text{sig} = \text{Some } m \wedge \text{accmodi } m \neq \text{Private} \wedge$
 $\text{declrefT } emh = \text{ClassT } \text{Object} \wedge \text{mhead } (mthd \ m) = mhd \ emh)$

$\langle \text{proof} \rangle$

lemma *mheads-cases*:

assumes $emh \in \text{mheads } G \ S \ t \ \text{sig}$ **and** $\text{wf-prog } G$

obtains $(\text{Class-methd}) \ C \ D \ m$ **where**

$t = \text{ClassT } C \ \text{declrefT } emh = \text{ClassT } D \ \text{accmethd } G \ S \ C \ \text{sig} = \text{Some } m$
 $\text{declclass } m = D \ \text{mhead } (mthd \ m) = mhd \ emh$

| $(\text{Iface-methd}) \ I \ im$ **where** $t = \text{IfaceT } I$

$im \in \text{accimethds } G \ (pid \ S) \ I \ \text{sig} \ mthd \ im = mhd \ emh$

| $(\text{Iface-Object-methd}) \ I \ m$ **where**

$t = \text{IfaceT } I \ G \vdash \text{Iface } I \text{ accessible-in } (pid \ S)$
 $\text{accmethd } G \ S \ \text{Object } \text{sig} = \text{Some } m \ \text{accmodi } m \neq \text{Private}$
 $\text{declrefT } emh = \text{ClassT } \text{Object} \ \text{mhead } (mthd \ m) = mhd \ emh$

| $(\text{Array-Object-methd}) \ T \ m$ **where**

$t = \text{ArrayT } T \ G \vdash \text{Array } T \text{ accessible-in } (pid \ S)$
 $\text{accmethd } G \ S \ \text{Object } \text{sig} = \text{Some } m \ \text{accmodi } m \neq \text{Private}$
 $\text{declrefT } emh = \text{ClassT } \text{Object} \ \text{mhead } (mthd \ m) = mhd \ emh$

$\langle \text{proof} \rangle$

lemma *declclassD*[rule-format]:

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

```

class G (declclass m) = Some d]]
⇒⇒ table-of (methods d) sig = Some (methd m)
⟨proof⟩

```

lemma *dynmethd-Object*:

```

assumes statM: methd G Object sig = Some statM and
  private: accmodi statM = Private and
  is-cls-C: is-class G C and
  wf: wf-prog G
shows dynmethd G Object C sig = Some statM
⟨proof⟩

```

lemma *wf-imethds-hiding-objmethdsD*:

```

assumes old: methd G Object sig = Some old and
  is-if-I: is-iface G I and
  wf: wf-prog G and
  not-private: accmodi old ≠ Private and
  new: new ∈ imethds G I sig
shows G ⊢ resTy new ≤ resTy old ∧ is-static new = is-static old (is ?P new)
⟨proof⟩

```

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 $G \vdash \text{Class } dynC \preceq statT$. 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

primrec *valid-lookup-cls*:: prog ⇒ ref-ty ⇒ qtname ⇒ bool ⇒ bool
 (⟨-, - ⟩ - valid'-lookup'-cls'-for -> [61,61,61,61] 60)

where

```

G, NullT ⊢ dynC valid-lookup-cls-for static-membr = False
| G, IfaceT I ⊢ dynC valid-lookup-cls-for static-membr
  = (if static-membr
     then dynC=Object
     else G ⊢ Class dynC ≤ Iface I)
| G, ClassT C ⊢ dynC valid-lookup-cls-for static-membr = G ⊢ Class dynC ≤ Class C
| G, ArrayT T ⊢ dynC valid-lookup-cls-for static-membr = (dynC=Object)

```

lemma *valid-lookup-cls-is-class*:

assumes $\text{dynC}: G, \text{statT} \vdash \text{dynC valid-lookup-cls-for static-membr}$ **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 :
 $\llbracket \text{emh} \in \text{mheads } G \text{ S statT sig};$
 $G, \text{statT} \vdash \text{dynC valid-lookup-cls-for (is-static emh)};$
 $\text{isrtype } G \text{ statT}; \text{wf-prog } G$
 $\rrbracket \implies \exists m \in \text{dynlookup } G \text{ statT dynC sig}:$
 $\text{is-static } m = \text{is-static emh} \wedge G \vdash \text{resTy } m \preceq \text{resTy emh}$
 $\langle \text{proof} \rangle$
declare split-paired-All [simp] split-paired-Ex [simp]
 $\langle \text{ML} \rangle$

lemma methd-declclass :
 $\llbracket \text{class } G \text{ C} = \text{Some } c; \text{wf-prog } G; \text{methd } G \text{ C sig} = \text{Some } m \rrbracket$
 $\implies \text{methd } G (\text{declclass } m) \text{ sig} = \text{Some } m$
 $\langle \text{proof} \rangle$

lemma $\text{dynmethd-declclass}$:
 $\llbracket \text{dynmethd } G \text{ statC dynC sig} = \text{Some } m;$
 $\text{wf-prog } G; \text{is-class } G \text{ statC}$
 $\rrbracket \implies \text{methd } G (\text{declclass } m) \text{ sig} = \text{Some } m$
 $\langle \text{proof} \rangle$

lemma dynlookup-declC :
 $\llbracket \text{dynlookup } G \text{ statT dynC sig} = \text{Some } m; \text{wf-prog } G;$
 $\text{is-class } G \text{ dynC}; \text{isrtype } G \text{ statT}$
 $\rrbracket \implies G \vdash \text{dynC} \preceq_C (\text{declclass } m) \wedge \text{is-class } G (\text{declclass } m)$
 $\langle \text{proof} \rangle$

lemma $\text{dynlookup-Array-declclassD}$ [simp]:
 $\llbracket \text{dynlookup } G (\text{ArrayT } T) \text{ Object sig} = \text{Some } dm; \text{wf-prog } G \rrbracket$
 $\implies \text{declclass } dm = \text{Object}$
 $\langle \text{proof} \rangle$

declare split-paired-All [simp del] split-paired-Ex [simp del]
 $\langle \text{ML} \rangle$

lemma wt-is-type : $E, dt \models v::T \implies \text{wf-prog } (\text{prg } E) \longrightarrow$
 $dt = \text{empty-dt} \longrightarrow (\text{case } T \text{ of}$
 $\quad \text{Inl } T \Rightarrow \text{is-type } (\text{prg } E) \text{ } T$
 $\quad | \text{Inr } Ts \Rightarrow \text{Ball } (\text{set } Ts) (\text{is-type } (\text{prg } E)))$
 $\langle \text{proof} \rangle$

declare split-paired-All [simp] split-paired-Ex [simp]
 $\langle \text{ML} \rangle$

lemma ty-expr-is-type :
 $\llbracket E \vdash e::T; \text{wf-prog } (\text{prg } E) \rrbracket \implies \text{is-type } (\text{prg } E) \text{ } T$

$\langle \text{proof} \rangle$

lemma *ty-var-is-type*:

$\llbracket E \vdash v :: T; \text{wf-prog } (\text{prg } E) \rrbracket \implies \text{is-type } (\text{prg } E) \ T$

$\langle \text{proof} \rangle$

lemma *ty-exprs-is-type*:

$\llbracket E \vdash es :: Ts; \text{wf-prog } (\text{prg } E) \rrbracket \implies \text{Ball } (\text{set } Ts) \ (\text{is-type } (\text{prg } E))$

$\langle \text{proof} \rangle$

lemma *static-mheadsD*:

$\llbracket \text{emh} \in \text{mheads } G \ S \ t \ \text{sig}; \text{wf-prog } G; E \vdash e :: \text{RefT } t; \text{prg } E = G ;$
 $\text{invmode } (\text{mhd } \text{emh}) \ e \neq \text{IntVir}$
 $\rrbracket \implies \exists m. ((\exists C. t = \text{ClassT } C \wedge \text{accmethd } G \ S \ C \ \text{sig} = \text{Some } m)$
 $\vee (\forall C. t \neq \text{ClassT } C \wedge \text{accmethd } G \ S \ \text{Object } \text{sig} = \text{Some } m)) \wedge$
 $\text{declrefT } \text{emh} = \text{ClassT } (\text{declclass } m) \wedge \text{mhead } (\text{mthd } m) = (\text{mhd } \text{emh})$
 $\langle \text{proof} \rangle$

lemma *wt-MethdI*:

$\llbracket \text{methd } G \ C \ \text{sig} = \text{Some } m; \text{wf-prog } G;$
 $\text{class } G \ C = \text{Some } c \rrbracket \implies$
 $\exists T. (\text{prg} = G, \text{cls} = (\text{declclass } m),$
 $\text{lcl} = \text{callee-lcl } (\text{declclass } m) \ \text{sig } (\text{mthd } m)) \vdash \text{Methd } C \ \text{sig} :: \text{RefT } T \wedge G \vdash T \preceq_{\text{resTy}} m$
 $\langle \text{proof} \rangle$

2 accessibility concerns

lemma *mheads-type-accessible*:

$\llbracket \text{emh} \in \text{mheads } G \ S \ T \ \text{sig}; \text{wf-prog } G \rrbracket$
 $\implies G \vdash \text{RefT } T \text{ accessible-in } (\text{pid } S)$
 $\langle \text{proof} \rangle$

lemma *static-to-dynamic-accessible-from-aux*:

$\llbracket G \vdash m \text{ of } C \text{ accessible-from } \text{accC}; \text{wf-prog } G \rrbracket$
 $\implies G \vdash m \text{ in } C \text{ dyn-accessible-from } \text{accC}$
 $\langle \text{proof} \rangle$

lemma *static-to-dynamic-accessible-from*:

assumes *stat-acc*: $G \vdash m \text{ of } \text{statC} \text{ accessible-from } \text{accC}$ **and**
 $\text{subclseq}: G \vdash \text{dynC} \preceq_C \text{statC}$ **and**
 $\text{wf}: \text{wf-prog } G$
shows $G \vdash m \text{ in } \text{dynC} \text{ dyn-accessible-from } \text{accC}$
 $\langle \text{proof} \rangle$

lemma *static-to-dynamic-accessible-from-static*:

assumes *stat-acc*: $G \vdash m \text{ of } \text{statC} \text{ accessible-from } \text{accC}$ **and**
 $\text{static}: \text{is-static } m$ **and**
 $\text{wf}: \text{wf-prog } G$
shows $G \vdash m \text{ in } (\text{declclass } m) \text{ dyn-accessible-from } \text{accC}$
 $\langle \text{proof} \rangle$

lemma *dynmethd-member-in*:

assumes $m: \text{dynmethd } G \ \text{statC} \ \text{dynC} \ \text{sig} = \text{Some } m$ **and**
 $\text{iscls-statC}: \text{is-class } G \ \text{statC}$ **and**
 $\text{wf}: \text{wf-prog } G$
shows $G \vdash \text{Methd } \text{sig } m \text{ member-in } \text{dynC}$
 $\langle \text{proof} \rangle$

lemma *dynmethd-access-prop*:

assumes *statM*: $\text{methd } G \ \text{statC} \ \text{sig} = \text{Some } \text{statM}$ **and**

$stat\text{-}acc: G \vdash \text{Methd } sig \text{ statM of statC accessible-from } accC \text{ and}$
 $dynM: dynmethd \ G \ statC \ dynC \ sig = \text{Some } dynM \text{ and}$
 $wf: wf\text{-}prog \ G$
shows $G \vdash \text{Methd } sig \ dynM \text{ in } dynC \ dyn\text{-}accessible\text{-from } accC$
 $\langle proof \rangle$

lemma *implmt-methd-access*:
fixes $accC::qtname$
assumes $iface\text{-}methd: imethds \ G \ I \ sig \neq \{\}$ **and**
 $implements: G \vdash dynC \rightsquigarrow I$ **and**
 $isif\text{-}I: is\text{-}iface \ G \ I$ **and**
 $wf: wf\text{-}prog \ G$
shows $\exists \ dynM. \text{methd } G \ dynC \ sig = \text{Some } dynM \wedge$
 $G \vdash \text{Methd } sig \ dynM \text{ in } dynC \ dyn\text{-}accessible\text{-from } accC$
 $\langle proof \rangle$

corollary *implmt-dynimethd-access*:
fixes $accC::qtname$
assumes $iface\text{-}methd: imethds \ G \ I \ sig \neq \{\}$ **and**
 $implements: G \vdash dynC \rightsquigarrow I$ **and**
 $isif\text{-}I: is\text{-}iface \ G \ I$ **and**
 $wf: wf\text{-}prog \ G$
shows $\exists \ dynM. \text{dynimethd } G \ I \ dynC \ sig = \text{Some } dynM \wedge$
 $G \vdash \text{Methd } sig \ dynM \text{ in } dynC \ dyn\text{-}accessible\text{-from } accC$
 $\langle proof \rangle$

lemma *dynlookup-access-prop*:
assumes $emh: emh \in mheads \ G \ accC \ statT \ sig$ **and**
 $dynM: dynlookup \ G \ statT \ dynC \ sig = \text{Some } dynM$ **and**
 $dynC\text{-}prop: G, statT \vdash dynC \text{ valid-lookup-cls-for } is\text{-}static \ emh$ **and**
 $isT\text{-}statT: isrtype \ G \ statT$ **and**
 $wf: wf\text{-}prog \ G$
shows $G \vdash \text{Methd } sig \ dynM \text{ in } dynC \ dyn\text{-}accessible\text{-from } accC$
 $\langle proof \rangle$

lemma *dynlookup-access*:
assumes $emh: emh \in mheads \ G \ accC \ statT \ sig$ **and**
 $dynC\text{-}prop: G, statT \vdash dynC \text{ valid-lookup-cls-for } (is\text{-}static \ emh)$ **and**
 $isT\text{-}statT: isrtype \ G \ statT$ **and**
 $wf: wf\text{-}prog \ G$
shows $\exists \ dynM. \text{dynlookup } G \ statT \ dynC \ sig = \text{Some } dynM \wedge$
 $G \vdash \text{Methd } sig \ dynM \text{ in } dynC \ dyn\text{-}accessible\text{-from } accC$
 $\langle proof \rangle$

lemma *stat-overrides-Package-old*:
assumes $stat\text{-}override: G \vdash \text{new overrides}_S \ old$ **and**
 $accmodi\text{-}new: accmodi \ new = \text{Package}$ **and**
 $wf: wf\text{-}prog \ G$
shows $accmodi \ old = \text{Package}$
 $\langle proof \rangle$

Properties of dynamic accessibility

lemma *dyn-accessible-Private*:
assumes $dyn\text{-}acc: G \vdash m \text{ in } C \ dyn\text{-}accessible\text{-from } accC$ **and**
 $priv: accmodi \ m = \text{Private}$
shows $accC = \text{declclass } m$
 $\langle proof \rangle$

dyn-accessible-Package only works with the *wf-prog* assumption. Without it. it is easy to leaf the Package!

lemma *dyn-accessible-Package*:

$\llbracket G \vdash m \text{ in } C \text{ dyn-accessible-from } accC; accmodi\ m = Package;$
 $\quad wf\text{-prog } G \rrbracket$
 $\implies pid\ accC = pid\ (declclass\ m)$
 $\langle proof \rangle$

For fields we don't need the wellformedness of the program, since there is no overriding

lemma *dyn-accessible-field-Package*:

assumes *dyn-acc*: $G \vdash f \text{ in } C \text{ dyn-accessible-from } accC$ **and**
 $\quad pack: accmodi\ f = Package$ **and**
 $\quad field: is\text{-field } f$
shows $pid\ accC = pid\ (declclass\ f)$
 $\langle proof \rangle$

dyn-accessible-instance-field-Protected only works for fields since methods can break the package bounds due to overriding

lemma *dyn-accessible-instance-field-Protected*:

assumes *dyn-acc*: $G \vdash f \text{ in } C \text{ dyn-accessible-from } accC$ **and**
 $\quad prot: accmodi\ f = Protected$ **and**
 $\quad field: is\text{-field } f$ **and**
 $\quad instance\text{-field}: \neg is\text{-static } f$ **and**
 $\quad outside: pid\ (declclass\ f) \neq pid\ accC$
shows $G \vdash C \preceq_C accC$
 $\langle proof \rangle$

lemma *dyn-accessible-static-field-Protected*:

assumes *dyn-acc*: $G \vdash f \text{ in } C \text{ dyn-accessible-from } accC$ **and**
 $\quad prot: accmodi\ f = Protected$ **and**
 $\quad field: is\text{-field } f$ **and**
 $\quad static\text{-field}: is\text{-static } f$ **and**
 $\quad outside: pid\ (declclass\ f) \neq pid\ accC$
shows $G \vdash accC \preceq_C declclass\ f \wedge G \vdash C \preceq_C declclass\ f$
 $\langle proof \rangle$

end

Chapter 14

State

1 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)
```

```
type-synonym vn = fspec + int      — variable name
record obj =
  tag :: obj-tag                  — generalized object
  values :: (vn, val) table
```

translations

```
(type) fspec <= (type) vname × qname
(type) vn    <= (type) fspec + int
(type) obj   <= (type) (tag::obj-tag, values::vn ⇒ val option)
(type) obj   <= (type) (tag::obj-tag, values::vn ⇒ val option,...::'a)
```

definition

```
the-Arr :: obj option ⇒ ty × int × (vn, val) table
where the-Arr obj = (SOME (T,k,t). obj = Some (tag=Arr T k, values=t))
```

```
lemma the-Arr-Arr [simp]: the-Arr (Some (tag=Arr T k, values=cs)) = (T,k,cs)
⟨proof⟩
```

```
lemma the-Arr-Arr1 [simp,intro,dest]:
  [tag obj = Arr T k] ⇒ the-Arr (Some obj) = (T,k, values obj)
⟨proof⟩
```

definition

$upd\text{-}obj :: vn \Rightarrow val \Rightarrow obj \Rightarrow obj$
where $upd\text{-}obj\ n\ v = (\lambda obj. obj\ (\downarrow values := (values\ obj)(n \mapsto v)))$

lemma $upd\text{-}obj\text{-}def2$ $[simp]$:

$upd\text{-}obj\ n\ v\ obj = obj\ (\downarrow values := (values\ obj)(n \mapsto v))$
 $\langle proof \rangle$

definition

$obj\text{-}ty :: obj \Rightarrow ty$ **where**
 $obj\text{-}ty\ obj = (case\ tag\ obj\ of$
 $\quad CInst\ C \Rightarrow Class\ C$
 $\quad | Arr\ T\ k \Rightarrow T.\Box)$

lemma $obj\text{-}ty\text{-}eq$ $[intro!]$: $obj\text{-}ty\ (\downarrow tag = oi, values = x) = obj\text{-}ty\ (\downarrow tag = oi, values = y)$

$\langle proof \rangle$

lemma $obj\text{-}ty\text{-}eq1$ $[intro!, dest]$:

$tag\ obj = tag\ obj' \implies obj\text{-}ty\ obj = obj\text{-}ty\ obj'$
 $\langle proof \rangle$

lemma $obj\text{-}ty\text{-}cong$ $[simp]$:

$obj\text{-}ty\ (obj\ (\downarrow values := vs)) = obj\text{-}ty\ obj$
 $\langle proof \rangle$

lemma $obj\text{-}ty\text{-}CInst$ $[simp]$:

$obj\text{-}ty\ (\downarrow tag = CInst\ C, values = vs) = Class\ C$
 $\langle proof \rangle$

lemma $obj\text{-}ty\text{-}CInst1$ $[simp, intro!, dest]$:

$\llbracket tag\ obj = CInst\ C \rrbracket \implies obj\text{-}ty\ obj = Class\ C$
 $\langle proof \rangle$

lemma $obj\text{-}ty\text{-}Arr$ $[simp]$:

$obj\text{-}ty\ (\downarrow tag = Arr\ T\ i, values = vs) = T.\Box$
 $\langle proof \rangle$

lemma $obj\text{-}ty\text{-}Arr1$ $[simp, intro!, dest]$:

$\llbracket tag\ obj = Arr\ T\ i \rrbracket \implies obj\text{-}ty\ obj = T.\Box$
 $\langle proof \rangle$

lemma $obj\text{-}ty\text{-}widenD$:

$\vdash obj\text{-}ty\ obj \preceq RefT\ t \implies (\exists C. tag\ obj = CInst\ C) \vee (\exists T\ k. tag\ obj = Arr\ T\ k)$
 $\langle proof \rangle$

definition

$obj\text{-}class :: obj \Rightarrow qname$ **where**
 $obj\text{-}class\ obj = (case\ tag\ obj\ of$
 $\quad CInst\ C \Rightarrow C$
 $\quad | Arr\ T\ k \Rightarrow Object)$

lemma $obj\text{-}class\text{-}CInst$ $[simp]$: $obj\text{-}class\ (\downarrow tag = CInst\ C, values = vs) = C$

$\langle proof \rangle$

lemma $obj\text{-}class\text{-}CInst1$ $[simp, intro!, dest]$:

$tag\ obj = CInst\ C \implies obj\text{-}class\ obj = C$

$\langle \text{proof} \rangle$

lemma *obj-class-Arr* [simp]: *obj-class* ($\text{tag}=\text{Arr } T \ k, \text{values}=vs$) = *Object*
 $\langle \text{proof} \rangle$

lemma *obj-class-Arr1* [simp,intro!,dest]:
 $\text{tag } \text{obj} = \text{Arr } T \ k \implies \text{obj-class } \text{obj} = \text{Object}$
 $\langle \text{proof} \rangle$

lemma *obj-ty-obj-class*: $G \vdash \text{obj-ty } \text{obj} \preceq \text{Class statC} = G \vdash \text{obj-class } \text{obj} \preceq_C \text{statC}$
 $\langle \text{proof} \rangle$

object references

type-synonym *oref* = *loc* + *qname* — generalized object reference

translations

$(\text{type}) \text{ oref} \leq (\text{type}) \text{ loc} + \text{qname}$

abbreviation (input)

$\text{Heap} :: \text{loc} \Rightarrow \text{oref} \textbf{ where } \text{Heap} \equiv \text{Inl}$

abbreviation (input)

$\text{Stat} :: \text{qname} \Rightarrow \text{oref} \textbf{ where } \text{Stat} \equiv \text{Inr}$

definition

$\text{fields-table} :: \text{prog} \Rightarrow \text{qname} \Rightarrow (\text{fspec} \Rightarrow \text{field} \Rightarrow \text{bool}) \Rightarrow (\text{fspec}, \text{ty}) \text{ table} \textbf{ where}$
 $\text{fields-table } G \ C \ P =$
 $\text{map-option type} \circ \text{table-of } (\text{filter } (\text{case-prod } P) (\text{DeclConcepts.fields } G \ C))$

lemma fields-table-SomeI:

$\llbracket \text{table-of } (\text{DeclConcepts.fields } G \ C) \ n = \text{Some } f; P \ n \ f \rrbracket$
 $\implies \text{fields-table } G \ C \ P \ n = \text{Some } (\text{type } f)$
 $\langle \text{proof} \rangle$

lemma fields-table-SomeD':

$\exists f. (fn, f) \in \text{set}(\text{DeclConcepts.fields } G \ C) \wedge \text{type } f = T$
 $\langle \text{proof} \rangle$

lemma fields-table-SomeD:

$\llbracket \text{fields-table } G \ C \ P \ fn = \text{Some } T; \text{unique } (\text{DeclConcepts.fields } G \ C) \rrbracket \implies$
 $\exists f. \text{table-of } (\text{DeclConcepts.fields } G \ C) \ fn = \text{Some } f \wedge \text{type } f = T$
 $\langle \text{proof} \rangle$

definition

$\text{in-bounds} :: \text{int} \Rightarrow \text{int} \Rightarrow \text{bool} \ (\langle - / \text{in'-bounds } - \rangle [50, 51] \ 50)$
 $\textbf{where } i \text{ in-bounds } k = (0 \leq i \wedge i < k)$

definition

$\text{arr-comps} :: 'a \Rightarrow \text{int} \Rightarrow \text{int} \Rightarrow 'a \text{ option}$
 $\textbf{where } \text{arr-comps } T \ k = (\lambda i. \text{if } i \text{ in-bounds } k \text{ then } \text{Some } T \text{ else } \text{None})$

definition

$\text{var-tys} :: \text{prog} \Rightarrow \text{obj-tag} \Rightarrow \text{oref} \Rightarrow (\text{vn}, \text{ty}) \text{ table} \textbf{ where}$
 $\text{var-tys } G \ oi \ r =$
 $(\text{case } r \text{ of}$
 $\text{Heap } a \Rightarrow (\text{case } oi \text{ of}$
 $\text{CInst } C \Rightarrow \text{fields-table } G \ C \ (\lambda n \ f. \neg \text{static } f) \ (+) \ \text{Map.empty}$
 $\mid \text{Arr } T \ k \Rightarrow \text{Map.empty } (+) \ \text{arr-comps } T \ k)$

| $Stat\ C \Rightarrow fields_table\ G\ C\ (\lambda fn\ f. declclassf\ fn = C \wedge static\ f)$
 (+) $Map.empty$)

lemma *var-tys-Some-eq*:

$var_tys\ G\ oi\ r\ n = Some\ T$
 $= (case\ r\ of$
 $Inl\ a \Rightarrow (case\ oi\ of$
 $CInst\ C \Rightarrow (\exists\ nt. n = Inl\ nt \wedge fields_table\ G\ C\ (\lambda n\ f.$
 $\neg static\ f)\ nt = Some\ T)$
 | $Arr\ t\ k \Rightarrow (\exists\ i. n = Inr\ i \wedge i\ in_bounds\ k \wedge t = T))$
 | $Inr\ C \Rightarrow (\exists\ nt. n = Inl\ nt \wedge$
 $fields_table\ G\ C\ (\lambda fn\ f. declclassf\ fn = C \wedge static\ f)\ nt$
 $= Some\ T))$

$\langle proof \rangle$

stores

type-synonym *globs* — global variables: heap and static variables

$= (oref\ ,\ obj)\ table$

type-synonym *heap*

$= (loc\ ,\ obj)\ table$

translations

$(type)\ globs\ \leq (type)\ (oref\ ,\ obj)\ table$

$(type)\ heap\ \leq (type)\ (loc\ ,\ obj)\ table$

datatype *st* =

st globs locals

2 access

definition

$globs :: st \Rightarrow globs$

where $globs = case_st\ (\lambda g\ l. g)$

definition

$locals :: st \Rightarrow locals$

where $locals = case_st\ (\lambda g\ l. l)$

definition $heap :: st \Rightarrow heap$ **where**

$heap\ s = globs\ s \circ Heap$

lemma *globs-def2* [simp]: $globs\ (st\ g\ l) = g$

$\langle proof \rangle$

lemma *locals-def2* [simp]: $locals\ (st\ g\ l) = l$

$\langle proof \rangle$

lemma *heap-def2* [simp]: $heap\ s\ a = globs\ s\ (Heap\ a)$

$\langle proof \rangle$

abbreviation $val_this :: st \Rightarrow val$

where $val_this\ s == the\ (locals\ s\ This)$

abbreviation $lookup_obj :: st \Rightarrow val \Rightarrow obj$

where $\text{lookup-obj } s \ a' == \text{the } (\text{heap } s \ (\text{the-Addr } a'))$

3 memory allocation

definition

$\text{new-Addr} :: \text{heap} \Rightarrow \text{loc option}$ **where**
 $\text{new-Addr } h = (\text{if } (\forall a. h \ a \neq \text{None}) \text{ then } \text{None} \text{ else } \text{Some } (\text{SOME } a. h \ a = \text{None}))$

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

$\langle \text{proof} \rangle$

lemma new-AddrD2 : $\text{new-Addr } h = \text{Some } a \implies \forall b. h \ b \neq \text{None} \longrightarrow b \neq a$

$\langle \text{proof} \rangle$

lemma new-Addr-SomeI : $h \ a = \text{None} \implies \exists b. \text{new-Addr } h = \text{Some } b \wedge h \ b = \text{None}$

$\langle \text{proof} \rangle$

4 initialization

abbreviation $\text{init-vals} :: ('a, \text{ty}) \text{ table} \Rightarrow ('a, \text{val}) \text{ table}$

where $\text{init-vals } vs == \text{map-option default-val } \circ \text{ vs}$

lemma $\text{init-arr-comps-base}$ $[\text{simp}]$: $\text{init-vals } (\text{arr-comps } T \ 0) = \text{Map.empty}$

$\langle \text{proof} \rangle$

lemma $\text{init-arr-comps-step}$ $[\text{simp}]$:

$0 < j \implies \text{init-vals } (\text{arr-comps } T \ j) =$
 $(\text{init-vals } (\text{arr-comps } T \ (j - 1)))(j - 1 \mapsto \text{default-val } T)$

$\langle \text{proof} \rangle$

5 update

definition

$\text{gupd} :: \text{oref} \Rightarrow \text{obj} \Rightarrow \text{st} \Rightarrow \text{st } (\langle \text{gupd}'(\mapsto -) \rangle [10, 10] \ 1000)$
where $\text{gupd } r \ \text{obj} = \text{case-st } (\lambda g \ l. \text{st } (g(r \mapsto \text{obj}))) \ l$

definition

$\text{lupd} :: \text{lname} \Rightarrow \text{val} \Rightarrow \text{st} \Rightarrow \text{st } (\langle \text{lupd}'(\mapsto -) \rangle [10, 10] \ 1000)$
where $\text{lupd } vn \ v = \text{case-st } (\lambda g \ l. \text{st } g \ (l(vn \mapsto v)))$

definition

$\text{upd-gobj} :: \text{oref} \Rightarrow \text{vn} \Rightarrow \text{val} \Rightarrow \text{st} \Rightarrow \text{st}$
where $\text{upd-gobj } r \ n \ v = \text{case-st } (\lambda g \ l. \text{st } (\text{chg-map } (\text{upd-obj } n \ v) \ r \ g) \ l)$

definition

$\text{set-locals} :: \text{locals} \Rightarrow \text{st} \Rightarrow \text{st}$
where $\text{set-locals } l = \text{case-st } (\lambda g \ l'. \text{st } g \ l)$

definition

$\text{init-obj} :: \text{prog} \Rightarrow \text{obj-tag} \Rightarrow \text{oref} \Rightarrow \text{st} \Rightarrow \text{st}$
where $\text{init-obj } G \ oi \ r = \text{gupd}(r \mapsto \langle \text{tag} = oi, \text{values} = \text{init-vals } (\text{var-tys } G \ oi \ r) \rangle)$

abbreviation

$\text{init-class-obj} :: \text{prog} \Rightarrow \text{qtname} \Rightarrow \text{st} \Rightarrow \text{st}$
where $\text{init-class-obj } G \ C == \text{init-obj } G \ \text{undefined } (\text{Inr } C)$

lemma gupd-def2 $[\text{simp}]$: $\text{gupd}(r \mapsto \text{obj}) \ (\text{st } g \ l) = \text{st } (g(r \mapsto \text{obj})) \ l$

$\langle \text{proof} \rangle$

lemma *lupd-def2* [simp]: $\text{lupd}(vn \mapsto v) (st\ g\ l) = st\ g\ (l(vn \mapsto v))$
 <proof>

lemma *globals-gupd* [simp]: $\text{globals}\ (\text{gupd}(r \mapsto obj)\ s) = (\text{globals}\ s)(r \mapsto obj)$
 <proof>

lemma *globals-lupd* [simp]: $\text{globals}\ (\text{lupd}(vn \mapsto v)\ s) = \text{globals}\ s$
 <proof>

lemma *locals-gupd* [simp]: $\text{locals}\ (\text{gupd}(r \mapsto obj)\ s) = \text{locals}\ s$
 <proof>

lemma *locals-lupd* [simp]: $\text{locals}\ (\text{lupd}(vn \mapsto v)\ s) = (\text{locals}\ s)(vn \mapsto v)$
 <proof>

lemma *globals-upd-gobj-new* [rule-format (no-asm), simp]:
 $\text{globals}\ s\ r = \text{None} \longrightarrow \text{globals}\ (\text{upd-gobj}\ r\ n\ v\ s) = \text{globals}\ s$
 <proof>

lemma *globals-upd-gobj-upd* [rule-format (no-asm), simp]:
 $\text{globals}\ s\ r = \text{Some}\ obj \longrightarrow \text{globals}\ (\text{upd-gobj}\ r\ n\ v\ s) = (\text{globals}\ s)(r \mapsto \text{upd-obj}\ n\ v\ obj)$
 <proof>

lemma *locals-upd-gobj* [simp]: $\text{locals}\ (\text{upd-gobj}\ r\ n\ v\ s) = \text{locals}\ s$
 <proof>

lemma *globals-init-obj* [simp]: $\text{globals}\ (\text{init-obj}\ G\ oi\ r\ s)\ t =$
 (if $t=r$ then $\text{Some}\ (\text{tag}=oi, \text{values}=\text{init-vals}\ (\text{var-tys}\ G\ oi\ r))$ else $\text{globals}\ s\ t$)
 <proof>

lemma *locals-init-obj* [simp]: $\text{locals}\ (\text{init-obj}\ G\ oi\ r\ s) = \text{locals}\ s$
 <proof>

lemma *surjective-st* [simp]: $st\ (\text{globals}\ s)\ (\text{locals}\ s) = s$
 <proof>

lemma *surjective-st-init-obj*:
 $st\ (\text{globals}\ (\text{init-obj}\ G\ oi\ r\ s))\ (\text{locals}\ s) = \text{init-obj}\ G\ oi\ r\ s$
 <proof>

lemma *heap-heap-upd* [simp]:
 $\text{heap}\ (st\ (g(\text{Inl}\ a \mapsto obj))\ l) = (\text{heap}\ (st\ g\ l))(a \mapsto obj)$
 <proof>

lemma *heap-stat-upd* [simp]: $\text{heap}\ (st\ (g(\text{Inr}\ C \mapsto obj))\ l) = \text{heap}\ (st\ g\ l)$
 <proof>

lemma *heap-local-upd* [simp]: $\text{heap}\ (st\ g\ (l(vn \mapsto v))) = \text{heap}\ (st\ g\ l)$
 <proof>

lemma *heap-gupd-Heap* [simp]: $\text{heap}\ (\text{gupd}(\text{Heap}\ a \mapsto obj)\ s) = (\text{heap}\ s)(a \mapsto obj)$
 <proof>

lemma *heap-gupd-Stat* [simp]: $\text{heap}\ (\text{gupd}(\text{Stat}\ C \mapsto obj)\ s) = \text{heap}\ s$
 <proof>

lemma *heap-lupd* [simp]: $\text{heap}\ (\text{lupd}(vn \mapsto v)\ s) = \text{heap}\ s$
 <proof>

lemma *heap-upd-gobj-Stat* [simp]: $\text{heap}\ (\text{upd-gobj}\ (\text{Stat}\ C)\ n\ v\ s) = \text{heap}\ s$
 <proof>

lemma *set-locals-def2* [simp]: *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 *globals-set-locals* [simp]: *globals* (*set-locals l s*) = *globals s*
 ⟨proof⟩

lemma *heap-set-locals* [simp]: *heap* (*set-locals l s*) = *heap s*
 ⟨proof⟩

abrupt completion

primrec *the-Xcpt* :: *abrupt* \Rightarrow *xcpt*
 where *the-Xcpt* (*Xcpt x*) = *x*

primrec *the-Jump* :: *abrupt* \Rightarrow *jump*
 where *the-Jump* (*Jump j*) = *j*

primrec *the-Loc* :: *xcpt* \Rightarrow *loc*
 where *the-Loc* (*Loc a*) = *a*

primrec *the-Std* :: *xcpt* \Rightarrow *xname*
 where *the-Std* (*Std x*) = *x*

definition

abrupt-if :: *bool* \Rightarrow *abopt* \Rightarrow *abopt* \Rightarrow *abopt*
 where *abrupt-if* *c x' x* = (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⟩

abbreviation *raise-if* :: *bool* \Rightarrow *xname* \Rightarrow *abopt* \Rightarrow *abopt*

where *raise-if* $c\ xn == \text{abrupt-if } c\ (\text{Some } (Xcpt\ (Std\ xn)))$

abbreviation *np* :: $val \Rightarrow abopt \Rightarrow abopt$

where *np* $v == \text{raise-if } (v = \text{Null})\ \text{NullPointer}$

abbreviation *check-neg* :: $val \Rightarrow abopt \Rightarrow abopt$

where *check-neg* $i' == \text{raise-if } (\text{the-Intg } i' < 0)\ \text{NegArrSize}$

abbreviation *error-if* :: $bool \Rightarrow error \Rightarrow abopt \Rightarrow abopt$

where *error-if* $c\ e == \text{abrupt-if } c\ (\text{Some } (\text{Error } e))$

lemma *raise-if-None* [simp]: $(\text{raise-if } c\ x\ y = \text{None}) = (\neg c \wedge y = \text{None})$

<proof>

declare *raise-if-None* [THEN iffD1, dest!]

lemma *if-raise-if-None* [simp]:

$((\text{if } b \text{ then } y \text{ else } \text{raise-if } c\ x\ y) = \text{None}) = ((c \longrightarrow b) \wedge y = \text{None})$

<proof>

lemma *raise-if-SomeD* [dest!]:

$\text{raise-if } c\ x\ y = \text{Some } z \implies c \wedge z = (Xcpt\ (Std\ x)) \wedge y = \text{None} \vee (y = \text{Some } z)$

<proof>

lemma *error-if-None* [simp]: $(\text{error-if } c\ e\ y = \text{None}) = (\neg c \wedge y = \text{None})$

<proof>

declare *error-if-None* [THEN iffD1, dest!]

lemma *if-error-if-None* [simp]:

$((\text{if } b \text{ then } y \text{ else } \text{error-if } c\ e\ y) = \text{None}) = ((c \longrightarrow b) \wedge y = \text{None})$

<proof>

lemma *error-if-SomeD* [dest!]:

$\text{error-if } c\ e\ y = \text{Some } z \implies c \wedge z = (\text{Error } e) \wedge y = \text{None} \vee (y = \text{Some } z)$

<proof>

definition

absorb :: $jump \Rightarrow abopt \Rightarrow abopt$

where *absorb* $j\ a = (\text{if } a = \text{Some } (Jump\ j) \text{ then } \text{None} \text{ else } a)$

lemma *absorb-SomeD* [dest!]: $\text{absorb } j\ a = \text{Some } x \implies a = \text{Some } x$

<proof>

lemma *absorb-same* [simp]: $\text{absorb } j\ (\text{Some } (Jump\ j)) = \text{None}$

<proof>

lemma *absorb-other* [simp]: $a \neq \text{Some } (Jump\ j) \implies \text{absorb } j\ a = a$

<proof>

lemma *absorb-Some-NoneD*: $\text{absorb } j\ (\text{Some } abr) = \text{None} \implies abr = Jump\ j$

<proof>

lemma *absorb-Some-JumpD*: $\text{absorb } j\ s = \text{Some } (Jump\ j') \implies j' \neq j$

<proof>

full program state

type-synonym

state = $abopt \times st$ — state including abrupton information

translations

$(type) \text{ abopt} \leq (type) \text{ abrupt option}$
 $(type) \text{ state} \leq (type) \text{ abopt} \times st$

abbreviation

$Norm :: st \Rightarrow state$
where $Norm \ s == (None, s)$

abbreviation *(input)*

$abrupt :: state \Rightarrow abopt$
where $abrupt == fst$

abbreviation *(input)*

$store :: state \Rightarrow st$
where $store == snd$

lemma *single-stateE*: $\forall Z. Z = (s::state) \implies False$
 $\langle proof \rangle$

lemma *state-not-single*: $All ((=) (x::state)) \implies R$
 $\langle proof \rangle$

definition

$normal :: state \Rightarrow bool$
where $normal = (\lambda s. abrupt \ s = None)$

lemma *normal-def2* $[simp]$: $normal \ s = (abrupt \ s = None)$
 $\langle proof \rangle$

definition

$heap-free :: nat \Rightarrow state \Rightarrow bool$
where $heap-free \ n = (\lambda s. atleast-free \ (heap \ (store \ s)) \ n)$

lemma *heap-free-def2* $[simp]$: $heap-free \ n \ s = atleast-free \ (heap \ (store \ s)) \ n$
 $\langle proof \rangle$

6 update**definition**

$abupd :: (abopt \Rightarrow abopt) \Rightarrow state \Rightarrow state$
where $abupd \ f = map-prod \ f \ id$

definition

$supd :: (st \Rightarrow st) \Rightarrow state \Rightarrow state$
where $supd = map-prod \ id$

lemma *abupd-def2* $[simp]$: $abupd \ f \ (x,s) = (f \ x,s)$
 $\langle proof \rangle$

lemma *abupd-abrupt-if-False* $[simp]$: $\bigwedge s. abupd \ (abrupt-if \ False \ x_0) \ s = s$
 $\langle proof \rangle$

lemma *supd-def2* $[simp]$: $supd \ f \ (x,s) = (x,f \ s)$
 $\langle proof \rangle$

lemma *supd-lupd* $[simp]$:

$\bigwedge s. supd \ (lupd \ vn \ v) \ s = (abrupt \ s, lupd \ vn \ v \ (store \ s))$
 $\langle proof \rangle$

lemma *supd-gupd* [simp]:

$\bigwedge s. \text{supd } (\text{gupd } r \text{ obj}) s = (\text{abrupt } s, \text{gupd } r \text{ obj } (\text{store } s))$
 $\langle \text{proof} \rangle$

lemma *supd-init-obj* [simp]:

$\text{supd } (\text{init-obj } G \text{ oi } r) s = (\text{abrupt } s, \text{init-obj } G \text{ oi } r (\text{store } s))$
 $\langle \text{proof} \rangle$

lemma *abupd-store-invariant* [simp]: $\text{store } (\text{abupd } f s) = \text{store } s$

$\langle \text{proof} \rangle$

lemma *supd-abrupt-invariant* [simp]: $\text{abrupt } (\text{supd } f s) = \text{abrupt } s$

$\langle \text{proof} \rangle$

abbreviation *set-lvars* :: $\text{locals} \Rightarrow \text{state} \Rightarrow \text{state}$

where *set-lvars* $l == \text{supd } (\text{set-locals } l)$

abbreviation *restore-lvars* :: $\text{state} \Rightarrow \text{state} \Rightarrow \text{state}$

where *restore-lvars* $s' s == \text{set-lvars } (\text{locals } (\text{store } s')) s$

lemma *set-set-lvars* [simp]: $\bigwedge s. \text{set-lvars } l (\text{set-lvars } l' s) = \text{set-lvars } l s$

$\langle \text{proof} \rangle$

lemma *set-lvars-id* [simp]: $\bigwedge s. \text{set-lvars } (\text{locals } (\text{store } s)) s = s$

$\langle \text{proof} \rangle$

initialisation test

definition

initd :: $qname \Rightarrow \text{globs} \Rightarrow \text{bool}$

where *initd* $C g = (g (\text{Stat } C) \neq \text{None})$

definition

initd :: $qname \Rightarrow \text{state} \Rightarrow \text{bool}$

where *initd* $C = \text{initd } C \circ \text{globs} \circ \text{store}$

lemma *not-initd-empty* [simp]: $\neg \text{initd } C \text{ Map.empty}$

$\langle \text{proof} \rangle$

lemma *initd-gupdate* [simp]: $\text{initd } C (g(r \mapsto \text{obj})) = (\text{initd } C g \vee r = \text{Stat } C)$

$\langle \text{proof} \rangle$

lemma *initd-init-class-obj* [intro!]: $\text{initd } C (\text{globs } (\text{init-class-obj } G C s))$

$\langle \text{proof} \rangle$

lemma *not-initdD*: $\neg \text{initd } C g \implies g (\text{Stat } C) = \text{None}$

$\langle \text{proof} \rangle$

lemma *initdD*: $\text{initd } C g \implies \exists \text{ obj. } g (\text{Stat } C) = \text{Some obj}$

$\langle \text{proof} \rangle$

lemma *initd-def2* [simp]: $\text{initd } C s = \text{initd } C (\text{globs } (\text{store } s))$

$\langle \text{proof} \rangle$

error-free

definition

error-free :: $\text{state} \Rightarrow \text{bool}$

where $\text{error-free } s = (\neg (\exists \text{ err. abrupt } s = \text{Some } (\text{Error err})))$

lemma $\text{error-free-Norm } [\text{simp}, \text{intro}]: \text{error-free } (\text{Norm } s)$
 $\langle \text{proof} \rangle$

lemma $\text{error-free-normal } [\text{simp}, \text{intro}]: \text{normal } s \implies \text{error-free } s$
 $\langle \text{proof} \rangle$

lemma $\text{error-free-Xcpt } [\text{simp}]: \text{error-free } (\text{Some } (\text{Xcpt } x), s)$
 $\langle \text{proof} \rangle$

lemma $\text{error-free-Jump } [\text{simp}, \text{intro}]: \text{error-free } (\text{Some } (\text{Jump } j), s)$
 $\langle \text{proof} \rangle$

lemma $\text{error-free-Error } [\text{simp}]: \text{error-free } (\text{Some } (\text{Error } e), s) = \text{False}$
 $\langle \text{proof} \rangle$

lemma $\text{error-free-Some } [\text{simp}, \text{intro}]:$
 $\neg (\exists \text{ err. } x = \text{Error err}) \implies \text{error-free } ((\text{Some } x), s)$
 $\langle \text{proof} \rangle$

lemma $\text{error-free-abupd-absorb } [\text{simp}, \text{intro}]:$
 $\text{error-free } s \implies \text{error-free } (\text{abupd } (\text{absorb } j) s)$
 $\langle \text{proof} \rangle$

lemma $\text{error-free-absorb } [\text{simp}, \text{intro}]:$
 $\text{error-free } (a, s) \implies \text{error-free } (\text{absorb } j a, s)$
 $\langle \text{proof} \rangle$

lemma $\text{error-free-abrupt-if } [\text{simp}, \text{intro}]:$
 $\llbracket \text{error-free } s; \neg (\exists \text{ err. } x = \text{Error err}) \rrbracket$
 $\implies \text{error-free } (\text{abupd } (\text{abrupt-if } p (\text{Some } x)) s)$
 $\langle \text{proof} \rangle$

lemma $\text{error-free-abrupt-if1 } [\text{simp}, \text{intro}]:$
 $\llbracket \text{error-free } (a, s); \neg (\exists \text{ err. } x = \text{Error err}) \rrbracket$
 $\implies \text{error-free } (\text{abrupt-if } p (\text{Some } x) a, s)$
 $\langle \text{proof} \rangle$

lemma $\text{error-free-abrupt-if-Xcpt } [\text{simp}, \text{intro}]:$
 $\text{error-free } s$
 $\implies \text{error-free } (\text{abupd } (\text{abrupt-if } p (\text{Some } (\text{Xcpt } x))) s)$
 $\langle \text{proof} \rangle$

lemma $\text{error-free-abrupt-if-Xcpt1 } [\text{simp}, \text{intro}]:$
 $\text{error-free } (a, s)$
 $\implies \text{error-free } (\text{abrupt-if } p (\text{Some } (\text{Xcpt } x)) a, s)$
 $\langle \text{proof} \rangle$

lemma $\text{error-free-abrupt-if-Jump } [\text{simp}, \text{intro}]:$
 $\text{error-free } s$
 $\implies \text{error-free } (\text{abupd } (\text{abrupt-if } p (\text{Some } (\text{Jump } j))) s)$
 $\langle \text{proof} \rangle$

lemma $\text{error-free-abrupt-if-Jump1 } [\text{simp}, \text{intro}]:$
 $\text{error-free } (a, s)$
 $\implies \text{error-free } (\text{abrupt-if } p (\text{Some } (\text{Jump } j)) a, s)$
 $\langle \text{proof} \rangle$

lemma *error-free-raise-if* [*simp,intro*]:
 $\text{error-free } s \implies \text{error-free } (\text{abupd } (\text{raise-if } p \ x) \ s)$
 ⟨proof⟩

lemma *error-free-raise-if1* [*simp,intro*]:
 $\text{error-free } (a,s) \implies \text{error-free } ((\text{raise-if } p \ x \ a), \ s)$
 ⟨proof⟩

lemma *error-free-supd* [*simp,intro*]:
 $\text{error-free } s \implies \text{error-free } (\text{supd } f \ s)$
 ⟨proof⟩

lemma *error-free-supd1* [*simp,intro*]:
 $\text{error-free } (a,s) \implies \text{error-free } (a,f \ s)$
 ⟨proof⟩

lemma *error-free-set-lvars* [*simp,intro*]:
 $\text{error-free } s \implies \text{error-free } ((\text{set-lvars } l) \ s)$
 ⟨proof⟩

lemma *error-free-set-locals* [*simp,intro*]:
 $\text{error-free } (x, \ s) \implies \text{error-free } (x, \ \text{set-locals } l \ s')$
 ⟨proof⟩

end

Chapter 15

Eval

1 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*)) = *undefined*.
 - ++ 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`

type-synonym $vvar = val \times (val \Rightarrow state \Rightarrow state)$

type-synonym $vals = (val, vvar, val\ list)\ sum3$

translations

$(type)\ vvar \leq (type)\ val \times (val \Rightarrow state \Rightarrow state)$

$(type)\ vals \leq (type)\ (val, vvar, val\ list)\ sum3$

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.

abbreviation

dummy-res :: *vals* ($\langle \diamond \rangle$)
where $\diamond == \text{In1 Unit}$

abbreviation (*input*)

val-inj-vals ($\langle \lfloor _ \rfloor_e \rangle 1000$)
where $\lfloor e \rfloor_e == \text{In1 } e$

abbreviation (*input*)

var-inj-vals ($\langle \lfloor _ \rfloor_v \rangle 1000$)
where $\lfloor v \rfloor_v == \text{In2 } v$

abbreviation (*input*)

lst-inj-vals ($\langle \lfloor _ \rfloor_l \rangle 1000$)
where $\lfloor es \rfloor_l == \text{In3 } es$

definition *undefined3* :: ('a1 + 'ar, 'b, 'c) *sum3* \Rightarrow *vals* **where**

undefined3 = *case-sum3* (*In1* \circ *case-sum* ($\lambda x. \text{undefined}$) ($\lambda x. \text{Unit}$))
 $(\lambda x. \text{In2 undefined}) (\lambda x. \text{In3 undefined})$

lemma [*simp*]: *undefined3* (*In1l* *x*) = *In1 undefined*
 $\langle \text{proof} \rangle$

lemma [*simp*]: *undefined3* (*In1r* *x*) = \diamond
 $\langle \text{proof} \rangle$

lemma [*simp*]: *undefined3* (*In2* *x*) = *In2 undefined*
 $\langle \text{proof} \rangle$

lemma [*simp*]: *undefined3* (*In3* *x*) = *In3 undefined*
 $\langle \text{proof} \rangle$

exception throwing and catching

definition

throw :: *val* \Rightarrow *abopt* \Rightarrow *abopt* **where**
throw *a'* *x* = *abrupt-if* *True* (*Some* (*Xcpt* (*Loc* (*the-Addr* *a'*)))) (*np* *a' x*)

lemma *throw-def2*:

throw *a' x* = *abrupt-if* *True* (*Some* (*Xcpt* (*Loc* (*the-Addr* *a'*)))) (*np* *a' x*)
 $\langle \text{proof} \rangle$

definition

fits :: *prog* \Rightarrow *st* \Rightarrow *val* \Rightarrow *ty* \Rightarrow *bool* ($\langle _, \vdash _ \text{fits} _ \rangle [61, 61, 61, 61] 60$)
where $G, s \vdash a' \text{ fits } T = ((\exists rt. T = \text{RefT } rt) \longrightarrow a' = \text{Null} \vee G \vdash \text{obj-ty}(\text{lookup-obj } s \ a') \preceq T)$

lemma *fits-Null* [*simp*]: $G, s \vdash \text{Null fits } T$
 $\langle \text{proof} \rangle$

lemma *fits-Addr-RefT* [*simp*]:

$G, s \vdash \text{Addr } a \text{ fits RefT } t = G \vdash \text{obj-ty}(\text{the}(\text{heap } s \ a)) \preceq \text{RefT } t$
 $\langle \text{proof} \rangle$

lemma *fitsD*: $\bigwedge X. G, s \vdash a' \text{ fits } T \implies (\exists pt. T = \text{PrimT } pt) \vee$

$(\exists t. T = \text{RefT } t) \wedge a' = \text{Null} \vee$
 $(\exists t. T = \text{RefT } t) \wedge a' \neq \text{Null} \wedge G \vdash \text{obj-ty } (\text{lookup-obj } s \ a') \preceq T$
 $\langle \text{proof} \rangle$

definition

$\text{catch} :: \text{prog} \Rightarrow \text{state} \Rightarrow \text{qtname} \Rightarrow \text{bool} \ (\hookleftarrow -, \vdash \text{catch} \rightarrow [61, 61, 61] 60)$ **where**
 $G, s \vdash \text{catch } C = (\exists xc. \text{abrupt } s = \text{Some } (\text{Xcpt } xc) \wedge$
 $G, \text{store } s \vdash \text{Addr } (\text{the-Loc } xc) \text{ fits Class } C)$

lemma catch-Norm $[\text{simp}]$: $\neg G, \text{Norm } s \vdash \text{catch } tn$
 $\langle \text{proof} \rangle$

lemma catch-XcptLoc $[\text{simp}]$:

$G, (\text{Some } (\text{Xcpt } (\text{Loc } a)), s) \vdash \text{catch } C = G, s \vdash \text{Addr } a \text{ fits Class } C$
 $\langle \text{proof} \rangle$

lemma catch-Jump $[\text{simp}]$: $\neg G, (\text{Some } (\text{Jump } j), s) \vdash \text{catch } tn$
 $\langle \text{proof} \rangle$

lemma catch-Error $[\text{simp}]$: $\neg G, (\text{Some } (\text{Error } e), s) \vdash \text{catch } tn$
 $\langle \text{proof} \rangle$

definition

$\text{new-xcpt-var} :: \text{vname} \Rightarrow \text{state} \Rightarrow \text{state}$ **where**
 $\text{new-xcpt-var } vn = (\lambda(x, s). \text{Norm } (\text{lupd}(\text{VName } vn \mapsto \text{Addr } (\text{the-Loc } (\text{the-Xcpt } (\text{the } x)))) s))$

lemma new-xcpt-var-def2 $[\text{simp}]$:

$\text{new-xcpt-var } vn \ (x, s) =$
 $\text{Norm } (\text{lupd}(\text{VName } vn \mapsto \text{Addr } (\text{the-Loc } (\text{the-Xcpt } (\text{the } x)))) s)$
 $\langle \text{proof} \rangle$

misc**definition**

$\text{assign} :: ('a \Rightarrow \text{state} \Rightarrow \text{state}) \Rightarrow 'a \Rightarrow \text{state} \Rightarrow \text{state}$ **where**
 $\text{assign } f \ v = (\lambda(x, s). \text{let } (x', s') = (\text{if } x = \text{None} \text{ then } f \ v \text{ else } \text{id}) \ (x, s)$
 $\text{in } (x', \text{if } x' = \text{None} \text{ then } s' \text{ else } s))$

lemma assign-Norm-Norm $[\text{simp}]$:

$f \ v \ (\text{Norm } s) = \text{Norm } s' \implies \text{assign } f \ v \ (\text{Norm } s) = \text{Norm } s'$
 $\langle \text{proof} \rangle$

lemma assign-Norm-Some $[\text{simp}]$:

$\llbracket \text{abrupt } (f \ v \ (\text{Norm } s)) = \text{Some } y \rrbracket$
 $\implies \text{assign } f \ v \ (\text{Norm } s) = (\text{Some } y, s)$
 $\langle \text{proof} \rangle$

lemma assign-Some $[\text{simp}]$:

$\text{assign } f \ v \ (\text{Some } x, s) = (\text{Some } x, s)$
 $\langle \text{proof} \rangle$

lemma assign-Some1 $[\text{simp}]$: $\neg \text{normal } s \implies \text{assign } f \ v \ s = s$

$\langle \text{proof} \rangle$

lemma *assign-supd* [simp]:
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]:
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⟩

definition

init-comp-ty :: $ty \Rightarrow stmt$
where *init-comp-ty* $T = (\text{if } (\exists C. T = \text{Class } C) \text{ then } \text{Init } (\text{the-Class } T) \text{ else } \text{Skip})$

lemma *init-comp-ty-PrimT* [simp]: *init-comp-ty* ($\text{PrimT } pt$) = *Skip*
 ⟨proof⟩

definition

invocation-class :: $inv\text{-mode} \Rightarrow st \Rightarrow val \Rightarrow ref\text{-ty} \Rightarrow qname$ **where**
invocation-class $m \ s \ a' \ statT =$
 (case m of
 $\text{Static} \Rightarrow \text{if } (\exists statC. statT = \text{ClassT } statC)$
 then $\text{the-Class } (\text{RefT } statT)$
 else Object
 $\text{SuperM} \Rightarrow \text{the-Class } (\text{RefT } statT)$
 $\text{IntVir} \Rightarrow \text{obj-class } (\text{lookup-obj } s \ a')$)

definition

invocation-declclass :: $prog \Rightarrow inv\text{-mode} \Rightarrow st \Rightarrow val \Rightarrow ref\text{-ty} \Rightarrow sig \Rightarrow qname$ **where**
invocation-declclass $G \ m \ s \ a' \ statT \ sig =$
 $\text{declclass } (\text{the } (\text{dynlookup } G \ statT$
 $(\text{invocation-class } m \ s \ a' \ statT)$
 $\text{sig}))$

lemma *invocation-class-IntVir* [simp]:
invocation-class $\text{IntVir } s \ a' \ statT = \text{obj-class } (\text{lookup-obj } s \ a')$
 ⟨proof⟩

lemma *dynclass-SuperM* [simp]:
invocation-class $\text{SuperM } s \ a' \ statT = \text{the-Class } (\text{RefT } statT)$
 ⟨proof⟩

lemma *invocation-class-Static* [simp]:
invocation-class $\text{Static } s \ a' \ statT = (\text{if } (\exists statC. statT = \text{ClassT } statC)$
 then $\text{the-Class } (\text{RefT } statT)$
 else Object)
 ⟨proof⟩

definition

init-lvars :: $prog \Rightarrow qname \Rightarrow sig \Rightarrow inv\text{-mode} \Rightarrow val \Rightarrow val \text{ list} \Rightarrow state \Rightarrow state$
where
init-lvars $G \ C \ sig \ mode \ a' \ pvs =$
 ($\lambda(x, s).$
 let $m = \text{methd } (\text{the } (\text{methd } G \ C \ sig));$
 $l = \lambda \ k.$
 (case k of

```

      EName e
      ⇒ (case e of
          VName v ⇒ (Map.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
          VName v
          ⇒ (Map.empty ((pars (methd (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⟩

```

definition

```

body :: prog ⇒ qname ⇒ sig ⇒ expr where
body G C sig =
  (let m = the (methd G C sig)
   in Body (declclass m) (stmt (mbody (methd m))))

```

lemma *body-def2*: — better suited for simplification

```

body G C sig = Body (declclass (the (methd G C sig)))
  (stmt (mbody (methd (the (methd G C sig)))))
  ⟨proof⟩

```

variables

definition

```

lvar :: lname ⇒ st ⇒ vvar
where lvar vn s = (the (locals s vn), λv. supd (lupd(vn↦v)))

```

definition

```

fvar :: qname ⇒ bool ⇒ vname ⇒ val ⇒ state ⇒ vvar × state where
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))

```

definition

```

avar :: prog ⇒ val ⇒ val ⇒ state ⇒ vvar × state where
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)

```

$$\text{ArrStore } x$$

$$, \text{upd-gobj } \text{oref } n \ v \ s))$$

$$\text{in } ((\text{the } (cs \ n), f), \text{abupd } (\text{raise-if } (\neg i \text{ in-bounds } k) \text{ IndOutBound } \circ \text{np } a') \ s))$$

lemma *fvar-def2*: — better suited for simplification

fvar $C \text{ stat } fn \ a' \ s =$
 $((\text{the}$
 $(\text{values}$
 $(\text{the } (\text{globs } (\text{store } s) \text{ (if stat then Stat } C \text{ else Heap } (\text{the-Addr } a'))))$
 $(\text{Inl } (fn, C)))$
 $, (\lambda v. \text{supd } (\text{upd-gobj } (\text{if stat then Stat } C \text{ else Heap } (\text{the-Addr } a'))$
 $(\text{Inl } (fn, C))$
 $v)))$
 $, \text{abupd } (\text{if stat then id else np } a') \ s)$

$\langle \text{proof} \rangle$

lemma *avar-def2*: — better suited for simplification

avar $G \ i' \ a' \ s =$
 $((\text{the } ((\text{snd}(\text{snd}(\text{the-Arr } (\text{globs } (\text{store } s) \text{ (Heap } (\text{the-Addr } a'))))))$
 $(\text{Inr } (\text{the-Intg } i'))$
 $, (\lambda v \ (x, s'). \text{ (raise-if } (\neg G, s \vdash v \text{ fits } (\text{fst}(\text{the-Arr } (\text{globs } (\text{store } s)$
 $(\text{Heap } (\text{the-Addr } a'))))))$
 $\text{ArrStore } x$
 $, \text{upd-gobj } (\text{Heap } (\text{the-Addr } a'))$
 $(\text{Inr } (\text{the-Intg } i')) \ v \ s'))$
 $, \text{abupd } (\text{raise-if } (\neg (\text{the-Intg } i') \text{ in-bounds } (\text{fst}(\text{snd}(\text{the-Arr } (\text{globs } (\text{store } s)$
 $(\text{Heap } (\text{the-Addr } a')))))) \text{ IndOutBound } \circ \text{np } a')$
 $s)$

$\langle \text{proof} \rangle$

definition

check-field-access :: *prog* \Rightarrow *qname* \Rightarrow *qname* \Rightarrow *vname* \Rightarrow *bool* \Rightarrow *val* \Rightarrow *state* \Rightarrow *state* **where**
check-field-access $G \ \text{accC} \ \text{statDeclC} \ fn \ \text{stat} \ a' \ s =$
 $(\text{let } \text{oref} = \text{if stat then Stat statDeclC}$
 $\text{else Heap } (\text{the-Addr } a');$
 $\text{dynC} = \text{case } \text{oref} \text{ of}$
 $\text{Heap } a \Rightarrow \text{obj-class } (\text{the } (\text{globs } (\text{store } s) \ \text{oref}))$
 $| \text{Stat } C \Rightarrow C;$
 $f = (\text{the } (\text{table-of } (\text{DeclConcepts.fields } G \ \text{dynC}) \ (fn, \text{statDeclC})))$
 $\text{in } \text{abupd}$
 $(\text{error-if } (\neg G \vdash \text{Field } fn \ (\text{statDeclC}, f) \text{ in } \text{dynC} \ \text{dyn-accessible-from } \text{accC})$
 $\text{AccessViolation})$
 $s)$

definition

check-method-access :: *prog* \Rightarrow *qname* \Rightarrow *ref-ty* \Rightarrow *inv-mode* \Rightarrow *sig* \Rightarrow *val* \Rightarrow *state* \Rightarrow *state* **where**
check-method-access $G \ \text{accC} \ \text{statT} \ \text{mode} \ \text{sig} \ a' \ s =$
 $(\text{let } \text{invC} = \text{invocation-class } \text{mode } (\text{store } s) \ a' \ \text{statT};$
 $\text{dynM} = \text{the } (\text{dynlookup } G \ \text{statT} \ \text{invC} \ \text{sig})$
 $\text{in } \text{abupd}$
 $(\text{error-if } (\neg G \vdash \text{Methd } \text{sig} \ \text{dynM} \text{ in } \text{invC} \ \text{dyn-accessible-from } \text{accC})$
 $\text{AccessViolation})$
 $s)$

evaluation judgments

inductive

halloc :: [*prog, state, obj-tag, loc, state*] \Rightarrow *bool* ($\vdash - \text{halloc} \dashv \rightarrow - \rightarrow [61, 61, 61, 61, 61] 60$) **for** $G :: \text{prog}$

where — allocating objects on the heap, cf. 12.5

Abrupt:

$G \vdash (\text{Some } x, s) \text{ --halloc } oi \rightarrow \text{undefined} \rightarrow (\text{Some } x, s)$

| *New:* $\llbracket \text{new-Addr } (\text{heap } s) = \text{Some } a;$
 $(x, oi') = (\text{if atleast-free } (\text{heap } s) (\text{Suc } (\text{Suc } 0)) \text{ then } (\text{None}, oi)$
 $\text{else } (\text{Some } (\text{Xcpt } (\text{Loc } a)), \text{CInst } (\text{SXcpt OutOfMemory}))) \rrbracket$
 \implies
 $G \vdash \text{Norm } s \text{ --halloc } oi \rightarrow a \rightarrow (x, \text{init-obj } G \text{ } oi' (\text{Heap } a) \text{ } s)$

inductive *sxalloc* :: $[\text{prog}, \text{state}, \text{state}] \Rightarrow \text{bool}$ ($\langle \vdash - \text{ --sxalloc} \rightarrow - \rangle [61, 61, 61] 60$) **for** $G :: \text{prog}$
where — allocating exception objects for standard exceptions (other than OutOfMemory)

Norm: $G \vdash \text{Norm} \quad s \text{ --sxalloc} \rightarrow \text{Norm} \quad s$

| *Jmp:* $G \vdash (\text{Some } (\text{Jump } j), s) \text{ --sxalloc} \rightarrow (\text{Some } (\text{Jump } j), s)$

| *Error:* $G \vdash (\text{Some } (\text{Error } e), s) \text{ --sxalloc} \rightarrow (\text{Some } (\text{Error } e), s)$

| *XcptL:* $G \vdash (\text{Some } (\text{Xcpt } (\text{Loc } a)), s) \text{ --sxalloc} \rightarrow (\text{Some } (\text{Xcpt } (\text{Loc } a)), s)$

| *SXcpt:* $\llbracket G \vdash \text{Norm } s0 \text{ --halloc } (\text{CInst } (\text{SXcpt } xn)) \rightarrow a \rightarrow (x, s1) \rrbracket \implies$
 $G \vdash (\text{Some } (\text{Xcpt } (\text{Std } xn)), s0) \text{ --sxalloc} \rightarrow (\text{Some } (\text{Xcpt } (\text{Loc } a)), s1)$

inductive

eval :: $[\text{prog}, \text{state}, \text{term}, \text{vals}, \text{state}] \Rightarrow \text{bool}$ ($\langle \vdash - \text{ --eval} \rightarrow '(-, -)' \rangle [61, 61, 80, 0, 0] 60$)
and *exec* :: $[\text{prog}, \text{state}, \text{stmt}, \text{state}] \Rightarrow \text{bool}$ ($\langle \vdash - \text{ --exec} \rightarrow - \rangle [61, 61, 65, \quad 61] 60$)
and *evar* :: $[\text{prog}, \text{state}, \text{var}, \text{vvar}, \text{state}] \Rightarrow \text{bool}$ ($\langle \vdash - \text{ --eval} \rightarrow - \rangle [61, 61, 90, 61, 61] 60$)
and *eval'* :: $[\text{prog}, \text{state}, \text{expr}, \text{val}, \text{state}] \Rightarrow \text{bool}$ ($\langle \vdash - \text{ --eval} \rightarrow - \rangle [61, 61, 80, 61, 61] 60$)
and *evals* :: $[\text{prog}, \text{state}, \text{expr list}, \text{val list}, \text{state}] \Rightarrow \text{bool}$ ($\langle \vdash - \text{ --evals} \rightarrow - \rangle [61, 61, 61, 61, 61] 60$)

for $G :: \text{prog}$

where

$G \vdash s \text{ --c} \rightarrow s' \equiv G \vdash s \text{ --In1r } c \rightarrow (\Diamond, s')$
| $G \vdash s \text{ --e} \rightarrow v \rightarrow s' \equiv G \vdash s \text{ --In1l } e \rightarrow (\text{In1 } v, s')$
| $G \vdash s \text{ --e} \rightarrow vf \rightarrow s' \equiv G \vdash s \text{ --In2 } e \rightarrow (\text{In2 } vf, s')$
| $G \vdash s \text{ --e} \rightarrow v \rightarrow s' \equiv G \vdash s \text{ --In3 } e \rightarrow (\text{In3 } v, s')$

— propagation of abrupt completion

— cf. 14.1, 15.5

| *Abrupt:*

$G \vdash (\text{Some } xc, s) \text{ --t} \rightarrow (\text{undefined3 } t, (\text{Some } xc, s))$

— execution of statements

— cf. 14.5

| *Skip:* $G \vdash \text{Norm } s \text{ --Skip} \rightarrow \text{Norm } s$

— cf. 14.7

| *Expr:* $\llbracket G \vdash \text{Norm } s0 \text{ --e} \rightarrow v \rightarrow s1 \rrbracket \implies$
 $G \vdash \text{Norm } s0 \text{ --Expr } e \rightarrow s1$

| *Lab:* $\llbracket G \vdash \text{Norm } s0 \text{ --c} \rightarrow s1 \rrbracket \implies$
 $G \vdash \text{Norm } s0 \text{ --l. } c \rightarrow \text{abupd } (\text{absorb } l) \text{ } s1$

- cf. 14.2
- | *Comp*: $\llbracket G \vdash \text{Norm } s0 - c1 \rightarrow s1; \\ G \vdash s1 - c2 \rightarrow s2 \rrbracket \Longrightarrow \\ G \vdash \text{Norm } s0 - c1;; c2 \rightarrow s2$
- cf. 14.8.2
- | *If*: $\llbracket G \vdash \text{Norm } s0 - e \rightarrow b \rightarrow s1; \\ G \vdash s1 - (\text{if the-Bool } b \text{ then } c1 \text{ else } c2) \rightarrow s2 \rrbracket \Longrightarrow \\ G \vdash \text{Norm } s0 - \text{If}(e) \ c1 \ \text{Else } c2 \rightarrow s2$
- 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*: $\llbracket G \vdash \text{Norm } s0 - e \rightarrow b \rightarrow s1; \\ \text{if the-Bool } b \\ \text{then } (G \vdash s1 - c \rightarrow s2 \wedge \\ G \vdash (\text{abupd } (\text{absorb } (\text{Cont } l)) \ s2) - l \cdot \text{While}(e) \ c \rightarrow s3) \\ \text{else } s3 = s1 \rrbracket \Longrightarrow \\ G \vdash \text{Norm } s0 - l \cdot \text{While}(e) \ c \rightarrow s3$
- | *Jmp*: $G \vdash \text{Norm } s - \text{Jmp } j \rightarrow (\text{Some } (\text{Jump } j), s)$
- cf. 14.16
- | *Throw*: $\llbracket G \vdash \text{Norm } s0 - e \rightarrow a' \rightarrow s1 \rrbracket \Longrightarrow \\ G \vdash \text{Norm } s0 - \text{Throw } e \rightarrow \text{abupd } (\text{throw } a') \ s1$
- cf. 14.18.1
- | *Try*: $\llbracket G \vdash \text{Norm } s0 - c1 \rightarrow s1; G \vdash s1 - \text{xalloc} \rightarrow s2; \\ \text{if } G, s2 \vdash \text{catch } C \text{ then } G \vdash \text{new-xcpt-var } vn \ s2 - c2 \rightarrow s3 \text{ else } s3 = s2 \rrbracket \Longrightarrow \\ G \vdash \text{Norm } s0 - \text{Try } c1 \ \text{Catch}(C \ vn) \ c2 \rightarrow s3$
- cf. 14.18.2
- | *Fin*: $\llbracket G \vdash \text{Norm } s0 - c1 \rightarrow (x1, s1); \\ G \vdash \text{Norm } s1 - c2 \rightarrow s2; \\ s3 = (\text{if } (\exists \text{ err. } x1 = \text{Some } (\text{Error } \text{err})) \\ \text{then } (x1, s1) \\ \text{else } \text{abupd } (\text{abrupt-if } (x1 \neq \text{None}) \ x1) \ s2) \rrbracket \\ \Longrightarrow \\ G \vdash \text{Norm } s0 - c1 \ \text{Finally } c2 \rightarrow s3$
- cf. 12.4.2, 8.5
- | *Init*: $\llbracket \text{the } (\text{class } G \ C) = c; \\ \text{if } \text{inited } C \ (\text{globs } s0) \text{ then } s3 = \text{Norm } s0 \\ \text{else } (G \vdash \text{Norm } (\text{init-class-obj } G \ C \ s0) \\ - (\text{if } C = \text{Object then Skip else Init } (\text{super } c)) \rightarrow s1 \wedge \\ G \vdash \text{set-lvars Map.empty } s1 - \text{init } c \rightarrow s2 \wedge s3 = \text{restore-lvars } s1 \ s2) \rrbracket \\ \Longrightarrow \\ G \vdash \text{Norm } s0 - \text{Init } C \rightarrow s3$

— This class initialisation rule is a little bit inaccurate. Look at the exact sequence: (1) The current class object (the static fields) are initialised (*init-class-obj*), (2) the superclasses are initialised, (3) the static initialiser of the current class is invoked. More precisely we should expect another ordering, namely 2 1 3. But we can't just naively toggle 1 and 2. By calling *init-class-obj* before initialising the superclasses, we also implicitly record that we have started to initialise the current class (by setting an value for the class object). This becomes crucial for the completeness proof of the axiomatic semantics *AxCompl.thy*. Static initialisation requires an induction on the number of classes not yet initialised (or to be more precise, classes where the initialisation has not yet begun). So we could first assign a dummy value to the class before superclass initialisation and afterwards set the correct values. But as long as we don't take memory overflow into account when allocating class objects, we can leave things as they are for convenience.

— evaluation of expressions

— cf. 15.8.1, 12.4.1

| *NewC*: $\llbracket G \vdash \text{Norm } s0 \text{ --Init } C \rightarrow s1; \\ G \vdash s1 \text{ --halloc } (C \text{Inst } C) \succ a \rightarrow s2 \rrbracket \implies \\ G \vdash \text{Norm } s0 \text{ --NewC } C \text{ --} \succ \text{Addr } a \rightarrow s2$

— cf. 15.9.1, 12.4.1

| *NewA*: $\llbracket G \vdash \text{Norm } s0 \text{ --init-comp-ty } T \rightarrow s1; G \vdash s1 \text{ --e--} \succ i' \rightarrow s2; \\ G \vdash \text{abupd } (\text{check-neg } i') s2 \text{ --halloc } (\text{Arr } T (\text{the-Intg } i')) \succ a \rightarrow s3 \rrbracket \implies \\ G \vdash \text{Norm } s0 \text{ --New } T[e] \text{ --} \succ \text{Addr } a \rightarrow s3$

— cf. 15.15

| *Cast*: $\llbracket G \vdash \text{Norm } s0 \text{ --e--} \succ v \rightarrow s1; \\ s2 = \text{abupd } (\text{raise-if } (\neg G, \text{store } s1 \vdash v \text{ fits } T) \text{ ClassCast}) s1 \rrbracket \implies \\ G \vdash \text{Norm } s0 \text{ --Cast } T \text{ --e--} \succ v \rightarrow s2$

— cf. 15.19.2

| *Inst*: $\llbracket G \vdash \text{Norm } s0 \text{ --e--} \succ v \rightarrow s1; \\ b = (v \neq \text{Null} \wedge G, \text{store } s1 \vdash v \text{ fits RefT } T) \rrbracket \implies \\ G \vdash \text{Norm } s0 \text{ --e } \text{InstOf } T \text{ --} \succ \text{Bool } b \rightarrow s1$

— cf. 15.7.1

| *Lit*: $G \vdash \text{Norm } s \text{ --Lit } v \text{ --} \succ v \rightarrow \text{Norm } s$

| *UnOp*: $\llbracket G \vdash \text{Norm } s0 \text{ --e--} \succ v \rightarrow s1 \rrbracket \\ \implies G \vdash \text{Norm } s0 \text{ --UnOp } \text{unop } e \text{ --} \succ (\text{eval-unop } \text{unop } v) \rightarrow s1$

| *BinOp*: $\llbracket G \vdash \text{Norm } s0 \text{ --e1--} \succ v1 \rightarrow s1; \\ G \vdash s1 \text{ --(if need-second-arg binop } v1 \text{ then } (In1l \ e2) \text{ else } (In1r \text{ Skip})) \\ \succ \rightarrow (In1 \ v2, s2) \\ \rrbracket \\ \implies G \vdash \text{Norm } s0 \text{ --BinOp } \text{binop } e1 \ e2 \text{ --} \succ (\text{eval-binop } \text{binop } v1 \ v2) \rightarrow s2$

— cf. 15.10.2

| *Super*: $G \vdash \text{Norm } s \text{ --Super--} \succ \text{val-this } s \rightarrow \text{Norm } s$

— cf. 15.2

| *Acc*: $\llbracket G \vdash \text{Norm } s0 \text{ --va==} \succ (v, f) \rightarrow s1 \rrbracket \implies \\ G \vdash \text{Norm } s0 \text{ --Acc } va \text{ --} \succ v \rightarrow s1$

— cf. 15.25.1

| *Ass*: $\llbracket G \vdash \text{Norm } s0 \text{ --va==} \succ (w, f) \rightarrow s1; \\ G \vdash s1 \text{ --e--} \succ v \rightarrow s2 \rrbracket \implies \\ G \vdash \text{Norm } s0 \text{ --va:=e--} \succ v \rightarrow \text{assign } f \ v \ s2$

— cf. 15.24

| *Cond*: $\llbracket G \vdash \text{Norm } s0 \text{ --e0--} \succ b \rightarrow s1; \\ G \vdash s1 \text{ --(if the-Bool } b \text{ then } e1 \text{ else } e2) \text{ --} \succ v \rightarrow s2 \rrbracket \implies \\ G \vdash \text{Norm } s0 \text{ --e0 ? } e1 : e2 \text{ --} \succ v \rightarrow s2$

— 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

| *Call*:

$$\begin{aligned} & \llbracket G \vdash \text{Norm } s0 - e \rightarrow a' \rightarrow s1; G \vdash s1 - \text{args} \rightarrow vs \rightarrow s2; \\ & D = \text{invocation-declclass } G \text{ mode } (store\ s2) \ a' \ \text{statT} \ (\llbracket name=mn, parTs=pTs \rrbracket); \\ & s3 = \text{init-lvars } G \ D \ (\llbracket name=mn, parTs=pTs \rrbracket) \ \text{mode } a' \ vs \ s2; \\ & s3' = \text{check-method-access } G \ accC \ \text{statT} \ \text{mode} \ (\llbracket name=mn, parTs=pTs \rrbracket) \ a' \ s3; \\ & G \vdash s3' - \text{Methd } D \ (\llbracket name=mn, parTs=pTs \rrbracket) \rightarrow v \rightarrow s4 \rrbracket \\ & \implies \\ & G \vdash \text{Norm } s0 - \{accC, statT, mode\} e.mn(\{pTs\}args) \rightarrow v \rightarrow (\text{restore-lvars } s2 \ s4) \end{aligned}$$

— 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.

$$\begin{aligned} | \text{Methd:} \quad & \llbracket G \vdash \text{Norm } s0 - \text{body } G \ D \ \text{sig} \rightarrow v \rightarrow s1 \rrbracket \implies \\ & G \vdash \text{Norm } s0 - \text{Methd } D \ \text{sig} \rightarrow v \rightarrow s1 \end{aligned}$$

$$\begin{aligned} | \text{Body:} \quad & \llbracket G \vdash \text{Norm } s0 - \text{Init } D \rightarrow s1; G \vdash s1 - c \rightarrow s2; \\ & s3 = (\text{if } (\exists \ l. \ \text{abrupt } s2 = \text{Some } (\text{Jump } (\text{Break } l))) \vee \\ & \quad \text{abrupt } s2 = \text{Some } (\text{Jump } (\text{Cont } l))) \\ & \quad \text{then } \text{abupd } (\lambda \ x. \ \text{Some } (\text{Error CrossMethodJump})) \ s2 \\ & \quad \text{else } s2) \rrbracket \implies \\ & G \vdash \text{Norm } s0 - \text{Body } D \ c \rightarrow \text{the } (locals \ (store\ s2) \ \text{Result}) \\ & \rightarrow \text{abupd } (\text{absorb } \text{Ret}) \ s3 \end{aligned}$$

— 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

$$| \text{LVar: } G \vdash \text{Norm } s - \text{LVar } vn \rightarrow \text{lvar } vn \ s \rightarrow \text{Norm } s$$

— cf. 15.10.1, 12.4.1

$$\begin{aligned} | \text{FVar:} \quad & \llbracket G \vdash \text{Norm } s0 - \text{Init } \text{statDeclC} \rightarrow s1; G \vdash s1 - e \rightarrow a \rightarrow s2; \\ & (v, s2') = \text{fvar } \text{statDeclC} \ \text{stat} \ \text{fn } a \ s2; \\ & s3 = \text{check-field-access } G \ accC \ \text{statDeclC} \ \text{fn } \text{stat } a \ s2' \rrbracket \implies \\ & G \vdash \text{Norm } s0 - \{accC, \text{statDeclC}, \text{stat}\} e..fn \rightarrow v \rightarrow s3 \end{aligned}$$

— 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

— cf. 15.12.1, 15.25.1

$$\begin{aligned} | \text{AVar:} \quad & \llbracket G \vdash \text{Norm } s0 - e1 \rightarrow a \rightarrow s1; G \vdash s1 - e2 \rightarrow i \rightarrow s2; \\ & (v, s2') = \text{avar } G \ i \ a \ s2 \rrbracket \implies \\ & G \vdash \text{Norm } s0 - e1.[e2] \rightarrow v \rightarrow s2' \end{aligned}$$

— evaluation of expression lists

— cf. 15.11.4.2

| *Nil*:

$$G \vdash \text{Norm } s0 - [] \rightarrow [] \rightarrow \text{Norm } s0$$

— cf. 15.6.4

$$\begin{aligned} | \text{Cons:} \quad & \llbracket G \vdash \text{Norm } s0 - e \rightarrow v \rightarrow s1; \\ & G \vdash s1 - es \rightarrow vs \rightarrow s2 \rrbracket \implies \\ & G \vdash \text{Norm } s0 - e \# es \rightarrow v \# vs \rightarrow s2 \end{aligned}$$

$\langle ML \rangle$

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

inductive-cases *halloc-elim-cases*:

$G \vdash (\text{Some } xc, s) - \text{halloc } oi \succ a \rightarrow s'$
 $G \vdash (\text{Norm } s) - \text{halloc } oi \succ a \rightarrow s'$

inductive-cases *sxalloc-elim-cases*:

$G \vdash \text{Norm } s - \text{sxalloc} \rightarrow s'$
 $G \vdash (\text{Some } (\text{Jump } j), s) - \text{sxalloc} \rightarrow s'$
 $G \vdash (\text{Some } (\text{Error } e), s) - \text{sxalloc} \rightarrow s'$
 $G \vdash (\text{Some } (\text{Xcpt } (\text{Loc } a)), s) - \text{sxalloc} \rightarrow s'$
 $G \vdash (\text{Some } (\text{Xcpt } (\text{Std } xn)), s) - \text{sxalloc} \rightarrow s'$

inductive-cases *sxalloc-cases*: $G \vdash s - \text{sxalloc} \rightarrow s'$

lemma *sxalloc-elim-cases2*: $\llbracket G \vdash s - \text{sxalloc} \rightarrow s' \rrbracket$;

$\bigwedge s. \llbracket s' = \text{Norm } s \rrbracket \implies P$;
 $\bigwedge j s. \llbracket s' = (\text{Some } (\text{Jump } j), s) \rrbracket \implies P$;
 $\bigwedge e s. \llbracket s' = (\text{Some } (\text{Error } e), s) \rrbracket \implies P$;
 $\bigwedge a s. \llbracket s' = (\text{Some } (\text{Xcpt } (\text{Loc } a)), s) \rrbracket \implies P$
 $\rrbracket \implies P$

$\langle \text{proof} \rangle$

declare *not-None-eq* [*simp del*]

declare *split-paired-All* [*simp del*] *split-paired-Ex* [*simp del*]

$\langle ML \rangle$

inductive-cases *eval-cases*: $G \vdash s - t \succ \rightarrow (v, s')$

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

$G \vdash (\text{Some } xc, s) - t$	$\succ \rightarrow (v, s')$
$G \vdash \text{Norm } s - \text{In1r Skip}$	$\succ \rightarrow (x, s')$
$G \vdash \text{Norm } s - \text{In1r } (\text{Jmp } j)$	$\succ \rightarrow (x, s')$
$G \vdash \text{Norm } s - \text{In1r } (l \cdot c)$	$\succ \rightarrow (x, s')$
$G \vdash \text{Norm } s - \text{In3 } (\llbracket \rrbracket)$	$\succ \rightarrow (v, s')$
$G \vdash \text{Norm } s - \text{In3 } (e \# es)$	$\succ \rightarrow (v, s')$
$G \vdash \text{Norm } s - \text{In1l } (\text{Lit } w)$	$\succ \rightarrow (v, s')$
$G \vdash \text{Norm } s - \text{In1l } (\text{UnOp } unop \ e)$	$\succ \rightarrow (v, s')$
$G \vdash \text{Norm } s - \text{In1l } (\text{BinOp } binop \ e1 \ e2)$	$\succ \rightarrow (v, s')$
$G \vdash \text{Norm } s - \text{In2 } (\text{LVar } vn)$	$\succ \rightarrow (v, s')$
$G \vdash \text{Norm } s - \text{In1l } (\text{Cast } T \ e)$	$\succ \rightarrow (v, s')$
$G \vdash \text{Norm } s - \text{In1l } (e \ \text{InstOf } T)$	$\succ \rightarrow (v, s')$
$G \vdash \text{Norm } s - \text{In1l } (\text{Super})$	$\succ \rightarrow (v, s')$
$G \vdash \text{Norm } s - \text{In1l } (\text{Acc } va)$	$\succ \rightarrow (v, s')$
$G \vdash \text{Norm } s - \text{In1r } (\text{Expr } e)$	$\succ \rightarrow (x, s')$
$G \vdash \text{Norm } s - \text{In1r } (c1;; c2)$	$\succ \rightarrow (x, s')$
$G \vdash \text{Norm } s - \text{In1l } (\text{Methd } C \ \text{sig})$	$\succ \rightarrow (x, s')$
$G \vdash \text{Norm } s - \text{In1l } (\text{Body } D \ c)$	$\succ \rightarrow (x, s')$
$G \vdash \text{Norm } s - \text{In1l } (e0 \ ? \ e1 : e2)$	$\succ \rightarrow (v, s')$
$G \vdash \text{Norm } s - \text{In1r } (\text{If}(e) \ c1 \ \text{Else } c2)$	$\succ \rightarrow (x, s')$
$G \vdash \text{Norm } s - \text{In1r } (l \cdot \text{While}(e) \ c)$	$\succ \rightarrow (x, s')$
$G \vdash \text{Norm } s - \text{In1r } (c1 \ \text{Finally } c2)$	$\succ \rightarrow (x, s')$
$G \vdash \text{Norm } s - \text{In1r } (\text{Throw } e)$	$\succ \rightarrow (x, s')$
$G \vdash \text{Norm } s - \text{In1l } (\text{NewC } C)$	$\succ \rightarrow (v, s')$

$$\begin{array}{ll}
G \vdash \text{Norm } s - \text{In1l } (\text{New } T[e]) & \succ \rightarrow (v, s') \\
G \vdash \text{Norm } s - \text{In1l } (\text{Ass } va \ e) & \succ \rightarrow (v, s') \\
G \vdash \text{Norm } s - \text{In1r } (\text{Try } c1 \ \text{Catch}(tn \ vn) \ c2) & \succ \rightarrow (x, s') \\
G \vdash \text{Norm } s - \text{In2 } (\{accC, statDeclC, stat\}e..fn) & \succ \rightarrow (v, s') \\
G \vdash \text{Norm } s - \text{In2 } (e1.[e2]) & \succ \rightarrow (v, s') \\
G \vdash \text{Norm } s - \text{In1l } (\{accC, statT, mode\}e.mn(\{pT\}p)) & \succ \rightarrow (v, s') \\
G \vdash \text{Norm } s - \text{In1r } (\text{Init } C) & \succ \rightarrow (x, s')
\end{array}$$

declare *not-None-eq* [simp]

declare *split-paired-All* [simp] *split-paired-Ex* [simp]

$\langle ML \rangle$

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

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

lemma *eval-Inj-elim*:

$G \vdash s - t \succ \rightarrow (w, s')$

$\implies \text{case } t \text{ of}$

$\text{In1 } ec \Rightarrow (\text{case } ec \text{ of}$
 $\quad \text{Inl } e \Rightarrow (\exists v. w = \text{In1 } v)$
 $\quad | \text{Inr } c \Rightarrow w = \Diamond)$
 $| \text{In2 } e \Rightarrow (\exists v. w = \text{In2 } v)$
 $| \text{In3 } e \Rightarrow (\exists v. w = \text{In3 } v)$

$\langle \text{proof} \rangle$

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*: $G \vdash s - \text{In1l } t \succ \rightarrow (w, s') = (\exists v. w = \text{In1 } v \wedge G \vdash s - t \succ v \rightarrow s')$

$\langle \text{proof} \rangle$

lemma *eval-var-eq*: $G \vdash s - \text{In2 } t \succ \rightarrow (w, s') = (\exists vf. w = \text{In2 } vf \wedge G \vdash s - t = \succ vf \rightarrow s')$

$\langle \text{proof} \rangle$

lemma *eval-exprs-eq*: $G \vdash s - \text{In3 } t \succ \rightarrow (w, s') = (\exists vs. w = \text{In3 } vs \wedge G \vdash s - t = \succ vs \rightarrow s')$

$\langle \text{proof} \rangle$

lemma *eval-stmt-eq*: $G \vdash s - \text{In1r } t \succ \rightarrow (w, s') = (w = \Diamond \wedge G \vdash s - t \rightarrow s')$

$\langle \text{proof} \rangle$

$\langle ML \rangle$

declare *halloc.Abrupt* [intro!] *eval.Abrupt* [intro!] *AbruptIs* [intro!]

Callee, *InsInitE*, *InsInitV*, *FinA* are only used in smallstep semantics, not in the bigstep semantics. So there is no valid evaluation of these terms

lemma *eval-Callee*: $G \vdash \text{Norm } s - \text{Callee } l \ e - \succ v \rightarrow s' = \text{False}$

$\langle \text{proof} \rangle$

lemma *eval-InsInitE*: $G \vdash \text{Norm } s - \text{InsInitE } c \ e - \succ v \rightarrow s' = \text{False}$

$\langle \text{proof} \rangle$

lemma *eval-InsInitV*: $G \vdash \text{Norm } s - \text{InsInitV } c \ w = \succ v \rightarrow s' = \text{False}$

$\langle \text{proof} \rangle$

lemma *eval-FinA*: $G \vdash \text{Norm } s - \text{FinA } a \ c \rightarrow s' = \text{False}$

$\langle \text{proof} \rangle$

lemma *eval-no-abrupt-lemma*:

$\bigwedge s s'. G \vdash s - t \rightarrow (w, s') \implies \text{normal } s' \longrightarrow \text{normal } s$
 $\langle \text{proof} \rangle$

lemma *eval-no-abrupt*:

$G \vdash (x, s) - t \rightarrow (w, \text{Norm } s') =$
 $(x = \text{None} \wedge G \vdash \text{Norm } s - t \rightarrow (w, \text{Norm } s'))$
 $\langle \text{proof} \rangle$

$\langle \text{ML} \rangle$

lemma *eval-abrupt-lemma*:

$G \vdash s - t \rightarrow (v, s') \implies \text{abrupt } s = \text{Some } xc \longrightarrow s' = s \wedge v = \text{undefined3 } t$
 $\langle \text{proof} \rangle$

lemma *eval-abrupt*:

$G \vdash (\text{Some } xc, s) - t \rightarrow (w, s') =$
 $(s' = (\text{Some } xc, s) \wedge w = \text{undefined3 } t \wedge$
 $G \vdash (\text{Some } xc, s) - t \rightarrow (\text{undefined3 } t, (\text{Some } xc, s)))$
 $\langle \text{proof} \rangle$

$\langle \text{ML} \rangle$

lemma *LitI*: $G \vdash s - \text{Lit } v \rightarrow (\text{if normal } s \text{ then } v \text{ else undefined}) \rightarrow s$
 $\langle \text{proof} \rangle$

lemma *SkipI* [intro!]: $G \vdash s - \text{Skip} \rightarrow s$
 $\langle \text{proof} \rangle$

lemma *ExprI*: $G \vdash s - e \rightarrow v \rightarrow s' \implies G \vdash s - \text{Expr } e \rightarrow s'$
 $\langle \text{proof} \rangle$

lemma *CompI*: $\llbracket G \vdash s - c1 \rightarrow s1; G \vdash s1 - c2 \rightarrow s2 \rrbracket \implies G \vdash s - c1;; c2 \rightarrow s2$
 $\langle \text{proof} \rangle$

lemma *CondI*:

$\bigwedge s1. \llbracket G \vdash s - e \rightarrow b \rightarrow s1; G \vdash s1 - (\text{if the-Bool } b \text{ then } e1 \text{ else } e2) \rightarrow v \rightarrow s2 \rrbracket \implies$
 $G \vdash s - e ? e1 : e2 \rightarrow (\text{if normal } s1 \text{ then } v \text{ else undefined}) \rightarrow s2$
 $\langle \text{proof} \rangle$

lemma *IfI*: $\llbracket G \vdash s - e \rightarrow v \rightarrow s1; G \vdash s1 - (\text{if the-Bool } v \text{ then } c1 \text{ else } c2) \rightarrow s2 \rrbracket$
 $\implies G \vdash s - \text{If}(e) c1 \text{ Else } c2 \rightarrow s2$
 $\langle \text{proof} \rangle$

lemma *MethdI*: $G \vdash s - \text{body } G \ C \ \text{sig} \rightarrow v \rightarrow s'$
 $\implies G \vdash s - \text{Methd } C \ \text{sig} \rightarrow v \rightarrow s'$
 $\langle \text{proof} \rangle$

lemma *eval-Call*:

$\llbracket G \vdash \text{Norm } s0 - e \rightarrow a' \rightarrow s1; G \vdash s1 - ps \dot{\rightarrow} pvs \rightarrow s2;$
 $D = \text{invocation-declclass } G \ \text{mode } (\text{store } s2) \ a' \ \text{statT } (\llbracket \text{name} = mn, \text{parTs} = pTs \rrbracket);$
 $s3 = \text{init-lvars } G \ D \ (\llbracket \text{name} = mn, \text{parTs} = pTs \rrbracket) \ \text{mode } a' \ pvs \ s2;$
 $s3' = \text{check-method-access } G \ \text{accC } \text{statT } \text{mode } (\llbracket \text{name} = mn, \text{parTs} = pTs \rrbracket) \ a' \ s3;$
 $G \vdash s3' - \text{Methd } D \ (\llbracket \text{name} = mn, \text{parTs} = pTs \rrbracket) \rightarrow v \rightarrow s4;$
 $s4' = \text{restore-lvars } s2 \ s4 \rrbracket \implies$
 $G \vdash \text{Norm } s0 - \{\text{accC}, \text{statT}, \text{mode}\} e. mn(\{pTs\} ps) \rightarrow v \rightarrow s4'$
 $\langle \text{proof} \rangle$

lemma *eval-Init*:

$\llbracket \text{if } \text{initd } C \text{ (globs } s0) \text{ then } s3 = \text{Norm } s0$
 $\text{else } G \vdash \text{Norm } (\text{init-class-obj } G \ C \ s0)$
 $\quad \neg(\text{if } C = \text{Object then Skip else Init } (\text{super } (\text{the } (\text{class } G \ C)))) \rightarrow s1 \wedge$
 $\quad G \vdash \text{set-lvars Map.empty } s1 \neg(\text{init } (\text{the } (\text{class } G \ C))) \rightarrow s2 \wedge$
 $\quad s3 = \text{restore-lvars } s1 \ s2 \rrbracket \implies$
 $G \vdash \text{Norm } s0 \neg \text{Init } C \rightarrow s3$
 $\langle \text{proof} \rangle$

lemma *init-done*: $\text{initd } C \ s \implies G \vdash s \neg \text{Init } C \rightarrow s$
 $\langle \text{proof} \rangle$

lemma *eval-StatRef*:
 $G \vdash s \neg \text{StatRef } rt \neg (\text{if abrupt } s = \text{None then Null else undefined}) \rightarrow s$
 $\langle \text{proof} \rangle$

lemma *SkipD* [*dest!*]: $G \vdash s \neg \text{Skip} \rightarrow s' \implies s' = s$
 $\langle \text{proof} \rangle$

lemma *Skip-eq* [*simp*]: $G \vdash s \neg \text{Skip} \rightarrow s' = (s = s')$
 $\langle \text{proof} \rangle$

lemma *init-retains-locals* [*rule-format (no-asm)*]: $G \vdash s \neg t \rightarrow (w, s') \implies$
 $(\forall C. t = \text{In1r } (\text{Init } C) \longrightarrow \text{locals } (\text{store } s) = \text{locals } (\text{store } s'))$
 $\langle \text{proof} \rangle$

lemma *halloc-xcpt* [*dest!*]:
 $\bigwedge s'. G \vdash (\text{Some } xc, s) \neg \text{halloc } oi \rightarrow a \rightarrow s' \implies s' = (\text{Some } xc, s)$
 $\langle \text{proof} \rangle$

lemma *eval-Methd*:
 $G \vdash s \neg \text{In1l}(\text{body } G \ C \ sig) \rightarrow (w, s')$
 $\implies G \vdash s \neg \text{In1l}(\text{Methd } C \ sig) \rightarrow (w, s')$
 $\langle \text{proof} \rangle$

lemma *eval-Body*: $\llbracket G \vdash \text{Norm } s0 \neg \text{Init } D \rightarrow s1; G \vdash s1 \neg c \rightarrow s2;$
 $\text{res} = \text{the } (\text{locals } (\text{store } s2) \ \text{Result});$
 $s3 = (\text{if } (\exists l. \text{abrupt } s2 = \text{Some } (\text{Jump } (\text{Break } l))) \vee$
 $\quad \text{abrupt } s2 = \text{Some } (\text{Jump } (\text{Cont } l)))$
 $\text{then abupd } (\lambda x. \text{Some } (\text{Error CrossMethodJump})) \ s2$
 $\text{else } s2);$
 $s4 = \text{abupd } (\text{absorb Ret}) \ s3 \rrbracket \implies$
 $G \vdash \text{Norm } s0 \neg \text{Body } D \ c \neg \text{res} \rightarrow s4$
 $\langle \text{proof} \rangle$

lemma *eval-binop-arg2-indep*:
 $\neg \text{need-second-arg binop } v1 \implies \text{eval-binop binop } v1 \ x = \text{eval-binop binop } v1 \ y$
 $\langle \text{proof} \rangle$

lemma *eval-BinOp-arg2-indepI*:
assumes *eval-e1*: $G \vdash \text{Norm } s0 \neg e1 \neg v1 \rightarrow s1$ **and**
 $\text{no-need: } \neg \text{need-second-arg binop } v1$
shows $G \vdash \text{Norm } s0 \neg \text{BinOp binop } e1 \ e2 \neg (\text{eval-binop binop } v1 \ v2) \rightarrow s1$
 $(\text{is } ?\text{EvalBinOp } v2)$
 $\langle \text{proof} \rangle$

single valued

lemma *unique-halloc* [rule-format (no-asm)]:

$G \vdash s -\text{halloc } oi \succ a \rightarrow s' \implies G \vdash s -\text{halloc } oi \succ a' \rightarrow s'' \longrightarrow a' = a \wedge s'' = s'$
 ⟨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)]:

$G \vdash s -t \rightarrow (w, s') \implies (\forall w' s''. G \vdash s -t \rightarrow (w', s'') \longrightarrow w' = w \wedge s'' = s')$
 ⟨proof⟩

lemma *single-valued-eval*:

single-valued $\{((s, t), (v, s')). G \vdash s -t \rightarrow (v, s')\}$
 ⟨proof⟩

end

Chapter 16

Example

1 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*: $\bigwedge fd. wf-fdecl\ G\ P\ fd = is-acc-type\ G\ P\ (type\ (snd\ fd))$
 $\langle proof \rangle$

declare *wf-fdecl-def2* [*iff*]

type and expression names

datatype *tnam'* = *HasFoo'* | *Base'* | *Ext'* | *Main'*

datatype *vnam'* = *arr'* | *vee'* | *z'* | *e'*

datatype *label'* = *lab1'*

axiomatization

tnam' :: *tnam'* \Rightarrow *tnam* **and**
vnam' :: *vnam'* \Rightarrow *vname* **and**
label' :: *label'* \Rightarrow *label*

where

inj-tnam' [*simp*]: $\bigwedge x\ y. (tnam'\ x = tnam'\ y) = (x = y)$ **and**
inj-vnam' [*simp*]: $\bigwedge x\ y. (vnam'\ x = vnam'\ y) = (x = y)$ **and**
inj-label' [*simp*]: $\bigwedge x\ y. (label'\ x = label'\ y) = (x = y)$ **and**

surj-tnam': $\bigwedge n. \exists m. n = tnam'\ m$ **and**
surj-vnam': $\bigwedge n. \exists m. n = vnam'\ m$ **and**
surj-label': $\bigwedge n. \exists m. n = label'\ m$

abbreviation

HasFoo :: *qname* **where**
HasFoo == ($\langle pid=java-lang, tid=TName\ (tnam'\ HasFoo') \rangle$)

abbreviation

Base :: *qname* **where**
Base == ($\langle pid=java-lang, tid=TName\ (tnam'\ Base') \rangle$)

abbreviation

Ext :: *qname* **where**
Ext == ($\langle pid=java-lang, tid=TName\ (tnam'\ Ext') \rangle$)

abbreviation

Main :: *qname* **where**
Main == ($\langle pid=java-lang, tid=TName\ (tnam'\ Main') \rangle$)

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'

```

lemma *neq-Base-Object* [simp]: *Base*≠*Object*
 ⟨proof⟩

lemma *neq-Ext-Object* [simp]: *Ext*≠*Object*
 ⟨proof⟩

lemma *neq-Main-Object* [simp]: *Main*≠*Object*
 ⟨proof⟩

lemma *neq-Base-SXcpt* [simp]: *Base*≠*SXcpt* *xn*
 ⟨proof⟩

lemma *neq-Ext-SXcpt* [simp]: *Ext*≠*SXcpt* *xn*
 ⟨proof⟩

lemma *neq-Main-SXcpt* [simp]: *Main*≠*SXcpt* *xn*
 ⟨proof⟩

classes and interfaces**overloading**

```

Object-mdecls ≡ Object-mdecls
SXcpt-mdecls ≡ SXcpt-mdecls

```

begin

```

definition Object-mdecls ≡ []
definition SXcpt-mdecls ≡ []

```

end**axiomatization**

```

foo    :: mname

```

definition

```

foo-sig :: sig
where foo-sig = (λname=foo,parTs=[Class Base])

```

definition

```

foo-mhead :: mhead
where foo-mhead = (λaccess=Public,static=False,pars=[z],resT=Class Base)

```

definition

```

Base-foo :: mdecl
where Base-foo = (foo-sig, (λaccess=Public,static=False,pars=[z],resT=Class Base,

```

mbody=(*lcls*=[],*stmt*=Return (!*z*))

definition *Ext-foo* :: *mdecl*

where *Ext-foo* = (*foo-sig*,
 (*access*=Public,*static*=False,*pars*=*z*,*resT*=Class *Ext*,
mbody=(*lcls*=[]
,stmt=Expr({*Ext*,*Ext*,False} Cast (Class *Ext*) (!*z*)..*vee* :=
Lit (Intg 1)) ;;
Return (Lit Null))
)

definition

arr-viewed-from :: *qtname* ⇒ *qtname* ⇒ *var*
where *arr-viewed-from* *accC C* = {*accC*,*Base*,*True*}StatRef (ClassT *C*)..*arr*

definition

BaseCl :: class **where**
BaseCl = (*access*=Public,
cfields=[(*arr*, (*access*=Public,*static*=True ,*type*=PrimT Boolean.[])),
(*vee*, (*access*=Public,*static*=False,*type*=Iface HasFoo [])),
methods=[*Base-foo*],
init=Expr(*arr-viewed-from* *Base* *Base*
:=New (PrimT Boolean)[Lit (Intg 2)]),
super=Object,
superIfs=[*HasFoo*])

definition

ExtCl :: class **where**
ExtCl = (*access*=Public,
cfields=[(*vee*, (*access*=Public,*static*=False,*type*= PrimT Integer))],
methods=[*Ext-foo*],
init=Skip,
super=*Base*,
superIfs=[])

definition

MainCl :: class **where**
MainCl = (*access*=Public,
cfields=[],
methods=[],
init=Skip,
super=Object,
superIfs=[])

definition

HasFooInt :: *iface*
where *HasFooInt* = (*access*=Public,*imethods*=[(*foo-sig*, *foo-mhead*)],*isuperIfs*=[])

definition

Ifaces :: *idecl* list
where *Ifaces* = [(*HasFoo*,*HasFooInt*)]

definition

Classes :: *cdecl* list
where *Classes* = [(*Base*,*BaseCl*),(*Ext*,*ExtCl*),(*Main*,*MainCl*)]@standard-classes

lemmas *table-classes-defs* =

Classes-def *standard-classes-def* *ObjectC-def* *SXcptC-def*

lemma *table-ifaces* [simp]: *table-of Ifaces* = *Map.empty*(*HasFoo*→*HasFooInt*)
 ⟨proof⟩

lemma *table-classes-Object* [simp]:
table-of Classes Object = *Some* (|*access*=*Public*,*cfields*=[]
 ,*methods*=*Object-mdecls*
 ,*init*=*Skip*,*super*=*undefined*,*superIfs*=[]|)
 ⟨proof⟩

lemma *table-classes-SXcpt* [simp]:
table-of Classes (SXcpt xn)
 = *Some* (|*access*=*Public*,*cfields*=[],*methods*=*SXcpt-mdecls*,
 init=*Skip*,
 super=if *xn* = *Throwable* then *Object* else *SXcpt Throwable*,
 superIfs=[]|)
 ⟨proof⟩

lemma *table-classes-HasFoo* [simp]: *table-of Classes HasFoo* = *None*
 ⟨proof⟩

lemma *table-classes-Base* [simp]: *table-of Classes Base* = *Some BaseCl*
 ⟨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 == (|*ifaces*=*Ifaces*,*classes*=*Classes*|)

definition

test :: (*ty*)*list* ⇒ *stmt* **where**
test pTs = (*e*==*NewC Ext*;;
 Try Expr({|*Main*,*ClassT Base*,*IntVir*}!!*e.foo*({|*pTs*}[*Lit Null*]))
 Catch((*SXcpt NullPointerException*) *z*)
 (*lab1*• *While*(*Acc*
 (*Acc* (*arr-viewed-from Main Ext*).[*Lit (Intg 2)*])) *Skip*))

well-structuredness

lemma *not-Object-subcls-any* [elim!]: (*Object*, *C*) ∈ (*subcls1 tprg*)⁺ ⇒ *R*
 ⟨proof⟩

lemma *not-Throwable-subcls-SXcpt* [elim!]:
 (*SXcpt Throwable*, *SXcpt xn*) ∈ (*subcls1 tprg*)⁺ ⇒ *R*
 ⟨proof⟩

lemma *not-SXcpt-n-subcls-SXcpt-n* [elim!]:
 (*SXcpt xn*, *SXcpt xn*) ∈ (*subcls1 tprg*)⁺ ⇒ *R*
 ⟨proof⟩

lemma *not-Base-subcls-Ext* [elim!]: (*Base*, *Ext*) ∈ (*subcls1 tprg*)⁺ ⇒ *R*
 ⟨proof⟩

lemma *not-TName-n-subcls-TName-n* [rule-format (no-asm), elim!]:
 $((\langle pid=java-lang, tid=TName\ tn \rangle), (\langle pid=java-lang, tid=TName\ tn \rangle))$
 $\in (subcls1\ tprg)^+ \longrightarrow R$
 <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*:
 $G=tprg \implies (\forall (I,i) \in set\ Ifaces.\ ws-idecl\ G\ I\ (isuperIfs\ i))$
 <proof>

lemma *ws-cdecl-all*: $G=tprg \implies (\forall (C,c) \in set\ Classes.\ ws-cdecl\ G\ C\ (super\ c))$
 <proof>

lemma *ws-tprg*: *ws-prog tprg*
 <proof>

misc program properties (independent of well-structuredness)

lemma *single-iface* [simp]: *is-iface tprg I = (I = HasFoo)*
 <proof>

lemma *empty-subint1* [simp]: *subint1 tprg = {}*
 <proof>

lemma *unique-ifaces*: *unique Ifaces*
 <proof>

lemma *unique-classes*: *unique Classes*
 <proof>

lemma *SXcpt-subcls-Throwable* [simp]: $\text{tprg} \vdash \text{SXcpt } xn \preceq_C \text{ SXcpt Throwable}$
 ⟨proof⟩

lemma *Ext-subclseq-Base* [simp]: $\text{tprg} \vdash \text{Ext} \preceq_C \text{ Base}$
 ⟨proof⟩

lemma *Ext-subcls-Base* [simp]: $\text{tprg} \vdash \text{Ext} \prec_C \text{ Base}$
 ⟨proof⟩

fields and method lookup

lemma *fields-tprg-Object* [simp]: $\text{DeclConcepts.fields tprg Object} = []$
 ⟨proof⟩

lemma *fields-tprg-Throwable* [simp]:
 $\text{DeclConcepts.fields tprg (SXcpt Throwable)} = []$
 ⟨proof⟩

lemma *fields-tprg-SXcpt* [simp]: $\text{DeclConcepts.fields tprg (SXcpt } xn) = []$
 ⟨proof⟩

lemmas *fields-rec' = fields-rec* [OF - ws-tprg]

lemma *fields-Base* [simp]:
 $\text{DeclConcepts.fields tprg Base}$
 $= [((\text{arr}, \text{Base}), (\text{access}=\text{Public}, \text{static}=\text{True}, \text{type}=\text{PrimT Boolean}, [])),$
 $((\text{vee}, \text{Base}), (\text{access}=\text{Public}, \text{static}=\text{False}, \text{type}=\text{Iface HasFoo } \text{ }))]]$
 ⟨proof⟩

lemma *fields-Ext* [simp]:
 $\text{DeclConcepts.fields tprg Ext}$
 $= [((\text{vee}, \text{Ext}), (\text{access}=\text{Public}, \text{static}=\text{False}, \text{type}=\text{PrimT Integer}))]$
 $@ \text{DeclConcepts.fields tprg Base}$
 ⟨proof⟩

lemmas *imethds-rec' = imethds-rec* [OF - ws-tprg]

lemmas *methd-rec' = methd-rec* [OF - ws-tprg]

lemma *imethds-HasFoo* [simp]:
 $\text{imethds tprg HasFoo} = \text{set-option} \circ \text{Map.empty}(\text{foo-sig} \mapsto (\text{HasFoo}, \text{foo-mhead}))$
 ⟨proof⟩

lemma *methd-tprg-Object* [simp]: $\text{methd tprg Object} = \text{Map.empty}$
 ⟨proof⟩

lemma *methd-Base* [simp]:
 $\text{methd tprg Base} = \text{table-of } [(\lambda(s, m). (s, \text{Base}, m)) \text{ Base-foo}]$
 ⟨proof⟩

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

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

lemma *Base-declares-foo*:

tprg ⊢ mdecl Base-foo declared-in Base
 ⟨proof⟩

lemma *foo-sig-not-undeclared-in-Base*:
 ¬ *tprg* ⊢ mid foo-sig undeclared-in Base
 ⟨proof⟩

lemma *Ext-declares-foo*:
tprg ⊢ mdecl Ext-foo declared-in Ext
 ⟨proof⟩

lemma *foo-sig-not-undeclared-in-Ext*:
 ¬ *tprg* ⊢ mid foo-sig undeclared-in Ext
 ⟨proof⟩

lemma *Base-foo-not-inherited-in-Ext*:
 ¬ *tprg* ⊢ Ext inherits (Base, mdecl Base-foo)
 ⟨proof⟩

lemma *Ext-method-inheritance*:
 filter-tab (λ sig m. *tprg* ⊢ Ext inherits method sig m)
 (Map.empty(fst ((λ(s, m). (s, Base, m)) Base-foo) ↦
 snd ((λ(s, m). (s, Base, m)) Base-foo)))
 = Map.empty
 ⟨proof⟩

lemma *methd-Ext [simp]*: methd *tprg* Ext =
 table-of [(λ(s, m). (s, Ext, m)) Ext-foo]
 ⟨proof⟩

accessibility

lemma *classesDefined*:
 ⟦class *tprg* C = Some c; C ≠ Object⟧ ⇒ ∃ sc. class *tprg* (super c) = 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

$\langle \text{proof} \rangle$

lemma *Base-is-acc-type[simp]: is-acc-type tprg P (Class Base)*
 $\langle \text{proof} \rangle$

lemma *Ext-accessible[simp]: tprg ⊢ (Class Ext) accessible-in P*
 $\langle \text{proof} \rangle$

lemma *Ext-is-acc-class[simp]: is-acc-class tprg P Ext*
 $\langle \text{proof} \rangle$

lemma *Ext-is-acc-type[simp]: is-acc-type tprg P (Class Ext)*
 $\langle \text{proof} \rangle$

lemma *accmethd-tprg-Object [simp]: accmethd tprg S Object = Map.empty*
 $\langle \text{proof} \rangle$

lemma *snd-special-simp: snd ((λ(s, m). (s, a, m)) x) = (a, snd x)*
 $\langle \text{proof} \rangle$

lemma *fst-special-simp: fst ((λ(s, m). (s, a, m)) x) = fst x*
 $\langle \text{proof} \rangle$

lemma *foo-sig-undeclared-in-Object:*
tprg ⊢ mid foo-sig undeclared-in Object
 $\langle \text{proof} \rangle$

lemma *unique-sig-Base-foo:*
tprg ⊢ mdecl (sig, snd Base-foo) declared-in Base \implies sig=foo-sig
 $\langle \text{proof} \rangle$

lemma *Base-foo-no-override:*
tprg, sig ⊢ (Base, (snd Base-foo)) overrides old \implies P
 $\langle \text{proof} \rangle$

lemma *Base-foo-no-stat-override:*
tprg, sig ⊢ (Base, (snd Base-foo)) overrides_S old \implies P
 $\langle \text{proof} \rangle$

lemma *Base-foo-no-hide:*
tprg, sig ⊢ (Base, (snd Base-foo)) hides old \implies P
 $\langle \text{proof} \rangle$

lemma *Ext-foo-no-hide:*
tprg, sig ⊢ (Ext, (snd Ext-foo)) hides old \implies P
 $\langle \text{proof} \rangle$

lemma *unique-sig-Ext-foo:*
tprg ⊢ mdecl (sig, snd Ext-foo) declared-in Ext \implies sig=foo-sig
 $\langle \text{proof} \rangle$

lemma *Ext-foo-override:*
tprg, sig ⊢ (Ext, (snd Ext-foo)) overrides old
 \implies *old = (Base, (snd Base-foo))*
 $\langle \text{proof} \rangle$

lemma *Ext-foo-stat-override:*
tprg, sig ⊢ (Ext, (snd Ext-foo)) overrides_S old

$\implies old = (Base, (snd\ Base-foo))$
 $\langle proof \rangle$

lemma *Base-foo-member-of-Base*:
 $tprg \vdash (Base, mdecl\ Base-foo)\ member-of\ Base$
 $\langle proof \rangle$

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]*:
 $accmethd\ tprg\ S\ Ext = methd\ tprg\ Ext$
 $\langle proof \rangle$

lemma *cls-Ext*: $class\ tprg\ Ext = Some\ ExtCl$
 $\langle proof \rangle$

lemma *dynmethd-Ext-foo*:
 $dynmethd\ tprg\ Base\ Ext\ (\name = foo, parTs = [Class\ Base])$

$= \text{Some } (\text{Ext}, \text{snd Ext-foo})$
 $\langle \text{proof} \rangle$

lemma *Base-fields-accessible*[simp]:
 $\text{accfield tprg } S \text{ Base}$
 $= \text{table-of}((\text{map } (\lambda((n,d),f).(n,(d,f)))) (\text{DeclConcepts.fields tprg Base}))$
 $\langle \text{proof} \rangle$

lemma *arr-member-of-Base*:
 $\text{tprg} \vdash (\text{Base}, \text{fdecl } (\text{arr},$
 $\quad \langle \text{access} = \text{Public}, \text{static} = \text{True}, \text{type} = \text{PrimT Boolean} \rangle))$
 member-of Base
 $\langle \text{proof} \rangle$

lemma *arr-member-in-Base*:
 $\text{tprg} \vdash (\text{Base}, \text{fdecl } (\text{arr},$
 $\quad \langle \text{access} = \text{Public}, \text{static} = \text{True}, \text{type} = \text{PrimT Boolean} \rangle))$
 member-in Base
 $\langle \text{proof} \rangle$

lemma *arr-member-of-Ext*:
 $\text{tprg} \vdash (\text{Base}, \text{fdecl } (\text{arr},$
 $\quad \langle \text{access} = \text{Public}, \text{static} = \text{True}, \text{type} = \text{PrimT Boolean} \rangle))$
 member-of Ext
 $\langle \text{proof} \rangle$

lemma *arr-member-in-Ext*:
 $\text{tprg} \vdash (\text{Base}, \text{fdecl } (\text{arr},$
 $\quad \langle \text{access} = \text{Public}, \text{static} = \text{True}, \text{type} = \text{PrimT Boolean} \rangle))$
 member-in Ext
 $\langle \text{proof} \rangle$

lemma *Ext-fields-accessible*[simp]:
 $\text{accfield tprg } S \text{ Ext}$
 $= \text{table-of}((\text{map } (\lambda((n,d),f).(n,(d,f)))) (\text{DeclConcepts.fields tprg Ext}))$
 $\langle \text{proof} \rangle$

lemma *arr-Base-dyn-accessible* [simp]:
 $\text{tprg} \vdash (\text{Base}, \text{fdecl } (\text{arr}, \langle \text{access} = \text{Public}, \text{static} = \text{True}, \text{type} = \text{PrimT Boolean} \rangle))$
 $\text{in Base dyn-accessible-from } S$
 $\langle \text{proof} \rangle$

lemma *arr-Ext-dyn-accessible*[simp]:
 $\text{tprg} \vdash (\text{Base}, \text{fdecl } (\text{arr}, \langle \text{access} = \text{Public}, \text{static} = \text{True}, \text{type} = \text{PrimT Boolean} \rangle))$
 $\text{in Ext dyn-accessible-from } S$
 $\langle \text{proof} \rangle$

lemma *array-of-PrimT-acc* [simp]:
 $\text{is-acc-type tprg java-lang } (\text{PrimT } t.)$
 $\langle \text{proof} \rangle$

lemma *PrimT-acc* [simp]:
 $\text{is-acc-type tprg java-lang } (\text{PrimT } t)$
 $\langle \text{proof} \rangle$

lemma *Object-acc* [simp]:
 $\text{is-acc-class tprg java-lang Object}$
 $\langle \text{proof} \rangle$

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*]

lemma *wf-BaseC*: *wf-cdecl tprg (Base, BaseCl)*
<proof>

lemma *wf-ExtC*: *wf-cdecl tprg (Ext, ExtCl)*
<proof>

lemma *wf-MainC*: *wf-cdecl tprg (Main, MainCl)*
<proof>

lemma *wf-idecl-all*: *p=tprg \implies Ball (set Ifaces) (wf-idecl p)*
<proof>

lemma *wf-cdecl-all-standard-classes*:
Ball (set standard-classes) (wf-cdecl tprg)
<proof>

lemma *wf-cdecl-all*: *p=tprg \implies 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) (\llbracket name=foo, parTs=[NT] \rrbracket) =
{((ClassT Base, (\llbracket access=Public,static=False,pars=[z],resT=Class Base \rrbracket),
[Class Base])}
<proof>

lemma *max-spec-Base-foo*: *max-spec tprg S (ClassT Base) (\llbracket name=foo,parTs=[NT] \rrbracket) =*

```
{((ClassT Base, (access=Public,static=False,pars=[z],resT=Class Base))
, [Class Base])}
⟨proof⟩
```

well-typedness

schematic-goal *wt-test*: $(\text{prg}=\text{tprg}, \text{cls}=\text{Main}, \text{lcl}=\text{Map.empty}(\text{VName } e \mapsto \text{Class Base})) \vdash \text{test } ?pTs :: \checkmark$
 ⟨proof⟩

definite assignment

schematic-goal *da-test*: $(\text{prg}=\text{tprg}, \text{cls}=\text{Main}, \text{lcl}=\text{Map.empty}(\text{VName } e \mapsto \text{Class Base})) \vdash \{ \} \gg \langle \text{test } ?pTs \rangle \gg (\text{nrm}=\{ \text{VName } e \}, \text{brk}=\lambda l. \text{UNIV})$
 ⟨proof⟩

execution

lemma *alloc-one*: $\bigwedge a \text{ obj. } \llbracket \text{the } (\text{new-Addr } h) = a; \text{atleast-free } h \text{ (Suc } n) \rrbracket \implies \text{new-Addr } h = \text{Some } a \wedge \text{atleast-free } (h(a \mapsto \text{obj})) \text{ } n$
 ⟨proof⟩

```
declare fvar-def2 [simp] avar-def2 [simp] init-lvars-def2 [simp]
declare init-obj-def [simp] var-tys-def [simp] fields-table-def [simp]
declare BaseCl-def [simp] ExtCl-def [simp] Ext-foo-def [simp]
      Base-foo-defs [simp]
```

⟨ML⟩

lemmas *eval-Is* = *eval-Init eval-StatRef AbruptIs eval-intros*

axiomatization

```
a :: loc and
b :: loc and
c :: loc
```

abbreviation *one* == *Suc 0*
abbreviation *two* == *Suc one*
abbreviation *three* == *Suc two*
abbreviation *four* == *Suc three*

abbreviation

```
obj-a == (tag=Arr (PrimT Boolean) 2
, values= Map.empty(Inr 0 ↦ Bool False, Inr 1 ↦ Bool False))
```

abbreviation

```
obj-b == (tag=CInst Ext
, values=(Map.empty(Inl (vee, Base) ↦ Null, Inl (vee, Ext ) ↦ Intg 0)))
```

abbreviation

```
obj-c == (tag=CInst (SXcpt NullPointer), values=CONST Map.empty)
```

abbreviation *arr-N* == *Map.empty(Inl (arr, Base) ↦ Null)*

abbreviation *arr-a* == *Map.empty(Inl (arr, Base) ↦ Addr a)*

abbreviation

```
globs1 == Map.empty(Inr Ext ↦ (tag=undefined, values=Map.empty),
      Inr Base ↦ (tag=undefined, values=arr-N),
      Inr Object ↦ (tag=undefined, values=Map.empty))
```

abbreviation

```

globs2 == Map.empty(Inr Ext  ↦(tag=undefined, values=Map.empty)),
          Inr Object↦(tag=undefined, values=Map.empty),
          Inl a↦obj-a,
          Inr Base  ↦(tag=undefined, values=arr-a))

```

```

abbreviation globs3 == globs2(Inl b↦obj-b)
abbreviation globs8 == globs3(Inl c↦obj-c)
abbreviation locs3 == Map.empty(VName e↦Addr b)
abbreviation locs8 == locs3(VName z↦Addr c)

```

```

abbreviation s0 == st Map.empty Map.empty
abbreviation s0' == Norm s0
abbreviation s1 == st globs1 Map.empty
abbreviation s1' == Norm s1
abbreviation s2 == st globs2 Map.empty
abbreviation s2' == Norm s2
abbreviation s3 == st globs3 locs3
abbreviation s3' == Norm s3
abbreviation s7' == (Some (Xcpt (Std NullPointer)), s3)
abbreviation s8 == st globs8 locs8
abbreviation s8' == Norm s8
abbreviation s9' == (Some (Xcpt (Std IndOutBound)), s8)

```

```

declare prod.inject [simp del]
schematic-goal 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 prod.inject [simp]

```

end

Chapter 17

Conform

1 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.

type-synonym $env' = prog \times (lname, ty) \text{ table}$

extension of global store

definition $gext :: st \Rightarrow st \Rightarrow bool$ ($\hookrightarrow \leq \mid \rightarrow$ [71,71] 70) **where**
 $s \leq \mid s' \equiv \forall r. \forall obj \in globs \ s \ r : \exists obj' \in globs \ s' \ r : tag \ obj' = tag \ obj$

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*:

$\llbracket s \leq \mid s'; globs \ s \ r = Some \ obj \rrbracket$
 $\implies \exists obj'. globs \ s' \ r = Some \ obj' \wedge tag \ obj' = tag \ obj$
<proof>

lemma *rev-gext-objD*:

$\llbracket globs \ s \ r = Some \ obj; s \leq \mid s' \rrbracket$
 $\implies \exists obj'. globs \ s' \ r = Some \ obj' \wedge tag \ obj' = tag \ obj$
<proof>

lemma *init-class-obj-initd*:

$init_class_obj \ G \ C \ s1 \leq \mid s2 \implies initd \ C \ (globs \ s2)$
<proof>

lemma *gext-refl* [*intro!*, *simp*]: $s \leq \mid s$

<proof>

lemma *gext-gupd* [*simp*, *elim!*]: $\bigwedge s. globs \ s \ r = None \implies s \leq \mid gupd(r \mapsto x) s$

<proof>

lemma *gext-new* [*simp*, *elim!*]: $\bigwedge s. globs \ s \ r = None \implies s \leq \mid init_obj \ G \ oi \ r \ s$

<proof>

lemma *gext-trans* [elim]: $\bigwedge X. \llbracket s \leq |s'; s' \leq |s'' \rrbracket \implies s \leq |s''$
 ⟨proof⟩

lemma *gext-upd-gobj* [intro!]: $s \leq | \text{upd-gobj } r \ n \ v \ s$
 ⟨proof⟩

lemma *gext-cong1* [simp]: $\text{set-locals } l \ s1 \leq | s2 = s1 \leq | s2$
 ⟨proof⟩

lemma *gext-cong2* [simp]: $s1 \leq | \text{set-locals } l \ s2 = s1 \leq | s2$
 ⟨proof⟩

lemma *gext-lupd1* [simp]: $\text{lupd}(vn \mapsto v) s1 \leq | s2 = s1 \leq | s2$
 ⟨proof⟩

lemma *gext-lupd2* [simp]: $s1 \leq | \text{lupd}(vn \mapsto v) s2 = s1 \leq | s2$
 ⟨proof⟩

lemma *inited-gext*: $\llbracket \text{inited } C \ (\text{globs } s); s \leq | s' \rrbracket \implies \text{inited } C \ (\text{globs } s')$
 ⟨proof⟩

value conformance

definition *conf* :: $\text{prog} \Rightarrow \text{st} \Rightarrow \text{val} \Rightarrow \text{ty} \Rightarrow \text{bool}$ ($\hookleftarrow, \vdash, \preceq, \preceq$) [71, 71, 71, 71] 70)
 where $G, s \vdash v :: \preceq T = (\exists T' \in \text{typeof} \ (\lambda a. \text{map-option obj-ty} \ (\text{heap } s \ a)) \ v : G \vdash T' \preceq T)$

lemma *conf-cong* [simp]: $G, \text{set-locals } l \ s \vdash v :: \preceq T = G, s \vdash v :: \preceq T$
 ⟨proof⟩

lemma *conf-lupd* [simp]: $G, \text{lupd}(vn \mapsto va) s \vdash v :: \preceq T = G, s \vdash v :: \preceq T$
 ⟨proof⟩

lemma *conf-PrimT* [simp]: $\forall dt. \text{typeof } dt \ v = \text{Some} \ (\text{PrimT } t) \implies G, s \vdash v :: \preceq \text{PrimT } t$
 ⟨proof⟩

lemma *conf-Boolean*: $G, s \vdash v :: \preceq \text{PrimT Boolean} \implies \exists b. v = \text{Bool } b$
 ⟨proof⟩

lemma *conf-litval* [rule-format (no-asm)]:
 $\text{typeof} \ (\lambda a. \text{None}) \ v = \text{Some } T \longrightarrow G, s \vdash v :: \preceq T$
 ⟨proof⟩

lemma *conf-Null* [simp]: $G, s \vdash \text{Null} :: \preceq T = G \vdash NT \preceq T$
 ⟨proof⟩

lemma *conf-Addr*:
 $G, s \vdash \text{Addr } a :: \preceq T = (\exists \text{obj}. \text{heap } s \ a = \text{Some obj} \wedge G \vdash \text{obj-ty obj} \preceq T)$
 ⟨proof⟩

lemma *conf-AddrI*: $\llbracket \text{heap } s \ a = \text{Some obj}; G \vdash \text{obj-ty obj} \preceq T \rrbracket \implies G, s \vdash \text{Addr } a :: \preceq T$
 ⟨proof⟩

lemma *defval-conf* [rule-format (no-asm), elim]:
 $\text{is-type } G \ T \longrightarrow G, s \vdash \text{default-val } T :: \preceq T$
 ⟨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]:

$G, s \vdash v :: \preceq T \longrightarrow s \leq s' \longrightarrow G, s \vdash v :: \preceq T$
 ⟨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)]:

$G, s \vdash a' :: \preceq \text{RefT } T$
 $\longrightarrow a' = \text{Null} \vee (\exists a \text{ obj } T'. a' = \text{Addr } a \wedge \text{heap } s \ a = \text{Some obj} \wedge$
 $\text{obj-ty obj} = T' \wedge G \vdash T' \preceq \text{RefT } T)$
 ⟨proof⟩

value list conformance

definition

$\text{lconf} :: \text{prog} \Rightarrow \text{st} \Rightarrow ('a, \text{val}) \text{ table} \Rightarrow ('a, \text{ty}) \text{ table} \Rightarrow \text{bool} \ (\hookrightarrow, \vdash, \vdash :: \preceq) \rightarrow [71, 71, 71, 71] \ 70)$
where $G, s \vdash \text{vs} [\preceq] Ts = (\forall n. \forall T \in Ts \ n: \exists v \in \text{vs} \ n: G, s \vdash v :: \preceq T)$

lemma *lconfD*: $\llbracket G, s \vdash \text{vs} [\preceq] Ts; Ts \ n = \text{Some } T \rrbracket \implies G, s \vdash (\text{the } (vs \ n)) :: \preceq T$

⟨proof⟩

lemma *lconf-cong* [simp]: $\bigwedge s. G, \text{set-locals } x \ s \vdash l [\preceq] L = G, s \vdash l [\preceq] L$

⟨proof⟩

lemma *lconf-lupd* [simp]: $G, \text{lupd}(vn \mapsto v) \ s \vdash l [\preceq] L = G, s \vdash l [\preceq] L$

⟨proof⟩

lemma *lconf-new*: $\llbracket L \ vn = \text{None}; G, s \vdash l [\preceq] L \rrbracket \implies G, s \vdash l(vn \mapsto v) [\preceq] L$

⟨proof⟩

lemma *lconf-upd*: $\llbracket G, s \vdash l [\preceq] L; G, s \vdash v :: \preceq T; L \ vn = \text{Some } T \rrbracket \implies$

$G, s \vdash l(vn \mapsto v) [\preceq] L$
 ⟨proof⟩

lemma *lconf-ext*: $\llbracket G, s \vdash l [\preceq] L; G, s \vdash v :: \preceq T \rrbracket \implies G, s \vdash l(vn \mapsto v) [\preceq] L(vn \mapsto T)$

⟨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)]:

$\bigwedge X. \llbracket G, s \vdash l [\preceq] L \rrbracket \implies$
 $\forall vs \ Ts. \text{distinct } vns \longrightarrow \text{length } Ts = \text{length } vns$
 $\longrightarrow \text{list-all2 } (\text{conf } G \ s) \ \text{vs } Ts \longrightarrow G, s \vdash l(vns \mapsto vs) [\preceq] L(vns \mapsto Ts)$
 ⟨proof⟩

lemma *lconf-deallocL*: $\llbracket G, s \vdash l[\sim::\preceq] L (vn \mapsto T); L \text{ } vn = \text{None} \rrbracket \implies G, s \vdash l[\sim::\preceq] L$
 $\langle \text{proof} \rangle$

lemma *lconf-gext [elim]*: $\llbracket G, s \vdash l[\sim::\preceq] L; s \leq s' \rrbracket \implies G, s' \vdash l[\sim::\preceq] L$
 $\langle \text{proof} \rangle$

lemma *lconf-empty [simp, intro!]*: $G, s \vdash vs[\sim::\preceq] \text{Map.empty}$
 $\langle \text{proof} \rangle$

lemma *lconf-init-vals [intro!]*:
 $\forall n. \forall T \in fs \text{ } n \text{ is-type } G \text{ } T \implies G, s \vdash \text{init-vals } fs[\sim::\preceq] fs$
 $\langle \text{proof} \rangle$

weak value list conformance

Only if the value is defined it has to conform to its type. This is the contribution of the definite assignment analysis to the notion of conformance. The definite assignment analysis ensures that the program only attempts to access local variables that actually have a defined value in the state. So conformance must only ensure that the defined values are of the right type, and not also that the value is defined.

definition

wlconf :: *prog* \Rightarrow *st* \Rightarrow (*a*, *val*) *table* \Rightarrow (*a*, *ty*) *table* \Rightarrow *bool* ($\langle -, \vdash -[\sim::\preceq] \rangle \rightarrow [71, 71, 71, 71]$ 70)
where $G, s \vdash vs[\sim::\preceq] Ts = (\forall n. \forall T \in Ts \text{ } n: \forall v \in vs \text{ } n: G, s \vdash v::\preceq T)$

lemma *wlconfD*: $\llbracket G, s \vdash vs[\sim::\preceq] Ts; Ts \text{ } n = \text{Some } T; vs \text{ } n = \text{Some } v \rrbracket \implies G, s \vdash v::\preceq T$
 $\langle \text{proof} \rangle$

lemma *wlconf-cong [simp]*: $\bigwedge s. G, \text{set-locals } x \text{ } s \vdash l[\sim::\preceq] L = G, s \vdash l[\sim::\preceq] L$
 $\langle \text{proof} \rangle$

lemma *wlconf-lupd [simp]*: $G, \text{lupd}(vn \mapsto v) s \vdash l[\sim::\preceq] L = G, s \vdash l[\sim::\preceq] L$
 $\langle \text{proof} \rangle$

lemma *wlconf-upd*: $\llbracket G, s \vdash l[\sim::\preceq] L; G, s \vdash v::\preceq T; L \text{ } vn = \text{Some } T \rrbracket \implies$
 $G, s \vdash l(vn \mapsto v)[\sim::\preceq] L$
 $\langle \text{proof} \rangle$

lemma *wlconf-ext*: $\llbracket G, s \vdash l[\sim::\preceq] L; G, s \vdash v::\preceq T \rrbracket \implies G, s \vdash l(vn \mapsto v)[\sim::\preceq] L (vn \mapsto T)$
 $\langle \text{proof} \rangle$

lemma *wlconf-map-sum [simp]*:
 $G, s \vdash l1 (+) l2[\sim::\preceq] L1 (+) L2 = (G, s \vdash l1[\sim::\preceq] L1 \wedge G, s \vdash l2[\sim::\preceq] L2)$
 $\langle \text{proof} \rangle$

lemma *wlconf-ext-list [rule-format (no-asm)]*:
 $\bigwedge X. \llbracket G, s \vdash l[\sim::\preceq] L \rrbracket \implies$
 $\forall vs \text{ } Ts. \text{distinct } vns \longrightarrow \text{length } Ts = \text{length } vns$
 $\longrightarrow \text{list-all2 } (\text{conf } G \text{ } s) \text{ } vs \text{ } Ts \longrightarrow G, s \vdash l(vns[\mapsto] vs)[\sim::\preceq] L (vns[\mapsto] Ts)$
 $\langle \text{proof} \rangle$

lemma *wlconf-deallocL*: $\llbracket G, s \vdash l[\sim::\preceq] L (vn \mapsto T); L \text{ } vn = \text{None} \rrbracket \implies G, s \vdash l[\sim::\preceq] L$
 $\langle \text{proof} \rangle$

lemma *wlconf-geat* [elim]: $\llbracket G, s \vdash l[\sim::\preceq] L; s \leq |s' \rrbracket \implies G, s' \vdash l[\sim::\preceq] L$
 ⟨proof⟩

lemma *wlconf-empty* [simp, intro!]: $G, s \vdash vs[\sim::\preceq] \text{Map.empty}$
 ⟨proof⟩

lemma *wlconf-empty-vals*: $G, s \vdash \text{Map.empty}[\sim::\preceq] ts$
 ⟨proof⟩

lemma *wlconf-init-vals* [intro!]:
 $\forall n. \forall T \in fs \ n:is\text{-}type \ G \ T \implies G, s \vdash \text{init-vals} \ fs[\sim::\preceq] fs$
 ⟨proof⟩

lemma *lconf-wlconf*:
 $G, s \vdash l[\sim::\preceq] L \implies G, s \vdash l[\sim::\preceq] L$
 ⟨proof⟩

object conformance

definition

oconf :: *prog* \Rightarrow *st* \Rightarrow *obj* \Rightarrow *oref* \Rightarrow *bool* ($\hookrightarrow, \vdash, \sim::\preceq, \sqrt{\cdot}$) [71, 71, 71, 71] 70) **where**
 $(G, s \vdash obj::\preceq \sqrt{r}) = (G, s \vdash \text{values } obj[\sim::\preceq] \text{var-tys } G \text{ (tag obj) } r \wedge$
 (case *r* of
 Heap *a* \Rightarrow *is-type* *G* (*obj-ty* *obj*)
 | Stat *C* \Rightarrow *True*))

lemma *oconf-is-type*: $G, s \vdash obj::\preceq \sqrt{\text{Heap } a} \implies is\text{-}type \ G \ (obj\text{-}ty \ obj)$
 ⟨proof⟩

lemma *oconf-lconf*: $G, s \vdash obj::\preceq \sqrt{r} \implies G, s \vdash \text{values } obj[\sim::\preceq] \text{var-tys } G \text{ (tag obj) } r$
 ⟨proof⟩

lemma *oconf-cong* [simp]: $G, \text{set-locals } l \vdash obj::\preceq \sqrt{r} = G, s \vdash obj::\preceq \sqrt{r}$
 ⟨proof⟩

lemma *oconf-init-obj-lemma*:

$\llbracket \bigwedge C \ c. \text{class } G \ C = \text{Some } c \implies \text{unique } (\text{DeclConcepts.fields } G \ C);$
 $\bigwedge C \ c \ f \text{ fld. } \llbracket \text{class } G \ C = \text{Some } c;$
 $\text{table-of } (\text{DeclConcepts.fields } G \ C) \ f = \text{Some fld} \rrbracket$
 $\implies is\text{-}type \ G \ (type \ fld);$
 (case *r* of
 Heap *a* \Rightarrow *is-type* *G* (*obj-ty* *obj*)
 | Stat *C* \Rightarrow *is-class* *G* *C*)
 $\rrbracket \implies G, s \vdash obj \ (values := init\text{-}vals \ (var\text{-}tys \ G \ (tag \ obj) \ r))::\preceq \sqrt{r}$
 ⟨proof⟩

state conformance

definition

conforms :: *state* \Rightarrow *env'* \Rightarrow *bool* ($\hookrightarrow::\preceq$) [71, 71] 70) **where**
 $xs::\preceq E =$
 (let (*G*, *L*) = *E*; *s* = *snd* *xs*; *l* = *locals* *s* in
 ($\forall r. \forall obj \in \text{globs } s \ r: G, s \vdash obj::\preceq \sqrt{r}) \wedge G, s \vdash l[\sim::\preceq] L \wedge$
 ($\forall a. \text{fst } xs = \text{Some}(Xcpt \ (\text{Loc } a)) \longrightarrow G, s \vdash \text{Addr } a::\preceq \text{Class } (SXcpt \ \text{Throwable})) \wedge$
 ($\text{fst } xs = \text{Some}(\text{Jump } \text{Ret}) \longrightarrow l \ \text{Result} \neq \text{None}))$

conforms**lemma** *conforms-globsD*:
$$\llbracket (x, s) :: \preceq (G, L); \text{globs } s \text{ } r = \text{Some } obj \rrbracket \implies G, s \vdash obj :: \preceq \sqrt{r}$$

<proof>

lemma *conforms-localD*: $(x, s) :: \preceq (G, L) \implies G, s \vdash \text{locals } s [\sim :: \preceq] L$ *<proof>***lemma** *conforms-XcptLocD*: $\llbracket (x, s) :: \preceq (G, L); x = \text{Some } (Xcpt (\text{Loc } a)) \rrbracket \implies$
 $G, s \vdash \text{Addr } a :: \preceq \text{Class } (SXcpt \text{ Throwable})$ *<proof>***lemma** *conforms-RetD*: $\llbracket (x, s) :: \preceq (G, L); x = \text{Some } (\text{Jump Ret}) \rrbracket \implies$
 $(\text{locals } s) \text{ Result} \neq \text{None}$ *<proof>***lemma** *conforms-RefTD*:
$$\llbracket G, s \vdash a' :: \preceq \text{RefT } t; a' \neq \text{Null}; (x, s) :: \preceq (G, L) \rrbracket \implies$$

$$\exists a \text{ obj}. a' = \text{Addr } a \wedge \text{globs } s (\text{Inl } a) = \text{Some } obj \wedge$$

$$G \vdash \text{obj-ty } obj \preceq \text{RefT } t \wedge \text{is-type } G (\text{obj-ty } obj)$$
*<proof>***lemma** *conforms-Jump [iff]*:
$$j = \text{Ret} \longrightarrow \text{locals } s \text{ Result} \neq \text{None}$$

$$\implies ((\text{Some } (\text{Jump } j), s) :: \preceq (G, L)) = (\text{Norm } s :: \preceq (G, L))$$
*<proof>***lemma** *conforms-StdXcpt [iff]*:
$$((\text{Some } (Xcpt (\text{Std } xn)), s) :: \preceq (G, L)) = (\text{Norm } s :: \preceq (G, L))$$
*<proof>***lemma** *conforms-Err [iff]*:
$$((\text{Some } (\text{Error } e), s) :: \preceq (G, L)) = (\text{Norm } s :: \preceq (G, L))$$
*<proof>***lemma** *conforms-raise-if [iff]*:
$$((\text{raise-if } c \text{ } xn \text{ } x, s) :: \preceq (G, L)) = ((x, s) :: \preceq (G, L))$$
*<proof>***lemma** *conforms-error-if [iff]*:
$$((\text{error-if } c \text{ } err \text{ } x, s) :: \preceq (G, L)) = ((x, s) :: \preceq (G, L))$$
*<proof>***lemma** *conforms-NormI*: $(x, s) :: \preceq (G, L) \implies \text{Norm } s :: \preceq (G, L)$ *<proof>***lemma** *conforms-absorb [rule-format]*:
$$(a, b) :: \preceq (G, L) \longrightarrow (\text{absorb } j \text{ } a, b) :: \preceq (G, L)$$
*<proof>***lemma** *conformsI*: $\llbracket \forall r. \forall obj \in \text{globs } s \text{ } r: G, s \vdash obj :: \preceq \sqrt{r};$ $G, s \vdash \text{locals } s [\sim :: \preceq] L;$ $\forall a. x = \text{Some } (Xcpt (\text{Loc } a)) \longrightarrow G, s \vdash \text{Addr } a :: \preceq \text{Class } (SXcpt \text{ Throwable});$ $x = \text{Some } (\text{Jump Ret}) \longrightarrow \text{locals } s \text{ Result} \neq \text{None} \rrbracket \implies$ $(x, s) :: \preceq (G, L)$ *<proof>***lemma** *conforms-xconf*: $\llbracket (x, s) :: \preceq (G, L);$

$\forall a. x' = \text{Some } (Xcpt \text{ (Loc } a)) \longrightarrow G, s \vdash \text{Addr } a :: \preceq \text{Class } (SXcpt \text{ Throwable});$
 $x' = \text{Some } (\text{Jump Ret}) \longrightarrow \text{locals } s \text{ Result} \neq \text{None} \rVert \implies$
 $(x', s) :: \preceq (G, L)$
 $\langle \text{proof} \rangle$

lemma *conforms-lupd*:

$\rVert (x, s) :: \preceq (G, L); L \text{ vn} = \text{Some } T; G, s \vdash v :: \preceq T \rVert \implies (x, \text{lupd}(vn \mapsto v)s) :: \preceq (G, L)$
 $\langle \text{proof} \rangle$

lemmas *conforms-allocL-aux* = *conforms-localD* [THEN *wlconf-ext*]

lemma *conforms-allocL*:

$\rVert (x, s) :: \preceq (G, L); G, s \vdash v :: \preceq T \rVert \implies (x, \text{lupd}(vn \mapsto v)s) :: \preceq (G, L(vn \mapsto T))$
 $\langle \text{proof} \rangle$

lemmas *conforms-deallocL-aux* = *conforms-localD* [THEN *wlconf-deallocL*]

lemma *conforms-deallocL*: $\bigwedge s. \rVert s :: \preceq (G, L(vn \mapsto T)); L \text{ vn} = \text{None} \rVert \implies s :: \preceq (G, L)$

$\langle \text{proof} \rangle$

lemma *conforms-gext*: $\rVert (x, s) :: \preceq (G, L); s \leq |s'|;$

$\forall r. \forall \text{obj} \in \text{globs } s' \text{ } r: G, s \vdash \text{obj} :: \preceq \sqrt{r};$
 $\text{locals } s' = \text{locals } s \rVert \implies (x, s') :: \preceq (G, L)$

$\langle \text{proof} \rangle$

lemma *conforms-xgext*:

$\rVert (x, s) :: \preceq (G, L); (x', s') :: \preceq (G, L); s' \leq |s; \text{dom } (\text{locals } s') \subseteq \text{dom } (\text{locals } s) \rVert$
 $\implies (x', s) :: \preceq (G, L)$

$\langle \text{proof} \rangle$

lemma *conforms-gupd*: $\bigwedge \text{obj}. \rVert (x, s) :: \preceq (G, L); G, s \vdash \text{obj} :: \preceq \sqrt{r}; s \leq | \text{gupd}(r \mapsto \text{obj})s \rVert$

$\implies (x, \text{gupd}(r \mapsto \text{obj})s) :: \preceq (G, L)$

$\langle \text{proof} \rangle$

lemma *conforms-upd-gobj*: $\rVert (x, s) :: \preceq (G, L); \text{globs } s \text{ } r = \text{Some } \text{obj};$

$\text{var-tys } G \text{ (tag obj) } r \text{ n} = \text{Some } T; G, s \vdash v :: \preceq T \rVert \implies (x, \text{upd-gobj } r \text{ n } v \text{ } s) :: \preceq (G, L)$
 $\langle \text{proof} \rangle$

lemma *conforms-set-locals*:

$\rVert (x, s) :: \preceq (G, L); G, s \vdash l [\sim :: \preceq] L; x = \text{Some } (\text{Jump Ret}) \longrightarrow l \text{ Result} \neq \text{None} \rVert$
 $\implies (x, \text{set-locals } l \text{ } s) :: \preceq (G, L)$

$\langle \text{proof} \rangle$

lemma *conforms-locals*:

$\rVert (a, b) :: \preceq (G, L); L \text{ } x = \text{Some } T; \text{locals } b \text{ } x \neq \text{None} \rVert$
 $\implies G, b \vdash \text{the } (\text{locals } b \text{ } x) :: \preceq T$

$\langle \text{proof} \rangle$

lemma *conforms-return*:

$\bigwedge s'. \rVert (x, s) :: \preceq (G, L); (x', s') :: \preceq (G, L); s \leq |s'; x' \neq \text{Some } (\text{Jump Ret}) \rVert \implies$
 $(x', \text{set-locals } (\text{locals } s) \text{ } s') :: \preceq (G, L)$

$\langle \text{proof} \rangle$

end

Chapter 18

DefiniteAssignmentCorrect

1 Correctness of Definite Assignment

theory *DefiniteAssignmentCorrect* **imports** *WellForm Eval* **begin**

declare $[[simproc\ del:\ wt\text{-}expr\ wt\text{-}var\ wt\text{-}exprs\ wt\text{-}stmt]]$

lemma *sxalloc-no-jump*:

assumes *sxalloc*: $G \vdash s0 \text{ --sxalloc--} \rightarrow s1$ **and**
no-jmp: $abrupt\ s0 \neq Some\ (Jump\ j)$
shows $abrupt\ s1 \neq Some\ (Jump\ j)$

$\langle proof \rangle$

lemma *sxalloc-no-jump'*:

assumes *sxalloc*: $G \vdash s0 \text{ --sxalloc--} \rightarrow s1$ **and**
jump: $abrupt\ s1 = Some\ (Jump\ j)$
shows $abrupt\ s0 = Some\ (Jump\ j)$

$\langle proof \rangle$

lemma *halloc-no-jump*:

assumes *halloc*: $G \vdash s0 \text{ --halloc } oi \succ a \rightarrow s1$ **and**
no-jmp: $abrupt\ s0 \neq Some\ (Jump\ j)$
shows $abrupt\ s1 \neq Some\ (Jump\ j)$

$\langle proof \rangle$

lemma *halloc-no-jump'*:

assumes *halloc*: $G \vdash s0 \text{ --halloc } oi \succ a \rightarrow s1$ **and**
jump: $abrupt\ s1 = Some\ (Jump\ j)$
shows $abrupt\ s0 = Some\ (Jump\ j)$

$\langle proof \rangle$

lemma *Body-no-jump*:

assumes *eval*: $G \vdash s0 \text{ --Body } D\ c \text{ --} \succ v \rightarrow s1$ **and**
jump: $abrupt\ s0 \neq Some\ (Jump\ j)$
shows $abrupt\ s1 \neq Some\ (Jump\ j)$

$\langle proof \rangle$

lemma *Methd-no-jump*:

assumes *eval*: $G \vdash s0 \text{ --Methd } D\ sig \text{ --} \succ v \rightarrow s1$ **and**
jump: $abrupt\ s0 \neq Some\ (Jump\ j)$
shows $abrupt\ s1 \neq Some\ (Jump\ j)$

$\langle proof \rangle$

lemma *jumpNestingOkS-mono*:

assumes *jumpNestingOk-l'*: $jumpNestingOkS\ jmps'\ c$

and $subset: jmps' \subseteq jmps$
shows $jumpNestingOkS\ jmps\ c$
 $\langle proof \rangle$

corollary $jumpNestingOk-mono$:
assumes $jumpOk: jumpNestingOk\ jmps'\ t$
and $subset: jmps' \subseteq jmps$
shows $jumpNestingOk\ jmps\ t$
 $\langle proof \rangle$

lemma $assign-abrupt-propagation$:
assumes $f-ok: abrupt\ (f\ n\ s) \neq x$
and $ass: abrupt\ (assign\ f\ n\ s) = x$
shows $abrupt\ s = x$
 $\langle proof \rangle$

lemma $wt-init-comp-ty'$:
 $is-acc-type\ (prg\ Env)\ (pid\ (cls\ Env))\ T \implies Env \vdash init-comp-ty\ T :: \checkmark$
 $\langle proof \rangle$

lemma $fvar-upd-no-jump$:
assumes $upd: upd = snd\ (fst\ (fvar\ statDeclC\ stat\ fn\ a\ s'))$
and $noJump: abrupt\ s \neq Some\ (Jump\ j)$
shows $abrupt\ (upd\ val\ s) \neq Some\ (Jump\ j)$
 $\langle proof \rangle$

lemma $avar-state-no-jump$:
assumes $jmp: abrupt\ (snd\ (avar\ G\ i\ a\ s)) = Some\ (Jump\ j)$
shows $abrupt\ s = Some\ (Jump\ j)$
 $\langle proof \rangle$

lemma $avar-upd-no-jump$:
assumes $upd: upd = snd\ (fst\ (avar\ G\ i\ a\ s'))$
and $noJump: abrupt\ s \neq Some\ (Jump\ j)$
shows $abrupt\ (upd\ val\ s) \neq Some\ (Jump\ j)$
 $\langle proof \rangle$

The next theorem expresses: If jumps (breaks, continues, returns) are nested correctly, we won't find an unexpected jump in the result state of the evaluation. For example, a break can't leave its enclosing loop, an return can't leave its enclosing method. To prove this, the method call is critical. Although the wellformedness of the whole program guarantees that the jumps (breaks, continues and returns) are nested correctly in all method bodies, the call rule alone does not guarantee that I will call a method or even a class that is part of the program due to dynamic binding! To be able to ensure this we need a kind of conformance of the state, like in the typesafety proof. But then we will redo the typesafety proof here. It would be nice if we could find an easy precondition that will guarantee that all calls will actually call classes and methods of the current program, which can be instantiated in the typesafety proof later on. To fix this problem, I have instrumented the semantic definition of a call to filter out any breaks in the state and to throw an error instead.

To get an induction hypothesis which is strong enough to perform the proof, we can't just assume $jumpNestingOk$ for the empty set and conclude, that no jump at all will be in the resulting state, because the set is altered by the statements *Lab* and *While*.

The wellformedness of the program is used to ensure that for all classinitialisations and methods the nesting of jumps is wellformed, too.

theorem $jumpNestingOk-eval$:
assumes $eval: G \vdash s0 \multimap \rightarrow (v, s1)$

and *jmpOk*: *jumpNestingOk jmps t*
and *wt*: *Env* ⊢ *t* :: *T*
and *wf*: *wf-prog G*
and *G*: *prg Env* = *G*
and *no-jmp*: $\forall j. \text{abrupt } s0 = \text{Some } (\text{Jump } j) \longrightarrow j \in \text{jmps}$
 (**is** ?*Jmp jmps s0*)
shows $(\forall j. \text{fst } s1 = \text{Some } (\text{Jump } j) \longrightarrow j \in \text{jmps}) \wedge$
 (*normal s1* \longrightarrow
 $(\forall w \text{ upd}. v = \text{In2 } (w, \text{upd})$
 $\longrightarrow (\forall s \ j \text{ val}.$
 $\text{abrupt } s \neq \text{Some } (\text{Jump } j) \longrightarrow$
 $\text{abrupt } (\text{upd val } s) \neq \text{Some } (\text{Jump } j))))$
 (**is** ?*Jmp jmps s1* \wedge ?*Upd v s1*)
⟨*proof*⟩

lemmas *jumpNestingOk-evalE* = *jumpNestingOk-eval* [*THEN conjE, rule-format*]

lemma *jumpNestingOk-eval-no-jump*:
assumes *eval*: *prg Env* ⊢ *s0* $\dashv\vdash t \rightarrow (v, s1)$ **and**
 jmpOk: *jumpNestingOk* {*t*} **and**
 no-jmp: *abrupt s0* $\neq \text{Some } (\text{Jump } j)$ **and**
 wt: *Env* ⊢ *t* :: *T* **and**
 wf: *wf-prog (prg Env)*
shows *abrupt s1* $\neq \text{Some } (\text{Jump } j) \wedge$
 (*normal s1* $\longrightarrow v = \text{In2 } (w, \text{upd})$
 $\longrightarrow \text{abrupt } s \neq \text{Some } (\text{Jump } j')$
 $\longrightarrow \text{abrupt } (\text{upd val } s) \neq \text{Some } (\text{Jump } j')$)
⟨*proof*⟩

lemmas *jumpNestingOk-eval-no-jumpE*
= *jumpNestingOk-eval-no-jump* [*THEN conjE, rule-format*]

corollary *eval-expression-no-jump*:
assumes *eval*: *prg Env* ⊢ *s0* $\dashv\vdash e \rightarrow v \rightarrow s1$ **and**
 no-jmp: *abrupt s0* $\neq \text{Some } (\text{Jump } j)$ **and**
 wt: *Env* ⊢ *e* :: $\neg T$ **and**
 wf: *wf-prog (prg Env)*
shows *abrupt s1* $\neq \text{Some } (\text{Jump } j)$
⟨*proof*⟩

corollary *eval-var-no-jump*:
assumes *eval*: *prg Env* ⊢ *s0* $\dashv\vdash \text{var} = \text{In2 } (w, \text{upd}) \rightarrow s1$ **and**
 no-jmp: *abrupt s0* $\neq \text{Some } (\text{Jump } j)$ **and**
 wt: *Env* ⊢ *var* :: *T* **and**
 wf: *wf-prog (prg Env)*
shows *abrupt s1* $\neq \text{Some } (\text{Jump } j) \wedge$
 (*normal s1* \longrightarrow
 $(\text{abrupt } s \neq \text{Some } (\text{Jump } j')$
 $\longrightarrow \text{abrupt } (\text{upd val } s) \neq \text{Some } (\text{Jump } j'))$)
⟨*proof*⟩

lemmas *eval-var-no-jumpE* = *eval-var-no-jump* [*THEN conjE, rule-format*]

corollary *eval-statement-no-jump*:
assumes *eval*: *prg Env* ⊢ *s0* $\dashv\vdash c \rightarrow s1$ **and**
 jmpOk: *jumpNestingOkS* {*c*} **and**
 no-jmp: *abrupt s0* $\neq \text{Some } (\text{Jump } j)$ **and**
 wt: *Env* ⊢ *c* :: $\sqrt{}$ **and**

$wf: wf\text{-}prog\ (prg\ Env)$
shows $abrupt\ s1 \neq Some\ (Jump\ j)$
 $\langle proof \rangle$

corollary *eval-expression-list-no-jump:*

assumes $eval: prg\ Env \vdash s0 \multimap es \multimap v \rightarrow s1$ **and**
 $no\text{-}jmp: abrupt\ s0 \neq Some\ (Jump\ j)$ **and**
 $wt: Env \vdash es :: T$ **and**
 $wf: wf\text{-}prog\ (prg\ Env)$
shows $abrupt\ s1 \neq Some\ (Jump\ j)$
 $\langle proof \rangle$

lemma *union-subseteq-elim* [elim]: $\llbracket A \cup B \subseteq C; \llbracket A \subseteq C; B \subseteq C \rrbracket \Longrightarrow P \rrbracket \Longrightarrow P$
 $\langle proof \rangle$

lemma *dom-locals-halloc-mono:*

assumes $halloc: G \vdash s0 \multimap halloc\ oi \multimap a \rightarrow s1$
shows $dom\ (locals\ (store\ s0)) \subseteq dom\ (locals\ (store\ s1))$
 $\langle proof \rangle$

lemma *dom-locals-sxalloc-mono:*

assumes $sxalloc: G \vdash s0 \multimap sxalloc \rightarrow s1$
shows $dom\ (locals\ (store\ s0)) \subseteq dom\ (locals\ (store\ s1))$
 $\langle proof \rangle$

lemma *dom-locals-assign-mono:*

assumes $f\text{-}ok: dom\ (locals\ (store\ s)) \subseteq dom\ (locals\ (store\ (f\ n\ s)))$
shows $dom\ (locals\ (store\ s)) \subseteq dom\ (locals\ (store\ (assign\ f\ n\ s)))$
 $\langle proof \rangle$

lemma *dom-locals-lvar-mono:*

$dom\ (locals\ (store\ s)) \subseteq dom\ (locals\ (store\ (snd\ (lvar\ vn\ s')\ val\ s)))$
 $\langle proof \rangle$

lemma *dom-locals-fvar-vvar-mono:*

$dom\ (locals\ (store\ s))$
 $\subseteq dom\ (locals\ (store\ (snd\ (fst\ (fvar\ statDeclC\ stat\ fn\ a\ s'))\ val\ s)))$
 $\langle proof \rangle$

lemma *dom-locals-fvar-mono:*

$dom\ (locals\ (store\ s))$
 $\subseteq dom\ (locals\ (store\ (snd\ (fvar\ statDeclC\ stat\ fn\ a\ s))))$
 $\langle proof \rangle$

lemma *dom-locals-avar-vvar-mono:*

$dom\ (locals\ (store\ s))$
 $\subseteq dom\ (locals\ (store\ (snd\ (fst\ (avar\ G\ i\ a\ s'))\ val\ s)))$
 $\langle proof \rangle$

lemma *dom-locals-avar-mono:*

$dom\ (locals\ (store\ s))$
 $\subseteq dom\ (locals\ (store\ (snd\ (avar\ G\ i\ a\ s))))$
 $\langle proof \rangle$

Since assignments are modelled as functions from states to states, we must take into account these functions. They appear only in the assignment rule and as result from evaluating a variable. That's why we need the complicated second part of the conjunction in the goal. The reason for the very generic way to treat assignments was the aim to omit redundancy. There is only one evaluation rule for each kind of variable (locals, fields, arrays). These rules are used for both accessing variables and updating variables. That's why the evaluation rules for variables result in a pair consisting of a value and an update function. Of course we could also think of a pair of a value and a reference in the store, instead of the generic update function. But as only array updates can cause a special exception (if the types mismatch) and not array reads we then have to introduce two different rules to handle array reads and updates

lemma *dom-locals-eval-mono*:

assumes *eval*: $G \vdash s0 \multimap t \rightarrow (v, s1)$
shows $\text{dom } (\text{locals } (\text{store } s0)) \subseteq \text{dom } (\text{locals } (\text{store } s1)) \wedge$
 $(\forall vv. v = \text{In2 } vv \wedge \text{normal } s1$
 $\rightarrow (\forall s \text{ val. } \text{dom } (\text{locals } (\text{store } s))$
 $\subseteq \text{dom } (\text{locals } (\text{store } ((\text{snd } vv) \text{ val } s))))$

<proof>

lemma *dom-locals-eval-mono-elim*:

assumes *eval*: $G \vdash s0 \multimap t \rightarrow (v, s1)$
obtains $\text{dom } (\text{locals } (\text{store } s0)) \subseteq \text{dom } (\text{locals } (\text{store } s1))$ **and**
 $\bigwedge vv \ s \ \text{val. } \llbracket v = \text{In2 } vv; \text{normal } s1 \rrbracket$
 $\implies \text{dom } (\text{locals } (\text{store } s))$
 $\subseteq \text{dom } (\text{locals } (\text{store } ((\text{snd } vv) \text{ val } s)))$

<proof>

lemma *halloc-no-abrupt*:

assumes *halloc*: $G \vdash s0 \multimap \text{halloc } oi \rightarrow a \rightarrow s1$ **and**
normal: $\text{normal } s1$
shows $\text{normal } s0$

<proof>

lemma *sxalloc-mono-no-abrupt*:

assumes *sxalloc*: $G \vdash s0 \multimap \text{sxalloc} \rightarrow s1$ **and**
normal: $\text{normal } s1$
shows $\text{normal } s0$

<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 *subseq-union-transl* [*trans*]: $\llbracket A \subseteq B; B \cup C \subseteq D \rrbracket \implies A \cup C \subseteq D$

<proof>

lemma *subseq-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: $G \vdash s0 \multimap t \rightarrow (v, s1)$ **and**
normal: *normal* *s1*
shows *assigns* $t \subseteq \text{dom} (\text{locals} (\text{store } s1))$
 ⟨*proof*⟩

corollary *assignsE-good-approx*:
assumes
eval: $\text{prg } Env \vdash s0 \multimap e \multimap v \rightarrow s1$ **and**
normal: *normal* *s1*
shows *assignsE* $e \subseteq \text{dom} (\text{locals} (\text{store } s1))$
 ⟨*proof*⟩

corollary *assignsV-good-approx*:
assumes
eval: $\text{prg } Env \vdash s0 \multimap v \multimap vf \rightarrow s1$ **and**
normal: *normal* *s1*
shows *assignsV* $v \subseteq \text{dom} (\text{locals} (\text{store } s1))$
 ⟨*proof*⟩

corollary *assignsEs-good-approx*:
assumes
eval: $\text{prg } Env \vdash s0 \multimap es \multimap vs \rightarrow s1$ **and**
normal: *normal* *s1*
shows *assignsEs* $es \subseteq \text{dom} (\text{locals} (\text{store } s1))$
 ⟨*proof*⟩

lemma *constVal-eval*:
assumes *const*: *constVal* $e = \text{Some } c$ **and**
eval: $G \vdash \text{Norm } s0 \multimap e \multimap v \rightarrow s$
shows $v = c \wedge \text{normal } s$
 ⟨*proof*⟩

lemmas *constVal-eval-elim* = *constVal-eval* [*THEN conjE*]

lemma *eval-unop-type*:
typeof $dt (\text{eval-unop } unop \ v) = \text{Some } (\text{PrimT } (\text{unop-type } unop))$
 ⟨*proof*⟩

lemma *eval-binop-type*:
typeof $dt (\text{eval-binop } binop \ v1 \ v2) = \text{Some } (\text{PrimT } (\text{binop-type } binop))$
 ⟨*proof*⟩

lemma *constVal-Boolean*:
assumes *const*: *constVal* $e = \text{Some } c$ **and**
wt: $Env \vdash e :: \neg \text{PrimT Boolean}$
shows *typeof* *empty-dt* $c = \text{Some } (\text{PrimT Boolean})$
 ⟨*proof*⟩

lemma *assigns-if-good-approx*:
assumes
eval: $\text{prg } Env \vdash s0 \multimap e \multimap b \rightarrow s1$ **and**
normal: *normal* *s1* **and**
bool: $Env \vdash e :: \neg \text{PrimT Boolean}$
shows *assigns-if* $(\text{the-Bool } b) \ e \subseteq \text{dom} (\text{locals} (\text{store } s1))$
 ⟨*proof*⟩

lemma *assigns-if-good-approx'*:
assumes *eval*: $G \vdash s0 \multimap e \multimap b \rightarrow s1$

and *normal*: *normal s1*
and *bool*: $(\llbracket \text{prg} = G, \text{cls} = C, \text{lcl} = L \rrbracket \vdash e :: - \text{ (PrimT Boolean)})$
shows *assigns-if* (*the-Bool b*) $e \subseteq \text{dom} (\text{locals} (\text{store } s1))$
 $\langle \text{proof} \rangle$

lemma *subset-Intl*: $A \subseteq C \implies A \cap B \subseteq C$
 $\langle \text{proof} \rangle$

lemma *subset-Intr*: $B \subseteq C \implies A \cap B \subseteq C$
 $\langle \text{proof} \rangle$

lemma *da-good-approx*:
assumes *eval*: $\text{prg } \text{Env} \vdash s0 \dashv\rightarrow (v, s1)$ **and**
wt: $\text{Env} \vdash t :: T$ (**is** *?Wt Env t T*) **and**
da: $\text{Env} \vdash \text{dom} (\text{locals} (\text{store } s0)) \gg t \gg A$ (**is** *?Da Env s0 t A*) **and**
wf: *wf-prog* (*prg Env*)
shows $(\text{normal } s1 \implies (\text{nrm } A \subseteq \text{dom} (\text{locals} (\text{store } s1)))) \wedge$
 $(\forall l. \text{abrupt } s1 = \text{Some } (\text{Jump } (\text{Break } l)) \wedge \text{normal } s0$
 $\implies (\text{brk } A l \subseteq \text{dom} (\text{locals} (\text{store } s1)))) \wedge$
 $(\text{abrupt } s1 = \text{Some } (\text{Jump } \text{Ret}) \wedge \text{normal } s0$
 $\implies \text{Result} \in \text{dom} (\text{locals} (\text{store } s1)))$
(is *?NormalAssigned s1 A* \wedge *?BreakAssigned s0 s1 A* \wedge *?ResAssigned s0 s1*)
 $\langle \text{proof} \rangle$

lemma *da-good-approxE*:
assumes
 $\text{prg } \text{Env} \vdash s0 \dashv\rightarrow (v, s1)$ **and** $\text{Env} \vdash t :: T$ **and**
 $\text{Env} \vdash \text{dom} (\text{locals} (\text{store } s0)) \gg t \gg A$ **and** *wf-prog* (*prg Env*)
obtains
 $\text{normal } s1 \implies \text{nrm } A \subseteq \text{dom} (\text{locals} (\text{store } s1))$ **and**
 $\bigwedge l. \llbracket \text{abrupt } s1 = \text{Some } (\text{Jump } (\text{Break } l)); \text{normal } s0 \rrbracket$
 $\implies \text{brk } A l \subseteq \text{dom} (\text{locals} (\text{store } s1))$ **and**
 $\llbracket \text{abrupt } s1 = \text{Some } (\text{Jump } \text{Ret}); \text{normal } s0 \rrbracket \implies \text{Result} \in \text{dom} (\text{locals} (\text{store } s1))$
 $\langle \text{proof} \rangle$

lemma *da-good-approxE'*:
assumes *eval*: $G \vdash s0 \dashv\rightarrow (v, s1)$
and *wt*: $(\llbracket \text{prg} = G, \text{cls} = C, \text{lcl} = L \rrbracket \vdash t :: T)$
and *da*: $(\llbracket \text{prg} = G, \text{cls} = C, \text{lcl} = L \rrbracket \vdash \text{dom} (\text{locals} (\text{store } s0)) \gg t \gg A)$
and *wf*: *wf-prog G*
obtains $\text{normal } s1 \implies \text{nrm } A \subseteq \text{dom} (\text{locals} (\text{store } s1))$ **and**
 $\bigwedge l. \llbracket \text{abrupt } s1 = \text{Some } (\text{Jump } (\text{Break } l)); \text{normal } s0 \rrbracket$
 $\implies \text{brk } A l \subseteq \text{dom} (\text{locals} (\text{store } s1))$ **and**
 $\llbracket \text{abrupt } s1 = \text{Some } (\text{Jump } \text{Ret}); \text{normal } s0 \rrbracket$
 $\implies \text{Result} \in \text{dom} (\text{locals} (\text{store } s1))$
 $\langle \text{proof} \rangle$

declare $\llbracket \text{simproc add: wt-expr wt-var wt-exprs wt-stmt} \rrbracket$

end

Chapter 19

TypeSafe

1 The type soundness proof for Java

theory *TypeSafe*

imports *DefiniteAssignmentCorrect Conform*

begin

error free

lemma *error-free-halloc*:

assumes *halloc*: $G \vdash s0 \text{ --halloc } oi \succ a \rightarrow s1$ **and**

error-free-s0: *error-free* *s0*

shows *error-free* *s1*

$\langle \text{proof} \rangle$

lemma *error-free-sxalloc*:

assumes *sxalloc*: $G \vdash s0 \text{ --sxalloc} \rightarrow s1$ **and** *error-free-s0*: *error-free* *s0*

shows *error-free* *s1*

$\langle \text{proof} \rangle$

lemma *error-free-check-field-access-eq*:

error-free (*check-field-access* *G accC statDeclC fn stat a s*)

\implies (*check-field-access* *G accC statDeclC fn stat a s*) = *s*

$\langle \text{proof} \rangle$

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*

$\langle \text{proof} \rangle$

lemma *error-free-FVar-lemma*:

error-free *s*

\implies *error-free* (*abupd* (*if stat then id else np a*) *s*)

$\langle \text{proof} \rangle$

lemma *error-free-init-lvars* [*simp,intro*]:

error-free *s* \implies

error-free (*init-lvars* *G C sig mode a pvs s*)

$\langle \text{proof} \rangle$

lemma *error-free-LVar-lemma*:

error-free *s* \implies *error-free* (*assign* ($\lambda v. \text{supd lupd}(vn \mapsto v)$) *w s*)

$\langle \text{proof} \rangle$

lemma *error-free-throw* [*simp,intro*]:

$error\text{-}free\ s \implies error\text{-}free\ (abupd\ (throw\ x)\ s)$
 $\langle proof \rangle$

result conformance

definition

$assign\text{-}conforms :: st \Rightarrow (val \Rightarrow state \Rightarrow state) \Rightarrow ty \Rightarrow env' \Rightarrow bool\ (\langle \cdot \leq | \cdot \leq :: \cdot \rangle [71, 71, 71, 71]\ 70)$
where
 $s \leq | f \leq T :: \leq E =$
 $((\forall s' w. Norm\ s' :: \leq E \longrightarrow fst\ E, s \vdash w :: \leq T \longrightarrow s \leq | s' \longrightarrow assign\ f\ w\ (Norm\ s') :: \leq E) \wedge$
 $(\forall s' w. error\text{-}free\ s' \longrightarrow (error\text{-}free\ (assign\ f\ w\ s'))))$

definition

$rconf :: prog \Rightarrow lenv \Rightarrow st \Rightarrow term \Rightarrow vals \Rightarrow tys \Rightarrow bool\ (\langle \cdot, \cdot, \cdot, \cdot, \cdot, \cdot :: \cdot \rangle [71, 71, 71, 71, 71, 71]\ 70)$
where
 $G, L, s \vdash t \succ v :: \leq T =$
 $(case\ T\ of$
 $\quad Inl\ T \Rightarrow if\ (\exists\ var. t = In2\ var)$
 $\quad \quad then\ (\forall\ n. (the\text{-}In2\ t) = LVar\ n$
 $\quad \quad \quad \longrightarrow (fst\ (the\text{-}In2\ v) = the\ (locals\ s\ n)) \wedge$
 $\quad \quad \quad (locals\ s\ n \neq None \longrightarrow G, s \vdash fst\ (the\text{-}In2\ v) :: \leq T)) \wedge$
 $\quad \quad (\neg (\exists\ n. the\text{-}In2\ t = LVar\ n) \longrightarrow (G, s \vdash fst\ (the\text{-}In2\ v) :: \leq T)) \wedge$
 $\quad \quad (s \leq | snd\ (the\text{-}In2\ v) \leq T :: \leq (G, L))$
 $\quad \quad else\ G, s \vdash the\text{-}In1\ v :: \leq T$
 $\quad | Inr\ Ts \Rightarrow list\text{-}all2\ (conf\ G\ s)\ (the\text{-}In3\ v)\ Ts)$

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 $rconf\text{-}In1$ [simp]:

$G, L, s \vdash In1\ ec \succ In1\ v :: \leq Inl\ T = G, s \vdash v :: \leq T$
 $\langle proof \rangle$

lemma $rconf\text{-}In2\text{-}no\text{-}LVar$ [simp]:

$\forall\ n. va \neq LVar\ n \implies$
 $G, L, s \vdash In2\ va \succ In2\ vf :: \leq Inl\ T = (G, s \vdash fst\ vf :: \leq T \wedge s \leq | snd\ vf \leq T :: \leq (G, L))$
 $\langle proof \rangle$

lemma $rconf\text{-}In2\text{-}LVar$ [simp]:

$va = LVar\ n \implies$
 $G, L, s \vdash In2\ va \succ In2\ vf :: \leq Inl\ T$
 $= ((fst\ vf = the\ (locals\ s\ n)) \wedge$
 $(locals\ s\ n \neq None \longrightarrow G, s \vdash fst\ vf :: \leq T) \wedge s \leq | snd\ vf \leq T :: \leq (G, L))$
 $\langle proof \rangle$

lemma $rconf\text{-}In3$ [simp]:

$G, L, \vdash \text{In}3 \text{ es} \succ \text{In}3 \text{ vs} :: \preceq \text{In}r \text{ Ts} = \text{list-all2 } (\lambda v \text{ T. } G, \vdash v :: \preceq T) \text{ vs } \text{Ts}$
 $\langle \text{proof} \rangle$

fits and conf

lemma *conf-fits*: $G, \vdash v :: \preceq T \implies G, \vdash v \text{ fits } T$
 $\langle \text{proof} \rangle$

lemma *fits-conf*:
 $\llbracket G, \vdash v :: \preceq T; G \vdash T \preceq? T'; G, \vdash v \text{ fits } T'; \text{ws-prog } G \rrbracket \implies G, \vdash v :: \preceq T'$
 $\langle \text{proof} \rangle$

lemma *fits-Array*:
 $\llbracket G, \vdash v :: \preceq T; G \vdash T'.[] \preceq T.[]; G, \vdash v \text{ fits } T'; \text{ws-prog } G \rrbracket \implies G, \vdash v :: \preceq T'$
 $\langle \text{proof} \rangle$

gext

lemma *halloc-gext*: $\bigwedge s1 \text{ s2. } G \vdash s1 \text{ -halloc } oi \succ a \rightarrow s2 \implies \text{snd } s1 \leq | \text{snd } s2$
 $\langle \text{proof} \rangle$

lemma *sxalloc-gext*: $\bigwedge s1 \text{ s2. } G \vdash s1 \text{ -sxalloc } \rightarrow s2 \implies \text{snd } s1 \leq | \text{snd } s2$
 $\langle \text{proof} \rangle$

lemma *eval-gext-lemma* [rule-format (no-asm)]:
 $G \vdash s \text{ -t} \rightarrow (w, s') \implies \text{snd } s \leq | \text{snd } s' \wedge (\text{case } w \text{ of}$
 $\quad \text{In1 } v \Rightarrow \text{True}$
 $\quad | \text{In2 } vf \Rightarrow \text{normal } s \longrightarrow (\forall v \text{ x s. } s \leq | \text{snd } (\text{assign } (\text{snd } vf) \text{ v } (x, s)))$
 $\quad | \text{In3 } vs \Rightarrow \text{True})$
 $\langle \text{proof} \rangle$

lemma *evar-gext-f*:
 $G \vdash \text{Norm } s1 \text{ -e} \succ vf \rightarrow s2 \implies s \leq | \text{snd } (\text{assign } (\text{snd } vf) \text{ v } (x, s))$
 $\langle \text{proof} \rangle$

lemmas *eval-gext* = *eval-gext-lemma* [THEN conjunct1]

lemma *eval-gext'*: $G \vdash (x1, s1) \text{ -t} \rightarrow (w, (x2, s2)) \implies s1 \leq | s2$
 $\langle \text{proof} \rangle$

lemma *init-yields-initd*: $G \vdash \text{Norm } s1 \text{ -Init } C \rightarrow s2 \implies \text{initd } C \text{ s2}$
 $\langle \text{proof} \rangle$

Lemmas

lemma *obj-ty-obj-class1*:
 $\llbracket \text{wf-prog } G; \text{is-type } G \text{ (obj-ty obj)} \rrbracket \implies \text{is-class } G \text{ (obj-class obj)}$
 $\langle \text{proof} \rangle$

lemma *oconf-init-obj*:
 $\llbracket \text{wf-prog } G;$
 $\quad (\text{case } r \text{ of Heap } a \Rightarrow \text{is-type } G \text{ (obj-ty obj)} \mid \text{Stat } C \Rightarrow \text{is-class } G \text{ C})$
 $\rrbracket \implies G, \vdash \text{obj } \llbracket \text{values} := \text{init-vals } (\text{var-tys } G \text{ (tag obj } r)) \rrbracket :: \preceq \sqrt{r}$
 $\langle \text{proof} \rangle$

lemma *conforms-newG*: $\llbracket \text{globs } s \text{ oref} = \text{None}; (x, s) :: \preceq (G, L);$
 $\text{wf-prog } G; \text{case oref of Heap } a \Rightarrow \text{is-type } G \text{ (obj-ty } \llbracket \text{tag} = oi, \text{values} = vs \rrbracket)$
 $\quad \mid \text{Stat } C \Rightarrow \text{is-class } G \text{ C} \rrbracket \implies$
 $(x, \text{init-obj } G \text{ oi oref } s) :: \preceq (G, L)$

$\langle \text{proof} \rangle$

lemma *conforms-init-class-obj*:

$\llbracket (x,s)::\preceq(G, L); \text{wf-prog } G; \text{class } G \ C=\text{Some } y; \neg \text{inited } C \ (\text{globs } s) \rrbracket \implies$
 $(x, \text{init-class-obj } G \ C \ s)::\preceq(G, L)$
 $\langle \text{proof} \rangle$

lemma *fst-init-lvars[simp]*:

$\text{fst} \ (\text{init-lvars } G \ C \ \text{sig} \ (\text{invmode } m \ e) \ a' \ \text{pvs} \ (x,s)) =$
 $(\text{if } \text{is-static } m \text{ then } x \text{ else } (\text{np } a') \ x)$
 $\langle \text{proof} \rangle$

lemma *halloc-conforms*: $\bigwedge s1. \llbracket G \vdash s1 \text{ -halloc } oi \succ a \rightarrow s2; \text{wf-prog } G; s1::\preceq(G, L);$
 $\text{is-type } G \ (\text{obj-ty} \ (\text{tag}=oi, \text{values}=fs)) \rrbracket \implies s2::\preceq(G, L)$
 $\langle \text{proof} \rangle$

lemma *halloc-type-sound*:

$\bigwedge s1. \llbracket G \vdash s1 \text{ -halloc } oi \succ a \rightarrow (x,s); \text{wf-prog } G; s1::\preceq(G, L);$
 $T = \text{obj-ty} \ (\text{tag}=oi, \text{values}=fs); \text{is-type } G \ T \rrbracket \implies$
 $(x,s)::\preceq(G, L) \wedge (x = \text{None} \longrightarrow G, s \vdash \text{Addr } a::\preceq T)$
 $\langle \text{proof} \rangle$

lemma *salloc-type-sound*:

$\bigwedge s1 \ s2. \llbracket G \vdash s1 \text{ -salloc } \rightarrow s2; \text{wf-prog } G \rrbracket \implies$
 $\text{case } \text{fst } s1 \text{ of}$
 $\quad \text{None} \Rightarrow s2 = s1$
 $\quad | \text{Some } \text{abr} \Rightarrow (\text{case } \text{abr} \text{ of}$
 $\quad \quad \text{Xcpt } x \Rightarrow (\exists a. \text{fst } s2 = \text{Some}(\text{Xcpt } (\text{Loc } a)) \wedge$
 $\quad \quad (\forall L. s1::\preceq(G, L) \longrightarrow s2::\preceq(G, L)))$
 $\quad \quad | \text{Jump } j \Rightarrow s2 = s1$
 $\quad \quad | \text{Error } e \Rightarrow s2 = s1)$
 $\langle \text{proof} \rangle$

lemma *wt-init-comp-ty*:

$\text{is-acc-type } G \ (\text{pid } C) \ T \implies (\text{prg}=G, \text{cls}=C, \text{lcl}=L) \vdash \text{init-comp-ty } T::\checkmark$
 $\langle \text{proof} \rangle$

declare *fun-upd-same* [simp]

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

definition

$\text{DynT-prop} :: [\text{prog}, \text{inv-mode}, \text{qtname}, \text{ref-ty}] \Rightarrow \text{bool} \ (\hookrightarrow \vdash \dashv \rightarrow \preceq \dashv \rightarrow [71, 71, 71, 71] 70)$
where
 $G \vdash \text{mode} \rightarrow D \preceq t = (\text{mode} = \text{IntVir} \longrightarrow \text{is-class } G \ D \wedge$
 $(\text{if } (\exists T. t = \text{ArrayT } T) \text{ then } D = \text{Object} \text{ else } G \vdash \text{Class } D \preceq \text{RefT } t))$

lemma *DynT-propI*:

$\llbracket (x,s)::\preceq(G, L); G, s \vdash a'::\preceq \text{RefT } \text{statT}; \text{wf-prog } G; \text{mode} = \text{IntVir} \longrightarrow a' \neq \text{Null} \rrbracket$
 $\implies G \vdash \text{mode} \rightarrow \text{invocation-class } \text{mode } s \ a' \ \text{statT} \preceq \text{statT}$
 $\langle \text{proof} \rangle$

lemma *invocation-methd*:

$\llbracket \text{wf-prog } G; \text{statT} \neq \text{NullT};$
 $(\forall \text{statC}. \text{statT} = \text{ClassT } \text{statC} \longrightarrow \text{is-class } G \ \text{statC});$
 $(\forall I. \text{statT} = \text{IfaceT } I \longrightarrow \text{is-iface } G \ I \wedge \text{mode} \neq \text{SuperM});$

$(\forall T. \text{stat}T = \text{Array}T \longrightarrow \text{mode} \neq \text{Super}M);$
 $G \vdash \text{mode} \rightarrow \text{invocation-class mode } s \text{ a' stat}T \preceq \text{stat}T;$
 $\text{dynlookup } G \text{ stat}T (\text{invocation-class mode } s \text{ a' stat}T) \text{ sig} = \text{Some } m \parallel$
 $\implies \text{methd } G (\text{invocation-declclass } G \text{ mode } s \text{ a' stat}T \text{ sig}) \text{ sig} = \text{Some } m$
 $\langle \text{proof} \rangle$

lemma *DynT-mheadsD*:

$\parallel G \vdash \text{invmode } sm \text{ e} \rightarrow \text{inv}C \preceq \text{stat}T;$
 $\text{wf-prog } G; (\text{prg}=G, \text{cls}=C, \text{lcl}=L) \vdash e :: -\text{Ref}T \text{ stat}T;$
 $(\text{statDecl}T, sm) \in \text{mheads } G \text{ C stat}T \text{ sig};$
 $\text{inv}C = \text{invocation-class } (\text{invmode } sm \text{ e}) \text{ s a' stat}T;$
 $\text{decl}C = \text{invocation-declclass } G (\text{invmode } sm \text{ e}) \text{ s a' stat}T \text{ sig}$
 $\parallel \implies$
 $\exists dm.$
 $\text{methd } G \text{ decl}C \text{ sig} = \text{Some } dm \wedge \text{dynlookup } G \text{ stat}T \text{ inv}C \text{ sig} = \text{Some } dm \wedge$
 $G \vdash \text{resTy } (mthd \text{ dm}) \preceq \text{resTy } sm \wedge$
 $\text{wf-mdecl } G \text{ decl}C (\text{sig}, mthd \text{ dm}) \wedge$
 $\text{decl}C = \text{declclass } dm \wedge$
 $\text{is-static } dm = \text{is-static } sm \wedge$
 $\text{is-class } G \text{ inv}C \wedge \text{is-class } G \text{ decl}C \wedge G \vdash \text{inv}C \preceq_C \text{ decl}C \wedge$
 $(\text{if invmode } sm \text{ e} = \text{IntVir}$
 $\text{ then } (\forall \text{ stat}C. \text{stat}T = \text{Class}T \text{ stat}C \longrightarrow G \vdash \text{inv}C \preceq_C \text{ stat}C)$
 $\text{ else } ((\exists \text{ stat}C. \text{stat}T = \text{Class}T \text{ stat}C \wedge G \vdash \text{stat}C \preceq_C \text{ decl}C)$
 $\quad \vee (\forall \text{ stat}C. \text{stat}T \neq \text{Class}T \text{ stat}C \wedge \text{decl}C = \text{Object})) \wedge$
 $\quad \text{statDecl}T = \text{Class}T (\text{declclass } dm))$
 $\langle \text{proof} \rangle$

corollary *DynT-mheadsE* [consumes 7]:

— Same as *DynT-mheadsD* but better suited for application in typesafety proof

assumes *invC-compatible*: $G \vdash \text{mode} \rightarrow \text{inv}C \preceq \text{stat}T$

and *wf*: $\text{wf-prog } G$
and *wt-e*: $(\text{prg}=G, \text{cls}=C, \text{lcl}=L) \vdash e :: -\text{Ref}T \text{ stat}T$
and *mheads*: $(\text{statDecl}T, sm) \in \text{mheads } G \text{ C stat}T \text{ sig}$
and *mode*: $\text{mode} = \text{invmode } sm \text{ e}$
and *invC*: $\text{inv}C = \text{invocation-class mode } s \text{ a' stat}T$
and *declC*: $\text{decl}C = \text{invocation-declclass } G \text{ mode } s \text{ a' stat}T \text{ sig}$
and *dm*: $\bigwedge dm. \parallel \text{methd } G \text{ decl}C \text{ sig} = \text{Some } dm;$
 $\text{dynlookup } G \text{ stat}T \text{ inv}C \text{ sig} = \text{Some } dm;$
 $G \vdash \text{resTy } (mthd \text{ dm}) \preceq \text{resTy } sm;$
 $\text{wf-mdecl } G \text{ decl}C (\text{sig}, mthd \text{ dm});$
 $\text{decl}C = \text{declclass } dm;$
 $\text{is-static } dm = \text{is-static } sm;$
 $\text{is-class } G \text{ inv}C; \text{is-class } G \text{ decl}C; G \vdash \text{inv}C \preceq_C \text{ decl}C;$
 $(\text{if invmode } sm \text{ e} = \text{IntVir}$
 $\text{ then } (\forall \text{ stat}C. \text{stat}T = \text{Class}T \text{ stat}C \longrightarrow G \vdash \text{inv}C \preceq_C \text{ stat}C)$
 $\text{ else } ((\exists \text{ stat}C. \text{stat}T = \text{Class}T \text{ stat}C \wedge G \vdash \text{stat}C \preceq_C \text{ decl}C)$
 $\quad \vee (\forall \text{ stat}C. \text{stat}T \neq \text{Class}T \text{ stat}C \wedge \text{decl}C = \text{Object})) \wedge$
 $\quad \text{statDecl}T = \text{Class}T (\text{declclass } dm)) \parallel \implies P$

shows *P*

$\langle \text{proof} \rangle$

lemma *DynT-conf*: $\parallel G \vdash \text{invocation-class mode } s \text{ a' stat}T \preceq_C \text{ decl}C; \text{wf-prog } G;$
 $\text{isrtype } G (\text{stat}T);$

$G, s \vdash a' :: \preceq \text{Ref}T \text{ stat}T; \text{mode} = \text{IntVir} \longrightarrow a' \neq \text{Null};$
 $\text{mode} \neq \text{IntVir} \longrightarrow ((\exists \text{ stat}C. \text{stat}T = \text{Class}T \text{ stat}C \wedge G \vdash \text{stat}C \preceq_C \text{ decl}C)$
 $\quad \vee (\forall \text{ stat}C. \text{stat}T \neq \text{Class}T \text{ stat}C \wedge \text{decl}C = \text{Object})) \parallel$

$\implies G, s \vdash a' :: \preceq \text{Class } \text{decl}C$

$\langle \text{proof} \rangle$

lemma *Ass-lemma*:

$\llbracket G \vdash \text{Norm } s0 \text{ } -\text{var} \Rightarrow (w, f) \rightarrow \text{Norm } s1; G \vdash \text{Norm } s1 \text{ } -e -\succ v \rightarrow \text{Norm } s2;$
 $G, s2 \vdash v :: \preceq eT; s1 \leq s2 \longrightarrow \text{assign } f \ v \ (\text{Norm } s2) :: \preceq (G, L) \rrbracket$
 $\implies \text{assign } f \ v \ (\text{Norm } s2) :: \preceq (G, L) \wedge$
 $(\text{normal } (\text{assign } f \ v \ (\text{Norm } s2))) \longrightarrow G, \text{store } (\text{assign } f \ v \ (\text{Norm } s2)) \vdash v :: \preceq eT$
 $\langle \text{proof} \rangle$

lemma *Throw-lemma*: $\llbracket G \vdash tn \preceq_C \text{SXcpt Throwable}; \text{wf-prog } G; (x1, s1) :: \preceq (G, L);$
 $x1 = \text{None} \longrightarrow G, s1 \vdash a' :: \preceq \text{Class } tn \rrbracket \implies (\text{throw } a' \ x1, s1) :: \preceq (G, L)$
 $\langle \text{proof} \rangle$

lemma *Try-lemma*: $\llbracket G \vdash \text{obj-ty } (\text{the } (\text{globs } s1' \ (\text{Heap } a))) \preceq \text{Class } tn;$
 $(\text{Some } (\text{Xcpt } (\text{Loc } a)), s1') :: \preceq (G, L); \text{wf-prog } G \rrbracket$
 $\implies \text{Norm } (\text{lupd}(vn \mapsto \text{Addr } a) \ s1') :: \preceq (G, L(vn \mapsto \text{Class } tn))$
 $\langle \text{proof} \rangle$

lemma *Fin-lemma*:

$\llbracket G \vdash \text{Norm } s1 \text{ } -c2 \rightarrow (x2, s2); \text{wf-prog } G; (\text{Some } a, s1) :: \preceq (G, L); (x2, s2) :: \preceq (G, L);$
 $\text{dom } (\text{locals } s1) \subseteq \text{dom } (\text{locals } s2) \rrbracket$
 $\implies (\text{abrupt-if } \text{True } (\text{Some } a) \ x2, s2) :: \preceq (G, L)$
 $\langle \text{proof} \rangle$

lemma *FVar-lemma1*:

$\llbracket \text{table-of } (\text{DeclConcepts.fields } G \ \text{statC}) \ (fn, \text{statDeclC}) = \text{Some } f ;$
 $x2 = \text{None} \longrightarrow G, s2 \vdash a :: \preceq \text{Class } \text{statC}; \text{wf-prog } G; G \vdash \text{statC} \preceq_C \text{statDeclC};$
 $\text{statDeclC} \neq \text{Object};$
 $\text{class } G \ \text{statDeclC} = \text{Some } y; (x2, s2) :: \preceq (G, L); s1 \leq s2;$
 $\text{inited } \text{statDeclC} \ (\text{globs } s1);$
 $(\text{if static } f \text{ then id else np } a) \ x2 = \text{None} \rrbracket$
 \implies
 $\exists \text{obj. } \text{globs } s2 \ (\text{if static } f \text{ then Inr } \text{statDeclC} \text{ else Inl } (\text{the-Addr } a))$
 $= \text{Some obj} \wedge$
 $\text{var-tys } G \ (\text{tag obj}) \ (\text{if static } f \text{ then Inr } \text{statDeclC} \text{ else Inl } (\text{the-Addr } a))$
 $(\text{Inl}(fn, \text{statDeclC})) = \text{Some } (\text{type } f)$
 $\langle \text{proof} \rangle$

lemma *FVar-lemma2*: *error-free state*

$\implies \text{error-free}$
 $(\text{assign}$
 $(\lambda v. \text{supd}$
 $(\text{upd-gobj}$
 $(\text{if static field then Inr } \text{statDeclC}$
 $\text{else Inl } (\text{the-Addr } a))$
 $(\text{Inl } (fn, \text{statDeclC})) \ v))$
 $w \ \text{state})$
 $\langle \text{proof} \rangle$

declare *split-paired-All* [simp del] *split-paired-Ex* [simp del]

declare *if-split* [split del] *if-split-asm* [split del]
 $\text{option.split} \ [\text{split del}] \ \text{option.split-asm} \ [\text{split del}]$
 $\langle \text{ML} \rangle$

lemma *FVar-lemma*:

$\llbracket ((v, f), \text{Norm } s2') = \text{fvar } \text{statDeclC} \ (\text{static field}) \ fn \ a \ (x2, s2);$
 $G \vdash \text{statC} \preceq_C \text{statDeclC};$
 $\text{table-of } (\text{DeclConcepts.fields } G \ \text{statC}) \ (fn, \text{statDeclC}) = \text{Some field};$
 $\text{wf-prog } G;$
 $x2 = \text{None} \longrightarrow G, s2 \vdash a :: \preceq \text{Class } \text{statC};$

$statDeclC \neq Object$; $class\ G\ statDeclC = Some\ y$;
 $(x2, s2) :: \preceq (G, L)$; $s1 \leq s2$; $initd\ statDeclC\ (globs\ s1) \implies$
 $G, s2 \vdash v :: \preceq type\ field \wedge s2' \leq f \preceq type\ field :: \preceq (G, L)$
 $\langle proof \rangle$
declare *split-paired-All* [simp] *split-paired-Ex* [simp]
declare *if-split* [split] *if-split-asm* [split]
 $option.split$ [split] $option.split-asm$ [split]
 $\langle ML \rangle$

lemma *AVar-lemma1*: $\llbracket globs\ s\ (Inl\ a) = Some\ obj; tag\ obj = Arr\ ty\ i;$
 $the-Intg\ i' \text{ in-bounds } i; wf-prog\ G; G \vdash ty.[] \preceq Tb.[]; Norm\ s :: \preceq (G, L)$
 $\rrbracket \implies G, s \vdash the\ ((values\ obj)\ (Inr\ (the-Intg\ i')) :: \preceq Tb$
 $\langle proof \rangle$

lemma *obj-split*: $\exists\ t\ vs.\ obj = \langle tag=t, values=vs \rangle$
 $\langle proof \rangle$

lemma *AVar-lemma2: error-free state*
 $\implies error-free$
 $(assign$
 $(\lambda v\ (x, s').$
 $((raise-if\ (\neg\ G, s \vdash v\ fits\ T)\ ArrStore)\ x,$
 $upd-gobj\ (Inl\ a)\ (Inr\ (the-Intg\ i))\ v\ s'))$
 $w\ state)$
 $\langle proof \rangle$

lemma *AVar-lemma*: $\llbracket wf-prog\ G; G \vdash (x1, s1) -e2-\triangleright i \rightarrow (x2, s2);$
 $((v, f), Norm\ s2') = avar\ G\ i\ a\ (x2, s2); x1 = None \longrightarrow G, s1 \vdash a :: \preceq Ta.[];$
 $(x2, s2) :: \preceq (G, L); s1 \leq s2 \rrbracket \implies G, s2 \vdash v :: \preceq Ta \wedge s2' \leq f \preceq Ta :: \preceq (G, L)$
 $\langle proof \rangle$

Call

lemma *conforms-init-lvars-lemma*: $\llbracket wf-prog\ G;$
 $wf-mhead\ G\ P\ sig\ mh;$
 $list-all2\ (conf\ G\ s)\ pvs\ pTsa; G \vdash pTsa[\preceq](parTs\ sig) \rrbracket \implies$
 $G, s \vdash Map.empty\ (pars\ mh[\mapsto]pvs)$
 $[\sim :: \preceq](table-of\ lvars)(pars\ mh[\mapsto]parTs\ sig)$
 $\langle proof \rangle$

lemma *lconf-map-lname* [simp]:
 $G, s \vdash (case-lname\ l1\ l2)[:: \preceq](case-lname\ L1\ L2)$
 $=$
 $(G, s \vdash l1[:: \preceq]L1 \wedge G, s \vdash (\lambda x :: unit.\ l2)[:: \preceq](\lambda x :: unit.\ L2))$
 $\langle proof \rangle$

lemma *wlconf-map-lname* [simp]:
 $G, s \vdash (case-lname\ l1\ l2)[\sim :: \preceq](case-lname\ L1\ L2)$
 $=$
 $(G, s \vdash l1[\sim :: \preceq]L1 \wedge G, s \vdash (\lambda x :: unit.\ l2)[\sim :: \preceq](\lambda x :: unit.\ L2))$
 $\langle proof \rangle$

lemma *lconf-map-ename* [simp]:
 $G, s \vdash (case-ename\ l1\ l2)[:: \preceq](case-ename\ L1\ L2)$
 $=$
 $(G, s \vdash l1[:: \preceq]L1 \wedge G, s \vdash (\lambda x :: unit.\ l2)[:: \preceq](\lambda x :: unit.\ L2))$
 $\langle proof \rangle$

lemma *wlconf-map-ename* [simp]:
 $G, s \vdash (\text{case-ename } l1 \ l2) [\sim :: \preceq] (\text{case-ename } L1 \ L2)$
 $=$
 $(G, s \vdash l1 [\sim :: \preceq] L1 \wedge G, s \vdash (\lambda x :: \text{unit}. l2) [\sim :: \preceq] (\lambda x :: \text{unit}. L2))$
 <proof>

lemma *defval-conf1* [rule-format (no-asm), elim]:
 $\text{is-type } G \ T \longrightarrow (\exists v \in \text{Some } (\text{default-val } T): G, s \vdash v :: \preceq T)$
 <proof>

lemma *np-no-jump*: $x \neq \text{Some } (\text{Jump } j) \implies (\text{np } a') \ x \neq \text{Some } (\text{Jump } j)$
 <proof>

declare *split-paired-All* [simp del] *split-paired-Ex* [simp del]
declare *if-split* [split del] *if-split-asm* [split del]
option.split [split del] *option.split-asm* [split del]
 <ML>

lemma *conforms-init-lvars*:
 $\llbracket \text{wf-mhead } G \ (\text{pid } \text{declC}) \ \text{sig} \ (\text{mhead } (\text{mthd } dm)); \text{wf-prog } G;$
 $\text{list-all2 } (\text{conf } G \ s) \ \text{pvs } pTsa; \ G \vdash pTsa [\preceq] (\text{parTs } \text{sig});$
 $(x, s) :: \preceq (G, L);$
 $\text{methd } G \ \text{declC} \ \text{sig} = \text{Some } dm;$
 $\text{isrtype } G \ \text{statT};$
 $G \vdash \text{invC} \preceq_C \ \text{declC};$
 $G, s \vdash a' :: \preceq \text{RefT } \text{statT};$
 $\text{invmode } (\text{mhd } sm) \ e = \text{IntVir} \longrightarrow a' \neq \text{Null};$
 $\text{invmode } (\text{mhd } sm) \ e \neq \text{IntVir} \longrightarrow$
 $(\exists \text{statC}. \text{statT} = \text{ClassT } \text{statC} \wedge G \vdash \text{statC} \preceq_C \ \text{declC})$
 $\vee (\forall \text{statC}. \text{statT} \neq \text{ClassT } \text{statC} \wedge \text{declC} = \text{Object});$
 $\text{invC} = \text{invocation-class } (\text{invmode } (\text{mhd } sm) \ e) \ s \ a' \ \text{statT};$
 $\text{declC} = \text{invocation-declclass } G \ (\text{invmode } (\text{mhd } sm) \ e) \ s \ a' \ \text{statT} \ \text{sig};$
 $x \neq \text{Some } (\text{Jump } \text{Ret})$
 $\rrbracket \implies$
 $\text{init-lvars } G \ \text{declC} \ \text{sig} \ (\text{invmode } (\text{mhd } sm) \ e) \ a'$
 $\text{pvs } (x, s) :: \preceq (G, \lambda k.$
 $\quad (\text{case } k \text{ of}$
 $\quad \quad \text{EName } e \Rightarrow (\text{case } e \text{ of}$
 $\quad \quad \quad \text{VName } v$
 $\quad \quad \quad \Rightarrow ((\text{table-of } (\text{lcls } (\text{mbody } (\text{mthd } dm))))$
 $\quad \quad \quad (\text{pars } (\text{mthd } dm) [\mapsto] \text{parTs } \text{sig})) \ v$
 $\quad \quad \quad | \text{Res} \Rightarrow \text{Some } (\text{resTy } (\text{mthd } dm)))$
 $\quad \quad | \text{This} \Rightarrow \text{if } (\text{is-static } (\text{mthd } sm))$
 $\quad \quad \quad \text{then None else Some } (\text{Class } \text{declC})))$
 <proof>

declare *split-paired-All* [simp] *split-paired-Ex* [simp]
declare *if-split* [split] *if-split-asm* [split]
option.split [split] *option.split-asm* [split]
 <ML>

2 accessibility

theorem *dynamic-field-access-ok*:
assumes *wf*: $\text{wf-prog } G$ **and**
 $\text{not-Null}: \neg \text{stat} \longrightarrow a \neq \text{Null}$ **and**
conform-a: $G, (\text{store } s) \vdash a :: \preceq \text{Class } \text{statC}$ **and**

conform-s: $s :: \preceq (G, L)$ **and**
normal-s: *normal s* **and**
wt-e: $(\text{prg} = G, \text{cls} = \text{accC}, \text{lcl} = L) \vdash e :: -\text{Class statC}$ **and**
f: $\text{accfield } G \text{ accC statC fn} = \text{Some } f$ **and**
dynC: if *stat* then $\text{dynC} = \text{declclass } f$
 else $\text{dynC} = \text{obj-class (lookup-obj (store } s) a)$ **and**
stat: if *stat* then $(\text{is-static } f)$ else $(\neg \text{is-static } f)$
shows $\text{table-of (DeclConcepts.fields } G \text{ dynC) (fn, declclass } f) = \text{Some (fld } f) \wedge$
 $G \vdash \text{Field fn } f \text{ in dynC dyn-accessible-from accC}$
 <proof>

lemma *error-free-field-access*:

assumes *accfield*: $\text{accfield } G \text{ accC statC fn} = \text{Some (statDeclC, } f)$ **and**
wt-e: $(\text{prg} = G, \text{cls} = \text{accC}, \text{lcl} = L) \vdash e :: -\text{Class statC}$ **and**
eval-init: $G \vdash \text{Norm } s0 \text{ --Init statDeclC} \rightarrow s1$ **and**
eval-e: $G \vdash s1 \text{ --e} \rightarrow a \rightarrow s2$ **and**
conf-s2: $s2 :: \preceq (G, L)$ **and**
conf-a: $\text{normal } s2 \implies G, \text{store } s2 \vdash a :: \preceq \text{Class statC}$ **and**
fvar: $(v, s2') = \text{fvar statDeclC (is-static } f) \text{ fn } a \text{ } s2$ **and**
wf: *wf-prog G*
shows $\text{check-field-access } G \text{ accC statDeclC fn (is-static } f) a \text{ } s2' = s2'$
 <proof>

lemma *call-access-ok*:

assumes *invC-prop*: $G \vdash \text{invmode statM } e \rightarrow \text{invC} \preceq \text{statT}$
and *wf*: *wf-prog G*
and *wt-e*: $(\text{prg} = G, \text{cls} = C, \text{lcl} = L) \vdash e :: -\text{RefT statT}$
and *statM*: $(\text{statDeclT}, \text{statM}) \in \text{mheads } G \text{ accC statT sig}$
and *invC*: $\text{invC} = \text{invocation-class (invmode statM } e) s a \text{ statT}$
shows $\exists \text{ dynM. dynlookup } G \text{ statT invC sig} = \text{Some dynM} \wedge$
 $G \vdash \text{Methd sig dynM in invC dyn-accessible-from accC}$
 <proof>

lemma *error-free-call-access*:

assumes
eval-args: $G \vdash s1 \text{ --args} \rightarrow vs \rightarrow s2$ **and**
wt-e: $(\text{prg} = G, \text{cls} = \text{accC}, \text{lcl} = L) \vdash e :: -(\text{RefT statT})$ **and**
statM: $\text{max-spec } G \text{ accC statT (name = mn, parTs = pTs)}$
 $= \{((\text{statDeclT}, \text{statM}), pTs')\}$ **and**
conf-s2: $s2 :: \preceq (G, L)$ **and**
conf-a: $\text{normal } s1 \implies G, \text{store } s1 \vdash a :: \preceq \text{RefT statT}$ **and**
invProp: $\text{normal } s3 \implies$
 $G \vdash \text{invmode statM } e \rightarrow \text{invC} \preceq \text{statT}$ **and**
s3: $s3 = \text{init-lvars } G \text{ invDeclC (name = mn, parTs = pTs')}$
 $(\text{invmode statM } e) a \text{ vs } s2$ **and**
invC: $\text{invC} = \text{invocation-class (invmode statM } e) (\text{store } s2) a \text{ statT}$ **and**
invDeclC: $\text{invDeclC} = \text{invocation-declclass } G (\text{invmode statM } e) (\text{store } s2)$
 $a \text{ statT (name = mn, parTs = pTs')}$ **and**
wf: *wf-prog G*
shows $\text{check-method-access } G \text{ accC statT (invmode statM } e) (\text{name=mn, parTs=pTs'}) a \text{ } s3$
 $= s3$
 <proof>

lemma *map-upds-eq-length-append-simp*:

$\bigwedge \text{tab } qs. \text{length } ps = \text{length } qs \implies \text{tab}(ps[\mapsto] qs @ zs) = \text{tab}(ps[\mapsto] qs)$
 <proof>

lemma *map-upds-upd-eq-length-simp*:

$\bigwedge \text{tab } qs \text{ } x \text{ } y. \text{length } ps = \text{length } qs$

$\implies \text{tab}(ps[\mapsto]qs, x \mapsto y) = \text{tab}(ps@[x][\mapsto]qs@[y])$
 $\langle \text{proof} \rangle$

lemma *map-upd-cong*: $\text{tab} = \text{tab}' \implies \text{tab}(x \mapsto y) = \text{tab}'(x \mapsto y)$
 $\langle \text{proof} \rangle$

lemma *map-upd-cong-ext*: $\text{tab } z = \text{tab}' z \implies (\text{tab}(x \mapsto y)) z = (\text{tab}'(x \mapsto y)) z$
 $\langle \text{proof} \rangle$

lemma *map-upds-cong*: $\text{tab} = \text{tab}' \implies \text{tab}(xs[\mapsto]ys) = \text{tab}'(xs[\mapsto]ys)$
 $\langle \text{proof} \rangle$

lemma *map-upds-cong-ext*:
 $\bigwedge \text{tab } \text{tab}' \text{ } ys. \text{tab } z = \text{tab}' z \implies (\text{tab}(xs[\mapsto]ys)) z = (\text{tab}'(xs[\mapsto]ys)) z$
 $\langle \text{proof} \rangle$

lemma *map-upd-override*: $(\text{tab}(x \mapsto y)) x = (\text{tab}'(x \mapsto y)) x$
 $\langle \text{proof} \rangle$

lemma *map-upds-eq-length-suffix*: $\bigwedge \text{tab } qs.$
 $\text{length } ps = \text{length } qs \implies \text{tab}(ps@xs[\mapsto]qs) = \text{tab}(ps[\mapsto]qs, xs[\mapsto][\])$
 $\langle \text{proof} \rangle$

lemma *map-upds-upds-eq-length-prefix-simp*:
 $\bigwedge \text{tab } qs. \text{length } ps = \text{length } qs$
 $\implies \text{tab}(ps[\mapsto]qs, xs[\mapsto]ys) = \text{tab}(ps@xs[\mapsto]qs@ys)$
 $\langle \text{proof} \rangle$

lemma *map-upd-cut-irrelevant*:
 $\llbracket (\text{tab}(x \mapsto y)) \text{ } vn = \text{Some } el; (\text{tab}'(x \mapsto y)) \text{ } vn = \text{None} \rrbracket$
 $\implies \text{tab } vn = \text{Some } el$
 $\langle \text{proof} \rangle$

lemma *map-upd-Some-expand*:
 $\llbracket \text{tab } vn = \text{Some } z \rrbracket$
 $\implies \exists z. (\text{tab}(x \mapsto y)) \text{ } vn = \text{Some } z$
 $\langle \text{proof} \rangle$

lemma *map-upds-Some-expand*:
 $\bigwedge \text{tab } ys \text{ } z. \llbracket \text{tab } vn = \text{Some } z \rrbracket$
 $\implies \exists z. (\text{tab}(xs[\mapsto]ys)) \text{ } vn = \text{Some } z$
 $\langle \text{proof} \rangle$

lemma *map-upd-Some-swap*:
 $(\text{tab}(r \mapsto w, u \mapsto v)) \text{ } vn = \text{Some } z \implies \exists z. (\text{tab}(u \mapsto v, r \mapsto w)) \text{ } vn = \text{Some } z$
 $\langle \text{proof} \rangle$

lemma *map-upd-None-swap*:
 $(\text{tab}(r \mapsto w, u \mapsto v)) \text{ } vn = \text{None} \implies (\text{tab}(u \mapsto v, r \mapsto w)) \text{ } vn = \text{None}$
 $\langle \text{proof} \rangle$

lemma *map-eq-upd-eq*: $\text{tab } vn = \text{tab}' \text{ } vn \implies (\text{tab}(x \mapsto y)) \text{ } vn = (\text{tab}'(x \mapsto y)) \text{ } vn$
 $\langle \text{proof} \rangle$

lemma *map-upd-in-expansion-map-swap*:

$$\llbracket (tab(x \mapsto y)) \rrbracket vn = Some\ z; tab\ vn \neq Some\ z \rrbracket$$

$$\implies \llbracket (tab'(x \mapsto y)) \rrbracket vn = Some\ z$$

$$\langle proof \rangle$$

lemma *map-upds-in-expansion-map-swap*:

$$\bigwedge tab\ tab'\ ys\ z. \llbracket (tab(xs[\mapsto]ys)) \rrbracket vn = Some\ z; tab\ vn \neq Some\ z \rrbracket$$

$$\implies \llbracket (tab'(xs[\mapsto]ys)) \rrbracket vn = Some\ z$$

$$\langle proof \rangle$$

lemma *map-upds-Some-swap*:
assumes $r \mapsto u$: $(tab(r \mapsto w, u \mapsto v, xs[\mapsto]ys))\ vn = Some\ z$
shows $\exists z. (tab(u \mapsto v, r \mapsto w, xs[\mapsto]ys))\ vn = Some\ z$

$$\langle proof \rangle$$

lemma *map-upds-Some-insert*:
assumes z : $(tab(xs[\mapsto]ys))\ vn = Some\ z$
shows $\exists z. (tab(u \mapsto v, xs[\mapsto]ys))\ vn = Some\ z$

$$\langle proof \rangle$$

lemma *map-upds-None-cut*:
assumes *expand-None*: $(tab(xs[\mapsto]ys))\ vn = None$
shows $tab\ vn = None$

$$\langle proof \rangle$$

lemma *map-upds-cut-irrelevant*:

$$\bigwedge tab\ tab'\ ys. \llbracket (tab(xs[\mapsto]ys)) \rrbracket vn = Some\ el; (tab'(xs[\mapsto]ys))\ vn = None \rrbracket$$

$$\implies tab\ vn = Some\ el$$

$$\langle proof \rangle$$

lemma *dom-vname-split*:

$$dom\ (case-lname\ (case-ename\ (tab(x \mapsto y, xs[\mapsto]ys))\ a)\ b)$$

$$= dom\ (case-lname\ (case-ename\ (tab(x \mapsto y))\ a)\ b) \cup$$

$$dom\ (case-lname\ (case-ename\ (tab(xs[\mapsto]ys))\ a)\ b)$$

$$(\text{is } ?List\ x\ xs\ y\ ys = ?Hd\ x\ y \cup ?Tl\ xs\ ys)$$

$$\langle proof \rangle$$

lemma *dom-map-upd*: $\bigwedge tab. dom\ (tab(x \mapsto y)) = dom\ tab \cup \{x\}$

$$\langle proof \rangle$$

lemma *dom-map-upds*: $\bigwedge tab\ ys. length\ xs = length\ ys$

$$\implies dom\ (tab(xs[\mapsto]ys)) = dom\ tab \cup set\ xs$$

$$\langle proof \rangle$$

lemma *dom-case-ename-None-simp*:

$$dom\ (case-ename\ vname-tab\ None) = VName\ '\ (dom\ vname-tab)$$

$$\langle proof \rangle$$

lemma *dom-case-ename-Some-simp*:

$$dom\ (case-ename\ vname-tab\ (Some\ a)) = VName\ '\ (dom\ vname-tab) \cup \{Res\}$$

$$\langle proof \rangle$$

lemma *dom-case-lname-None-simp*:

$$dom\ (case-lname\ ename-tab\ None) = EName\ '\ (dom\ ename-tab)$$

$$\langle proof \rangle$$

lemma *dom-case-lname-Some-simp*:

$$dom\ (case-lname\ ename-tab\ (Some\ a)) = EName\ '\ (dom\ ename-tab) \cup \{This\}$$

$\langle \text{proof} \rangle$

lemmas *dom-lname-case-ename-simps* =
 dom-case-ename-None-simp dom-case-ename-Some-simp
 dom-case-lname-None-simp dom-case-lname-Some-simp

lemma *image-comp*:
 $f \circ g \circ A = (f \circ g) \circ A$
 $\langle \text{proof} \rangle$

lemma *dom-locals-init-lvars*:
 assumes *m*: $m = (\text{methd } (\text{the } (\text{methd } G \ C \ \text{sig})))$
 assumes *len*: $\text{length } (\text{pars } m) = \text{length } pvs$
 shows *dom* (*locals* (*store* (*init-lvars* *G C sig* (*invmode m e*) *a pvs s*)))
 = *parameters m*
 $\langle \text{proof} \rangle$

lemma *da-e2-BinOp*:
 assumes *da*: $(\text{prg} = G, \text{cls} = \text{acc } C, \text{lcl} = L) \vdash \text{dom } (\text{locals } (\text{store } s0)) \gg \langle \text{BinOp binop } e1 \ e2 \rangle_e \gg A$
 and *wt-e1*: $(\text{prg} = G, \text{cls} = \text{acc } C, \text{lcl} = L) \vdash e1 :: -e1T$
 and *wt-e2*: $(\text{prg} = G, \text{cls} = \text{acc } C, \text{lcl} = L) \vdash e2 :: -e2T$
 and *wt-binop*: $\text{wt-binop } G \ \text{binop } e1T \ e2T$
 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*: $G, \text{store } s1 \vdash v1 :: \preceq e1T$
 and *wf*: *wf-prog G*
 shows $\exists \ E2. (\text{prg} = G, \text{cls} = \text{acc } C, \text{lcl} = L) \vdash \text{dom } (\text{locals } (\text{store } s1)) \gg (\text{if need-second-arg binop } v1 \text{ then } \langle e2 \rangle_e \text{ else } \langle \text{Skip} \rangle_s) \gg E2$
 $\langle \text{proof} \rangle$

main proof of type safety

lemma *eval-type-sound*:
 assumes *eval*: $G \vdash s0 \ -t \multimap (v, s1)$
 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 *conf-s0*: $s0 :: \preceq (G, L)$
 shows $s1 :: \preceq (G, L) \wedge (\text{normal } s1 \longrightarrow G, L, \text{store } s1 \vdash t \multimap v :: \preceq T) \wedge$
 $(\text{error-free } s0 = \text{error-free } s1)$
 $\langle \text{proof} \rangle$

corollary *eval-type-soundE* [*consumes 5*]:
 assumes *eval*: $G \vdash s0 \ -t \multimap (v, s1)$
 and *conf*: $s0 :: \preceq (G, L)$
 and *wt*: $(\text{prg} = G, \text{cls} = \text{acc } C, \text{lcl} = L) \vdash t :: T$
 and *da*: $(\text{prg} = G, \text{cls} = \text{acc } C, \text{lcl} = L) \vdash \text{dom } (\text{locals } (\text{snd } s0)) \gg t \gg A$
 and *wf*: *wf-prog G*
 and *elim*: $\llbracket s1 :: \preceq (G, L); \text{normal } s1 \Longrightarrow G, L, \text{snd } s1 \vdash t \multimap v :: \preceq T; \text{error-free } s0 = \text{error-free } s1 \rrbracket \Longrightarrow P$
 shows *P*
 $\langle \text{proof} \rangle$

corollary *eval-ts*:

$\llbracket G \vdash s - e \rightarrow v \rightarrow s'; \text{wf-prog } G; s::\preceq(G, L); (\text{prg}=G, \text{cls}=C, \text{lcl}=L) \vdash e::-T;$
 $(\text{prg}=G, \text{cls}=C, \text{lcl}=L) \vdash_{\text{dom}} (\text{locals } (\text{store } s)) \gg \text{In1l } e \gg A \rrbracket$
 $\implies s'::\preceq(G, L) \wedge (\text{normal } s' \longrightarrow G, \text{store } s' \vdash v::\preceq T) \wedge$
 $(\text{error-free } s = \text{error-free } s')$
 $\langle \text{proof} \rangle$

corollary evals-ts:

$\llbracket G \vdash s - es \rightarrow vs \rightarrow s'; \text{wf-prog } G; s::\preceq(G, L); (\text{prg}=G, \text{cls}=C, \text{lcl}=L) \vdash es::Ts;$
 $(\text{prg}=G, \text{cls}=C, \text{lcl}=L) \vdash_{\text{dom}} (\text{locals } (\text{store } s)) \gg \text{In3 } es \gg A \rrbracket$
 $\implies s'::\preceq(G, L) \wedge (\text{normal } s' \longrightarrow \text{list-all2 } (\text{conf } G (\text{store } s')) \text{ vs } Ts) \wedge$
 $(\text{error-free } s = \text{error-free } s')$
 $\langle \text{proof} \rangle$

corollary evar-ts:

$\llbracket G \vdash s - v \rightarrow vf \rightarrow s'; \text{wf-prog } G; s::\preceq(G, L); (\text{prg}=G, \text{cls}=C, \text{lcl}=L) \vdash v::T;$
 $(\text{prg}=G, \text{cls}=C, \text{lcl}=L) \vdash_{\text{dom}} (\text{locals } (\text{store } s)) \gg \text{In2 } v \gg A \rrbracket \implies$
 $s'::\preceq(G, L) \wedge (\text{normal } s' \longrightarrow G, L, (\text{store } s') \vdash \text{In2 } v \gg \text{In2 } vf::\preceq \text{Inl } T) \wedge$
 $(\text{error-free } s = \text{error-free } s')$
 $\langle \text{proof} \rangle$

theorem exec-ts:

$\llbracket G \vdash s - c \rightarrow s'; \text{wf-prog } G; s::\preceq(G, L); (\text{prg}=G, \text{cls}=C, \text{lcl}=L) \vdash c::\checkmark;$
 $(\text{prg}=G, \text{cls}=C, \text{lcl}=L) \vdash_{\text{dom}} (\text{locals } (\text{store } s)) \gg \text{In1r } c \gg A \rrbracket$
 $\implies s'::\preceq(G, L) \wedge (\text{error-free } s \longrightarrow \text{error-free } s')$
 $\langle \text{proof} \rangle$

lemma wf-eval-Fin:

assumes $\text{wf: wf-prog } G$
and $\text{wt-c1: } (\text{prg} = G, \text{cls} = C, \text{lcl} = L) \vdash \text{In1r } c1::\text{Inl } (\text{PrimT } \text{Void})$
and $\text{da-c1: } (\text{prg}=G, \text{cls}=C, \text{lcl}=L) \vdash_{\text{dom}} (\text{locals } (\text{store } (\text{Norm } s0))) \gg \text{In1r } c1 \gg A$
and $\text{conf-s0: Norm } s0::\preceq(G, L)$
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 $G \vdash \text{Norm } s0 - c1 \text{ Finally } c2 \rightarrow s3$
 $\langle \text{proof} \rangle$

3 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 $\text{eval: } G \vdash s0 - t \rightarrow (v, s1)$
and $\text{wt: } (\text{prg}=G, \text{cls}=\text{acc } C, \text{lcl}=L) \vdash t::T$
and $\text{da: } (\text{prg}=G, \text{cls}=\text{acc } C, \text{lcl}=L) \vdash_{\text{dom}} (\text{locals } (\text{store } s0)) \gg t \gg A$
and $\text{wf: wf-prog } G$
and $\text{abrupt: } \bigwedge s \ t \ \text{abr } L \ \text{acc } C \ T \ A.$
 $\llbracket (\text{prg}=G, \text{cls}=\text{acc } C, \text{lcl}=L) \vdash t::T;$
 $(\text{prg}=G, \text{cls}=\text{acc } C, \text{lcl}=L) \vdash_{\text{dom}} (\text{locals } (\text{store } (\text{Some } \text{abr}, s))) \gg t \gg A$
 $\rrbracket \implies P \ L \ \text{acc } C \ (\text{Some } \text{abr}, s) \ t \ (\text{undefined3 } t) \ (\text{Some } \text{abr}, s)$

and $skip: \bigwedge s \ L \ accC. \ P \ L \ accC \ (Norm \ s) \ \langle Skip \rangle_s \ \Diamond \ (Norm \ s)$
and $expr: \bigwedge e \ s0 \ s1 \ v \ L \ accC \ eT \ E.$
 $\llbracket \langle prg=G, cls=accC, lcl=L \rangle \vdash e :: -eT; \$
 $\langle prg=G, cls=accC, lcl=L \rangle \vdash dom \ (locals \ (store \ ((Norm \ s0)::state))) \ \rangle \langle e \rangle_e \ \rangle E;$
 $P \ L \ accC \ (Norm \ s0) \ \langle e \rangle_e \ [v]_e \ s1 \rrbracket$
 $\implies P \ L \ accC \ (Norm \ s0) \ \langle Expr \ e \rangle_s \ \Diamond \ s1$
and $lab: \bigwedge c \ l \ s0 \ s1 \ L \ accC \ C.$
 $\llbracket \langle prg=G, cls=accC, lcl=L \rangle \vdash c :: \sqrt{}; \$
 $\langle prg=G, cls=accC, lcl=L \rangle \vdash dom \ (locals \ (store \ ((Norm \ s0)::state))) \ \rangle \langle c \rangle_s \ \rangle C;$
 $P \ L \ accC \ (Norm \ s0) \ \langle c \rangle_s \ \Diamond \ s1 \rrbracket$
 $\implies P \ L \ accC \ (Norm \ s0) \ \langle l \cdot c \rangle_s \ \Diamond \ (abupd \ (absorb \ l) \ s1)$
and $comp: \bigwedge c1 \ c2 \ s0 \ s1 \ s2 \ L \ accC \ C1.$
 $\llbracket G \vdash Norm \ s0 \ -c1 \rightarrow s1; G \vdash s1 \ -c2 \rightarrow s2; \$
 $\langle prg=G, cls=accC, lcl=L \rangle \vdash c1 :: \sqrt{}; \$
 $\langle prg=G, cls=accC, lcl=L \rangle \vdash c2 :: \sqrt{}; \$
 $\langle prg=G, cls=accC, lcl=L \rangle \vdash$
 $dom \ (locals \ (store \ ((Norm \ s0)::state))) \ \rangle \langle c1 \rangle_s \ \rangle C1;$
 $P \ L \ accC \ (Norm \ s0) \ \langle c1 \rangle_s \ \Diamond \ s1; \$
 $\bigwedge Q. \llbracket normal \ s1; \$
 $\bigwedge C2. \llbracket \langle prg=G, cls=accC, lcl=L \rangle \vdash$
 $dom \ (locals \ (store \ s1)) \ \rangle \langle c2 \rangle_s \ \rangle C2; \$
 $P \ L \ accC \ s1 \ \langle c2 \rangle_s \ \Diamond \ s2 \rrbracket \implies Q$
 $\rrbracket \implies Q$
 $\rrbracket \implies P \ L \ accC \ (Norm \ s0) \ \langle c1;; \ c2 \rangle_s \ \Diamond \ s2$
and $if: \bigwedge b \ c1 \ c2 \ e \ s0 \ s1 \ s2 \ L \ accC \ E.$
 $\llbracket G \vdash Norm \ s0 \ -e \multimap b \rightarrow s1; \$
 $G \vdash s1 \ -(if \ the\ Bool \ b \ then \ c1 \ else \ c2) \rightarrow s2; \$
 $\langle prg=G, cls=accC, lcl=L \rangle \vdash e :: -PrimT \ Boolean;$
 $\langle prg=G, cls=accC, lcl=L \rangle \vdash (if \ the\ Bool \ b \ then \ c1 \ else \ c2) :: \sqrt{}; \$
 $\langle prg=G, cls=accC, lcl=L \rangle \vdash$
 $dom \ (locals \ (store \ ((Norm \ s0)::state))) \ \rangle \langle e \rangle_e \ \rangle E;$
 $P \ L \ accC \ (Norm \ s0) \ \langle e \rangle_e \ [b]_e \ s1; \$
 $\bigwedge Q. \llbracket normal \ s1; \$
 $\bigwedge C. \llbracket \langle prg=G, cls=accC, lcl=L \rangle \vdash (dom \ (locals \ (store \ s1))) \$
 $\rangle \langle if \ the\ Bool \ b \ then \ c1 \ else \ c2 \rangle_s \ \rangle C; \$
 $P \ L \ accC \ s1 \ \langle if \ the\ Bool \ b \ then \ c1 \ else \ c2 \rangle_s \ \Diamond \ s2$
 $\rrbracket \implies Q$
 $\rrbracket \implies Q$
 $\rrbracket \implies P \ L \ accC \ (Norm \ s0) \ \langle If(e) \ c1 \ Else \ c2 \rangle_s \ \Diamond \ s2$
shows $P \ L \ accC \ s0 \ t \ v \ s1$
 $\langle proof \rangle$
end

Chapter 20

Evaln

1 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
  (⟦-⟧evaln [61,61,80,61,0,0] 60)
and evaln :: [prog, state, var, vvar, nat, state] ⇒ bool
  (⟦-⟧evaln [61,61,90,61,61,61] 60)
and evaln :: [prog, state, expr, val, nat, state] ⇒ bool
  (⟦-⟧evaln [61,61,80,61,61,61] 60)
and evaln :: [prog, state, expr list, val list, nat, state] ⇒ bool
  (⟦-⟧evaln [61,61,61,61,61,61] 60)
and execn :: [prog, state, stmt, nat, state] ⇒ bool
  (⟦-⟧execn [61,61,65, 61,61] 60)
for G :: prog
where

```

```

  G ⊢ s - c - n → s' ≡ G ⊢ s - In1r c > - n → (◇ s, 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 → (undefined3 t, (Some xc, s))

```

— evaluation of variables

```

| LVar: G ⊢ Norm s - LVar vn => lvar vn s - n → Norm s

```

| *FVar*: $\llbracket G \vdash \text{Norm } s0 \text{ --Init statDeclC--} n \rightarrow s1; G \vdash s1 \text{ --} e \text{--} \succ a \text{--} n \rightarrow s2;$
 $(v, s2') = \text{fvar statDeclC stat fn } a \text{ } s2;$
 $s3 = \text{check-field-access } G \text{ accC statDeclC fn stat } a \text{ } s2 \rrbracket \implies$
 $G \vdash \text{Norm } s0 \text{ --}\{accC, statDeclC, stat\}e..fn \text{--} \succ v \text{--} n \rightarrow s3$

| *AVar*: $\llbracket G \vdash \text{Norm } s0 \text{ --} e1 \text{--} \succ a \text{--} n \rightarrow s1; G \vdash s1 \text{ --} e2 \text{--} \succ i \text{--} n \rightarrow s2;$
 $(v, s2') = \text{avar } G \text{ } i \text{ } a \text{ } s2 \rrbracket \implies$
 $G \vdash \text{Norm } s0 \text{ --} e1.[e2] \text{--} \succ v \text{--} n \rightarrow s2'$

— evaluation of expressions

| *NewC*: $\llbracket G \vdash \text{Norm } s0 \text{ --Init } C \text{--} n \rightarrow s1;$
 $G \vdash s1 \text{ --halloc } (C \text{Inst } C) \text{--} \succ a \rightarrow s2 \rrbracket \implies$
 $G \vdash \text{Norm } s0 \text{ --NewC } C \text{--} \succ \text{Addr } a \text{--} n \rightarrow s2$

| *NewA*: $\llbracket G \vdash \text{Norm } s0 \text{ --init-comp-ty } T \text{--} n \rightarrow s1; G \vdash s1 \text{ --} e \text{--} \succ i' \text{--} n \rightarrow s2;$
 $G \vdash \text{abupd } (\text{check-neg } i') \text{ } s2 \text{ --halloc } (\text{Arr } T \text{ } (\text{the-Intg } i')) \text{--} \succ a \rightarrow s3 \rrbracket \implies$
 $G \vdash \text{Norm } s0 \text{ --New } T[e] \text{--} \succ \text{Addr } a \text{--} n \rightarrow s3$

| *Cast*: $\llbracket G \vdash \text{Norm } s0 \text{ --} e \text{--} \succ v \text{--} n \rightarrow s1;$
 $s2 = \text{abupd } (\text{raise-if } (\neg G, \text{snd } s1 \vdash v \text{ fits } T) \text{ } \text{ClassCast}) \text{ } s1 \rrbracket \implies$
 $G \vdash \text{Norm } s0 \text{ --Cast } T \text{ } e \text{--} \succ v \text{--} n \rightarrow s2$

| *Inst*: $\llbracket G \vdash \text{Norm } s0 \text{ --} e \text{--} \succ v \text{--} n \rightarrow s1;$
 $b = (v \neq \text{Null} \wedge G, \text{store } s1 \vdash v \text{ fits } \text{RefT } T) \rrbracket \implies$
 $G \vdash \text{Norm } s0 \text{ --} e \text{InstOf } T \text{--} \succ \text{Bool } b \text{--} n \rightarrow s1$

| *Lit*: $G \vdash \text{Norm } s \text{ --Lit } v \text{--} \succ v \text{--} n \rightarrow \text{Norm } s$

| *UnOp*: $\llbracket G \vdash \text{Norm } s0 \text{ --} e \text{--} \succ v \text{--} n \rightarrow s1 \rrbracket$
 $\implies G \vdash \text{Norm } s0 \text{ --UnOp unop } e \text{--} \succ (\text{eval-unop unop } v) \text{--} n \rightarrow s1$

| *BinOp*: $\llbracket G \vdash \text{Norm } s0 \text{ --} e1 \text{--} \succ v1 \text{--} n \rightarrow s1;$
 $G \vdash s1 \text{ --(if need-second-arg binop } v1 \text{ then (In1l } e2) \text{ else (In1r Skip))}$
 $\text{--} \succ n \rightarrow (\text{In1 } v2, s2) \rrbracket$
 $\implies G \vdash \text{Norm } s0 \text{ --BinOp binop } e1 \text{ } e2 \text{--} \succ (\text{eval-binop binop } v1 \text{ } v2) \text{--} n \rightarrow s2$

| *Super*: $G \vdash \text{Norm } s \text{ --Super--} \succ \text{val-this } s \text{--} n \rightarrow \text{Norm } s$

| *Acc*: $\llbracket G \vdash \text{Norm } s0 \text{ --} va \text{--} \succ (v, f) \text{--} n \rightarrow s1 \rrbracket \implies$
 $G \vdash \text{Norm } s0 \text{ --Acc } va \text{--} \succ v \text{--} n \rightarrow s1$

| *Ass*: $\llbracket G \vdash \text{Norm } s0 \text{ --} va \text{--} \succ (w, f) \text{--} n \rightarrow s1;$
 $G \vdash s1 \text{ --} e \text{--} \succ v \text{--} n \rightarrow s2 \rrbracket \implies$
 $G \vdash \text{Norm } s0 \text{ --} va \text{--} \text{:=} e \text{--} \succ v \text{--} n \rightarrow \text{assign } f \text{ } v \text{ } s2$

| *Cond*: $\llbracket G \vdash \text{Norm } s0 \text{ --} e0 \text{--} \succ b \text{--} n \rightarrow s1;$
 $G \vdash s1 \text{ --(if the-Bool } b \text{ then } e1 \text{ else } e2) \text{--} \succ v \text{--} n \rightarrow s2 \rrbracket \implies$
 $G \vdash \text{Norm } s0 \text{ --} e0 \text{ } ? \text{ } e1 : e2 \text{--} \succ v \text{--} n \rightarrow s2$

| *Call*:
 $\llbracket G \vdash \text{Norm } s0 \text{ --} e \text{--} \succ a' \text{--} n \rightarrow s1; G \vdash s1 \text{ --args--} \succ vs \text{--} n \rightarrow s2;$
 $D = \text{invocation-declclass } G \text{ mode } (\text{store } s2) \text{ } a' \text{ } \text{statT } (\text{name=mn, parTs=pTs});$
 $s3 = \text{init-lvars } G \text{ } D \text{ } (\text{name=mn, parTs=pTs}) \text{ mode } a' \text{ } vs \text{ } s2;$
 $s3' = \text{check-method-access } G \text{ accC statT mode } (\text{name=mn, parTs=pTs}) \text{ } a' \text{ } s3;$

- $$\begin{array}{l}
\begin{array}{c}
G \vdash s3' - \text{Methd } D \ (\backslash \text{name=mn, parTs=pTs}) - \succ v - n \rightarrow s4 \\
\parallel \\
\implies \\
G \vdash \text{Norm } s0 \ - \{accC, statT, mode\} e \cdot mn(\{pTs\} args) - \succ v - n \rightarrow (\text{restore-lvars } s2 \ s4)
\end{array} \\
| \text{Methd: } \llbracket G \vdash \text{Norm } s0 \ - \text{body } G \ D \ \text{sig} - \succ v - n \rightarrow s1 \rrbracket \implies \\
\quad G \vdash \text{Norm } s0 \ - \text{Methd } D \ \text{sig} - \succ v - \text{Suc } n \rightarrow s1 \\
| \text{Body: } \llbracket G \vdash \text{Norm } s0 - \text{Init } D - n \rightarrow s1; \ G \vdash s1 \ - c - n \rightarrow s2; \\
\quad s3 = (\text{if } (\exists \ l. \ \text{abrupt } s2 = \text{Some } (\text{Jump } (\text{Break } l))) \vee \\
\quad \quad \text{abrupt } s2 = \text{Some } (\text{Jump } (\text{Cont } l))) \\
\quad \text{then } \text{abupd } (\lambda \ x. \ \text{Some } (\text{Error } \text{CrossMethodJump})) \ s2 \\
\quad \text{else } s2 \rrbracket \implies \\
\quad G \vdash \text{Norm } s0 \ - \text{Body } D \ c \\
\quad \ - \succ \text{the } (\text{locals } (\text{store } s2) \ \text{Result}) - n \rightarrow \text{abupd } (\text{absorb } \text{Ret}) \ s3 \\
- \text{ evaluation of expression lists} \\
| \text{Nil:} \\
\quad G \vdash \text{Norm } s0 \ - \ [] \dot{=} \succ [] - n \rightarrow \text{Norm } s0 \\
| \text{Cons: } \llbracket G \vdash \text{Norm } s0 \ - e - \succ v - n \rightarrow s1; \\
\quad G \vdash \quad s1 \ - es \dot{=} \succ vs - n \rightarrow s2 \rrbracket \implies \\
\quad G \vdash \text{Norm } s0 \ - e \# es \dot{=} \succ v \# vs - n \rightarrow s2 \\
- \text{ execution of statements} \\
| \text{Skip:} \quad G \vdash \text{Norm } s \ - \text{Skip} - n \rightarrow \text{Norm } s \\
| \text{Expr: } \llbracket G \vdash \text{Norm } s0 \ - e - \succ v - n \rightarrow s1 \rrbracket \implies \\
\quad G \vdash \text{Norm } s0 \ - \text{Expr } e - n \rightarrow s1 \\
| \text{Lab: } \llbracket G \vdash \text{Norm } s0 \ - c - n \rightarrow s1 \rrbracket \implies \\
\quad G \vdash \text{Norm } s0 \ - l \cdot c - n \rightarrow \text{abupd } (\text{absorb } l) \ s1 \\
| \text{Comp: } \llbracket G \vdash \text{Norm } s0 \ - c1 \ - n \rightarrow s1; \\
\quad G \vdash \quad s1 \ - c2 \ - n \rightarrow s2 \rrbracket \implies \\
\quad G \vdash \text{Norm } s0 \ - c1 ;; c2 - n \rightarrow s2 \\
| \text{If: } \llbracket G \vdash \text{Norm } s0 \ - e - \succ b - n \rightarrow s1; \\
\quad G \vdash \quad s1 - (\text{if the-Bool } b \text{ then } c1 \text{ else } c2) - n \rightarrow s2 \rrbracket \implies \\
\quad G \vdash \text{Norm } s0 \ - \text{If}(e) \ c1 \ \text{Else } c2 \ - n \rightarrow s2 \\
| \text{Loop: } \llbracket G \vdash \text{Norm } s0 \ - e - \succ b - n \rightarrow s1; \\
\quad \text{if the-Bool } b \\
\quad \text{then } (G \vdash s1 \ - c - n \rightarrow s2 \wedge \\
\quad \quad G \vdash (\text{abupd } (\text{absorb } (\text{Cont } l)) \ s2) \ - l \cdot \text{While}(e) \ c - n \rightarrow s3) \\
\quad \text{else } s3 = s1 \rrbracket \implies \\
\quad G \vdash \text{Norm } s0 \ - l \cdot \text{While}(e) \ c - n \rightarrow s3 \\
| \text{Jmp: } G \vdash \text{Norm } s \ - \text{Jmp } j - n \rightarrow (\text{Some } (\text{Jump } j), \ s) \\
| \text{Throw: } \llbracket G \vdash \text{Norm } s0 \ - e - \succ a' - n \rightarrow s1 \rrbracket \implies \\
\quad G \vdash \text{Norm } s0 \ - \text{Throw } e - n \rightarrow \text{abupd } (\text{throw } a') \ s1 \\
| \text{Try: } \llbracket G \vdash \text{Norm } s0 \ - c1 - n \rightarrow s1; \ G \vdash s1 \ - s\text{xalloc} \rightarrow s2; \\
\quad \text{if } G, s2 \vdash \text{catch } tn \text{ then } G \vdash \text{new-xcpt-var } vn \ s2 \ - c2 - n \rightarrow s3 \text{ else } s3 = s2 \rrbracket \\
\implies
\end{array}$$

$$G \vdash \text{Norm } s0 \text{ --Try } c1 \text{ Catch}(tn \text{ } vn) \text{ } c2 \text{ --} n \rightarrow s3$$

| *Fin*: $\llbracket G \vdash \text{Norm } s0 \text{ --} c1 \text{ --} n \rightarrow (x1, s1);$
 $G \vdash \text{Norm } s1 \text{ --} c2 \text{ --} n \rightarrow s2;$
 $s3 = (\text{if } (\exists \text{ err. } x1 = \text{Some } (\text{Error err}))$
 $\text{then } (x1, s1)$
 $\text{else } \text{abupd } (\text{abrupt-if } (x1 \neq \text{None}) \text{ } x1) \text{ } s2) \rrbracket \implies$
 $G \vdash \text{Norm } s0 \text{ --} c1 \text{ Finally } c2 \text{ --} n \rightarrow s3$

| *Init*: $\llbracket \text{the } (\text{class } G \text{ } C) = c;$
 $\text{if } \text{inited } C \text{ (globs } s0) \text{ then } s3 = \text{Norm } s0$
 $\text{else } (G \vdash \text{Norm } (\text{init-class-obj } G \text{ } C \text{ } s0)$
 $\text{--} (\text{if } C = \text{Object then Skip else Init (super } c)) \text{ --} n \rightarrow s1 \wedge$
 $G \vdash \text{set-lvars Map.empty } s1 \text{ --init } c \text{ --} n \rightarrow s2 \wedge$
 $s3 = \text{restore-lvars } s1 \text{ } s2) \rrbracket$
 \implies
 $G \vdash \text{Norm } s0 \text{ --Init } C \text{ --} n \rightarrow s3$

monos

if-bool-eq-conj

declare *if-split* $[split \text{ del}]$ *if-split-asm* $[split \text{ del}]$
 $\text{option.split } [split \text{ del}] \text{ option.split-asm } [split \text{ del}]$
 $\text{not-None-eq } [simp \text{ del}]$
 $\text{split-paired-All } [simp \text{ del}] \text{ split-paired-Ex } [simp \text{ del}]$
 $\langle ML \rangle$

inductive-cases *evaln-cases*: $G \vdash s \text{ --} t \text{ --} n \rightarrow (v, s')$

inductive-cases *evaln-elim-cases*:

$G \vdash (\text{Some } xc, s) \text{ --} t$	$\succ \text{--} n \rightarrow (v, s')$
$G \vdash \text{Norm } s \text{ --In1r Skip}$	$\succ \text{--} n \rightarrow (x, s')$
$G \vdash \text{Norm } s \text{ --In1r (Jmp } j)$	$\succ \text{--} n \rightarrow (x, s')$
$G \vdash \text{Norm } s \text{ --In1r } (l \cdot c)$	$\succ \text{--} n \rightarrow (x, s')$
$G \vdash \text{Norm } s \text{ --In3 } (\llbracket \rrbracket)$	$\succ \text{--} n \rightarrow (v, s')$
$G \vdash \text{Norm } s \text{ --In3 } (e \# es)$	$\succ \text{--} n \rightarrow (v, s')$
$G \vdash \text{Norm } s \text{ --In1l (Lit } w)$	$\succ \text{--} n \rightarrow (v, s')$
$G \vdash \text{Norm } s \text{ --In1l (UnOp unop } e)$	$\succ \text{--} n \rightarrow (v, s')$
$G \vdash \text{Norm } s \text{ --In1l (BinOp binop } e1 \text{ } e2)$	$\succ \text{--} n \rightarrow (v, s')$
$G \vdash \text{Norm } s \text{ --In2 (LVar } vn)$	$\succ \text{--} n \rightarrow (v, s')$
$G \vdash \text{Norm } s \text{ --In1l (Cast } T \text{ } e)$	$\succ \text{--} n \rightarrow (v, s')$
$G \vdash \text{Norm } s \text{ --In1l } (e \text{ InstOf } T)$	$\succ \text{--} n \rightarrow (v, s')$
$G \vdash \text{Norm } s \text{ --In1l (Super)}$	$\succ \text{--} n \rightarrow (v, s')$
$G \vdash \text{Norm } s \text{ --In1l (Acc } va)$	$\succ \text{--} n \rightarrow (v, s')$
$G \vdash \text{Norm } s \text{ --In1r (Expr } e)$	$\succ \text{--} n \rightarrow (x, s')$
$G \vdash \text{Norm } s \text{ --In1r } (c1;; c2)$	$\succ \text{--} n \rightarrow (x, s')$
$G \vdash \text{Norm } s \text{ --In1l (Methd } C \text{ sig)}$	$\succ \text{--} n \rightarrow (x, s')$
$G \vdash \text{Norm } s \text{ --In1l (Body } D \text{ } c)$	$\succ \text{--} n \rightarrow (x, s')$
$G \vdash \text{Norm } s \text{ --In1l } (e0 \text{ ? } e1 : e2)$	$\succ \text{--} n \rightarrow (v, s')$
$G \vdash \text{Norm } s \text{ --In1r (If}(e) \text{ } c1 \text{ Else } c2)$	$\succ \text{--} n \rightarrow (x, s')$
$G \vdash \text{Norm } s \text{ --In1r } (l \cdot \text{While}(e) \text{ } c)$	$\succ \text{--} n \rightarrow (x, s')$
$G \vdash \text{Norm } s \text{ --In1r } (c1 \text{ Finally } c2)$	$\succ \text{--} n \rightarrow (x, s')$
$G \vdash \text{Norm } s \text{ --In1r (Throw } e)$	$\succ \text{--} n \rightarrow (x, s')$
$G \vdash \text{Norm } s \text{ --In1l (NewC } C)$	$\succ \text{--} n \rightarrow (v, s')$
$G \vdash \text{Norm } s \text{ --In1l (New } T[e])$	$\succ \text{--} n \rightarrow (v, s')$
$G \vdash \text{Norm } s \text{ --In1l (Ass } va \text{ } e)$	$\succ \text{--} n \rightarrow (v, s')$
$G \vdash \text{Norm } s \text{ --In1r (Try } c1 \text{ Catch}(tn \text{ } vn) \text{ } c2)$	$\succ \text{--} n \rightarrow (x, s')$
$G \vdash \text{Norm } s \text{ --In2 } (\{accC, statDeclC, stat\}e..fn)$	$\succ \text{--} n \rightarrow (v, s')$
$G \vdash \text{Norm } s \text{ --In2 } (e1.[e2])$	$\succ \text{--} n \rightarrow (v, s')$

$$\begin{array}{l} G \vdash \text{Norm } s - \text{In1l } (\{accC, statT, mode\} e \cdot mn(\{pT\}p)) \succ -n \rightarrow (v, s') \\ G \vdash \text{Norm } s - \text{In1r } (\text{Init } C) \succ -n \rightarrow (x, s') \end{array}$$

declare *if-split* [split] *if-split-asm* [split]
option.split [split] *option.split-asm* [split]
not-None-eq [simp]
split-paired-All [simp] *split-paired-Ex* [simp]
 <ML>

lemma *evaln-Inj-elim*: $G \vdash s - t \succ -n \rightarrow (w, s') \implies \text{case } t \text{ of } \text{In1 } ec \Rightarrow$
 $(\text{case } ec \text{ of } \text{Inl } e \Rightarrow (\exists v. w = \text{In1 } v) \mid \text{Inr } c \Rightarrow w = \Diamond)$
 $\mid \text{In2 } e \Rightarrow (\exists v. w = \text{In2 } v) \mid \text{In3 } e \Rightarrow (\exists v. w = \text{In3 } v)$
 <proof>

The following simplification procedures set up the proper injections of terms and their corresponding values in the evaluation relation: E.g. an expression (injection *In1l* into terms) always evaluates to ordinary values (injection *In1* into generalised values *vals*).

lemma *evaln-expr-eq*: $G \vdash s - \text{In1l } t \succ -n \rightarrow (w, s') = (\exists v. w = \text{In1 } v \wedge G \vdash s - t \succ v -n \rightarrow s')$
 <proof>

lemma *evaln-var-eq*: $G \vdash s - \text{In2 } t \succ -n \rightarrow (w, s') = (\exists vf. w = \text{In2 } vf \wedge G \vdash s - t \succ vf -n \rightarrow s')$
 <proof>

lemma *evaln-exprs-eq*: $G \vdash s - \text{In3 } t \succ -n \rightarrow (w, s') = (\exists vs. w = \text{In3 } vs \wedge G \vdash s - t \dot{\succ} vs -n \rightarrow s')$
 <proof>

lemma *evaln-stmt-eq*: $G \vdash s - \text{In1r } t \succ -n \rightarrow (w, s') = (w = \Diamond \wedge G \vdash s - t -n \rightarrow s')$
 <proof>

<ML>

declare *evaln-AbruptIs* [intro!]

lemma *evaln-Callee*: $G \vdash \text{Norm } s - \text{In1l } (\text{Callee } l \ e) \succ -n \rightarrow (v, s') = \text{False}$
 <proof>

lemma *evaln-InsInitE*: $G \vdash \text{Norm } s - \text{In1l } (\text{InsInitE } c \ e) \succ -n \rightarrow (v, s') = \text{False}$
 <proof>

lemma *evaln-InsInitV*: $G \vdash \text{Norm } s - \text{In2 } (\text{InsInitV } c \ w) \succ -n \rightarrow (v, s') = \text{False}$
 <proof>

lemma *evaln-FinA*: $G \vdash \text{Norm } s - \text{In1r } (\text{FinA } a \ c) \succ -n \rightarrow (v, s') = \text{False}$
 <proof>

lemma *evaln-abrupt-lemma*: $G \vdash s - e \succ -n \rightarrow (v, s') \implies$
 $\text{fst } s = \text{Some } xc \longrightarrow s' = s \wedge v = \text{undefined3 } e$
 <proof>

lemma *evaln-abrupt*:
 $\wedge s'. G \vdash (\text{Some } xc, s) - e \succ -n \rightarrow (w, s') = (s' = (\text{Some } xc, s) \wedge$
 $w = \text{undefined3 } e \wedge G \vdash (\text{Some } xc, s) - e \succ -n \rightarrow (\text{undefined3 } e, (\text{Some } xc, s)))$
 <proof>

<ML>

lemma *evaln-LitI*: $G \vdash s - \text{Lit } v \succ (\text{if normal } s \text{ then } v \text{ else undefined}) -n \rightarrow s$
 <proof>

lemma *CondI*:

$\bigwedge s1. \llbracket G \vdash s - e - \succ b - n \rightarrow s1; G \vdash s1 - (\text{if the-Bool } b \text{ then } e1 \text{ else } e2) - \succ v - n \rightarrow s2 \rrbracket \implies$
 $G \vdash s - e \text{ ? } e1 : e2 - \succ (\text{if normal } s1 \text{ then } v \text{ else undefined}) - n \rightarrow s2$
 $\langle \text{proof} \rangle$

lemma *evaln-SkipI* [*intro!*]: $G \vdash s - \text{Skip} - n \rightarrow s$

$\langle \text{proof} \rangle$

lemma *evaln-ExprI*: $G \vdash s - e - \succ v - n \rightarrow s' \implies G \vdash s - \text{Expr } e - n \rightarrow s'$

$\langle \text{proof} \rangle$

lemma *evaln-CompI*: $\llbracket G \vdash s - c1 - n \rightarrow s1; G \vdash s1 - c2 - n \rightarrow s2 \rrbracket \implies G \vdash s - c1;; c2 - n \rightarrow s2$

$\langle \text{proof} \rangle$

lemma *evaln-IfI*:

$\llbracket G \vdash s - e - \succ v - n \rightarrow s1; G \vdash s1 - (\text{if the-Bool } v \text{ then } c1 \text{ else } c2) - n \rightarrow s2 \rrbracket \implies$
 $G \vdash s - \text{If}(e) \ c1 \ \text{Else } c2 - n \rightarrow s2$

$\langle \text{proof} \rangle$

lemma *evaln-SkipD* [*dest!*]: $G \vdash s - \text{Skip} - n \rightarrow s' \implies s' = s$

$\langle \text{proof} \rangle$

lemma *evaln-Skip-eq* [*simp*]: $G \vdash s - \text{Skip} - n \rightarrow s' = (s = s')$

$\langle \text{proof} \rangle$

evaln implies eval

lemma *evaln-eval*:

assumes *evaln*: $G \vdash s0 - t \succ - n \rightarrow (v, s1)$

shows $G \vdash s0 - t \succ \rightarrow (v, s1)$

$\langle \text{proof} \rangle$

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*]:

$G \vdash s - t \succ - n \rightarrow (w, s') \implies \forall m. n \leq m \longrightarrow G \vdash s - t \succ - m \rightarrow (w, s')$

$\langle \text{proof} \rangle$

lemmas *evaln-nonstrict-Suc* = *evaln-nonstrict* [*OF - le-refl [THEN le-SucI]*]

lemma *evaln-max2*: $\llbracket G \vdash s1 - t1 \succ - n1 \rightarrow (w1, s1'); G \vdash s2 - t2 \succ - n2 \rightarrow (w2, s2') \rrbracket \implies$

$G \vdash s1 - t1 \succ - \max n1 \ n2 \rightarrow (w1, s1') \wedge G \vdash s2 - t2 \succ - \max n1 \ n2 \rightarrow (w2, s2')$

$\langle \text{proof} \rangle$

corollary *evaln-max2E* [*consumes 2*]:

$\llbracket G \vdash s1 - t1 \succ - n1 \rightarrow (w1, s1'); G \vdash s2 - t2 \succ - n2 \rightarrow (w2, s2') \rrbracket$

$\llbracket G \vdash s1 - t1 \succ - \max n1 \ n2 \rightarrow (w1, s1'); G \vdash s2 - t2 \succ - \max n1 \ n2 \rightarrow (w2, s2') \rrbracket \implies P \rrbracket \implies P$

$\langle \text{proof} \rangle$

lemma *evaln-max3*:

$\llbracket G \vdash s1 - t1 \succ - n1 \rightarrow (w1, s1'); G \vdash s2 - t2 \succ - n2 \rightarrow (w2, s2'); G \vdash s3 - t3 \succ - n3 \rightarrow (w3, s3') \rrbracket \implies$

$G \vdash s1 - t1 \succ - \max (\max n1 \ n2) \ n3 \rightarrow (w1, s1') \wedge$

$G \vdash s2 - t2 \succ - \max (\max n1 \ n2) \ n3 \rightarrow (w2, s2') \wedge$

$G \vdash s3 - t3 \succ - \max (\max n1 \ n2) \ n3 \rightarrow (w3, s3')$

$\langle \text{proof} \rangle$

corollary *evaln-max3E*:


```

[[G⊢ s1 -t1>-n1 → (w1, s1'); G⊢ s2 -t2>-n2 → (w2, s2'); G⊢ s3 -t3>-n3 → (w3, s3');
  [[G⊢ s1 -t1>-max (max n1 n2) n3 → (w1, s1');
    G⊢ s2 -t2>-max (max n1 n2) n3 → (w2, s2');
    G⊢ s3 -t3>-max (max n1 n2) n3 → (w3, s3')
  ]] ⇒ P
]] ⇒ P
⟨proof⟩

```

```

lemma le-max3I1: (n2::nat) ≤ max n1 (max n2 n3)
⟨proof⟩

```

```

lemma le-max3I2: (n3::nat) ≤ max n1 (max n2 n3)
⟨proof⟩

```

```

declare [[simproc del: wt-expr wt-var wt-exprs wt-stmt]]

```

eval implies evaln

```

lemma eval-evaln:
  assumes eval: G⊢ s0 -t>-→ (v,s1)
  shows ∃ n. G⊢ s0 -t>-n → (v,s1)
⟨proof⟩

```

end

Chapter 21

Trans

theory *Trans* **imports** *Evaln* **begin**

definition

groundVar :: *var* \Rightarrow *bool* **where**
groundVar *v* \longleftrightarrow (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*:

assumes *ground*: *groundVar* *v*
obtains (*LVar*) *ln* **where** *v*=*LVar* *ln*
 | (*FVar*) *accC* *statDeclC* *stat* *a* *fn* **where** *v*={*accC*,*statDeclC*,*stat*}(*Lit* *a*)..*fn*
 | (*AVar*) *a* *i* **where** *v*=(*Lit* *a*).[*Lit* *i*]
<proof>

definition

groundExprs :: *expr* *list* \Rightarrow *bool*
where *groundExprs* *es* $\longleftrightarrow (\forall e \in \text{set } es. \exists v. e = \text{Lit } v)$

primrec *the-val*:: *expr* \Rightarrow *val*

where *the-val* (*Lit* *v*) = *v*

primrec *the-var*:: *prog* \Rightarrow *state* \Rightarrow *var* \Rightarrow (*vvar* \times *state*) **where**

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*]

abbreviation

Ref :: *loc* \Rightarrow *expr*

where *Ref* *a* == *Lit* (*Addr* *a*)

abbreviation

$$SKIP :: \text{expr}$$

$$\text{where } SKIP == Lit\ Unit$$
inductive

$$\text{step} :: [prog, term \times state, term \times state] \Rightarrow bool \ (\lhd \vdash - \mapsto 1 \rightarrow [61, 82, 82] \ 81)$$

$$\text{for } G :: prog$$
where

$$\begin{aligned} \text{Abrupt:} \quad & \llbracket \forall v. t \neq \langle Lit\ v \rangle; \\ & \forall t. t \neq \langle l \cdot Skip \rangle; \\ & \forall C\ vn\ c. t \neq \langle Try\ Skip\ Catch(C\ vn)\ c \rangle; \\ & \forall x\ c. t \neq \langle Skip\ Finally\ c \rangle \wedge xc \neq Xcpt\ x; \\ & \forall a\ c. t \neq \langle FinA\ a\ c \rangle \rrbracket \\ \implies & \\ & G \vdash (t, Some\ xc, s) \mapsto 1 \ (\langle Lit\ undefined \rangle, Some\ xc, s) \end{aligned}$$

$$\begin{aligned} | \text{InsInitE: } & \llbracket G \vdash (\langle c \rangle, Norm\ s) \mapsto 1 \ (\langle c' \rangle, s') \rrbracket \\ \implies & \\ & G \vdash (\langle InsInitE\ c\ e \rangle, Norm\ s) \mapsto 1 \ (\langle InsInitE\ c'\ e' \rangle, s') \end{aligned}$$

$$\begin{aligned} | \text{NewC: } & G \vdash (\langle NewC\ C' \rangle, Norm\ s) \mapsto 1 \ (\langle InsInitE\ (Init\ C)\ (NewC\ C') \rangle, Norm\ s) \\ | \text{NewCInitd: } & \llbracket G \vdash Norm\ s \text{ --halloc } (CInst\ C) \text{ --} a \rightarrow s' \rrbracket \\ \implies & \\ & G \vdash (\langle InsInitE\ Skip\ (NewC\ C') \rangle, Norm\ s) \mapsto 1 \ (\langle Ref\ a \rangle, s') \end{aligned}$$

$$\begin{aligned} | \text{NewA:} & \\ & G \vdash (\langle New\ T[e] \rangle, Norm\ s) \mapsto 1 \ (\langle InsInitE\ (init\ comp\ ty\ T)\ (New\ T[e]) \rangle, Norm\ s) \\ | \text{InsInitNewAIdx:} & \\ & \llbracket G \vdash (\langle e \rangle, Norm\ s) \mapsto 1 \ (\langle e' \rangle, s') \rrbracket \\ \implies & \\ & G \vdash (\langle InsInitE\ Skip\ (New\ T[e]) \rangle, Norm\ s) \mapsto 1 \ (\langle InsInitE\ Skip\ (New\ T[e']) \rangle, s') \\ | \text{InsInitNewA:} & \\ & \llbracket G \vdash abupd\ (check\ neg\ i)\ (Norm\ s) \text{ --halloc } (Arr\ T\ (the\ Intg\ i)) \text{ --} a \rightarrow s' \rrbracket \\ \implies & \\ & G \vdash (\langle InsInitE\ Skip\ (New\ T[Lit\ i]) \rangle, Norm\ s) \mapsto 1 \ (\langle Ref\ a \rangle, s') \end{aligned}$$

$$\begin{aligned} | \text{CastE:} & \\ & \llbracket G \vdash (\langle e \rangle, Norm\ s) \mapsto 1 \ (\langle e' \rangle, s') \rrbracket \\ \implies & \\ & G \vdash (\langle Cast\ T\ e \rangle, None, s) \mapsto 1 \ (\langle Cast\ T\ e' \rangle, s') \\ | \text{Cast:} & \\ & \llbracket s' = abupd\ (raise\ if\ (\neg G, s \vdash v\ fits\ T)\ ClassCast)\ (Norm\ s) \rrbracket \\ \implies & \\ & G \vdash (\langle Cast\ T\ (Lit\ v) \rangle, Norm\ s) \mapsto 1 \ (\langle Lit\ v \rangle, s') \end{aligned}$$

$$\begin{array}{l}
| \text{InstE: } \llbracket G \vdash (\langle e \rangle, \text{Norm } s) \mapsto 1 \ (\langle e'::\text{expr} \rangle, s') \rrbracket \\
\quad \Longrightarrow \\
\quad G \vdash (\langle e \text{ InstOf } T \rangle, \text{Norm } s) \mapsto 1 \ (\langle e' \rangle, s') \\
| \text{Inst: } \llbracket b = (v \neq \text{Null} \wedge G, s \vdash v \text{ fits RefT } T) \rrbracket \\
\quad \Longrightarrow \\
\quad G \vdash (\langle (\text{Lit } v) \text{ InstOf } T \rangle, \text{Norm } s) \mapsto 1 \ (\langle \text{Lit } (\text{Bool } b) \rangle, s') \\
\\
| \text{UnOpE: } \llbracket G \vdash (\langle e \rangle, \text{Norm } s) \mapsto 1 \ (\langle e' \rangle, s') \rrbracket \\
\quad \Longrightarrow \\
\quad G \vdash (\langle \text{UnOp unop } e \rangle, \text{Norm } s) \mapsto 1 \ (\langle \text{UnOp unop } e' \rangle, s') \\
| \text{UnOp: } G \vdash (\langle \text{UnOp unop } (\text{Lit } v) \rangle, \text{Norm } s) \mapsto 1 \ (\langle \text{Lit } (\text{eval-unop unop } v) \rangle, \text{Norm } s) \\
\\
| \text{BinOpE1: } \llbracket G \vdash (\langle e1 \rangle, \text{Norm } s) \mapsto 1 \ (\langle e1' \rangle, s') \rrbracket \\
\quad \Longrightarrow \\
\quad G \vdash (\langle \text{BinOp binop } e1 \ e2 \rangle, \text{Norm } s) \mapsto 1 \ (\langle \text{BinOp binop } e1' \ e2 \rangle, s') \\
| \text{BinOpE2: } \llbracket \text{need-second-arg binop } v1; G \vdash (\langle e2 \rangle, \text{Norm } s) \mapsto 1 \ (\langle e2' \rangle, s') \rrbracket \\
\quad \Longrightarrow \\
\quad G \vdash (\langle \text{BinOp binop } (\text{Lit } v1) \ e2 \rangle, \text{Norm } s) \\
\quad \mapsto 1 \ (\langle \text{BinOp binop } (\text{Lit } v1) \ e2' \rangle, s') \\
| \text{BinOpTerm: } \llbracket \neg \text{need-second-arg binop } v1 \rrbracket \\
\quad \Longrightarrow \\
\quad G \vdash (\langle \text{BinOp binop } (\text{Lit } v1) \ e2 \rangle, \text{Norm } s) \\
\quad \mapsto 1 \ (\langle \text{Lit } v1 \rangle, \text{Norm } s) \\
| \text{BinOp: } G \vdash (\langle \text{BinOp binop } (\text{Lit } v1) \ (\text{Lit } v2) \rangle, \text{Norm } s) \\
\quad \mapsto 1 \ (\langle \text{Lit } (\text{eval-binop binop } v1 \ v2) \rangle, \text{Norm } s) \\
\\
| \text{Super: } G \vdash (\langle \text{Super} \rangle, \text{Norm } s) \mapsto 1 \ (\langle \text{Lit } (\text{val-this } s) \rangle, \text{Norm } s) \\
\\
| \text{AccVA: } \llbracket G \vdash (\langle va \rangle, \text{Norm } s) \mapsto 1 \ (\langle va' \rangle, s') \rrbracket \\
\quad \Longrightarrow \\
\quad G \vdash (\langle \text{Acc } va \rangle, \text{Norm } s) \mapsto 1 \ (\langle \text{Acc } va' \rangle, s') \\
| \text{Acc: } \llbracket \text{groundVar } va; ((v, vf), s') = \text{the-var } G \ (\text{Norm } s) \ va \rrbracket \\
\quad \Longrightarrow \\
\quad G \vdash (\langle \text{Acc } va \rangle, \text{Norm } s) \mapsto 1 \ (\langle \text{Lit } v \rangle, s') \\
\\
| \text{AssVA: } \llbracket G \vdash (\langle va \rangle, \text{Norm } s) \mapsto 1 \ (\langle va' \rangle, s') \rrbracket \\
\quad \Longrightarrow \\
\quad G \vdash (\langle va:=e \rangle, \text{Norm } s) \mapsto 1 \ (\langle va':=e \rangle, s') \\
| \text{AssE: } \llbracket \text{groundVar } va; G \vdash (\langle e \rangle, \text{Norm } s) \mapsto 1 \ (\langle e' \rangle, s') \rrbracket \\
\quad \Longrightarrow \\
\quad G \vdash (\langle va:=e \rangle, \text{Norm } s) \mapsto 1 \ (\langle va:=e' \rangle, s') \\
| \text{Ass: } \llbracket \text{groundVar } va; ((w, f), s') = \text{the-var } G \ (\text{Norm } s) \ va \rrbracket \\
\quad \Longrightarrow \\
\quad G \vdash (\langle va:=(\text{Lit } v) \rangle, \text{Norm } s) \mapsto 1 \ (\langle \text{Lit } v \rangle, \text{assign } f \ v \ s') \\
\\
| \text{CondC: } \llbracket G \vdash (\langle e0 \rangle, \text{Norm } s) \mapsto 1 \ (\langle e0' \rangle, s') \rrbracket \\
\quad \Longrightarrow \\
\quad G \vdash (\langle e0? \ e1:e2 \rangle, \text{Norm } s) \mapsto 1 \ (\langle e0'? \ e1:e2 \rangle, s') \\
| \text{Cond: } G \vdash (\langle \text{Lit } b? \ e1:e2 \rangle, \text{Norm } s) \mapsto 1 \ (\langle \text{if the-Bool } b \text{ then } e1 \text{ else } e2 \rangle, \text{Norm } s) \\
\\
| \text{CallTarget: } \llbracket G \vdash (\langle e \rangle, \text{Norm } s) \mapsto 1 \ (\langle e' \rangle, s') \rrbracket \\
\quad \Longrightarrow
\end{array}$$

$$\begin{array}{l}
\begin{array}{l}
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 \text{groundExprs } args; vs = \text{map the-val } args; \\
D = \text{invocation-declclass } G\ mode\ s\ a\ statT\ (\langle name=mn, parTs=pTs \rangle); \\
s' = \text{init-lvars } G\ D\ (\langle name=mn, parTs=pTs \rangle\ 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 \text{Callee } (locals\ s)\ (\text{Methd } D\ (\langle name=mn, parTs=pTs \rangle)) \rangle, s') \\
| \text{ Callee: } \llbracket G \vdash (\langle e \rangle, Norm\ s) \mapsto 1\ (\langle e'::expr \rangle, s') \rrbracket \\
\implies \\
G \vdash (\langle \text{Callee lcls-caller } e \rangle, Norm\ s) \mapsto 1\ (\langle e' \rangle, s') \\
| \text{ CalleeRet: } G \vdash (\langle \text{Callee lcls-caller } (Lit\ v) \rangle, Norm\ s) \\
\mapsto 1\ (\langle Lit\ v \rangle, (\text{set-lvars lcls-caller } (Norm\ s))) \\
| \text{ Methd: } G \vdash (\langle \text{Methd } D\ sig \rangle, Norm\ s) \mapsto 1\ (\langle \text{body } G\ D\ sig \rangle, Norm\ s) \\
| \text{ Body: } G \vdash (\langle \text{Body } D\ c \rangle, Norm\ s) \mapsto 1\ (\langle \text{InsInitE } (Init\ D)\ (\text{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 \text{InsInitE Skip } (\text{Body } D\ c) \rangle, Norm\ s) \mapsto 1\ (\langle \text{InsInitE Skip } (\text{Body } D\ c') \rangle, s') \\
| \text{ InsInitBodyRet: } \\
G \vdash (\langle \text{InsInitE Skip } (\text{Body } D\ Skip) \rangle, Norm\ s) \\
\mapsto 1\ (\langle Lit\ (\text{the } ((locals\ s)\ Result)) \rangle, \text{abupd } (\text{absorb Ret})\ (Norm\ s)) \\
| \text{ FVar: } \llbracket \neg\ \text{inited statDeclC } (globs\ s) \rrbracket \\
\implies \\
G \vdash (\langle \{accC, statDeclC, stat\} e..fn \rangle, Norm\ s) \\
\mapsto 1\ (\langle \text{InsInitV } (Init\ statDeclC)\ (\{accC, statDeclC, stat\} e..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 \text{InsInitV Skip } (\{accC, statDeclC, stat\} e..fn) \rangle, Norm\ s) \\
\mapsto 1\ (\langle \text{InsInitV Skip } (\{accC, statDeclC, stat\} e'..fn) \rangle, s') \\
| \text{ InsInitFVar: } \\
G \vdash (\langle \text{InsInitV Skip } (\{accC, statDeclC, stat\} Lit\ a..fn) \rangle, Norm\ s) \\
\mapsto 1\ (\langle \{accC, statDeclC, stat\} Lit\ a..fn \rangle, Norm\ s)
\end{array}
\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*.

$$\begin{array}{l}
| \text{ AVarE1: } \llbracket G \vdash (\langle e1 \rangle, Norm\ s) \mapsto 1\ (\langle e1' \rangle, s') \rrbracket \\
\implies \\
G \vdash (\langle e1.[e2] \rangle, Norm\ s) \mapsto 1\ (\langle e1'.[e2] \rangle, s') \\
| \text{ AVarE2: } G \vdash (\langle e2 \rangle, Norm\ s) \mapsto 1\ (\langle e2' \rangle, s') \\
\implies \\
G \vdash (\langle Lit\ a.[e2] \rangle, Norm\ s) \mapsto 1\ (\langle Lit\ a.[e2'] \rangle, s')
\end{array}$$

— *Nil* is fully evaluated

$$\begin{aligned} | \text{ConsHd}: & \llbracket G \vdash (\langle e :: \text{expr} \rangle, \text{Norm } s) \mapsto 1 \ (\langle e' :: \text{expr} \rangle, s') \rrbracket \\ & \implies \\ & G \vdash (\langle e \# \text{es} \rangle, \text{Norm } s) \mapsto 1 \ (\langle e' \# \text{es} \rangle, s') \end{aligned}$$

$$\begin{aligned} | \text{ConsTl}: & \llbracket G \vdash (\langle \text{es} \rangle, \text{Norm } s) \mapsto 1 \ (\langle \text{es}' \rangle, s') \rrbracket \\ & \implies \\ & G \vdash (\langle (\text{Lit } v) \# \text{es} \rangle, \text{Norm } s) \mapsto 1 \ (\langle (\text{Lit } v) \# \text{es}' \rangle, s') \end{aligned}$$

$$| \text{Skip}: G \vdash (\langle \text{Skip} \rangle, \text{Norm } s) \mapsto 1 \ (\langle \text{SKIP} \rangle, \text{Norm } s)$$

$$\begin{aligned} | \text{ExprE}: & \llbracket G \vdash (\langle e \rangle, \text{Norm } s) \mapsto 1 \ (\langle e' \rangle, s') \rrbracket \\ & \implies \\ & G \vdash (\langle \text{Expr } e \rangle, \text{Norm } s) \mapsto 1 \ (\langle \text{Expr } e' \rangle, s') \\ | \text{Expr}: & G \vdash (\langle \text{Expr } (\text{Lit } v) \rangle, \text{Norm } s) \mapsto 1 \ (\langle \text{Skip} \rangle, \text{Norm } s) \end{aligned}$$

$$\begin{aligned} | \text{LabC}: & \llbracket G \vdash (\langle c \rangle, \text{Norm } s) \mapsto 1 \ (\langle c' \rangle, s') \rrbracket \\ & \implies \\ & G \vdash (\langle l \cdot c \rangle, \text{Norm } s) \mapsto 1 \ (\langle l \cdot c' \rangle, s') \\ | \text{Lab}: & G \vdash (\langle l \cdot \text{Skip} \rangle, s) \mapsto 1 \ (\langle \text{Skip} \rangle, \text{abupd } (\text{absorb } l) \ s) \end{aligned}$$

$$\begin{aligned} | \text{CompC1}: & \llbracket G \vdash (\langle c1 \rangle, \text{Norm } s) \mapsto 1 \ (\langle c1' \rangle, s') \rrbracket \\ & \implies \\ & G \vdash (\langle c1 ;; c2 \rangle, \text{Norm } s) \mapsto 1 \ (\langle c1' ;; c2 \rangle, s') \end{aligned}$$

$$| \text{Comp}: G \vdash (\langle \text{Skip} ;; c2 \rangle, \text{Norm } s) \mapsto 1 \ (\langle c2 \rangle, \text{Norm } s)$$

$$\begin{aligned} | \text{IfE}: & \llbracket G \vdash (\langle e \rangle, \text{Norm } s) \mapsto 1 \ (\langle e' \rangle, s') \rrbracket \\ & \implies \\ & G \vdash (\langle \text{If}(e) \ s1 \ \text{Else } s2 \rangle, \text{Norm } s) \mapsto 1 \ (\langle \text{If}(e') \ s1 \ \text{Else } s2 \rangle, s') \\ | \text{If}: & G \vdash (\langle \text{If}(\text{Lit } v) \ s1 \ \text{Else } s2 \rangle, \text{Norm } s) \\ & \mapsto 1 \ (\langle \text{if the-Bool } v \ \text{then } s1 \ \text{else } s2 \rangle, \text{Norm } s) \end{aligned}$$

$$\begin{aligned} | \text{Loop}: & G \vdash (\langle l \cdot \text{While}(e) \ c \rangle, \text{Norm } s) \\ & \mapsto 1 \ (\langle \text{If}(e) \ (\text{Cont } l \cdot c ;; l \cdot \text{While}(e) \ c) \ \text{Else } \text{Skip} \rangle, \text{Norm } s) \end{aligned}$$

$$| \text{Jmp}: G \vdash (\langle \text{Jmp } j \rangle, \text{Norm } s) \mapsto 1 \ (\langle \text{Skip} \rangle, (\text{Some } (\text{Jump } j), \ s))$$

$$\begin{aligned} | \text{ThrowE}: & \llbracket G \vdash (\langle e \rangle, \text{Norm } s) \mapsto 1 \ (\langle e' \rangle, s') \rrbracket \\ & \implies \\ & G \vdash (\langle \text{Throw } e \rangle, \text{Norm } s) \mapsto 1 \ (\langle \text{Throw } e' \rangle, s') \\ | \text{Throw}: & G \vdash (\langle \text{Throw } (\text{Lit } a) \rangle, \text{Norm } s) \mapsto 1 \ (\langle \text{Skip} \rangle, \text{abupd } (\text{throw } a) \ (\text{Norm } s)) \end{aligned}$$

$$\begin{aligned} | \text{TryC1}: & \llbracket G \vdash (\langle c1 \rangle, \text{Norm } s) \mapsto 1 \ (\langle c1' \rangle, s') \rrbracket \\ & \implies \\ & G \vdash (\langle \text{Try } c1 \ \text{Catch}(C \ vn) \ c2 \rangle, \text{Norm } s) \mapsto 1 \ (\langle \text{Try } c1' \ \text{Catch}(C \ vn) \ c2 \rangle, s') \end{aligned}$$

| *Try*: $\llbracket G \vdash s \text{ -- } s\text{alloc} \rightarrow s \rrbracket$
 \implies
 $G \vdash (\langle \text{Try Skip Catch}(C \text{ vn}) \ c2 \rangle, s)$
 $\mapsto 1 \text{ (if } G, s \vdash \text{catch } C \text{ then } (\langle c2 \rangle, \text{new-}xcpt\text{-var } vn \ s') \text{ else } (\langle \text{Skip} \rangle, s'))$

| *FinC1*: $\llbracket G \vdash (\langle c1 \rangle, \text{Norm } s) \mapsto 1 \text{ } (\langle c1' \rangle, s') \rrbracket$
 \implies
 $G \vdash (\langle c1 \text{ Finally } c2 \rangle, \text{Norm } s) \mapsto 1 \text{ } (\langle c1' \text{ Finally } c2 \rangle, s')$

| *Fin*: $G \vdash (\langle \text{Skip Finally } c2 \rangle, (a, s)) \mapsto 1 \text{ } (\langle \text{FinA } a \ c2 \rangle, \text{Norm } s)$

| *FinAC*: $\llbracket G \vdash (\langle c \rangle, s) \mapsto 1 \text{ } (\langle c' \rangle, s') \rrbracket$
 \implies
 $G \vdash (\langle \text{FinA } a \ c \rangle, s) \mapsto 1 \text{ } (\langle \text{FinA } a \ c' \rangle, s')$

| *FinA*: $G \vdash (\langle \text{FinA } a \ \text{Skip} \rangle, s) \mapsto 1 \text{ } (\langle \text{Skip} \rangle, \text{abupd (abrupt-if (} a \neq \text{None) } a) \ s)$

| *Init1*: $\llbracket \text{inited } C \text{ (globs } s) \rrbracket$
 \implies
 $G \vdash (\langle \text{Init } C \rangle, \text{Norm } s) \mapsto 1 \text{ } (\langle \text{Skip} \rangle, \text{Norm } s)$

| *Init*: $\llbracket \text{the (class } G \ C) = c; \neg \text{inited } C \text{ (globs } s) \rrbracket$
 \implies
 $G \vdash (\langle \text{Init } C \rangle, \text{Norm } s)$
 $\mapsto 1 \text{ } ((\text{if } C = \text{Object then Skip else (Init (super } c))) ;$
 $\text{Expr (Callee (locals } s) (\text{InsInitE (init } c) \text{ SKIP}))}$
 $, \text{Norm (init-class-obj } G \ C \ s))$

— *InsInitE* is just used as trick to embed the statement *init c* into an expression

| *InsInitESKIP*:
 $G \vdash (\langle \text{InsInitE Skip SKIP} \rangle, \text{Norm } s) \mapsto 1 \text{ } (\langle \text{SKIP} \rangle, \text{Norm } s)$

abbreviation

stepn:: $[prog, term \times state, nat, term \times state] \Rightarrow bool \text{ } (\langle \vdash - \mapsto - \rangle [61, 82, 82] \ 81)$
where $G \vdash p \mapsto n \ p' \equiv (p, p') \in \{(x, y). \text{ step } G \ x \ y\}^{\sim n}$

abbreviation

steptr:: $[prog, term \times state, term \times state] \Rightarrow bool \text{ } (\langle \vdash - \mapsto * \rangle [61, 82, 82] \ 81)$
where $G \vdash p \mapsto * \ p' \equiv (p, p') \in \{(x, y). \text{ step } G \ x \ y\}^*$

end

Chapter 22

AxSem

1 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 (-> Thomas Kleymann)
- expressions not flattened to elementary assignments (as usual for axiomatic semantics) but treated first-class => 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)

type-synonym *res = vals* — result entry

abbreviation (*input*)

Val **where** *Val* *x* == *In1* *x*

abbreviation (*input*)

Var **where** *Var* *x* == *In2* *x*

abbreviation (*input*)

Vals **where** *Vals* *x* == *In3* *x*

syntax

- *Val* :: [*pttrn*] => *pttrn* (‹ *Val* :-› [951] 950)
- *Var* :: [*pttrn*] => *pttrn* (‹ *Var* :-› [951] 950)

-Vals :: [pttrn] => pttrn (⋄ Vals:-> [951] 950)

translations

$\lambda Val:v . b == (\lambda v. b) \circ CONST \text{ the-In1}$
 $\lambda Var:v . b == (\lambda v. b) \circ CONST \text{ the-In2}$
 $\lambda Vals:v. b == (\lambda v. b) \circ CONST \text{ the-In3}$

— relation on result values, state and auxiliary variables

type-synonym 'a assn = res \Rightarrow state \Rightarrow 'a \Rightarrow bool

translations

(type) 'a assn <= (type) vals \Rightarrow state \Rightarrow 'a \Rightarrow bool

definition

assn-imp :: 'a assn \Rightarrow 'a assn \Rightarrow bool (**infixr** ⋄⇒ 25)
where (P \Rightarrow Q) = ($\forall Y s Z. P Y s Z \longrightarrow Q Y s Z$)

lemma assn-imp-def2 [iff]: (P \Rightarrow Q) = ($\forall Y s Z. P Y s Z \longrightarrow Q Y s Z$)
 ⋄proof⋄

assertion transformers

2 peek-and

definition

peek-and :: 'a assn \Rightarrow (state \Rightarrow bool) \Rightarrow 'a assn (**infixl** ⋄∧. 13)
where (P ∧. p) = ($\lambda Y s Z. P Y s Z \wedge p s$)

lemma peek-and-def2 [simp]: peek-and P p Y s = ($\lambda Z. (P Y s Z \wedge p s)$)
 ⋄proof⋄

lemma peek-and-Not [simp]: (P ∧. ($\lambda s. \neg f s$)) = (P ∧. Not $\circ f$)
 ⋄proof⋄

lemma peek-and-and [simp]: peek-and (peek-and P p) p = peek-and P p
 ⋄proof⋄

lemma peek-and-commut: (P ∧. p ∧. q) = (P ∧. q ∧. p)
 ⋄proof⋄

abbreviation

Normal :: 'a assn \Rightarrow 'a assn
where Normal P == P ∧. normal

lemma peek-and-Normal [simp]: peek-and (Normal P) p = Normal (peek-and P p)
 ⋄proof⋄

3 assn-supd

definition

assn-supd :: 'a assn \Rightarrow (state \Rightarrow state) \Rightarrow 'a assn (**infixl** ⋄.;. 13)
where (P ;. f) = ($\lambda Y s' Z. \exists s. P Y s Z \wedge s' = f s$)

lemma assn-supd-def2 [simp]: assn-supd P f Y s' Z = ($\exists s. P Y s Z \wedge s' = f s$)
 ⋄proof⋄

4 supd-assn

definition

supd-assn :: (state \Rightarrow state) \Rightarrow 'a assn \Rightarrow 'a assn (**infixr** ⋄.;. 13)
where (f ;. P) = ($\lambda Y s. P Y (f s)$)

lemma *supd-assn-def2* [simp]: $(f .; P) Y s = P Y (f s)$
 $\langle proof \rangle$

lemma *supd-assn-supdD* [elim]: $((f .; Q) ;. f) Y s Z \implies Q Y s Z$
 $\langle proof \rangle$

lemma *supd-assn-supdI* [elim]: $Q Y s Z \implies (f .; (Q ;. f)) Y s Z$
 $\langle proof \rangle$

5 subst-res

definition

subst-res :: $'a \text{ assn} \Rightarrow \text{res} \Rightarrow 'a \text{ assn}$ ($\langle \leftarrow \rangle$ [60,61] 60)
where $P \leftarrow w = (\lambda Y. P w)$

lemma *subst-res-def2* [simp]: $(P \leftarrow w) Y = P w$
 $\langle proof \rangle$

lemma *subst-subst-res* [simp]: $P \leftarrow w \leftarrow v = P \leftarrow w$
 $\langle proof \rangle$

lemma *peek-and-subst-res* [simp]: $(P \wedge. p) \leftarrow w = (P \leftarrow w \wedge. p)$
 $\langle proof \rangle$

6 subst-Bool

definition

subst-Bool :: $'a \text{ assn} \Rightarrow \text{bool} \Rightarrow 'a \text{ assn}$ ($\langle \leftarrow = \rangle$ [60,61] 60)
where $P \leftarrow = b = (\lambda Y s Z. \exists v. P (Val v) s Z \wedge (normal\ s \longrightarrow the\text{-}Bool\ v=b))$

lemma *subst-Bool-def2* [simp]:
 $(P \leftarrow = b) Y s Z = (\exists v. P (Val v) s Z \wedge (normal\ s \longrightarrow the\text{-}Bool\ v=b))$
 $\langle proof \rangle$

lemma *subst-Bool-the-BoolI*: $P (Val b) s Z \implies (P \leftarrow = the\text{-}Bool\ b) Y s Z$
 $\langle proof \rangle$

7 peek-res

definition

peek-res :: $(\text{res} \Rightarrow 'a \text{ assn}) \Rightarrow 'a \text{ assn}$
where *peek-res* $Pf = (\lambda Y. Pf Y Y)$

syntax

-peek-res :: $pttrn \Rightarrow 'a \text{ assn} \Rightarrow 'a \text{ assn}$ ($\langle \lambda \cdot. \cdot \rangle$ [0,3] 3)

syntax-consts

-peek-res == *peek-res*

translations

$\lambda w. P == CONST\ peek\text{-}res\ (\lambda w. P)$

lemma *peek-res-def2* [simp]: $peek\text{-}res\ P\ Y = P\ Y\ Y$
 $\langle proof \rangle$

lemma *peek-res-subst-res* [simp]: $peek\text{-}res\ P \leftarrow w = P\ w \leftarrow w$
 $\langle proof \rangle$

lemma *peek-subst-res-allI*:

$(\bigwedge a. T\ a\ (P\ (f\ a) \leftarrow f\ a)) \implies \forall a. T\ a\ (peek\text{-}res\ P \leftarrow f\ a)$
 $\langle proof \rangle$

8 ign-res

definition

ign-res :: 'a assn \Rightarrow 'a assn $(\hookrightarrow \downarrow)$ [1000] 1000)
where $P \downarrow = (\lambda Y\ s\ Z. \exists Y. P\ Y\ s\ Z)$

lemma *ign-res-def2* [simp]: $P \downarrow\ Y\ s\ Z = (\exists Y. P\ Y\ s\ Z)$
 $\langle proof \rangle$

lemma *ign-ign-res* [simp]: $P \downarrow \downarrow = P \downarrow$
 $\langle proof \rangle$

lemma *ign-subst-res* [simp]: $P \downarrow \leftarrow w = P \downarrow$
 $\langle proof \rangle$

lemma *peek-and-ign-res* [simp]: $(P \wedge. p) \downarrow = (P \downarrow \wedge. p)$
 $\langle proof \rangle$

9 peek-st

definition

peek-st :: (st \Rightarrow 'a assn) \Rightarrow 'a assn
where *peek-st* $P = (\lambda Y\ s. P\ (store\ s)\ Y\ s)$

syntax

-peek-st :: pttrn \Rightarrow 'a assn \Rightarrow 'a assn $(\hookrightarrow \lambda \dots \rightarrow [0,3]\ 3)$

syntax-consts

-peek-st == *peek-st*

translations

$\lambda s.. P == CONST\ peek\text{-}st\ (\lambda s. P)$

lemma *peek-st-def2* [simp]: $(\lambda s.. Pf\ s)\ Y\ s = Pf\ (store\ s)\ Y\ s$
 $\langle proof \rangle$

lemma *peek-st-triv* [simp]: $(\lambda s.. P) = P$
 $\langle proof \rangle$

lemma *peek-st-st* [simp]: $(\lambda s.. \lambda s'.. P\ s\ s') = (\lambda s.. P\ s\ s)$
 $\langle proof \rangle$

lemma *peek-st-split* [simp]: $(\lambda s.. \lambda Y\ s'. P\ s\ Y\ s') = (\lambda Y\ s. P\ (store\ s)\ Y\ s)$
 $\langle proof \rangle$

lemma *peek-st-subst-res* [simp]: $(\lambda s.. P\ s) \leftarrow w = (\lambda s.. P\ s \leftarrow w)$
 $\langle proof \rangle$

lemma *peek-st-Normal* [simp]: $(\lambda s.. (Normal\ (P\ s))) = Normal\ (\lambda s.. P\ s)$
 $\langle proof \rangle$

10 ign-res-eq

definition

ign-res-eq :: 'a assn \Rightarrow res \Rightarrow 'a assn $(\hookrightarrow \downarrow \Leftarrow \rightarrow)$ [60,61] 60)
where $P \downarrow = w \equiv (\lambda Y.. P \downarrow \wedge. (\lambda s. Y = w))$

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⟩

11 RefVar

definition

RefVar :: $(state \Rightarrow vvar \times state) \Rightarrow 'a \text{ assn} \Rightarrow 'a \text{ assn}$ (**infixr** <..;> 13)
where $(vf \ ..; P) = (\lambda Y \ s. \text{let } (v, s') = vf \ s \text{ in } P \ (Var \ v) \ s')$

lemma *RefVar-def2* [simp]: $(vf \ ..; P) \ Y \ s =$
 $P \ (Var \ (fst \ (vf \ s))) \ (snd \ (vf \ s))$
 ⟨proof⟩

12 allocation

definition

Alloc :: $prog \Rightarrow obj\text{-}tag \Rightarrow 'a \text{ assn} \Rightarrow 'a \text{ assn}$
where $Alloc \ G \ otag \ P = (\lambda Y \ s \ Z. \forall s' \ a. G \vdash s \text{ --halloc } otag \succ a \rightarrow s' \longrightarrow P \ (Val \ (Addr \ a)) \ s' \ Z)$

definition

SXAlloc :: $prog \Rightarrow 'a \text{ assn} \Rightarrow 'a \text{ assn}$
where $SXAlloc \ G \ P = (\lambda Y \ s \ Z. \forall s'. G \vdash s \text{ --salloc } \rightarrow s' \longrightarrow P \ Y \ s' \ Z)$

lemma *Alloc-def2* [simp]: $Alloc \ G \ otag \ P \ Y \ s \ Z =$
 $(\forall s' \ a. G \vdash s \text{ --halloc } otag \succ a \rightarrow s' \longrightarrow P \ (Val \ (Addr \ a)) \ s' \ Z)$
 ⟨proof⟩

lemma *SXAlloc-def2* [simp]:
 $SXAlloc \ G \ P \ Y \ s \ Z = (\forall s'. G \vdash s \text{ --salloc } \rightarrow s' \longrightarrow P \ Y \ s' \ Z)$
 ⟨proof⟩

validity

definition

type-ok :: $prog \Rightarrow term \Rightarrow state \Rightarrow bool$ **where**
type-ok $G \ t \ s =$
 $(\exists L \ T \ C \ A. (normal \ s \longrightarrow (\llbracket prg=G, cls=C, lcl=L \rrbracket) \vdash t :: T \wedge$
 $\llbracket prg=G, cls=C, lcl=L \rrbracket \vdash dom \ (locals \ (store \ s)) \rrangle t \rrangle A) \wedge$
 $s :: \preceq (G, L))$

datatype $'a \text{ triple} = triple \ ('a \text{ assn}) \ term \ ('a \text{ assn})$
 $(\langle \{(1-)\} / \text{--}\rangle / \{(1-)\})$ [3,65,3] 75)

type-synonym $'a \text{ triples} = 'a \text{ triple set}$

abbreviation

var-triple :: $['a \text{ assn}, var \quad , 'a \text{ assn}] \Rightarrow 'a \text{ triple}$
 $(\langle \{(1-)\} / \text{--}\rangle / \{(1-)\})$ [3,80,3] 75)

where $\{P\} e \multimap \{Q\} == \{P\} \text{ In2 } e \multimap \{Q\}$

abbreviation

$\text{expr-triple} :: ['a \text{ assn}, \text{expr} \quad , 'a \text{ assn}] \Rightarrow 'a \text{ triple}$
 $(\langle \{(1-)\} / \multimap / \{(1-)\} \rangle \quad [3,80,3] \ 75)$

where $\{P\} e \multimap \{Q\} == \{P\} \text{ In1l } e \multimap \{Q\}$

abbreviation

$\text{exprs-triple} :: ['a \text{ assn}, \text{expr list} \quad , 'a \text{ assn}] \Rightarrow 'a \text{ triple}$
 $(\langle \{(1-)\} / \multimap / \{(1-)\} \rangle \quad [3,65,3] \ 75)$

where $\{P\} e \multimap \{Q\} == \{P\} \text{ In3 } e \multimap \{Q\}$

abbreviation

$\text{stmt-triple} :: ['a \text{ assn}, \text{stmt}, \quad 'a \text{ assn}] \Rightarrow 'a \text{ triple}$
 $(\langle \{(1-)\} / \multimap / \{(1-)\} \rangle \quad [3,65,3] \ 75)$

where $\{P\} .c. \{Q\} == \{P\} \text{ In1r } c \multimap \{Q\}$

notation (ASCII)

$\text{triple} \ (\langle \{(1-)\} / \multimap / \{(1-)\} \rangle \quad [3,65,3] \ 75) \text{ and}$
 $\text{var-triple} \ (\langle \{(1-)\} / \multimap / \{(1-)\} \rangle \quad [3,80,3] \ 75) \text{ and}$
 $\text{expr-triple} \ (\langle \{(1-)\} / \multimap / \{(1-)\} \rangle \quad [3,80,3] \ 75) \text{ and}$
 $\text{exprs-triple} \ (\langle \{(1-)\} / \multimap / \{(1-)\} \rangle \quad [3,65,3] \ 75)$

lemma inj-triple: $\text{inj} \ (\lambda(P,t,Q). \{P\} t \multimap \{Q\})$

$\langle \text{proof} \rangle$

lemma triple-inj-eq: $(\{P\} t \multimap \{Q\} = \{P'\} t' \multimap \{Q'\}) = (P=P' \wedge t=t' \wedge Q=Q')$

$\langle \text{proof} \rangle$

definition $\text{mtriples} :: ('c \Rightarrow 'a \text{ sig} \Rightarrow 'a \text{ assn}) \Rightarrow ('c \Rightarrow 'a \text{ sig} \Rightarrow \text{expr}) \Rightarrow$
 $('c \Rightarrow 'a \text{ sig} \Rightarrow 'a \text{ assn}) \Rightarrow ('c \times 'a \text{ sig}) \text{ set} \Rightarrow 'a \text{ triples} \ (\langle \{(1-)\} / \multimap / \{(1-)\} \mid - \rangle [3,65,3,65] \ 75)$

where

$\{\{P\} \text{ tf} \multimap \{Q\} \mid \text{ms}\} = (\lambda(C, \text{sig}). \{\text{Normal}(P \ C \ \text{sig})\} \text{ tf } C \ \text{sig} \multimap \{Q \ C \ \text{sig}\}) 'ms$

definition

$\text{triple-valid} :: \text{prog} \Rightarrow \text{nat} \Rightarrow 'a \text{ triple} \Rightarrow \text{bool} \ (\langle \vdash \multimap \vdash \rangle [61,0, \ 58] \ 57)$

where

$G \models n:t =$
 $(\text{case } t \text{ of } \{P\} t \multimap \{Q\} \Rightarrow$
 $\forall Y \ s \ Z. P \ Y \ s \ Z \longrightarrow \text{type-ok } G \ t \ s \longrightarrow$
 $(\forall Y' \ s'. G \vdash s - t \multimap -n \longrightarrow (Y', s') \longrightarrow Q \ Y' \ s' \ Z))$

abbreviation

$\text{triples-valid} :: \text{prog} \Rightarrow \text{nat} \Rightarrow 'a \text{ triples} \Rightarrow \text{bool} \ (\langle \vdash \mid \vdash \rangle [61,0, \ 58] \ 57)$

where $G \models n:ts == \text{Ball } ts \ (\text{triple-valid } G \ n)$

notation (ASCII)

$\text{triples-valid} \ (\langle \vdash \mid \vdash \rangle [61,0, \ 58] \ 57)$

definition

$\text{ax-valids} :: \text{prog} \Rightarrow 'b \text{ triples} \Rightarrow 'a \text{ triples} \Rightarrow \text{bool} \ (\langle \vdash, - \mid \vdash \rangle [61,58,58] \ 57)$

where $(G, A \models ts) = (\forall n. G \models n:A \longrightarrow G \models n:ts)$

abbreviation

$\text{ax-valid} :: \text{prog} \Rightarrow 'b \text{ triples} \Rightarrow 'a \text{ triple} \Rightarrow \text{bool} \ (\langle \vdash, - \mid \vdash \rangle [61,58,58] \ 57)$

where $G, A \models t == G, A \models \{t\}$

notation (ASCII)

ax-valid ($\langle -, - \rangle \vdash \rightarrow$ [61,58,58] 57)

lemma *triple-valid-def2*: $G \models n: \{P\} \ t \succ \{Q\} =$
 $(\forall Y \ s \ Z. \ P \ Y \ s \ Z$
 $\longrightarrow (\exists L. (normal \ s \longrightarrow (\exists \ C \ T \ A. (\text{prg}=G, \text{cls}=C, \text{lcl}=L) \vdash t::T \wedge$
 $(\text{prg}=G, \text{cls}=C, \text{lcl}=L) \vdash \text{dom} (\text{locals} (\text{store } s)) \gg t \gg A)) \wedge$
 $s::\preceq(G, L))$
 $\longrightarrow (\forall Y' \ s'. \ G \vdash s \rightarrow t \succ n \rightarrow (Y', s') \longrightarrow Q \ Y' \ s' \ Z))$
 $\langle \text{proof} \rangle$

declare *split-paired-All* [*simp del*] *split-paired-Ex* [*simp del*]
declare *if-split* [*split del*] *if-split-asm* [*split del*]
option.split [*split del*] *option.split-asm* [*split del*]
 $\langle ML \rangle$

inductive

ax-derivs :: $prog \Rightarrow 'a \text{ triples} \Rightarrow 'a \text{ triples} \Rightarrow bool$ ($\langle -, - \rangle \vdash \rightarrow$ [61,58,58] 57)
and *ax-deriv* :: $prog \Rightarrow 'a \text{ triples} \Rightarrow 'a \text{ triple} \Rightarrow bool$ ($\langle -, - \rangle \vdash \rightarrow$ [61,58,58] 57)
for $G :: prog$
where

$G, A \vdash t \equiv G, A \vdash \{t\}$

| *empty*: $G, A \vdash \{\}$
| *insert*: $\llbracket G, A \vdash t; G, A \vdash ts \rrbracket \Longrightarrow$
 $G, A \vdash \text{insert } t \ ts$

| *asm*: $ts \subseteq A \Longrightarrow G, A \vdash ts$

| *weaken*: $\llbracket G, A \vdash ts'; ts \subseteq ts' \rrbracket \Longrightarrow G, A \vdash ts$

| *conseq*: $\forall Y \ s \ Z. \ P \ Y \ s \ Z \longrightarrow (\exists P' \ Q'. \ G, A \vdash \{P'\} \ t \succ \{Q'\} \wedge (\forall Y' \ s'.$
 $(\forall Y' \ Z'. \ P' \ Y' \ s' \ Z' \longrightarrow Q' \ Y' \ s' \ Z')) \longrightarrow$
 $Q \ Y' \ s' \ Z))$
 $\Longrightarrow G, A \vdash \{P\} \ t \succ \{Q\}$

| *hazard*: $G, A \vdash \{P \wedge. \text{Not} \circ \text{type-ok } G \ t\} \ t \succ \{Q\}$

| *Abrupt*: $G, A \vdash \{P \leftarrow (\text{undefined3 } t) \wedge. \text{Not} \circ \text{normal}\} \ t \succ \{P\}$

— variables

| *LVar*: $G, A \vdash \{\text{Normal } (\lambda s.. P \leftarrow \text{Var } (\text{lvar } vn \ s))\} \ LVar \ vn = \succ \{P\}$

| *FVar*: $\llbracket G, A \vdash \{\text{Normal } P\} \ . \text{Init } C. \{Q\};$
 $G, A \vdash \{Q\} \ e \rightarrow \{\lambda Val:a.. \text{fvar } C \ \text{stat} \ \text{fn } a \ ..; R\} \rrbracket \Longrightarrow$
 $G, A \vdash \{\text{Normal } P\} \ \{\text{acc } C, C, \text{stat}\} e.. \text{fn} = \succ \{R\}$

| *AVar*: $\llbracket G, A \vdash \{\text{Normal } P\} \ e1 \rightarrow \{Q\};$
 $\forall a. \ G, A \vdash \{Q \leftarrow \text{Val } a\} \ e2 \rightarrow \{\lambda Val:i.. \text{avar } G \ i \ a \ ..; R\} \rrbracket \Longrightarrow$
 $G, A \vdash \{\text{Normal } P\} \ e1.[e2] = \succ \{R\}$

— expressions

| *NewC*: $\llbracket G, A \vdash \{\text{Normal } P\} \ . \text{Init } C. \{\text{Alloc } G \ (C \text{Inst } C) \ Q\} \rrbracket \Longrightarrow$
 $G, A \vdash \{\text{Normal } P\} \ \text{NewC } C \rightarrow \{Q\}$

| *NewA*: $\llbracket G, A \vdash \{\text{Normal } P\} \ . \text{init-comp-ty } T. \{Q\}; \ G, A \vdash \{Q\} \ e \rightarrow$

- $$\begin{aligned} & \{ \lambda Val:i:. \text{abupd } (\text{check-neg } i) .; \text{Alloc } G (\text{Arr } T (\text{the-Intg } i)) R \} \Longrightarrow \\ & \quad G, A \vdash \{ \text{Normal } P \} \text{ New } T[e] \multimap \{ R \} \\ | \text{ Cast: } & \llbracket G, A \vdash \{ \text{Normal } P \} e \multimap \{ \lambda Val:v:. \lambda s.. \\ & \quad \text{abupd } (\text{raise-if } (\neg G, s \vdash v \text{ fits } T) \text{ ClassCast}) .; Q \leftarrow \text{Val } v \} \rrbracket \Longrightarrow \\ & \quad G, A \vdash \{ \text{Normal } P \} \text{ Cast } T e \multimap \{ Q \} \\ | \text{ Inst: } & \llbracket G, A \vdash \{ \text{Normal } P \} e \multimap \{ \lambda Val:v:. \lambda s.. \\ & \quad Q \leftarrow \text{Val } (\text{Bool } (v \neq \text{Null} \wedge G, s \vdash v \text{ fits } \text{RefT } T)) \} \rrbracket \Longrightarrow \\ & \quad G, A \vdash \{ \text{Normal } P \} e \text{ InstOf } T \multimap \{ Q \} \\ | \text{ Lit: } & \quad G, A \vdash \{ \text{Normal } (P \leftarrow \text{Val } v) \} \text{ Lit } v \multimap \{ P \} \\ | \text{ UnOp: } & \llbracket G, A \vdash \{ \text{Normal } P \} e \multimap \{ \lambda Val:v:. Q \leftarrow \text{Val } (\text{eval-unop } \text{unop } v) \} \rrbracket \\ & \quad \Longrightarrow \\ & \quad G, A \vdash \{ \text{Normal } P \} \text{ UnOp } \text{unop } e \multimap \{ Q \} \\ | \text{ BinOp: } & \llbracket G, A \vdash \{ \text{Normal } P \} e1 \multimap \{ Q \}; \\ & \quad \forall v1. G, A \vdash \{ Q \leftarrow \text{Val } v1 \} \\ & \quad \quad (\text{if need-second-arg binop } v1 \text{ then } (\text{In1l } e2) \text{ else } (\text{In1r Skip})) \multimap \\ & \quad \quad \{ \lambda Val:v2:. R \leftarrow \text{Val } (\text{eval-binop } \text{binop } v1 v2) \} \rrbracket \\ & \quad \Longrightarrow \\ & \quad G, A \vdash \{ \text{Normal } P \} \text{ BinOp } \text{binop } e1 e2 \multimap \{ R \} \\ | \text{ Super: } & G, A \vdash \{ \text{Normal } (\lambda s.. P \leftarrow \text{Val } (\text{val-this } s)) \} \text{ Super} \multimap \{ P \} \\ | \text{ Acc: } & \llbracket G, A \vdash \{ \text{Normal } P \} va \multimap \{ \lambda Var:(v,f):. Q \leftarrow \text{Val } v \} \rrbracket \Longrightarrow \\ & \quad G, A \vdash \{ \text{Normal } P \} \text{ Acc } va \multimap \{ Q \} \\ | \text{ Ass: } & \llbracket G, A \vdash \{ \text{Normal } P \} va \multimap \{ Q \}; \\ & \quad \forall vf. G, A \vdash \{ Q \leftarrow \text{Var } vf \} e \multimap \{ \lambda Val:v:. \text{assign } (\text{snd } vf) v .; R \} \rrbracket \Longrightarrow \\ & \quad G, A \vdash \{ \text{Normal } P \} va := e \multimap \{ R \} \\ | \text{ Cond: } & \llbracket G, A \vdash \{ \text{Normal } P \} e0 \multimap \{ P' \}; \\ & \quad \forall b. G, A \vdash \{ P' \leftarrow b \} (\text{if } b \text{ then } e1 \text{ else } e2) \multimap \{ Q \} \rrbracket \Longrightarrow \\ & \quad G, A \vdash \{ \text{Normal } P \} e0 ? e1 : e2 \multimap \{ Q \} \\ | \text{ Call: } & \llbracket G, A \vdash \{ \text{Normal } P \} e \multimap \{ Q \}; \forall a. G, A \vdash \{ Q \leftarrow \text{Val } a \} \text{args} \multimap \{ R a \}; \\ & \quad \forall a \text{ vs } \text{invC declC } l. G, A \vdash \{ (R a \leftarrow \text{Vals } \text{vs} \wedge. \\ & \quad (\lambda s. \text{declC} = \text{invocation-declC } G \text{ mode } (\text{store } s) a \text{ statT } (\text{!name=mn,parTs=pTs}) \wedge \\ & \quad \text{invC} = \text{invocation-class } \text{mode } (\text{store } s) a \text{ statT } \wedge \\ & \quad l = \text{locals } (\text{store } s)) ; \\ & \quad \text{init-lvars } G \text{ declC } (\text{!name=mn,parTs=pTs}) \text{ mode } a \text{ vs}) \wedge. \\ & \quad (\lambda s. \text{normal } s \longrightarrow G \vdash \text{mode} \rightarrow \text{invC} \preceq \text{statT}) \} \rrbracket \\ & \quad \text{Methd declC } (\text{!name=mn,parTs=pTs}) \multimap \{ \text{set-lvars } l .; S \} \rrbracket \Longrightarrow \\ & \quad G, A \vdash \{ \text{Normal } P \} \{ \text{accC, statT, mode} \} e \cdot \text{mn}(\{ pTs \} \text{args}) \multimap \{ S \} \\ | \text{ Methd: } & \llbracket G, A \cup \{ \{ P \} \text{Methd} \multimap \{ Q \} \mid ms \} \vdash \{ \{ P \} \text{body } G \multimap \{ Q \} \mid ms \} \rrbracket \Longrightarrow \\ & \quad G, A \vdash \{ \{ P \} \text{Methd} \multimap \{ Q \} \mid ms \} \\ | \text{ Body: } & \llbracket G, A \vdash \{ \text{Normal } P \} .\text{Init } D. \{ Q \}; \\ & \quad 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 \\ & \quad \Longrightarrow \\ & \quad G, A \vdash \{ \text{Normal } P \} \text{Body } D c \multimap \{ R \} \end{aligned}$$

— expression lists

- | *Nil*: $G, A \vdash \{ \text{Normal } (P \leftarrow \text{Vals } []) \} \Vdash \{P\}$
- | *Cons*: $\llbracket G, A \vdash \{ \text{Normal } P \} \ e \multimap \{Q\};$
 $\forall v. G, A \vdash \{ Q \leftarrow \text{Val } v \} \ es \multimap \{ \lambda \text{Vals:vs}.. R \leftarrow \text{Vals } (v \# \text{vs}) \} \rrbracket \implies$
 $G, A \vdash \{ \text{Normal } P \} \ e \# es \multimap \{R\}$
- statements
- | *Skip*: $G, A \vdash \{ \text{Normal } (P \leftarrow \Diamond) \} \ .\text{Skip}. \{P\}$
- | *Expr*: $\llbracket G, A \vdash \{ \text{Normal } P \} \ e \multimap \{ Q \leftarrow \Diamond \} \rrbracket \implies$
 $G, A \vdash \{ \text{Normal } P \} \ .\text{Expr } e. \{Q\}$
- | *Lab*: $\llbracket G, A \vdash \{ \text{Normal } P \} \ .c. \{ \text{abupd } (\text{absorb } l) \ .; Q \} \rrbracket \implies$
 $G, A \vdash \{ \text{Normal } P \} \ .l. c. \{Q\}$
- | *Comp*: $\llbracket G, A \vdash \{ \text{Normal } P \} \ .c1. \{Q\};$
 $G, A \vdash \{ Q \} \ .c2. \{R\} \rrbracket \implies$
 $G, A \vdash \{ \text{Normal } P \} \ .c1;;c2. \{R\}$
- | *If*: $\llbracket G, A \vdash \{ \text{Normal } P \} \ e \multimap \{P'\};$
 $\forall b. G, A \vdash \{ P' \leftarrow b \} \ .(\text{if } b \text{ then } c1 \text{ else } c2). \{Q\} \rrbracket \implies$
 $G, A \vdash \{ \text{Normal } P \} \ .\text{If}(e) \ c1 \text{ Else } c2. \{Q\}$
- | *Loop*: $\llbracket G, A \vdash \{ P \} \ e \multimap \{P'\};$
 $G, A \vdash \{ \text{Normal } (P' \leftarrow \text{True}) \} \ .c. \{ \text{abupd } (\text{absorb } (\text{Cont } l)) \ .; P \} \rrbracket \implies$
 $G, A \vdash \{ P \} \ .l. \text{While}(e) \ c. \{ (P' \leftarrow \text{False}) \downarrow = \Diamond \}$
- | *Jmp*: $G, A \vdash \{ \text{Normal } (\text{abupd } (\lambda a. (\text{Some } (\text{Jump } j)))) \ .; P \leftarrow \Diamond \} \ .\text{Jmp } j. \{P\}$
- | *Throw*: $\llbracket G, A \vdash \{ \text{Normal } P \} \ e \multimap \{ \lambda \text{Val:a}.. \text{abupd } (\text{throw } a) \ .; Q \leftarrow \Diamond \} \rrbracket \implies$
 $G, A \vdash \{ \text{Normal } P \} \ .\text{Throw } e. \{Q\}$
- | *Try*: $\llbracket G, A \vdash \{ \text{Normal } P \} \ .c1. \{ \text{SXAlloc } G \ Q \};$
 $G, A \vdash \{ Q \wedge (\lambda s. G, s \vdash \text{catch } C) \ .; \text{new-xcpt-var } vn \} \ .c2. \{R\};$
 $(Q \wedge (\lambda s. \neg G, s \vdash \text{catch } C)) \Rightarrow R \rrbracket \implies$
 $G, A \vdash \{ \text{Normal } P \} \ .\text{Try } c1 \text{ Catch}(C \ vn) \ c2. \{R\}$
- | *Fin*: $\llbracket G, A \vdash \{ \text{Normal } P \} \ .c1. \{Q\};$
 $\forall x. G, A \vdash \{ Q \wedge (\lambda s. x = \text{fst } s) \ .; \text{abupd } (\lambda x. \text{None}) \}$
 $\ .c2. \{ \text{abupd } (\text{abrupt-if } (x \neq \text{None}) \ x) \ .; R \} \rrbracket \implies$
 $G, A \vdash \{ \text{Normal } P \} \ .c1 \text{ Finally } c2. \{R\}$
- | *Done*: $G, A \vdash \{ \text{Normal } (P \leftarrow \Diamond \wedge \text{initd } C) \} \ .\text{Init } C. \{P\}$
- | *Init*: $\llbracket \text{the } (\text{class } G \ C) = c;$
 $G, A \vdash \{ \text{Normal } ((P \wedge \text{Not } \circ \text{initd } C) \ .; \text{supd } (\text{init-class-obj } G \ C)) \}$
 $\ .(\text{if } C = \text{Object then Skip else Init } (\text{super } c)). \{Q\};$
 $\forall l. G, A \vdash \{ Q \wedge (\lambda s. l = \text{locals } (\text{store } s)) \ .; \text{set-lvars Map.empty} \}$
 $\ .\text{init } c. \{ \text{set-lvars } l \ .; R \} \rrbracket \implies$
 $G, A \vdash \{ \text{Normal } (P \wedge \text{Not } \circ \text{initd } C) \} \ .\text{Init } C. \{R\}$

— 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*: $G, A \vdash \{ \text{Normal } P \} \ .\text{FinA } a \ c. \{Q\}$

definition

$adapt_pre :: 'a\ assn \Rightarrow 'a\ assn \Rightarrow 'a\ assn \Rightarrow 'a\ assn$
where $adapt_pre\ P\ Q\ Q' = (\lambda Y\ s\ Z. \forall Y'\ s'. \exists Z'. P\ Y\ s\ Z' \wedge (Q\ Y'\ s'\ Z' \longrightarrow Q'\ Y'\ s'\ Z))$

rules derived by induction

lemma *cut-valid*: $\llbracket G, A' \rrbracket \models ts; G, A \rrbracket \models A \rrbracket \Longrightarrow G, A \rrbracket \models ts$
 $\langle proof \rangle$

lemma *ax-thin* [rule-format (no-asm)]:

$G, (A' :: 'a\ triple\ set) \vdash (ts :: 'a\ triple\ set) \Longrightarrow \forall A. A' \subseteq A \longrightarrow G, A \vdash ts$
 $\langle proof \rangle$

lemma *ax-thin-insert*: $G, (A :: 'a\ triple\ set) \vdash (t :: 'a\ triple) \Longrightarrow G, insert\ x\ A \vdash t$
 $\langle proof \rangle$

lemma *subset-mtriples-iff*:

$ts \subseteq \{\{P\}\ mb-\succ \{Q\} \mid ms\} = (\exists ms'. ms' \subseteq ms \wedge ts = \{\{P\}\ mb-\succ \{Q\} \mid ms'\})$
 $\langle proof \rangle$

lemma *weaken*:

$G, (A :: 'a\ triple\ set) \vdash (ts' :: 'a\ triple\ set) \Longrightarrow \forall ts. ts \subseteq ts' \longrightarrow G, A \vdash ts$
 $\langle proof \rangle$

rules derived from conseq

In the following rules we often have to give some type annotations like: $G, A \vdash \{P\}\ t \succ \{Q\}$. Given only the term above without annotations, Isabelle would infer a more general type were we could have different types of auxiliary variables in the assumption set (A) and in the triple itself (P and Q). But *ax-derivs.Methd* enforces the same type in the inductive definition of the derivation. So we have to restrict the types to be able to apply the rules.

lemma *conseq12*: $\llbracket G, (A :: 'a\ triple\ set) \vdash \{P' :: 'a\ assn\}\ t \succ \{Q'\};$
 $\forall Y\ s\ Z. P\ Y\ s\ Z \longrightarrow (\forall Y'\ s'. (\forall Y\ Z'. P'\ Y\ s\ Z' \longrightarrow Q'\ Y'\ s'\ Z') \longrightarrow$
 $Q\ Y'\ s'\ Z) \rrbracket$
 $\Longrightarrow G, A \vdash \{P :: 'a\ assn\}\ t \succ \{Q\}$
 $\langle proof \rangle$

lemma *conseq12'*: $\llbracket G, (A :: 'a\ triple\ set) \vdash \{P' :: 'a\ assn\}\ t \succ \{Q'\}; \forall s\ Y'\ s'.$
 $(\forall Y\ Z. P'\ Y\ s\ Z \longrightarrow Q'\ Y'\ s'\ Z) \longrightarrow$
 $(\forall Y\ Z. P\ Y\ s\ Z \longrightarrow Q\ Y'\ s'\ Z) \rrbracket$
 $\Longrightarrow G, A \vdash \{P :: 'a\ assn\}\ t \succ \{Q\}$
 $\langle proof \rangle$

lemma *conseq12-from-conseq12'*: $\llbracket G, (A :: 'a\ triple\ set) \vdash \{P' :: 'a\ assn\}\ t \succ \{Q'\};$
 $\forall Y\ s\ Z. P\ Y\ s\ Z \longrightarrow (\forall Y'\ s'. (\forall Y\ Z'. P'\ Y\ s\ Z' \longrightarrow Q'\ Y'\ s'\ Z') \longrightarrow$
 $Q\ Y'\ s'\ Z) \rrbracket$
 $\Longrightarrow G, A \vdash \{P :: 'a\ assn\}\ t \succ \{Q\}$
 $\langle proof \rangle$

lemma *conseq1*: $\llbracket G, (A :: 'a\ triple\ set) \vdash \{P' :: 'a\ assn\}\ t \succ \{Q\}; P \Rightarrow P' \rrbracket$
 $\Longrightarrow G, A \vdash \{P :: 'a\ assn\}\ t \succ \{Q\}$
 $\langle proof \rangle$

lemma *conseq2*: $\llbracket G, (A :: 'a\ triple\ set) \vdash \{P :: 'a\ assn\}\ t \succ \{Q'\}; Q' \Rightarrow Q \rrbracket$
 $\Longrightarrow G, A \vdash \{P :: 'a\ assn\}\ t \succ \{Q\}$
 $\langle proof \rangle$

lemma *ax-escape*:

$$\begin{aligned} & \llbracket \forall Y s Z. P Y s Z \\ & \quad \longrightarrow G, (A::'a \text{ triple set}) \vdash \{\lambda Y' s' (Z'::'a). (Y', s') = (Y, s)\} \\ & \quad \quad \quad t \succ \\ & \quad \quad \quad \{\lambda Y s Z'. Q Y s Z\} \\ & \rrbracket \Longrightarrow G, A \vdash \{P::'a \text{ assn}\} t \succ \{Q::'a \text{ assn}\} \\ & \langle \text{proof} \rangle \end{aligned}$$

$$\begin{aligned} & \textbf{lemma } ax\text{-constant: } \llbracket C \Longrightarrow G, (A::'a \text{ triple set}) \vdash \{P::'a \text{ assn}\} t \succ \{Q\} \rrbracket \\ & \Longrightarrow G, A \vdash \{\lambda Y s Z. C \wedge P Y s Z\} t \succ \{Q\} \\ & \langle \text{proof} \rangle \end{aligned}$$

lemma *ax-impossible* [intro]:

$$\begin{aligned} & G, (A::'a \text{ triple set}) \vdash \{\lambda Y s Z. \text{False}\} t \succ \{Q::'a \text{ assn}\} \\ & \langle \text{proof} \rangle \end{aligned}$$

$$\begin{aligned} & \textbf{lemma } ax\text{-nochange-lemma: } \llbracket P Y s; \text{All } ((=) w) \rrbracket \Longrightarrow P w s \\ & \langle \text{proof} \rangle \end{aligned}$$

lemma *ax-nochange*:

$$\begin{aligned} & G, (A::(\text{res} \times \text{state}) \text{ triple set}) \vdash \{\lambda Y s Z. (Y, s) = Z\} t \succ \{\lambda Y s Z. (Y, s) = Z\} \\ & \Longrightarrow G, A \vdash \{P::(\text{res} \times \text{state}) \text{ assn}\} t \succ \{P\} \\ & \langle \text{proof} \rangle \end{aligned}$$

$$\begin{aligned} & \textbf{lemma } ax\text{-trivial: } G, (A::'a \text{ triple set}) \vdash \{P::'a \text{ assn}\} t \succ \{\lambda Y s Z. \text{True}\} \\ & \langle \text{proof} \rangle \end{aligned}$$

lemma *ax-disj*:

$$\begin{aligned} & \llbracket G, (A::'a \text{ triple set}) \vdash \{P1::'a \text{ assn}\} t \succ \{Q1\}; G, A \vdash \{P2::'a \text{ assn}\} t \succ \{Q2\} \rrbracket \\ & \Longrightarrow G, A \vdash \{\lambda Y s Z. P1 Y s Z \vee P2 Y s Z\} t \succ \{\lambda Y s Z. Q1 Y s Z \vee Q2 Y s Z\} \\ & \langle \text{proof} \rangle \end{aligned}$$

lemma *ax-supd-shuffle*:

$$\begin{aligned} & (\exists Q. G, (A::'a \text{ triple set}) \vdash \{P::'a \text{ assn}\} .c1. \{Q\} \wedge G, A \vdash \{Q ;. f\} .c2. \{R\}) = \\ & (\exists Q'. G, A \vdash \{P\} .c1. \{f ;. Q'\} \wedge G, A \vdash \{Q'\} .c2. \{R\}) \\ & \langle \text{proof} \rangle \end{aligned}$$

lemma *ax-cases*:

$$\begin{aligned} & \llbracket G, (A::'a \text{ triple set}) \vdash \{P \wedge. C\} t \succ \{Q::'a \text{ assn}\}; \\ & \quad G, A \vdash \{P \wedge. \text{Not} \circ C\} t \succ \{Q\} \rrbracket \Longrightarrow G, A \vdash \{P\} t \succ \{Q\} \\ & \langle \text{proof} \rangle \end{aligned}$$

$$\begin{aligned} & \textbf{lemma } ax\text{-adapt: } G, (A::'a \text{ triple set}) \vdash \{P::'a \text{ assn}\} t \succ \{Q\} \\ & \Longrightarrow G, A \vdash \{\text{adapt-pre } P Q Q'\} t \succ \{Q'\} \\ & \langle \text{proof} \rangle \end{aligned}$$

$$\begin{aligned} & \textbf{lemma } adapt\text{-pre-adapts: } G, (A::'a \text{ triple set}) \models \{P::'a \text{ assn}\} t \succ \{Q\} \\ & \longrightarrow G, A \models \{\text{adapt-pre } P Q Q'\} t \succ \{Q'\} \\ & \langle \text{proof} \rangle \end{aligned}$$

lemma *adapt-pre-weakest*:

$\forall G (A::'a \text{ triple set}) t. G, A \models \{P\} t \succ \{Q\} \longrightarrow G, A \models \{P'\} t \succ \{Q'\} \implies$
 $P' \Rightarrow \text{adapt-pre } P \ Q \ (Q'::'a \text{ assn})$
 $\langle \text{proof} \rangle$

lemma *peek-and-forget1-Normal*:

$G, (A::'a \text{ triple set}) \vdash \{\text{Normal } P\} t \succ \{Q::'a \text{ assn}\}$
 $\implies G, A \vdash \{\text{Normal } (P \wedge p)\} t \succ \{Q\}$
 $\langle \text{proof} \rangle$

lemma *peek-and-forget1*:

$G, (A::'a \text{ triple set}) \vdash \{P::'a \text{ assn}\} t \succ \{Q\}$
 $\implies G, A \vdash \{P \wedge p\} t \succ \{Q\}$
 $\langle \text{proof} \rangle$

lemmas *ax-NormalD = peek-and-forget1 [of - - - - normal]*

lemma *peek-and-forget2*:

$G, (A::'a \text{ triple set}) \vdash \{P::'a \text{ assn}\} t \succ \{Q \wedge p\}$
 $\implies G, A \vdash \{P\} t \succ \{Q\}$
 $\langle \text{proof} \rangle$

lemma *ax-subst-Val-allI*:

$\forall v. G, (A::'a \text{ triple set}) \vdash \{(P' \quad v) \leftarrow \text{Val } v\} t \succ \{(Q \ v)::'a \text{ assn}\}$
 $\implies \forall v. G, A \vdash \{(\lambda w.. P' \ (\text{the-In1 } w)) \leftarrow \text{Val } v\} t \succ \{Q \ v\}$
 $\langle \text{proof} \rangle$

lemma *ax-subst-Var-allI*:

$\forall v. G, (A::'a \text{ triple set}) \vdash \{(P' \quad v) \leftarrow \text{Var } v\} t \succ \{(Q \ v)::'a \text{ assn}\}$
 $\implies \forall v. G, A \vdash \{(\lambda w.. P' \ (\text{the-In2 } w)) \leftarrow \text{Var } v\} t \succ \{Q \ v\}$
 $\langle \text{proof} \rangle$

lemma *ax-subst-Vals-allI*:

$(\forall v. G, (A::'a \text{ triple set}) \vdash \{(P' \quad v) \leftarrow \text{Vals } v\} t \succ \{(Q \ v)::'a \text{ assn}\})$
 $\implies \forall v. G, A \vdash \{(\lambda w.. P' \ (\text{the-In3 } w)) \leftarrow \text{Vals } v\} t \succ \{Q \ v\}$
 $\langle \text{proof} \rangle$

alternative axioms

lemma *ax-Lit2*:

$G, (A::'a \text{ triple set}) \vdash \{\text{Normal } P::'a \text{ assn}\} \text{Lit } v \succ \{\text{Normal } (P \downarrow = \text{Val } v)\}$
 $\langle \text{proof} \rangle$

lemma *ax-Lit2-test-complete*:

$G, (A::'a \text{ triple set}) \vdash \{\text{Normal } (P \leftarrow \text{Val } v)::'a \text{ assn}\} \text{Lit } v \succ \{P\}$
 $\langle \text{proof} \rangle$

lemma *ax-LVar2*: $G, (A::'a \text{ triple set}) \vdash \{\text{Normal } P::'a \text{ assn}\} \text{LVar } vn \Rightarrow \{\text{Normal } (\lambda s.. P \downarrow = \text{Var } (\text{lvar } vn \ s)))\}$
 $\langle \text{proof} \rangle$

lemma *ax-Super2*: $G, (A::'a \text{ triple set}) \vdash$

$\{\text{Normal } P::'a \text{ assn}\} \text{Super} \succ \{\text{Normal } (\lambda s.. P \downarrow = \text{Val } (\text{val-this } s)))\}$
 $\langle \text{proof} \rangle$

lemma *ax-Nil2*:

$G, (A::'a \text{ triple set}) \vdash \{\text{Normal } P::'a \text{ assn}\} [] \doteq \{\text{Normal } (P \downarrow = \text{Vals } [])\}$
 $\langle \text{proof} \rangle$

misc derived structural rules

lemma *ax-finite-mtriples-lemma*: $\llbracket F \subseteq ms; \text{finite } ms; \forall (C, sig) \in ms. G, (A::'a \text{ triple set}) \vdash \{ \text{Normal } (P \ C \ sig)::'a \text{ assn} \} \text{ mb } C \ sig \multimap \{ Q \ C \ sig \} \rrbracket \implies$
 $G, A \Vdash \{ \{ P \} \text{ mb } \multimap \{ Q \} \mid F \}$

$\langle \text{proof} \rangle$

lemmas *ax-finite-mtriples* = *ax-finite-mtriples-lemma* [*OF subset-refl*]

lemma *ax-derivs-insertD*:

$G, (A::'a \text{ triple set}) \Vdash \text{insert } (t::'a \text{ triple}) \ ts \implies G, A \vdash t \wedge G, A \Vdash ts$

$\langle \text{proof} \rangle$

lemma *ax-methods-spec*:

$\llbracket G, (A::'a \text{ triple set}) \vdash \text{case-prod } f \ ' \ ms; (C, sig) \in ms \rrbracket \implies G, A \vdash ((f \ C \ sig)::'a \text{ triple})$

$\langle \text{proof} \rangle$

lemma *ax-finite-pointwise-lemma* [*rule-format*]: $\llbracket F \subseteq ms; \text{finite } ms \rrbracket \implies$

$((\forall (C, sig) \in F. G, (A::'a \text{ triple set}) \vdash (f \ C \ sig)::'a \text{ triple})) \longrightarrow (\forall (C, sig) \in ms. G, A \vdash (g \ C \ sig)::'a \text{ triple})) \longrightarrow$
 $G, A \Vdash \text{case-prod } f \ ' \ F \longrightarrow G, A \Vdash \text{case-prod } g \ ' \ F$

$\langle \text{proof} \rangle$

lemmas *ax-finite-pointwise* = *ax-finite-pointwise-lemma* [*OF subset-refl*]

lemma *ax-no-hazard*:

$G, (A::'a \text{ triple set}) \vdash \{ P \wedge. \text{type-ok } G \ t \} \ t \succ \{ Q::'a \text{ assn} \} \implies G, A \vdash \{ P \} \ t \succ \{ Q \}$

$\langle \text{proof} \rangle$

lemma *ax-free-wt*:

$(\exists T \ L \ C. (\text{prg} = G, \text{cls} = C, \text{lcl} = L) \vdash t::T) \longrightarrow G, (A::'a \text{ triple set}) \vdash \{ \text{Normal } P \} \ t \succ \{ Q::'a \text{ assn} \} \implies$
 $G, A \vdash \{ \text{Normal } P \} \ t \succ \{ Q \}$

$\langle \text{proof} \rangle$

$\langle \text{ML} \rangle$

declare *ax-Abrupts* [*intro!*]

lemmas *ax-Normal-cases* = *ax-cases* [*of - - - normal*]

lemma *ax-Skip* [*intro!*]: $G, (A::'a \text{ triple set}) \vdash \{ P \leftarrow \Diamond \} . \text{Skip}. \{ P::'a \text{ assn} \}$

$\langle \text{proof} \rangle$

lemmas *ax-SkipI* = *ax-Skip* [*THEN conseq1*]

derived rules for methd call

lemma *ax-Call-known-DynT*:

$\llbracket G \vdash \text{IntVir} \rightarrow C \preceq \text{statT};$
 $\forall a \text{ vs } l. G, A \vdash \{ (R \ a \leftarrow \text{Vals } vs \wedge. (\lambda s. l = \text{locals } (store \ s))) ;.$
 $\text{init-lvars } G \ C \ (\text{name} = mn, \text{parTs} = pTs) \ \text{IntVir } a \text{ vs} \}$
 $\text{Methd } C \ (\text{name} = mn, \text{parTs} = pTs) \multimap \{ \text{set-lvars } l \ .; S \};$
 $\forall a. G, A \vdash \{ Q \leftarrow \text{Val } a \} \ \text{args} \multimap$
 $\{ R \ a \wedge. (\lambda s. C = \text{obj-class } (the \ (heap \ (store \ s)) \ (the\text{-Addr } a))) \wedge$
 $C = \text{invocation-declclass}$
 $G \ \text{IntVir } (store \ s) \ a \ \text{statT } (\text{name} = mn, \text{parTs} = pTs) \ \};$
 $G, (A::'a \text{ triple set}) \vdash \{ \text{Normal } P \} \ e \multimap \{ Q::'a \text{ assn} \} \rrbracket$
 $\implies G, A \vdash \{ \text{Normal } P \} \ \{ \text{acc } C, \text{statT}, \text{IntVir} \} e.mn(\{ pTs \} \text{args}) \multimap \{ S \}$

$\langle \text{proof} \rangle$

lemma *ax-Call-Static*:

$$\begin{aligned} & \llbracket \forall a \text{ vs } l. G, A \vdash \{R \leftarrow \text{Vals } vs \wedge. (\lambda s. l = \text{locals } (\text{store } s)) \}; \\ & \quad \text{init-lvars } G \ C \ (\llbracket \text{name}=\text{mn}, \text{parTs}=\text{pTs} \rrbracket) \ \text{Static any-Addr } vs \} \\ & \quad \text{Methd } C \ (\llbracket \text{name}=\text{mn}, \text{parTs}=\text{pTs} \rrbracket) \text{--} \succ \ \{ \text{set-lvars } l \ .; S \}; \\ & \quad G, A \vdash \{ \text{Normal } P \} \ e \text{--} \succ \ \{ Q \}; \\ & \quad \forall a. G, (A::'a \text{ triple set}) \vdash \{ Q \leftarrow \text{Val } a \} \ \text{args} \dot{=} \succ \ \{ (R::\text{val} \Rightarrow 'a \text{ assn}) \ a \\ & \quad \wedge. (\lambda s. C = \text{invocation-declclass} \\ & \quad \quad G \ \text{Static } (\text{store } s) \ a \ \text{statT } (\llbracket \text{name}=\text{mn}, \text{parTs}=\text{pTs} \rrbracket)) \} \\ & \rrbracket \implies G, A \vdash \{ \text{Normal } P \} \ \{ \text{accC}, \text{statT}, \text{Static} \} e \cdot \text{mn}(\{ \text{pTs} \} \text{args}) \text{--} \succ \ \{ S \} \\ & \langle \text{proof} \rangle \end{aligned}$$

lemma *ax-Methd1*:

$$\begin{aligned} & \llbracket G, A \cup \{ \{ P \} \ \text{Methd} \text{--} \succ \ \{ Q \} \mid ms \} \vdash \{ \{ P \} \ \text{body } G \text{--} \succ \ \{ Q \} \mid ms \}; (C, \text{sig}) \in ms \rrbracket \implies \\ & \quad G, A \vdash \{ \text{Normal } (P \ C \ \text{sig}) \} \ \text{Methd } C \ \text{sig} \text{--} \succ \ \{ Q \ C \ \text{sig} \} \\ & \langle \text{proof} \rangle \end{aligned}$$

lemma *ax-MethdN*:

$$\begin{aligned} & G, \text{insert}(\{ \text{Normal } P \} \ \text{Methd } C \ \text{sig} \text{--} \succ \ \{ Q \}) \ A \vdash \\ & \quad \{ \text{Normal } P \} \ \text{body } G \ C \ \text{sig} \text{--} \succ \ \{ Q \} \implies \\ & \quad G, A \vdash \{ \text{Normal } P \} \ \text{Methd } C \ \text{sig} \text{--} \succ \ \{ Q \} \\ & \langle \text{proof} \rangle \end{aligned}$$

lemma *ax-StatRef*:

$$G, (A::'a \text{ triple set}) \vdash \{ \text{Normal } (P \leftarrow \text{Val } \text{Null}) \} \ \text{StatRef } rt \text{--} \succ \ \{ P::'a \text{ assn} \}$$

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

rules derived from Init and Done

lemma *ax-InitS*: $\llbracket \text{the } (\text{class } G \ C) = c; C \neq \text{Object};$

$$\begin{aligned} & \forall l. G, A \vdash \{ Q \wedge. (\lambda s. l = \text{locals } (\text{store } s)) \}; \text{set-lvars } \text{Map.empty} \} \\ & \quad \cdot \text{init } c. \{ \text{set-lvars } l \ .; R \}; \\ & \quad G, A \vdash \{ \text{Normal } ((P \wedge. \text{Not} \circ \text{initd } C) \ .; \text{supd } (\text{init-class-obj } G \ C)) \} \\ & \quad \cdot \text{Init } (\text{super } c). \{ Q \} \rrbracket \implies \\ & \quad G, (A::'a \text{ triple set}) \vdash \{ \text{Normal } (P \wedge. \text{Not} \circ \text{initd } C) \} \cdot \text{Init } C. \{ R::'a \text{ assn} \} \\ & \langle \text{proof} \rangle \end{aligned}$$

lemma *ax-Init-Skip-lemma*:

$$\begin{aligned} & \forall l. G, (A::'a \text{ triple set}) \vdash \{ P \leftarrow \Diamond \wedge. (\lambda s. l = \text{locals } (\text{store } s)) \}; \text{set-lvars } l' \} \\ & \quad \cdot \text{Skip}. \{ (\text{set-lvars } l \ .; P)::'a \text{ assn} \} \\ & \langle \text{proof} \rangle \end{aligned}$$

lemma *ax-triv-InitS*: $\llbracket \text{the } (\text{class } G \ C) = c; \text{init } c = \text{Skip}; C \neq \text{Object};$

$$\begin{aligned} & P \leftarrow \Diamond \Rightarrow (\text{supd } (\text{init-class-obj } G \ C) \ .; P); \\ & \quad G, A \vdash \{ \text{Normal } (P \wedge. \text{initd } C) \} \cdot \text{Init } (\text{super } c). \{ (P \wedge. \text{initd } C) \leftarrow \Diamond \} \rrbracket \implies \\ & \quad G, (A::'a \text{ triple set}) \vdash \{ \text{Normal } P \leftarrow \Diamond \} \cdot \text{Init } C. \{ (P \wedge. \text{initd } C)::'a \text{ assn} \} \\ & \langle \text{proof} \rangle \end{aligned}$$

lemma *ax-Init-Object*: $\text{wf-prog } G \implies G, (A::'a \text{ triple set}) \vdash$

$$\begin{aligned} & \{ \text{Normal } ((\text{supd } (\text{init-class-obj } G \ \text{Object}) \ .; P \leftarrow \Diamond) \wedge. \text{Not} \circ \text{initd } \text{Object}) \} \\ & \quad \cdot \text{Init } \text{Object}. \{ (P \wedge. \text{initd } \text{Object})::'a \text{ assn} \} \\ & \langle \text{proof} \rangle \end{aligned}$$

lemma *ax-triv-Init-Object*: $\llbracket \text{wf-prog } G;$

$$\begin{aligned} & (P::'a \text{ assn}) \Rightarrow (\text{supd } (\text{init-class-obj } G \ \text{Object}) \ .; P) \rrbracket \implies \\ & \quad G, (A::'a \text{ triple set}) \vdash \{ \text{Normal } P \leftarrow \Diamond \} \cdot \text{Init } \text{Object}. \{ P \wedge. \text{initd } \text{Object} \} \\ & \langle \text{proof} \rangle \end{aligned}$$

introduction rules for Alloc and SXAlloc

lemma *ax-SXAlloc-Normal*:

$G, (A :: 'a \text{ triple set}) \vdash \{P :: 'a \text{ assn}\} .c. \{Normal\ Q\}$
 $\implies G, A \vdash \{P\} .c. \{SXAlloc\ G\ Q\}$
 $\langle \text{proof} \rangle$

lemma *ax-Alloc*:

$G, (A :: 'a \text{ triple set}) \vdash \{P :: 'a \text{ assn}\} \ t \succ$
 $\{Normal\ (\lambda Y\ (x, s)\ Z.\ (\forall a.\ new_Addr\ (heap\ s) = Some\ a \implies$
 $Q\ (Val\ (Addr\ a))\ (Norm\ (init_obj\ G\ (CInst\ C)\ (Heap\ a)\ s))\ Z)) \wedge.$
 $heap_free\ (Suc\ (Suc\ 0))\}$
 $\implies G, A \vdash \{P\} \ t \succ \{Alloc\ G\ (CInst\ C)\ Q\}$
 $\langle \text{proof} \rangle$

lemma *ax-Alloc-Arr*:

$G, (A :: 'a \text{ triple set}) \vdash \{P :: 'a \text{ assn}\} \ t \succ$
 $\{\lambda Val:i:.. Normal\ (\lambda Y\ (x, s)\ Z.\ \neg the_Intg\ i < 0 \wedge$
 $(\forall a.\ new_Addr\ (heap\ s) = Some\ a \implies$
 $Q\ (Val\ (Addr\ a))\ (Norm\ (init_obj\ G\ (Arr\ T\ (the_Intg\ i))\ (Heap\ a)\ s))\ Z)) \wedge.$
 $heap_free\ (Suc\ (Suc\ 0))\}$
 \implies
 $G, A \vdash \{P\} \ t \succ \{\lambda Val:i:.. abupd\ (check_neg\ i) \ ;\ Alloc\ G\ (Arr\ T\ (the_Intg\ i))\ Q\}$
 $\langle \text{proof} \rangle$

lemma *ax-SXAlloc-catch-SXcpt*:

$\llbracket G, (A :: 'a \text{ triple set}) \vdash \{P :: 'a \text{ assn}\} \ t \succ$
 $\{(\lambda Y\ (x, s)\ Z.\ x = Some\ (Xcpt\ (Std\ xn)) \wedge$
 $(\forall a.\ new_Addr\ (heap\ s) = Some\ a \implies$
 $Q\ Y\ (Some\ (Xcpt\ (Loc\ a)), init_obj\ G\ (CInst\ (SXcpt\ xn))\ (Heap\ a)\ s)\ Z))$
 $\wedge. heap_free\ (Suc\ (Suc\ 0))\} \rrbracket$
 \implies
 $G, A \vdash \{P\} \ t \succ \{SXAlloc\ G\ (\lambda Y\ s\ Z.\ Q\ Y\ s\ Z \wedge G, s \vdash catch\ SXcpt\ xn)\}$
 $\langle \text{proof} \rangle$

end

Chapter 23

AxSound

1 Soundness proof for Axiomatic semantics of Java expressions and statements

theory *AxSound* imports *AxSem* begin

validity

definition

triple-valid2 :: *prog* \Rightarrow *nat* \Rightarrow 'a *triple* \Rightarrow *bool* ($\langle -, - \rangle$ [61, 0, 58] 57)
 where

$$G \models_{n::t} =$$

$$(\text{case } t \text{ of } \{P\} \ t \succ \{Q\} \Rightarrow$$

$$\forall Y \ s \ Z. \ P \ Y \ s \ Z \longrightarrow (\forall L. \ s :: \preceq(G, L)$$

$$\longrightarrow (\forall T \ C \ A. \ (\text{normal } s \longrightarrow (\langle \text{prg} = G, \text{cls} = C, \text{lcl} = L \rangle \vdash t :: T \wedge$$

$$\langle \text{prg} = G, \text{cls} = C, \text{lcl} = L \rangle \vdash \text{dom } (\text{locals } (\text{store } s)) \rangle t \rangle A)) \longrightarrow$$

$$(\forall Y' \ s'. \ G \vdash s \ -t \succ -n \longrightarrow (Y', s') \longrightarrow Q \ Y' \ s' \ Z \wedge s' :: \preceq(G, L))))))$$

This definition differs from the ordinary *triple-valid-def* manly in the conclusion: We also ensures conformance of the result state. So we don't have to apply the type soundness lemma all the time during induction. This definition is only introduced for the soundness proof of the axiomatic semantics, in the end we will conclude to the ordinary definition.

definition

ax-valids2 :: *prog* \Rightarrow 'a *triples* \Rightarrow 'a *triples* \Rightarrow *bool* ($\langle -, - \rangle$ [61, 58, 58] 57)
 where $G, A \models_{::ts} = (\forall n. (\forall t \in A. G \models_{n::t}) \longrightarrow (\forall t \in ts. G \models_{n::t}))$

lemma *triple-valid2-def2*: $G \models_{n::\{P\}} \ t \succ \{Q\} =$

$(\forall Y \ s \ Z. \ P \ Y \ s \ Z \longrightarrow (\forall Y' \ s'. \ G \vdash s \ -t \succ -n \longrightarrow (Y', s') \longrightarrow$
 $(\forall L. \ s :: \preceq(G, L) \longrightarrow (\forall T \ C \ A. \ (\text{normal } s \longrightarrow (\langle \text{prg} = G, \text{cls} = C, \text{lcl} = L \rangle \vdash t :: T \wedge$
 $\langle \text{prg} = G, \text{cls} = C, \text{lcl} = L \rangle \vdash \text{dom } (\text{locals } (\text{store } s)) \rangle t \rangle A)) \longrightarrow$
 $Q \ Y' \ s' \ Z \wedge s' :: \preceq(G, L))))))$
 $\langle \text{proof} \rangle$

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*: $G \models 0 :: \{\text{Normal } P\} \ \text{Methd } C \ \text{sig} \succ \{Q\}$

$\langle \text{proof} \rangle$

lemma *Method-triple-valid2-SucI*:

$\llbracket G \models n :: \{ \text{Normal } P \} \text{ body } G \ C \ \text{sig} \multimap \{ Q \} \rrbracket$
 $\implies G \models \text{Suc } n :: \{ \text{Normal } P \} \text{ Method } C \ \text{sig} \multimap \{ Q \}$
 $\langle \text{proof} \rangle$

lemma *triples-valid2-Suc*:

$\text{Ball } ts \ (\text{triple-valid2 } G \ (\text{Suc } n)) \implies \text{Ball } ts \ (\text{triple-valid2 } G \ n)$
 $\langle \text{proof} \rangle$

lemma $G \models n :: \text{insert } t \ A = (G \models n :: t \wedge G \models n :: A)$

$\langle \text{proof} \rangle$

soundness

lemma *Method-sound*:

assumes *recursive*: $G, A \cup \{ \{ P \} \text{ Method } \multimap \{ Q \} \mid ms \} \models :: \{ \{ P \} \text{ body } G \multimap \{ Q \} \mid ms \}$
shows $G, A \models :: \{ \{ P \} \text{ Method } \multimap \{ Q \} \mid ms \}$
 $\langle \text{proof} \rangle$

lemma *valids2-inductI*: $\forall s \ t \ n \ Y' \ s'. \ G \vdash s \multimap t \multimap n \rightarrow (Y', s') \longrightarrow t = c \longrightarrow$

$\text{Ball } A \ (\text{triple-valid2 } G \ n) \longrightarrow (\forall Y \ Z. \ P \ Y \ s \ Z \longrightarrow$

$(\forall L. \ s :: \preceq (G, L) \longrightarrow$

$(\forall T \ C \ A. \ (\text{normal } s \longrightarrow (\llbracket \text{prg} = G, \text{cls} = C, \text{lcl} = L \rrbracket \vdash t :: T) \wedge$

$\llbracket \text{prg} = G, \text{cls} = C, \text{lcl} = L \rrbracket \vdash \text{dom } (\text{locals } (\text{store } s)) \gg t \gg A) \longrightarrow$

$Q \ Y' \ s' \ Z \wedge s' :: \preceq (G, L))) \implies$

$G, A \models :: \{ \{ P \} \ c \multimap \{ Q \} \}$

$\langle \text{proof} \rangle$

lemma *da-good-approx-evalnE* [consumes 4]:

assumes *evaln*: $G \vdash s0 \multimap t \multimap n \rightarrow (v, s1)$

and *wt*: $\llbracket \text{prg} = G, \text{cls} = C, \text{lcl} = L \rrbracket \vdash t :: T$

and *da*: $\llbracket \text{prg} = G, \text{cls} = C, \text{lcl} = L \rrbracket \vdash \text{dom } (\text{locals } (\text{store } s0)) \gg t \gg A$

and *wf*: *wf-prog* G

and *elim*: $\llbracket \text{normal } s1 \implies \text{nrm } A \subseteq \text{dom } (\text{locals } (\text{store } s1));$

$\wedge l. \llbracket \text{abrupt } s1 = \text{Some } (\text{Jump } (\text{Break } l)); \text{normal } s0 \rrbracket$

$\implies \text{brk } A \ l \subseteq \text{dom } (\text{locals } (\text{store } s1));$

$\llbracket \text{abrupt } s1 = \text{Some } (\text{Jump } \text{Ret}); \text{normal } s0 \rrbracket$

$\implies \text{Result} \in \text{dom } (\text{locals } (\text{store } s1))$

$\rrbracket \implies P$

shows P

$\langle \text{proof} \rangle$

lemma *validI*:

assumes $I: \bigwedge n \ s0 \ L \ \text{accC} \ T \ C \ v \ s1 \ Y \ Z.$

$\llbracket \forall t \in A. \ G \models n :: t; \ s0 :: \preceq (G, L);$

$\text{normal } s0 \implies \llbracket \text{prg} = G, \text{cls} = \text{accC}, \text{lcl} = L \rrbracket \vdash t :: T;$

$\text{normal } s0 \implies \llbracket \text{prg} = G, \text{cls} = \text{accC}, \text{lcl} = L \rrbracket \vdash \text{dom } (\text{locals } (\text{store } s0)) \gg t \gg C;$

$G \vdash s0 \multimap t \multimap n \rightarrow (v, s1); \ P \ Y \ s0 \ Z \rrbracket \implies Q \ v \ s1 \ Z \wedge s1 :: \preceq (G, L)$

shows $G, A \models :: \{ \{ P \} \ t \multimap \{ Q \} \}$

$\langle \text{proof} \rangle$

declare $[[\text{simproc } \text{add}: \text{wt-expr } \text{wt-var } \text{wt-exprs } \text{wt-stmt}]]$

lemma *valid-stmtI*:

assumes $I: \bigwedge n \ s0 \ L \ \text{accC} \ C \ s1 \ Y \ Z.$

$\llbracket \forall t \in A. \ G \models n :: t; \ s0 :: \preceq (G, L);$

$$\begin{aligned} \text{normal } s0 &\Longrightarrow \langle \text{prg}=G, \text{cls}=\text{acc}C, \text{lcl}=L \rangle \vdash c::\surd; \\ \text{normal } s0 &\Longrightarrow \langle \text{prg}=G, \text{cls}=\text{acc}C, \text{lcl}=L \rangle \vdash \text{dom } (\text{locals } (\text{store } s0)) \rangle \langle c \rangle_s \rangle C; \\ G \vdash s0 -c-n \rightarrow s1; P \ Y \ s0 \ Z &\Longrightarrow Q \Diamond s1 \ Z \wedge s1::\preceq(G, L) \end{aligned}$$

shows $G, A \models::\{ \{P\} \langle c \rangle_s \succ \{Q\} \}$

<proof>

lemma *valid-stmt-NormalI*:

assumes $I: \bigwedge n \ s0 \ L \ \text{acc}C \ C \ s1 \ Y \ Z.$

$\llbracket \forall t \in A. \ G \models n::t; s0::\preceq(G, L); \text{normal } s0; \langle \text{prg}=G, \text{cls}=\text{acc}C, \text{lcl}=L \rangle \vdash c::\surd;$

$\langle \text{prg}=G, \text{cls}=\text{acc}C, \text{lcl}=L \rangle \vdash \text{dom } (\text{locals } (\text{store } s0)) \rangle \langle c \rangle_s \rangle C;$

$G \vdash s0 -c-n \rightarrow s1; (\text{Normal } P) \ Y \ s0 \ Z \rrbracket \Longrightarrow Q \Diamond s1 \ Z \wedge s1::\preceq(G, L)$

shows $G, A \models::\{ \{ \text{Normal } P \} \langle c \rangle_s \succ \{Q\} \}$

<proof>

lemma *valid-var-NormalI*:

assumes $I: \bigwedge n \ s0 \ L \ \text{acc}C \ T \ C \ \text{vf} \ s1 \ Y \ Z.$

$\llbracket \forall t \in A. \ G \models n::t; s0::\preceq(G, L); \text{normal } s0;$

$\langle \text{prg}=G, \text{cls}=\text{acc}C, \text{lcl}=L \rangle \vdash t::=T;$

$\langle \text{prg}=G, \text{cls}=\text{acc}C, \text{lcl}=L \rangle \vdash \text{dom } (\text{locals } (\text{store } s0)) \rangle \langle t \rangle_v \rangle C;$

$G \vdash s0 -t=\succ \text{vf}-n \rightarrow s1; (\text{Normal } P) \ Y \ s0 \ Z \rrbracket$

$\Longrightarrow Q \ (\text{In2 } \text{vf}) \ s1 \ Z \wedge s1::\preceq(G, L)$

shows $G, A \models::\{ \{ \text{Normal } P \} \langle t \rangle_v \succ \{Q\} \}$

<proof>

lemma *valid-expr-NormalI*:

assumes $I: \bigwedge n \ s0 \ L \ \text{acc}C \ T \ C \ v \ s1 \ Y \ Z.$

$\llbracket \forall t \in A. \ G \models n::t; s0::\preceq(G, L); \text{normal } s0;$

$\langle \text{prg}=G, \text{cls}=\text{acc}C, \text{lcl}=L \rangle \vdash t::=T;$

$\langle \text{prg}=G, \text{cls}=\text{acc}C, \text{lcl}=L \rangle \vdash \text{dom } (\text{locals } (\text{store } s0)) \rangle \langle t \rangle_e \rangle C;$

$G \vdash s0 -t=\succ v-n \rightarrow s1; (\text{Normal } P) \ Y \ s0 \ Z \rrbracket$

$\Longrightarrow Q \ (\text{In1 } v) \ s1 \ Z \wedge s1::\preceq(G, L)$

shows $G, A \models::\{ \{ \text{Normal } P \} \langle t \rangle_e \succ \{Q\} \}$

<proof>

lemma *valid-expr-list-NormalI*:

assumes $I: \bigwedge n \ s0 \ L \ \text{acc}C \ T \ C \ \text{vs} \ s1 \ Y \ Z.$

$\llbracket \forall t \in A. \ G \models n::t; s0::\preceq(G, L); \text{normal } s0;$

$\langle \text{prg}=G, \text{cls}=\text{acc}C, \text{lcl}=L \rangle \vdash t::=T;$

$\langle \text{prg}=G, \text{cls}=\text{acc}C, \text{lcl}=L \rangle \vdash \text{dom } (\text{locals } (\text{store } s0)) \rangle \langle t \rangle_l \rangle C;$

$G \vdash s0 -t=\succ \text{vs}-n \rightarrow s1; (\text{Normal } P) \ Y \ s0 \ Z \rrbracket$

$\Longrightarrow Q \ (\text{In3 } \text{vs}) \ s1 \ Z \wedge s1::\preceq(G, L)$

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*: $G \vdash s0 -t=\succ -n \rightarrow (v, s1)$

and *wt*: $\text{normal } s0 \Longrightarrow \langle \text{prg}=G, \text{cls}=\text{acc}C, \text{lcl}=L \rangle \vdash t::=T$

and *da*: $\text{normal } s0 \Longrightarrow \langle \text{prg}=G, \text{cls}=\text{acc}C, \text{lcl}=L \rangle \vdash \text{dom } (\text{locals } (\text{store } s0)) \rangle t \rangle C$

and *elim*: $\llbracket Q \ v \ s1 \ Z; s1::\preceq(G, L) \rrbracket \Longrightarrow \text{concl}$

shows *concl*

<proof>

lemma *all-empty*: $(\forall x. \ P) = P$

<proof>

corollary *evaln-type-sound*:

assumes *evaln*: $G \vdash s0 \rightarrow -t \rightarrow -n \rightarrow (v, s1)$ **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**
 $wf: wf\text{-prog } G$
shows $s1 :: \preceq (G, L) \wedge (\text{normal } s1 \longrightarrow G, L, \text{store } s1 \vdash t \rightarrow v :: \preceq T) \wedge$
 $(\text{error-free } s0 = \text{error-free } s1)$

$\langle \text{proof} \rangle$

corollary *dom-locals-evaln-mono-elim* [consumes 1]:

assumes
evaln: $G \vdash s0 \rightarrow -t \rightarrow -n \rightarrow (v, s1)$ **and**
 $hyps: \llbracket \text{dom} (\text{locals} (\text{store } s0)) \subseteq \text{dom} (\text{locals} (\text{store } s1)) \rrbracket;$
 $\bigwedge vv \ s \ \text{val}. \llbracket v = \text{In2 } vv; \text{normal } s1 \rrbracket$
 $\implies \text{dom} (\text{locals} (\text{store } s))$
 $\subseteq \text{dom} (\text{locals} (\text{store } ((\text{snd } vv) \ \text{val } s))) \rrbracket \implies P$

shows P

$\langle \text{proof} \rangle$

lemma *evaln-no-abrupt*:

$\bigwedge s \ s'. \llbracket G \vdash s \rightarrow -t \rightarrow -n \rightarrow (w, s') \rrbracket \implies \text{normal } s$

$\langle \text{proof} \rangle$

declare *inj-term-simps* [simp]

lemma *ax-sound2*:

assumes $wf: wf\text{-prog } G$

and $\text{deriv}: G, A \vdash ts$

shows $G, A \models ts$

$\langle \text{proof} \rangle$

declare *inj-term-simps* [simp del]

theorem *ax-sound*:

$wf\text{-prog } G \implies G, (A :: 'a \text{ triple set}) \vdash (ts :: 'a \text{ triple set}) \implies G, A \models ts$

$\langle \text{proof} \rangle$

lemma *sound-valid2-lemma*:

$\llbracket \forall v \ n. \text{Ball } A (\text{triple-valid2 } G \ n) \longrightarrow P \ v \ n; \text{Ball } A (\text{triple-valid2 } G \ n) \rrbracket$

$\implies P \ v \ n$

$\langle \text{proof} \rangle$

end

Chapter 24

AxCompl

1 Completeness proof for Axiomatic semantics of Java expressions and statements

theory *AxCompl* **imports** *AxSem* **begin**

design issues:

- proof structured by Most General Formulas (-> Thomas Kleymann)

set of not yet initialized classes

definition

nyinitcls :: *prog* \Rightarrow *state* \Rightarrow *qname* *set*
where *nyinitcls* *G* *s* = { *C*. *is-class* *G* *C* \wedge \neg *initd* *C* *s* }

lemma *nyinitcls-subset-class*: *nyinitcls* *G* *s* \subseteq { *C*. *is-class* *G* *C* }
<proof>

lemmas *finite-nyinitcls* [*simp*] =
finite-is-class [*THEN nyinitcls-subset-class* [*THEN finite-subset*]]

lemma *card-nyinitcls-bound*: *card* (*nyinitcls* *G* *s*) \leq *card* { *C*. *is-class* *G* *C* }
<proof>

lemma *nyinitcls-set-locals-cong* [*simp*]:
nyinitcls *G* (*x*, *set-locals* *l* *s*) = *nyinitcls* *G* (*x*, *s*)
<proof>

lemma *nyinitcls-abrupt-cong* [*simp*]: *nyinitcls* *G* (*f* *x*, *y*) = *nyinitcls* *G* (*x*, *y*)
<proof>

lemma *nyinitcls-abupd-cong* [*simp*]: *nyinitcls* *G* (*abupd* *f* *s*) = *nyinitcls* *G* *s*
<proof>

lemma *card-nyinitcls-abrupt-congE* [*elim!*]:
card (*nyinitcls* *G* (*x*, *s*)) \leq *n* \implies *card* (*nyinitcls* *G* (*y*, *s*)) \leq *n*
<proof>

lemma *nyinitcls-new-xcpt-var* [*simp*]:
nyinitcls *G* (*new-xcpt-var* *vn* *s*) = *nyinitcls* *G* *s*
<proof>

lemma *nyinitcls-init-lvars* [*simp*]:

$nyinitcls\ G\ ((init-lvars\ G\ C\ sig\ mode\ a'\ pvs)\ s) = nyinitcls\ G\ s$
 $\langle proof \rangle$

lemma *nyinitcls-emptyD*: $\llbracket nyinitcls\ G\ s = \{\};\ is-class\ G\ C \rrbracket \implies initd\ C\ s$
 $\langle proof \rangle$

lemma *card-Suc-lemma*:
 $\llbracket card\ (insert\ a\ A) \leq Suc\ n;\ a \notin A;\ finite\ A \rrbracket \implies card\ A \leq n$
 $\langle proof \rangle$

lemma *nyinitcls-le-SucD*:
 $\llbracket card\ (nyinitcls\ G\ (x,s)) \leq Suc\ n;\ \neg initd\ C\ (globs\ s);\ class\ G\ C = Some\ y \rrbracket \implies$
 $card\ (nyinitcls\ G\ (x,init-class-obj\ G\ C\ s)) \leq n$
 $\langle proof \rangle$

lemma *initd-gext'*: $\llbracket s \leq |s'; initd\ C\ (globs\ s) \rrbracket \implies initd\ C\ (globs\ s')$
 $\langle proof \rangle$

lemma *nyinitcls-gext*: $snd\ s \leq |snd\ s' \implies nyinitcls\ G\ s' \subseteq nyinitcls\ G\ s$
 $\langle proof \rangle$

lemma *card-nyinitcls-gext*:
 $\llbracket snd\ s \leq |snd\ s'; card\ (nyinitcls\ G\ s) \leq n \rrbracket \implies card\ (nyinitcls\ G\ s') \leq n$
 $\langle proof \rangle$

init-le

definition

$init-le :: prog \Rightarrow nat \Rightarrow state \Rightarrow bool\ (\hookrightarrow + init \leq \hookrightarrow\ [51,51]\ 50)$
where $G \vdash init \leq n = (\lambda s. card\ (nyinitcls\ G\ s) \leq n)$

lemma *init-le-def2* [simp]: $(G \vdash init \leq n)\ s = (card\ (nyinitcls\ G\ s) \leq n)$
 $\langle proof \rangle$

lemma *All-init-leD*:
 $\forall n::nat. G, (A::'a\ triple\ set) \vdash \{P \wedge. G \vdash init \leq n\}\ t \succ \{Q::'a\ assn\}$
 $\implies G, A \vdash \{P\}\ t \succ \{Q\}$
 $\langle proof \rangle$

Most General Triples and Formulas

definition

$remember-init-state :: state\ assn\ (\hookrightarrow \dot{=})$
where $\dot{=} \equiv \lambda Y\ s\ Z. s = Z$

lemma *remember-init-state-def2* [simp]: $\dot{=}\ Y = (=)$
 $\langle proof \rangle$

definition

$MGF :: [state\ assn,\ term,\ prog] \Rightarrow state\ triple\ (\hookrightarrow \{-\} \succ \{-\rightarrow\}) [3,65,3] 62)$
where $\{P\}\ t \succ \{G \rightarrow\} = \{P\}\ t \succ \{\lambda Y\ s'\ s. G \vdash s - t \succ \rightarrow (Y, s')\}$

definition

$MGFn :: [nat,\ term,\ prog] \Rightarrow state\ triple\ (\hookrightarrow \{=: \} \succ \{-\rightarrow\}) [3,65,3] 62)$
where $\{=: n\}\ t \succ \{G \rightarrow\} = \{\dot{=}\ \wedge. G \vdash init \leq n\}\ t \succ \{G \rightarrow\}$

lemma *MGF-valid*: $wf-prog\ G \implies G, \{\} \models \{\dot{=}\}\ t \succ \{G \rightarrow\}$
 $\langle proof \rangle$

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*:

$$(\forall T L C. (\text{prg} = G, \text{cls} = C, \text{lcl} = L) \vdash t :: T)$$

$$\longrightarrow G, (A :: \text{state triple set})$$

$$\vdash \{\text{Normal}((\lambda Y' s' s. s' = s \wedge \text{normal } s) \wedge. G \vdash \text{init} \leq n) \wedge. (\lambda s. s :: \leq (G, L))\}$$

$$t \succ$$

$$\{\lambda Y s' s. G \vdash s - t \rightarrow (Y, s')\}$$

$$\implies G, A \vdash \{=:n\} t \succ \{G \rightarrow\}$$

⟨proof⟩

To derive the most general formula, we can restrict ourselves to welltyped terms 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*:

$(\forall T L C B. (\text{prg}=G, \text{cls}=C, \text{lcl}=L) \vdash t :: T$
 $\longrightarrow G, (A :: \text{state triple set})$
 $\vdash \{ \text{Normal}((\lambda Y' s' s. s'=s \wedge \text{normal } s) \wedge. G \vdash \text{init} \leq n) \wedge. (\lambda s. s :: \leq (G, L))$
 $\wedge. (\lambda s. (\text{prg}=G, \text{cls}=C, \text{lcl}=L) \vdash \text{dom } (\text{locals } (\text{store } s)) \gg t \gg B) \}$
 $t \succ$
 $\{ \lambda Y s' s. G \vdash s - t \succ \rightarrow (Y, s') \}$
 $\implies G, A \vdash \{ =: n \} t \succ \{ G \rightarrow \}$
 $\langle \text{proof} \rangle$

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 $G, (A :: \text{state triple set}) \vdash \{ =: n \} \langle \text{Init } C \rangle_s \succ \{ G \rightarrow \}$
 $\langle \text{proof} \rangle$
lemmas $\text{MGFn-InitD} = \text{MGFn-Init } [\text{THEN } \text{MGFnD}, \text{ THEN } \text{ax-NormalD}]$

lemma *MGFn-Call*:

assumes *mgf-methds*:
 $\forall C \text{ sig}. G, (A :: \text{state triple set}) \vdash \{ =: n \} \langle (\text{Methd } C \text{ sig}) \rangle_e \succ \{ G \rightarrow \}$
and *mgf-e*: $G, A \vdash \{ =: n \} \langle e \rangle_e \succ \{ G \rightarrow \}$
and *mgf-ps*: $G, A \vdash \{ =: n \} \langle ps \rangle_l \succ \{ G \rightarrow \}$
and *wf*: *wf-prog* G
shows $G, A \vdash \{ =: n \} \langle \{ \text{acc } C, \text{stat } T, \text{mode} \} e \cdot \text{mn}(\{ pTs' \} ps) \rangle_e \succ \{ G \rightarrow \}$
 $\langle \text{proof} \rangle$

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 $\text{abrupt } s1 \neq \text{Some } (\text{Jump } j)$
 $\langle \text{proof} \rangle$

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.

definition

unroll :: $\text{prog} \Rightarrow \text{label} \Rightarrow \text{expr} \Rightarrow \text{stmt} \Rightarrow (\text{state} \times \text{state}) \text{ set}$ **where**
 $\text{unroll } G \text{ l e c} = \{ (s, t). \exists v s1 s2.$
 $G \vdash s - e \rightarrow v \rightarrow s1 \wedge \text{the-Bool } v \wedge \text{normal } s1 \wedge$
 $G \vdash s1 - c \rightarrow s2 \wedge t = (\text{abupd } (\text{absorb } (\text{Cont } l)) s2) \}$

lemma *unroll-while*:

assumes *unroll*: $(s, t) \in (\text{unroll } G \text{ l e c})^*$
and *eval-e*: $G \vdash t - e \rightarrow v \rightarrow s'$
and *normal-termination*: $\text{normal } s' \longrightarrow \neg \text{the-Bool } v$
and *wt*: $(\text{prg}=G, \text{cls}=C, \text{lcl}=L) \vdash e :: -T$
and *wf*: *wf-prog* G
shows $G \vdash s - l \cdot \text{While}(e) \text{ c} \rightarrow s'$
 $\langle \text{proof} \rangle$

lemma *MGFn-Loop*:

assumes *mgf-e*: $G, (A :: \text{state triple set}) \vdash \{=:n\} \langle e \rangle_e \succ \{G \rightarrow\}$
and *mgf-c*: $G, A \vdash \{=:n\} \langle c \rangle_s \succ \{G \rightarrow\}$
and *wf*: *wf-prog* G
shows $G, A \vdash \{=:n\} \langle l \cdot \text{While}(e) \ c \rangle_s \succ \{G \rightarrow\}$
 $\langle \text{proof} \rangle$

lemma *MGFn-FVar*:

fixes $A :: \text{state triple set}$
assumes *mgf-init*: $G, A \vdash \{=:n\} \langle \text{Init statDeclC} \rangle_s \succ \{G \rightarrow\}$
and *mgf-e*: $G, A \vdash \{=:n\} \langle e \rangle_e \succ \{G \rightarrow\}$
and *wf*: *wf-prog* G
shows $G, A \vdash \{=:n\} \langle \{ \text{accC}, \text{statDeclC}, \text{stat} \} e..fn \rangle_v \succ \{G \rightarrow\}$
 $\langle \text{proof} \rangle$

lemma *MGFn-Fin*:

assumes *wf*: *wf-prog* G
and *mgf-c1*: $G, A \vdash \{=:n\} \langle c1 \rangle_s \succ \{G \rightarrow\}$
and *mgf-c2*: $G, A \vdash \{=:n\} \langle c2 \rangle_s \succ \{G \rightarrow\}$
shows $G, (A :: \text{state triple set}) \vdash \{=:n\} \langle c1 \text{ Finally } c2 \rangle_s \succ \{G \rightarrow\}$
 $\langle \text{proof} \rangle$

lemma *Body-no-break*:

assumes *eval-init*: $G \vdash \text{Norm } s0 \rightarrow \text{Init } D \rightarrow s1$
and *eval-c*: $G \vdash s1 \rightarrow c \rightarrow s2$
and *jmpOk*: $\text{jumpNestingOkS } \{ \text{Ret} \} \ c$
and *wt-c*: $(\text{prg} = G, \text{cls} = C, \text{lcl} = L) \vdash c :: \checkmark$
and *clsD*: $\text{class } G \ D = \text{Some } d$
and *wf*: *wf-prog* G
shows $\forall l. \text{abrupt } s2 \neq \text{Some } (\text{Jump } (\text{Break } l)) \wedge$
 $\text{abrupt } s2 \neq \text{Some } (\text{Jump } (\text{Cont } l))$
 $\langle \text{proof} \rangle$

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*:

$\llbracket \forall C \text{ sig. is-methd } G \ C \text{ sig} \longrightarrow G, A \vdash \{ \dot{=} \} \text{In1l } (\text{Methd } C \text{ sig}) \succ \{G \rightarrow\}; \text{wf-prog } G \rrbracket$
 $\implies G, (A :: \text{state triple set}) \vdash \{ \dot{=} \} \ t \succ \{G \rightarrow\}$
 $\langle \text{proof} \rangle$

nested version

lemma *nesting-lemma'* [rule-format (no-asm)]:

assumes *ax-derivs-asm*: $\bigwedge A \text{ ts. } ts \subseteq A \implies P \ A \ ts$
and *MGF-nested-Methd*: $\bigwedge A \text{ pn. } \forall b \in \text{bdy } pn. P \ (\text{insert } (\text{mgf-call } pn) \ A) \ \{ \text{mgf } b \}$
 $\implies P \ A \ \{ \text{mgf-call } pn \}$

and $MGF\text{-}asm: \bigwedge A \ t. \forall pn \in U. P \ A \ \{mgf\text{-}call \ pn\} \implies P \ A \ \{mgf \ t\}$
and $finU: finite \ U$
and $uA: uA = mgf\text{-}call' U$
shows $\forall A. A \subseteq uA \longrightarrow n \leq card \ uA \longrightarrow card \ A = card \ uA - n$
 $\longrightarrow (\forall t. P \ A \ \{mgf \ t\})$
 $\langle proof \rangle$

lemma *nesting-lemma* [rule-format (no-asm)]:
assumes $ax\text{-}derivs\text{-}asm: \bigwedge A \ ts. ts \subseteq A \implies P \ A \ ts$
and $MGF\text{-}nested\text{-}Methd: \bigwedge A \ pn. \forall b \in bdy \ pn. P \ (insert \ (mgf \ (f \ pn)) \ A) \ \{mgf \ b\}$
 $\implies P \ A \ \{mgf \ (f \ pn)\}$
and $MGF\text{-}asm: \bigwedge A \ t. \forall pn \in U. P \ A \ \{mgf \ (f \ pn)\} \implies P \ A \ \{mgf \ t\}$
and $finU: finite \ U$
shows $P \ \{\} \ \{mgf \ t\}$
 $\langle proof \rangle$

lemma *MGF-nested-Methd*: \llbracket
 $G, insert \ (\{Normal \ \doteq\} \ \langle Methd \ C \ sig \rangle_e \succ \{G \rightarrow\}) \ A$
 $\vdash \{Normal \ \doteq\} \ \langle body \ G \ C \ sig \rangle_e \succ \{G \rightarrow\}$
 $\rrbracket \implies G, A \vdash \{Normal \ \doteq\} \ \langle Methd \ C \ sig \rangle_e \succ \{G \rightarrow\}$
 $\langle proof \rangle$

lemma *MGF-deriv*: $wf\text{-}prog \ G \implies G, (\{\} :: state \ triple \ set) \vdash \{\doteq\} \ t \succ \{G \rightarrow\}$
 $\langle proof \rangle$

simultaneous version

lemma *MGF-simult-Methd-lemma*: $finite \ ms \implies$
 $G, A \cup (\lambda(C, sig). \ \{Normal \ \doteq\} \ \langle Methd \ C \ sig \rangle_e \succ \{G \rightarrow\}) \ 'ms$
 $\vdash (\lambda(C, sig). \ \{Normal \ \doteq\} \ \langle body \ G \ C \ sig \rangle_e \succ \{G \rightarrow\}) \ 'ms \implies$
 $G, A \vdash (\lambda(C, sig). \ \{Normal \ \doteq\} \ \langle Methd \ C \ sig \rangle_e \succ \{G \rightarrow\}) \ 'ms$
 $\langle proof \rangle$

lemma *MGF-simult-Methd*: $wf\text{-}prog \ G \implies$
 $G, (\{\} :: state \ triple \ set) \vdash (\lambda(C, sig). \ \{Normal \ \doteq\} \ \langle Methd \ C \ sig \rangle_e \succ \{G \rightarrow\})$
 $\ 'Collect \ (case\text{-}prod \ (is\text{-}methd \ G))$
 $\langle proof \rangle$

corollaries

lemma *eval-to-evaln*: $\llbracket G \vdash s \text{--} t \succ \rightarrow (Y', s'); type\text{-}ok \ G \ t \ s; wf\text{-}prog \ G \rrbracket$
 $\implies \exists n. G \vdash s \text{--} t \succ \text{--} n \rightarrow (Y', s')$
 $\langle proof \rangle$

lemma *MGF-complete*:
assumes $valid: G, \{\} \models \{P\} \ t \succ \{Q\}$
and $mgf: G, (\{\} :: state \ triple \ set) \vdash \{\doteq\} \ t \succ \{G \rightarrow\}$
and $wf: wf\text{-}prog \ G$
shows $G, (\{\} :: state \ triple \ set) \vdash \{P :: state \ assn\} \ t \succ \{Q\}$
 $\langle proof \rangle$

theorem *ax-complete*:
assumes $wf: wf\text{-}prog \ G$
and $valid: G, \{\} \models \{P :: state \ assn\} \ t \succ \{Q\}$
shows $G, (\{\} :: state \ triple \ set) \vdash \{P\} \ t \succ \{Q\}$
 $\langle proof \rangle$

end

Chapter 25

AxExample

1 Example of a proof based on the Bali axiomatic semantics

```
theory AxExample
imports AxSem Example
begin
```

definition

```
arr-inv :: st ⇒ bool where
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 *if-split-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
```

$(Expr\ (Ass\ (LVar\ i)\ (Acc\ (LVar\ j))))). \{Q\}$
 $\langle proof \rangle$

end