

Java Source and Bytecode Formalizations in Isabelle: Bali

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

August 15, 2018

Contents

1	Overview	7
2	Basis	11
1	Definitions extending HOL as logical basis of Bali	11
3	Table	17
1	Abstract tables and their implementation as lists	17
4	Name	25
1	Java names	25
5	Value	27
1	Java values	27
6	Type	29
1	Java types	29
7	Term	31
1	Java expressions and statements	31
8	Decl	41
1	Field, method, interface, and class declarations, whole Java programs	41
2	Modifier	41
3	Declaration (base "class" for member,interface and class declarations	43
4	Member (field or method)	43
5	Field	43
6	Method	43
7	Interface	46
8	Class	46
9	TypeRel	55
1	The relations between Java types	55
10	DeclConcepts	65
1	Advanced concepts on Java declarations like overriding, inheritance, dynamic method lookup	65
2	accessibility of types (cf. 6.6.1)	65
3	accessibility of members	66
4	imethds	87
5	accimethd	87
6	methd	88
7	accmethd	89
8	dynmethd	89

9	dynlookup	91
10	fields	91
11	accfield	92
12	is methd	92
11	WellType	95
1	Well-typedness of Java programs	95
12	DefiniteAssignment	107
1	Definite Assignment	107
2	Very restricted calculation fallback calculation	109
3	Analysis of constant expressions	110
4	Main analysis for boolean expressions	111
5	Lifting set operations to range of tables (map to a set)	112
13	WellForm	121
1	Well-formedness of Java programs	121
2	accessibility concerns	139
14	State	143
1	State for evaluation of Java expressions and statements	143
2	access	146
3	memory allocation	147
4	initialization	147
5	update	148
6	update	152
15	Eval	157
1	Operational evaluation (big-step) semantics of Java expressions and statements	157
16	Example	175
1	Example Bali program	175
17	Conform	191
1	Conformance notions for the type soundness proof for Java	191
18	DefiniteAssignmentCorrect	201
1	Correctness of Definite Assignment	201
19	TypeSafe	209
1	The type soundness proof for Java	209
2	accessibility	217
3	Ideas for the future	223
20	Evaln	225
1	Operational evaluation (big-step) semantics of Java expressions and statements	225
21	Trans	233
22	AxSem	239
1	Axiomatic semantics of Java expressions and statements (see also Eval.thy) .	239
2	peek-and	240
3	assn-supd	240
4	supd-assn	241

5	subst-res	241
6	subst-Bool	241
7	peek-res	241
8	ign-res	242
9	peek-st	242
10	ign-res-eq	243
11	RefVar	243
12	allocation	243

23 AxSound **255**

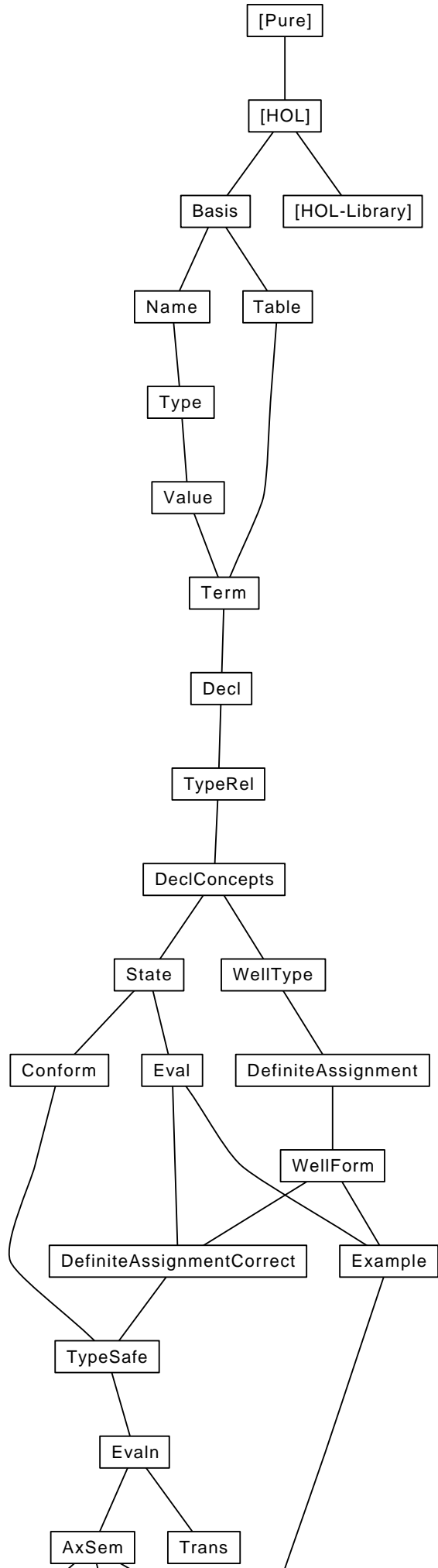
1	Soundness proof for Axiomatic semantics of Java expressions and statements	255
---	--	-----

24 AxCompl **261**

1	Completeness proof for Axiomatic semantics of Java expressions and statements	261
---	---	-----

25 AxExample **269**

1	Example of a proof based on the Bali axiomatic semantics	269
---	--	-----



Chapter 1

Overview

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

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

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

The following features are missing in Bali wrt. JavaCard:

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

- Inner Classes

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

Overview of the theories:

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

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

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

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

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

Term JavaCard terms. Variables, expressions and statements.

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

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

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

WellType Typesystem on the JavaCard term level.

DefiniteAssignment The definite assignment analysis on the JavaCard term level.

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

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

Eval Operational (big step) semantics for JavaCard.

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

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

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

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

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

Trans A smallstep operational semantics for JavaCard.

AxSem An axiomatic semantics (Hoare logic) for JavaCard.

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

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

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

Chapter 2

Basis

1 Definitions extending HOL as logical basis of Bali

```
theory Basis
imports Main
begin
```

```
misc
```

```
⟨ML⟩
```

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

```
⟨ML⟩
```

```
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)}
⟨proof⟩
```

```
lemma subset-insertD: A ⊆ insert x B ⟹ A ⊆ B ∧ x ∉ A ∨ (∃ B'. A = insert x B' ∧ B' ⊆ B)
⟨proof⟩
```

```
abbreviation nat3 :: nat (3) where 3 ≡ Suc 2
```

```
abbreviation nat4 :: nat (4) where 4 ≡ Suc 3
```

```
lemma irrefl-tranclI': r-1 ∩ r+ = {} ⟹ ∀ x. (x, x) ∉ r+
⟨proof⟩
```

```
lemma trancl-rtrancl-trancl: [(x, y) ∈ r+; (y, z) ∈ r*] ⟹ (x, z) ∈ r+
⟨proof⟩
```

```
lemma rtrancl-into-trancl3: [(a, b) ∈ r*; a ≠ b] ⟹ (a, b) ∈ r+
⟨proof⟩
```

```
lemma rtrancl-into-rtrancl2: [(a, b) ∈ r; (b, c) ∈ r*] ⟹ (a, c) ∈ r*
⟨proof⟩
```

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

lemma *rtrancl-cases*:

assumes $(a,b) \in r^*$
obtains $(\text{Refl}) a = b$
 $\mid (\text{Trancl}) (a,b) \in r^+$
<proof>

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

lemma *finite-SetCompr2*:

$\text{finite } (\text{Collect } P) \implies \forall y. P y \longrightarrow \text{finite } (\text{range } (f y)) \implies$
 $\text{finite } \{f y x \mid x y. P y\}$
<proof>

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

pairs

lemma *surjective-pairing5*:

$p = (\text{fst } p, \text{fst } (\text{snd } p), \text{fst } (\text{snd } (\text{snd } p)), \text{fst } (\text{snd } (\text{snd } (\text{snd } p))),$
 $\text{snd } (\text{snd } (\text{snd } (\text{snd } p))))$
<proof>

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

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

lemma *fst-in-set-lemma*: $(x, y) \in \text{set } l \implies x \in \text{fst } ` \text{set } l$
<proof>

quantifiers

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

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

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

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

lemma *All-Ex-refl-eq1 [simp]*: $(\forall x. (\exists b. x = f b) \longrightarrow P x) = (\forall b. P (f b))$
 ⟨proof⟩

sums

notation *case-sum* (**infixr** $'+'$) 80)

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

primrec *the-Inr* :: $'a + 'b \Rightarrow 'b$
 where *the-Inr* (*Inr* b) = b

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

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

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

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

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

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

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

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

⟨ML⟩

quantifiers for option type

syntax

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

syntax (symbols)

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

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 ⇒ 'b) ⇒ 'a ⇒ ('a → 'b) ⇒ ('a → 'b)
where *chg-map f a m* = (case m a of None ⇒ m | Some b ⇒ m(a→f b))

lemma *chg-map-new[simp]*: m a = None ⇒ *chg-map f a m* = m
 ⟨proof⟩

lemma *chg-map-upd[simp]*: m a = Some b ⇒ *chg-map f a m* = m(a→f b)
 ⟨proof⟩

lemma *chg-map-other [simp]*: a ≠ b ⇒ *chg-map f a m b* = m b
 ⟨proof⟩

unique association lists

definition *unique* :: ('a × 'b) list ⇒ bool
where *unique* = distinct ◦ map fst

lemma *uniqueD*: *unique l* ⇒ (x, y) ∈ set l ⇒ (x', y') ∈ set l ⇒ x = x' ⇒ y = y'
 ⟨proof⟩

lemma *unique-Nil [simp]*: *unique []*
 ⟨proof⟩

lemma *unique-Cons [simp]*: *unique ((x,y)#l)* = (*unique l* ∧ (∀ y. (x,y) ∉ set l))
 ⟨proof⟩

lemma *unique-ConsD*: *unique (x#xs)* ⇒ *unique xs*
 ⟨proof⟩

lemma *unique-single [simp]*: ∧p. *unique [p]*
 ⟨proof⟩

lemma *unique-append [rule-format (no-asm)]*: *unique l'* ⇒ *unique l* ⇒
 (∀ (x,y) ∈ set l. ∀ (x',y') ∈ set l'. x' ≠ x) → *unique (l @ l')*
 ⟨proof⟩

lemma *unique-map-inj*: *unique l* ⇒ inj f ⇒ *unique (map (λ(k,x). (f k, g k x)) l)*
 ⟨proof⟩

lemma *map-of-SomeI*: *unique l* ⇒ (k, x) ∈ set l ⇒ *map-of l k* = Some x
 ⟨proof⟩

list patterns

definition *lsplit* :: [['a, 'a list] ⇒ 'b, 'a list] ⇒ 'b

where $lsplit = (\lambda f l. f (hd l) (tl l))$

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

syntax

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

translations

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

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

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*:
 [[*old k = Some ov; new k = Some nv; C nv ov*]]
 ⇒ (*cond-override C old new*) *k = Some nv*
 ⟨proof⟩

lemma *cond-override-noOverride*:
 [[*old k = Some ov; new k = Some nv; ¬ (C nv ov)*]]
 ⇒ (*cond-override C old new*) *k = Some ov*
 ⟨proof⟩

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

lemma *finite-dom-cond-override*:
 [[*finite (dom s); finite (dom t)*]] ⇒ *finite (dom (cond-override C s 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*

$\langle \text{proof} \rangle$

lemma *filter-tab-False[simp]*: $\text{filter-tab } (\lambda x y. \text{False}) t = \text{Map.empty}$

$\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

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*:

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

lemma *cond-override-filter*:

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

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*:

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

declare *map-of-SomeD* [elim]

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

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

lemma *set-get-eq*:

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

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

$\langle \text{proof} \rangle$

lemma *table-of-mapconst-SomeI*:

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

lemma *table-of-mapconst-NoneI*:

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

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

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

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

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

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

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

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

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

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

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

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

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

definition
hidings-entails :: ('a, 'b) tables \Rightarrow ('a, 'c) tables \Rightarrow ('b \Rightarrow 'c \Rightarrow bool) \Rightarrow bool
 (- hidings - entails - 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 unique table:

$\text{hiding-entails} :: ('a, 'b) \text{ table} \Rightarrow ('a, 'c) \text{ table} \Rightarrow ('b \Rightarrow 'c \Rightarrow \text{bool}) \Rightarrow \text{bool}$
 $(- \text{ hiding - entails - } 20)$

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:

$\text{cond-hiding-entails} :: ('a, 'b) \text{ table} \Rightarrow ('a, 'c) \text{ table}$
 $\Rightarrow ('b \Rightarrow 'c \Rightarrow \text{bool}) \Rightarrow ('b \Rightarrow 'c \Rightarrow \text{bool}) \Rightarrow \text{bool}$
 $(- \text{ hiding - under - entails - } 20)$

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*]: $t \in ts \Longrightarrow x \in t k \Longrightarrow x \in \text{Un-tables } ts k$
 $\langle \text{proof} \rangle$

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

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

lemmas $\text{overrides-t-SomeD} = \text{overrides-t-Some-iff}$ [*THEN iffD1, dest!*]

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

lemma $\text{overrides-t-find-right}$ [*simp*]: $n k \neq \{\} \Longrightarrow (m \oplus \oplus n) k = n k$
 $\langle \text{proof} \rangle$

hiding entails

lemma hiding-entailsD :

$t \text{ hiding } s \text{ entails } R \Longrightarrow t k = \text{Some } x \Longrightarrow s k = \text{Some } y \Longrightarrow R x y$
 $\langle \text{proof} \rangle$

lemma *empty-hiding-entails* [simp]: *Map.empty hiding s entails R*
 ⟨proof⟩

lemma *hiding-empty-entails* [simp]: *t hiding Map.empty entails R*
 ⟨proof⟩

cond hiding entails

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

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

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

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

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

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

primrec *atleast-free* :: (*'a* \rightarrow *'b*) \implies *nat* \implies *bool*

where

atleast-free m 0 = True

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

lemma *atleast-free-weaken* [rule-format (no-asm)]:

$\forall m. \text{atleast-free } m \text{ (Suc } n) \longrightarrow \text{atleast-free } m \text{ n}$

⟨proof⟩

lemma *atleast-free-SucI*:

$\llbracket h \text{ a} = \text{None}; \forall \text{obj}. \text{atleast-free } (h(\text{a} \mapsto \text{obj})) \text{ n} \rrbracket \implies \text{atleast-free } h \text{ (Suc } n)$

⟨proof⟩

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

lemma *atleast-free-SucD-lemma* [rule-format (no-asm)]:

$\forall m \text{ a}. m \text{ a} = \text{None} \longrightarrow (\forall c. \text{atleast-free } (m(\text{a} \mapsto c)) \text{ n} \longrightarrow$

$(\forall b \text{ d}. \text{a} \neq b \longrightarrow \text{atleast-free } (m(\text{b} \mapsto \text{d})) \text{ n}))$

⟨proof⟩

declare *fun-upd-apply* [simp]

lemma *atleast-free-SucD*: $\text{atleast-free } h \text{ (Suc } n) \implies \text{atleast-free } (h(a|->b)) \ n$
<proof>

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

end

Chapter 4

Name

1 Java names

theory *Name* imports *Basis* begin

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

typedecl *pname* — package name

typedecl *mname* — method name

typedecl *vname* — variable or field name

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

datatype *ename* — expression name

= *VNam 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* (*VNam* *n*)

abbreviation *Result* :: *lname*

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

datatype *xname* — names of standard exceptions

= *Throwable*

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

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

lemma *xn-cases*:

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

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

$xn = \text{NegArrSize} \vee xn = \text{IndOutBound} \vee xn = \text{ArrStore}$

<proof>

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

= *Object'*

| *SXcpt'* *xname*

| *TName* *tnam*

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

pid :: *pname*

tid :: *tname*

class *has-pname* =
fixes *pname* :: 'a ⇒ *pname*

instantiation *pname* :: *has-pname*
begin

definition
pname-pname-def: *pname* (*p*::*pname*) ≡ *p*

instance ⟨*proof*⟩

end

class *has-tname* =
fixes *tname* :: 'a ⇒ *tname*

instantiation *tname* :: *has-tname*
begin

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

instance ⟨*proof*⟩

end

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

translations

(*type*) *qname* <= (*type*) (|*pid*::*pname*,*tid*::*tname*)
(*type*) 'a *qname-scheme* <= (*type*) (|*pid*::*pname*,*tid*::*tname*,...::'a)

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

definition
Object :: *qname*
where *Object* = (|*pid* = *java-lang*, *tid* = *Object*'|)

definition *SXcpt* :: *xname* ⇒ *qname*
where *SXcpt* = (λ*x*. (|*pid* = *java-lang*, *tid* = *SXcpt*' *x*|))

lemma *Object-neq-SXcpt* [*simp*]: *Object* ≠ *SXcpt* *xn*
⟨*proof*⟩

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

end

Chapter 5

Value

1 Java values

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

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

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

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* (-.) [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*
(*proof*)

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 exception object
 | *Std xname* — intermediate standard exception, see Eval.thy

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

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

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

<i>Ass var expr</i>	(<i>:-=</i> - [90,85]85)
	— variable assign
<i>Cond expr expr expr</i>	(<i>- ? - : -</i> [85,85,80]80)
	— conditional
<i>Call qname ref-ty inv-mode expr mname (ty list) (expr list)</i>	(<i>{-,,-,-}'(-)'</i> [10,10,10,85,99,10,10]85)
	— method call
	— <i>{accC,statT,mode}e.mn({pTs}args) "</i>
	— <i>accC</i> : accessing class (static class were
	— the call code is declared. Annotation only needed for
	— evaluation to check accessibility)
	— <i>statT</i> : static declaration class/interface of
	— method
	— <i>mode</i> : invocation mode
	— <i>e</i> : reference to object
	— <i>mn</i> : field name
	— <i>pTs</i> : types of parameters
	— <i>args</i> : the actual parameters/arguments
<i>Methd qname sig</i>	— (folded) method (see below)
<i>Body qname stmt</i>	— (unfolded) method body
<i>InsInitE stmt expr</i>	
	— insertion of initialization before
	— evaluation of expr (technical term for smallstep sem.)
<i>Callee locals expr</i>	— save callers locals in callee-Frame
	— (technical term for smallstep semantics)

and *stmt*

= <i>Skip</i>	— empty statement
<i>Expr expr</i>	— expression statement
<i>Lab jump stmt</i>	(<i>-· -</i> [99,66]66)
	— labeled statement; handles break
<i>Comp stmt stmt</i>	(<i>-;</i> - [66,65]65)
<i>If' expr stmt stmt</i>	(<i>If'(-)' - Else -</i> [80,79,79]70)
<i>Loop label expr stmt</i>	(<i>-· While'(-)' -</i> [99,80,79]70)
<i>Jmp jump</i>	— break, continue, return
<i>Throw expr</i>	
<i>TryC stmt qname vname stmt</i>	(<i>Try - Catch'(- -)' -</i> [79,99,80,79]70)
	— <i>Try c1 Catch(C vn) c2</i>
	— <i>c1</i> : block were 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 cateched
<i>Fin stmt stmt</i>	(<i>- Finally -</i> [79,79]70)
<i>FinA abopt stmt</i>	— Save abruption of first statement
	— technical term for smallstep sem.)
<i>Init qname</i>	— class initialization

datatype-compat *var expr stmt*

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

```
abbreviation
LAss :: vname ⇒ expr ⇒ stmt (-:==- [90,85] 85)
where v:=e == Expr (Ass (LVar (ENAME (VName v))) e)
```

```
abbreviation
Return :: expr ⇒ stmt
where Return e == Expr (Ass (LVar (ENAME 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-reus [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: $\langle c::\text{stmt} \rangle = \text{In1r } c$

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

end

lemma *stmt-inj-term-simp*: $\langle c::\text{stmt} \rangle = \text{In1r } c$

$\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

instantiation *expr :: inj-term*

begin

definition

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

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

end

lemma *expr-inj-term-simp*: $\langle e::\text{expr} \rangle = \text{In1l } e$

$\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

instantiation *var :: inj-term*

begin

definition

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

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

end

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

$\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

class *expr-of* =

fixes *expr-of* :: $'a \Rightarrow \text{expr}$

instantiation *expr :: expr-of*

begin

definition

$$\text{expr-of} = (\lambda(e::\text{expr}). e)$$
instance $\langle \text{proof} \rangle$
end
instantiation $\text{list} :: (\text{expr-of}) \text{inj-term}$
begin
definition

$$\langle \text{es}::'a \text{ list} \rangle = \text{In3} (\text{map expr-of es})$$
instance $\langle \text{proof} \rangle$
end
lemma $\text{expr-list-inj-term-def}$:

$$\langle \text{es}::\text{expr list} \rangle \equiv \text{In3 es}$$
 $\langle \text{proof} \rangle$
lemma $\text{expr-list-inj-term-simp}$: $\langle \text{es}::\text{expr list} \rangle = \text{In3 es}$
 $\langle \text{proof} \rangle$
lemma $\text{expr-list-inj-term [iff]}$: $\langle x::\text{expr list} \rangle = \langle y \rangle \equiv x = y$
 $\langle \text{proof} \rangle$
lemmas $\text{inj-term-simps} = \text{stmt-inj-term-simp expr-inj-term-simp var-inj-term-simp}$
 $\text{expr-list-inj-term-simp}$
lemmas $\text{inj-term-sym-simps} = \text{stmt-inj-term-simp [THEN sym]}$
 $\text{expr-inj-term-simp [THEN sym]}$
 $\text{var-inj-term-simp [THEN sym]}$
 $\text{expr-list-inj-term-simp [THEN sym]}$
lemma $\text{stmt-expr-inj-term [iff]}$: $\langle t::\text{stmt} \rangle \neq \langle w::\text{expr} \rangle$
 $\langle \text{proof} \rangle$
lemma $\text{expr-stmt-inj-term [iff]}$: $\langle t::\text{expr} \rangle \neq \langle w::\text{stmt} \rangle$
 $\langle \text{proof} \rangle$
lemma $\text{stmt-var-inj-term [iff]}$: $\langle t::\text{stmt} \rangle \neq \langle w::\text{var} \rangle$
 $\langle \text{proof} \rangle$
lemma $\text{var-stmt-inj-term [iff]}$: $\langle t::\text{var} \rangle \neq \langle w::\text{stmt} \rangle$
 $\langle \text{proof} \rangle$
lemma $\text{stmt-elist-inj-term [iff]}$: $\langle t::\text{stmt} \rangle \neq \langle w::\text{expr list} \rangle$
 $\langle \text{proof} \rangle$
lemma $\text{elist-stmt-inj-term [iff]}$: $\langle t::\text{expr list} \rangle \neq \langle w::\text{stmt} \rangle$
 $\langle \text{proof} \rangle$
lemma $\text{expr-var-inj-term [iff]}$: $\langle t::\text{expr} \rangle \neq \langle w::\text{var} \rangle$
 $\langle \text{proof} \rangle$

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

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

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

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

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

lemma *term-cases*:

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

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

$\llbracket \text{binop} \neq \text{CondAnd}; \text{binop} \neq \text{CondOr} \rrbracket \implies \text{need-second-arg binop } b$
<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 $\Rightarrow (b=Package \vee b=Protected \vee b=Public)$
 | *Package* $\Rightarrow (b=Protected \vee b=Public)$
 | *Protected* $\Rightarrow (b=Public)$
 | *Public* $\Rightarrow False$)

definition

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

instance

$\langle proof \rangle$

end

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

$\langle proof \rangle$

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

$\langle proof \rangle$

lemma *acc-modi-le-Public*:

$a \leq Public \Longrightarrow a=Private \vee a = Package \vee a=Protected \vee a=Public$

$\langle proof \rangle$

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

$\langle proof \rangle$

lemma *acc-modi-Private-le*:

$Private \leq a \Longrightarrow a=Private \vee a = Package \vee a=Protected \vee a=Public$

$\langle proof \rangle$

lemma *acc-modi-Package-le*:

$Package \leq a \Longrightarrow a = Package \vee a=Protected \vee a=Public$

$\langle proof \rangle$

lemma *acc-modi-le-Package*:

$a \leq Package \Longrightarrow a=Private \vee a = Package$

$\langle proof \rangle$

lemma *acc-modi-Protected-le*:

$Protected \leq a \Longrightarrow a=Protected \vee a=Public$

$\langle proof \rangle$

lemma *acc-modi-le-Protected*:

$a \leq Protected \Longrightarrow a=Private \vee a = Package \vee a = Protected$

$\langle proof \rangle$

```

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 Package ≤ m
  obtains (Package) m = Package
    | (Protected) m = Protected
    | (Public) m = Public
  ⟨proof⟩

```

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

```

4 Member (field or method)

```

record member = decl +
  static :: stat-modi

```

translations

```

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

```

5 Field

```

record field = member +
  type :: ty

```

translations

```

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

```

```

type-synonym fdecl
  = vname × field

```

translations

```

(type) fdecl <= (type) vname × field

```

6 Method

```

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

```

```

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

```

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

type-synonym *mdecl* = *sig* × *methd*

translations

(*type*) *mhead* <= (*type*) (|*access::acc-modi*, *static::bool*,
 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 $\langle proof \rangle$

end

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

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

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

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

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

definition

pair-memberid-def:

$memberid p = memberid (snd p)$

instance $\langle proof \rangle$

end

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

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

definition

is-field :: $qname \times memberdecl \Rightarrow bool$

where $is-field m = (\exists declC f. m=(declC,fdecl f))$

lemma *is-fieldD*: $is-field m \Longrightarrow \exists declC f. m=(declC,fdecl f)$
 $\langle proof \rangle$

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

definition

is-method :: $qname \times memberdecl \Rightarrow bool$

where $is-method membr = (\exists declC m. membr=(declC,mdecl m))$

lemma *is-methodD*: $is-method membr \Longrightarrow \exists declC m. membr=(declC,mdecl m)$
 $\langle proof \rangle$

lemma *is-methodI*: *is-method* ($C, mdecl\ m$)
 ⟨*proof*⟩

7 Interface

record *ibody* = *decl* + — interface body
imethods :: (*sig* × *mhead*) *list* — method heads

record *iface* = *ibody* + — interface
isuperIfs :: *qname list* — superinterface list

type-synonym
idecl — interface declaration, cf. 9.1
 = *qname* × *iface*

translations

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

definition

ibody :: *iface* ⇒ *ibody*
where *ibody* *i* = (|*access*=*access* *i*,*imethods*=*imethods* *i*)

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

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

8 Class

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

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

type-synonym
cdecl — class declaration, cf. 8.1
 = *qname* × *class*

translations

(*type*) *cbody* <= (*type*) (|*access*::*acc-modi*,*cfields*::*fdecl list*,
methods::*mdecl list*,*init*::*stmt*)
 (*type*) *cbody* <= (*type*) (|*access*::*acc-modi*,*cfields*::*fdecl list*,
methods::*mdecl list*,*init*::*stmt*,...:'*a*)
 (*type*) *class* <= (*type*) (|*access*::*acc-modi*,*cfields*::*fdecl list*,
methods::*mdecl list*,*init*::*stmt*,
super::*qname*,*superIfs*::*qname list*)
 (*type*) *class* <= (*type*) (|*access*::*acc-modi*,*cfields*::*fdecl list*,

$methods::mdecl\ list, init::stmt,$
 $super::qname, superIfs::qname\ list, \dots::'a)$
 $(type)\ cdecl \leq (type)\ qname \times class$

definition

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

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

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

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

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

standard classes**consts**

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

definition

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

definition

$SXcptC :: xname \Rightarrow cdecl$ — declarations of throwable classes **where**
 $SXcptC\ xn = (SXcpt\ xn, (\backslash access=Public, cfields=[], methods=SXcpt-mdecls,$
 $init=Skip,$
 $super=if\ xn = Throwable\ then\ Object$
 $else\ SXcpt\ Throwable,$
 $superIfs=[]))$

lemma $ObjectC-neq-SXcptC$ [simp]: $ObjectC \neq SXcptC\ xn$
 ⟨proof⟩

lemma $SXcptC-inject$ [simp]: $(SXcptC\ xn = SXcptC\ xm) = (xn = xm)$
 ⟨proof⟩

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-type* *G* (*Iface* *I*) ⇒ *is-iface* *G* *I*
 ⟨*proof*⟩

lemma *type-is-class*: *is-type* *G* (*Class* *C*) ⇒ *is-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*}1- *D* == (*C*,*D*) ∈ *subcls1* *G*

abbreviation

$subclseq\text{-}syntax :: prog \Rightarrow [qname, qname] \Rightarrow bool \ (+\prec_C - [71,71,71] 70)$
where $G \vdash C \preceq_C D == (C,D) \in (subcls1\ G)^*$

abbreviation

$subcls\text{-}syntax :: prog \Rightarrow [qname, qname] \Rightarrow bool \ (+\prec_C - [71,71,71] 70)$
where $G \vdash C \prec_C D == (C,D) \in (subcls1\ G)^+$

notation (ASCII)

$subcls1\text{-}syntax \ (-|\prec_C: C1- [71,71,71] 70)$ **and**
 $subclseq\text{-}syntax \ (-|\prec_C=: C-[71,71,71] 70)$ **and**
 $subcls\text{-}syntax \ (-|\prec_C: C-[71,71,71] 70)$

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

$\langle proof \rangle$

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

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

lemma $subcls1D$:

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

lemma $subint1\text{-}def2$:

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

lemma $subcls1\text{-}def2$:

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

lemma $subcls\text{-}is\text{-}class$:

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

lemma $no\text{-}subcls1\text{-}Object$: $G \vdash Object \prec_C D \implies P$
 $\langle proof \rangle$

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

well-structured programs**definition**

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

definition

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

definition

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

lemma $ws\text{-prog}I$:

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

lemma $ws\text{-prog-idecl}D$:

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

lemma $ws\text{-prog-cdecl}D$:

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

well-foundedness

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

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

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

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

lemma $subint1\text{-irrefl-lemma1}$:

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

lemma $subcls1\text{-irrefl-lemma1}$:

$ws\text{-prog } G \Longrightarrow (subcls1\ G)^{-1} \cap (subcls1\ G)^+ = \{\}$

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

\langle proof \rangle

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

\langle proof \rangle

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

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

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

\langle proof \rangle

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

\langle proof \rangle

lemma *subint1-induct*:

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

\langle proof \rangle

lemma *subcls1-induct* [*consumes 1*]:

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

\langle proof \rangle

lemma *ws-subint1-induct*:

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

\langle proof \rangle

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

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

$\rrbracket \implies P C$

\langle proof \rangle

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

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

\langle proof \rangle

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

lemma *ws-class-induct''* [consumes 2, case-names *Object Subcls*]:
 $\llbracket \text{class } G \ C = \text{Some } c; \text{ws-prog } G; \wedge \text{ co. class } G \ \text{Object} = \text{Some } \text{co} \implies P \ \text{Object } \text{co}; \wedge \ C \ c \ \text{sc.} \llbracket \text{class } G \ C = \text{Some } c; \text{class } G \ (\text{super } c) = \text{Some } \text{sc}; C \neq \text{Object}; P \ (\text{super } c) \ \text{sc} \rrbracket \implies P \ C \ c \rrbracket \implies P \ C \ c$
 ⟨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: $\wedge I \ i. \llbracket \text{iface } G \ I = \text{Some } i; \forall J \in \text{set } (\text{isuperIfs } i). (I, J) \in \text{subint1 } G \wedge P \ J \wedge \text{is-iface } G \ J \rrbracket \implies P \ I$
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*: $\llbracket \text{class } G \ C = \text{Some } c; \text{ ws-prog } G \rrbracket \implies$
class-rec G C t f =
f C c (if C = Object then t else class-rec G (super c) t f)
<proof>

definition

imethds :: *prog* \Rightarrow *qtname* \Rightarrow (*sig*, *qtname* \times *mhead*) *tables* **where**
 — methods of an interface, with overriding and inheritance, cf. 9.2
imethds G I = 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* => (*qname* × *qname*) set — direct implementation
— direct implementation, cf. 8.1.3
where *implmt1* *G* = {(*C*,*I*). *C* ≠ *Object* ∧ (∃ *c* ∈ *class G C*: *I* ∈ *set (superIfs c)*)}

abbreviation

subint1-syntax :: *prog* => [*qname*, *qname*] => bool (⊢-⊂*I1*- [71,71,71] 70)
where *G* ⊢ *I* ⊂*I1* *J* == (*I*,*J*) ∈ *subint1 G*

abbreviation

subint-syntax :: *prog* => [*qname*, *qname*] => bool (⊢-⊆*I*- [71,71,71] 70)
where *G* ⊢ *I* ⊆*I* *J* == (*I*,*J*) ∈ (*subint1 G*)^{*} — cf. 9.1.3

abbreviation

implmt1-syntax :: *prog* => [*qname*, *qname*] => bool (⊢-⊃*I1*- [71,71,71] 70)
where *G* ⊢ *C* ⊃*I1* *I* == (*C*,*I*) ∈ *implmt1 G*

notation (ASCII)

subint1-syntax (⊢-⊂*I1*- [71,71,71] 70) **and**
subint-syntax (⊢-⊆*I*- [71,71,71] 70) **and**
implmt1-syntax (⊢-⊃*I1*- [71,71,71] 70)

subclass and subinterface relations

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

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*:

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

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*:

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

implementation relation

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

inductive — implementation, cf. 8.1.4

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

where

direct: $G \vdash C \rightsquigarrow 1J \implies G \vdash C \rightsquigarrow J$
 | *subint*: $G \vdash C \rightsquigarrow 1I \implies G \vdash I \preceq_I J \implies G \vdash C \rightsquigarrow J$
 | *subcls1*: $G \vdash C \prec_C 1D \implies G \vdash D \rightsquigarrow J \implies G \vdash C \rightsquigarrow J$

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

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

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

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

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

widening relation

inductive

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

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

where

refl: $G \vdash T \preceq T$ — identity conversion, cf. 5.1.1
 | *subint*: $G \vdash I \preceq_I J \implies G \vdash \text{Iface } I \preceq \text{Iface } J$ — wid.ref.conv.,cf. 5.1.4
 | *int-obj*: $G \vdash \text{Iface } I \preceq \text{Class } \text{Object}$
 | *subcls*: $G \vdash C \preceq_C D \implies G \vdash \text{Class } C \preceq \text{Class } D$
 | *implmt*: $G \vdash C \rightsquigarrow I \implies G \vdash \text{Class } C \preceq \text{Iface } I$
 | *null*: $G \vdash \text{NT} \preceq \text{RefT } R$
 | *arr-obj*: $G \vdash T.\square \preceq \text{Class } \text{Object}$
 | *array*: $G \vdash \text{RefT } S \preceq \text{RefT } T \implies G \vdash \text{RefT } S.\square \preceq \text{RefT } T.\square$

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

declare *widen.intros* [*simp*]

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

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

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

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

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

Specialized versions for booleans also would work for real Java

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

lemma *widen-Array2*: $G \vdash S \preceq T.\square \implies S = NT \vee (\exists S'. S = S'.\square \wedge G \vdash S' \preceq T)$
 ⟨proof⟩

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

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

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

lemma *widen-Array-Array*: $G \vdash T.\square \preceq T'.\square \implies G \vdash T \preceq T'$
 ⟨proof⟩

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

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

lemma *widen-Object*:
 assumes *isrtype* G T and *ws-prog* G
 shows $G \vdash \text{RefT } T \preceq \text{Class Object}$
 ⟨proof⟩

lemma *widen-trans-lemma* [rule-format (no-asm)]:

$\llbracket G \vdash S \preceq U; \forall C. \text{is-class } G \ C \longrightarrow G \vdash C \preceq_C \text{Object} \rrbracket \Longrightarrow \forall T. G \vdash U \preceq T \longrightarrow G \vdash S \preceq T$
 ⟨proof⟩

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

⟨proof⟩

lemma *widen-antisym-lemma* [rule-format (no-asm)]: $\llbracket G \vdash S \preceq T;$

$\forall I \ J. G \vdash I \preceq I \ J \wedge G \vdash J \preceq I \ I \longrightarrow I = J;$
 $\forall C \ D. G \vdash C \preceq_C D \wedge G \vdash D \preceq_C C \longrightarrow C = D;$
 $\forall I. G \vdash \text{Object} \rightsquigarrow I \longrightarrow \text{False} \rrbracket \Longrightarrow G \vdash T \preceq S \longrightarrow S = T$
 ⟨proof⟩

lemmas *subint-antisym* =

subint1-acyclic [THEN *acyclic-impl-antisym-rtrancl*]

lemmas *subcls-antisym* =

subcls1-acyclic [THEN *acyclic-impl-antisym-rtrancl*]

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

⟨proof⟩

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

⟨proof⟩

definition

widens :: *prog* \Rightarrow [*ty list*, *ty list*] \Rightarrow *bool* (+-[\preceq]- [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*]: $G \vdash [][\preceq] []$

⟨proof⟩

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

⟨proof⟩

narrowing relation

inductive — narrowing reference conversion, cf. 5.1.5

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

for G :: *prog*

where

subcls: $G \vdash C \preceq_C D \Longrightarrow G \vdash \text{Class } D \succ \text{Class } C$

| *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* $G \ I$ *hidings* *imethds* $G \ J$ *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*: $G \vdash \text{RefT } R \succ T \Longrightarrow \exists t. T = \text{RefT } t$

⟨proof⟩

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

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

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

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

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

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

Specialized versions for booleans also would work for real Java

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

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

casting relation

inductive — casting conversion, cf. 5.5

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

for $G :: \text{prog}$

where

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

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

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*: $G \vdash \text{PrimT Boolean} \preceq? T$

shows $T = \text{PrimT Boolean}$

<proof>

lemma *cast-Boolean2*:

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

shows $S = \text{PrimT Boolean}$

<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* ⇒ *qname* ⇒ *bool* **where**
is-public *G* *qn* = (case class *G* *qn* of
 None ⇒ (case iface *G* *qn* of
 None ⇒ *False*
 | Some *i* ⇒ *access i = Public*)
 | Some *c* ⇒ *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* ⇒ *ty* ⇒ *pname* ⇒ *bool* (- ⊢ - *accessible'-in* - [61,61,61] 60) **and**
rt-accessible-in :: *prog* ⇒ *ref-ty* ⇒ *pname* ⇒ *bool* (- ⊢ - *accessible'-in'* - [61,61,61] 60)

where

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

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

definition

is-acc-class :: *prog* ⇒ *pname* ⇒ *qname* ⇒ *bool*
where *is-acc-class* *G* *P* *C* = (*is-class G C* ∧ *G*⊢(*Class C*) *accessible-in P*)

definition

is-acc-iface :: *prog* ⇒ *pname* ⇒ *qname* ⇒ *bool*
where *is-acc-iface* *G* *P* *I* = (*is-iface G I* ∧ *G*⊢(*Iface I*) *accessible-in P*)

definition

is-acc-type :: *prog* ⇒ *pname* ⇒ *ty* ⇒ *bool*
where *is-acc-type* *G* *P* *T* = (*is-type G T* ∧ *G*⊢*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

$$\text{decl-acc-modi-def: } \text{accmodi } (d::('a:: \text{type}) \text{ decl-scheme}) = \text{access } d$$
instance $\langle \text{proof} \rangle$ **end**

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

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

definition

$$\text{pair-acc-modi-def: } \text{accmodi } p = \text{accmodi } (\text{snd } p)$$
instance $\langle \text{proof} \rangle$ **end**

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

instantiation *memberdecl* :: has-accmodi
begin

definition

$$\text{memberdecl-acc-modi-def: } \text{accmodi } m = (\text{case } m \text{ of}$$

$$\quad \text{fdecl } f \Rightarrow \text{accmodi } f$$

$$\quad | \text{mdecl } m \Rightarrow \text{accmodi } m)$$
instance $\langle \text{proof} \rangle$ **end**

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

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

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

class *has-declclass* =
fixes *declclass*:: 'a \Rightarrow qname

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

definition

$$\text{declclass } q = (\mid \text{pid} = \text{pid } q, \text{tid} = \text{tid } q \mid)$$
instance $\langle \text{proof} \rangle$

end

lemma *qname-declclass-def*:
 $declclass\ q \equiv (q::qname)$
 $\langle proof \rangle$

lemma *qname-declclass-simp[simp]*: $declclass\ (q::qname) = q$
 $\langle proof \rangle$

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*
 ⟨*proof*⟩

lemma *pair-is-static-simp1*: *is-static* *p* = *is-static* (*snd* *p*)
 ⟨*proof*⟩

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

definition

memberdecl-is-static-def:
is-static *m* = (case *m* of
 fdecl *f* ⇒ *is-static* *f*
 | *mdecl* *m* ⇒ *is-static* *m*)

instance ⟨*proof*⟩

end

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

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

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

definition

decliface :: *qname* × 'a *decl-scheme* ⇒ *qname* **where**
decliface = *fst* — get the interface component

definition

mbr :: *qname* × *memberdecl* ⇒ *memberdecl* **where**
mbr = *snd* — get the memberdecl component

definition

mthd :: 'b × 'a ⇒ 'a **where**
mthd = *snd* — get the method component
 — also used for *mdecl*, *mhead*

definition

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

— some mnemonic selectors for (*vname* × *qname*)

definition

fname:: *vname* × 'a ⇒ *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
 ⟨proof⟩

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

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

definition

fldname :: vname × qname ⇒ vname
 where fldname = fst

definition

fldclass :: vname × qname ⇒ qname
 where fldclass = snd

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

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

lemma *ext-fieldname-simp* [simp]: (fldname f,fldclass f) = f
 ⟨proof⟩

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

definition

methdMembr :: qname × mdecl ⇒ qname × 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

$method :: sig \Rightarrow (qname \times method) \Rightarrow (qname \times memberdecl)$
where $method\ sig\ m = (declclass\ m, mdecl\ (sig, mhd\ m))$

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

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

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

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

lemma $mbr-method-simp[simp]$: $mbr\ (method\ sig\ m) = mdecl\ (sig, mhd\ m)$
 $\langle proof \rangle$

lemma $memberid-method-simp[simp]$: $memberid\ (method\ sig\ m) = mid\ sig$
 $\langle proof \rangle$

definition

$fieldm :: vname \Rightarrow (qname \times field) \Rightarrow (qname \times memberdecl)$
where $fieldm\ n\ f = (declclass\ f, fdecl\ (n, fld\ f))$

lemma $fieldm-simp[simp]$: $fieldm\ n\ (C, f) = (C, fdecl\ (n, f))$
 $\langle proof \rangle$

lemma $accmodi-fieldm-simp[simp]$: $accmodi\ (fieldm\ n\ f) = accmodi\ f$
 $\langle proof \rangle$

lemma $declclass-fieldm-simp[simp]$: $declclass\ (fieldm\ n\ f) = declclass\ f$
 $\langle proof \rangle$

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

lemma $mbr-fieldm-simp[simp]$: $mbr\ (fieldm\ n\ f) = fdecl\ (n, fld\ f)$
 $\langle proof \rangle$

lemma $memberid-fieldm-simp[simp]$: $memberid\ (fieldm\ n\ f) = fld\ n$
 $\langle proof \rangle$

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

definition

$msig :: (qname \times mdecl) \Rightarrow sig$
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

$qmdecl :: sig \Rightarrow (qname \times methd) \Rightarrow (qname \times mdecl)$
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 \Rightarrow 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 ⟨proof⟩

end

axiomatization where

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

lemma *mhead-resTy-simp*: $\text{resTy } (m::('a \text{ mhead-scheme})) = \text{resT } m$
 ⟨proof⟩

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

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

definition

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

instance ⟨proof⟩

end

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

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

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

inheritable-in

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

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

definition

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

where

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

(case (acmodi membr) of

 Private \Rightarrow False

 | Package \Rightarrow (pid (declclass membr)) = pack

 | Protected \Rightarrow True

| *Public* \Rightarrow *True*)

abbreviation

Method-inheritable-in-syntax::

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

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

abbreviation

Methd-inheritable-in::

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

where $G \vdash Methd s m inheritable-in p == G \vdash (method s m) inheritable-in p$

declared-in/undeclared-in**definition**

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

definition

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

definition

declared-in :: $prog \Rightarrow memberdecl \Rightarrow qname \Rightarrow bool$ ($- \vdash - declared'-in - [61,61,61] 60$)

where

$G \vdash m declared-in C = (case m of$
 $fdecl (fn, f) \Rightarrow cdeclaredfield G C fn = Some f$
 | $mdecl (sig, m) \Rightarrow cdeclaredmethd G C sig = Some m)$

abbreviation

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

where $G \vdash Method m declared-in C == G \vdash mdecl (mthd m) declared-in C$

abbreviation

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

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

lemma *declared-in-classD*:

$G \vdash m declared-in C \Longrightarrow is-class G C$
 <proof>

definition

undeclared-in :: $prog \Rightarrow memberid \Rightarrow qname \Rightarrow bool$ ($- \vdash - undeclared'-in - [61,61,61] 60$)

where

$G \vdash m undeclared-in C = (case m of$
 $fid fn \Rightarrow cdeclaredfield G C fn = None$
 | $mid sig \Rightarrow cdeclaredmethd G C sig = None)$

members**inductive**

$$\text{members} :: \text{prog} \Rightarrow (\text{qname} \times \text{memberdecl}) \Rightarrow \text{qname} \Rightarrow \text{bool}$$

$$(- \vdash - \text{member'-of} - [61,61,61] 60)$$

$$\text{for } G :: \text{prog}$$
where

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

$$\text{Inherited: } \llbracket G \vdash m \text{ inheritable-in } (\text{pid } C); G \vdash \text{memberid } m \text{ undeclared-in } C;$$

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

$$\rrbracket \Longrightarrow G \vdash m \text{ member-of } C$$

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

$$\text{method-member-of} :: \text{prog} \Rightarrow (\text{qname} \times \text{mdecl}) \Rightarrow \text{qname} \Rightarrow \text{bool}$$

$$(- \vdash \text{Method} - \text{member'-of} - [61,61,61] 60)$$

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

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

$$(- \vdash \text{Methd} - \text{member'-of} - [61,61,61,61] 60)$$

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

$$\text{fieldm-member-of} :: \text{prog} \Rightarrow \text{vname} \Rightarrow (\text{qname} \times \text{field}) \Rightarrow \text{qname} \Rightarrow \text{bool}$$

$$(- \vdash \text{Field} - \text{member'-of} - [61,61,61] 60)$$

$$\text{where } G \vdash \text{Field } n \text{ f member-of } C == G \vdash \text{fieldm } n \text{ f member-of } C$$
definition

$$\text{inherits} :: \text{prog} \Rightarrow \text{qname} \Rightarrow (\text{qname} \times \text{memberdecl}) \Rightarrow \text{bool} \quad (- \vdash - \text{inherits} - [61,61,61] 60)$$
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*: $G \vdash C \text{ inherits } m \Longrightarrow G \vdash m \text{ member-of } C$

<proof>

definition

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

$$\text{where } G \vdash m \text{ member-in } C = (\exists \text{ provC}. G \vdash C \preceq_C \text{ provC} \wedge G \vdash m \text{ member-of } \text{provC})$$

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

$$\text{method-member-in} :: \text{prog} \Rightarrow (\text{qname} \times \text{mdecl}) \Rightarrow \text{qname} \Rightarrow \text{bool}$$

$$(- \vdash \text{Method} - \text{member'-in} - [61,61,61] 60)$$

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

abbreviation

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

where $G \vdash Methd s m member\text{-}in C == G \vdash (method s m) member\text{-}in C$

lemma *member-inD*: $G \vdash m member\text{-}in C$

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

<proof>

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

<proof>

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

<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 :: $prog \Rightarrow (qname \times mdecl) \Rightarrow (qname \times mdecl) \Rightarrow bool$
 ($- \vdash - overrides_S - [61,61,61] 60$)

for $G :: prog$

where

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

| *Indirect*: $\llbracket G \vdash new\ overrides_S\ intr; G \vdash intr\ overrides_S\ old \rrbracket$
 $\implies G \vdash new\ overrides_S\ old$

Dynamic overriding (used during the typecheck of the compiler)

inductive

overridesR :: $prog \Rightarrow (qname \times mdecl) \Rightarrow (qname \times mdecl) \Rightarrow bool$
 ($- \vdash - overrides - [61,61,61] 60$)

for $G :: prog$

where

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

| *Indirect*: $\llbracket G \vdash new\ overrides\ intr; G \vdash intr\ overrides\ old \rrbracket$

$$\implies G \vdash \text{new overrides old}$$

abbreviation (*input*)

sig-stat-overrides::

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

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

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

abbreviation (*input*)

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

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

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

Hiding

definition

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

where

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

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

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

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

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

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

abbreviation

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

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

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

lemma *hidesI*:

$$\llbracket \text{is-static new}; \text{msig new} = \text{msig old};$$

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

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

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

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

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

<proof>

lemma *hidesD*:

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

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

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

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

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

<proof>

lemma *overrides-commonD*:

$$\llbracket G \vdash \text{new overrides old} \rrbracket \implies$$

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

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

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

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

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

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

<proof>

lemma *ws-overrides-commonD*:

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

lemma *overrides-eq-sigD*:

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

lemma *hides-eq-sigD*:

$\llbracket G \vdash \text{new hides old} \rrbracket \implies \text{msig old} = \text{msig new}$
 $\langle \text{proof} \rangle$

permits access

definition

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

where

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

The subcondition of the *Protected* case: $G \vdash \text{accclass} \prec_C \text{declclass mbr}$ could also be relaxed to: $G \vdash \text{accclass} \preceq_C \text{declclass mbr}$ since in case both classes are the same the other condition $\text{pid } (\text{declclass mbr}) = \text{pid accclass}$ 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 $\text{accessible-from} :: \text{prog} \Rightarrow (\text{qname} \times \text{memberdecl}) \Rightarrow \text{qname} \Rightarrow \text{qname} \Rightarrow \text{bool}$
 $(\text{- } \vdash \text{- of - accessible'-from - [61,61,61,61] 60})$
and $\text{method-accessible-from} :: \text{prog} \Rightarrow (\text{qname} \times \text{mdecl}) \Rightarrow \text{qname} \Rightarrow \text{qname} \Rightarrow \text{bool}$
 $(\text{- } \vdash \text{Method - of - accessible'-from - [61,61,61,61] 60})$

for $G :: \text{prog}$ **and** $\text{accclass} :: \text{qname}$
where
 $G \vdash \text{membr of cls accessible-from accclass} \equiv \text{accessible-fromR } G \text{ accclass membr cls}$

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

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

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

abbreviation

methd-accessible-from::

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

where

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

abbreviation

field-accessible-from::

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

where

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

inductive

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

where

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

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

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

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

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

$$(- \vdash \text{Methd} \text{ - - in - dyn'-accessible'-from - [61,61,61,61,61] 60)$$
where

$$G \vdash \text{Methd } s \ m \ \text{in } C \ \text{dyn-accessible-from } \text{acc}C \ ==$$

$$G \vdash (\text{method } s \ m) \ \text{in } C \ \text{dyn-accessible-from } \text{acc}C$$
abbreviation*field-dyn-accessible-from*::
$$\text{prog} \Rightarrow \text{vname} \Rightarrow (\text{qname} \times \text{field}) \Rightarrow \text{qname} \Rightarrow \text{qname} \Rightarrow \text{bool}$$

$$(- \vdash \text{Field} \text{ - - in - dyn'-accessible'-from - [61,61,61,61,61] 60)$$
where

$$G \vdash \text{Field } \text{fn } f \ \text{in } \text{dyn}C \ \text{dyn-accessible-from } \text{acc}C \ ==$$

$$G \vdash (\text{fieldm } \text{fn } f) \ \text{in } \text{dyn}C \ \text{dyn-accessible-from } \text{acc}C$$
lemma *accessible-from-commonD*: $G \vdash m$ 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*:
$$\neg G \vdash \text{memb-rid undeclared-in } C \implies (\exists m. G \vdash m \ \text{declared-in } C \wedge$$

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

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

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

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

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

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

lemma *stat-override-declclasses-relation*:
 $\llbracket G \vdash (\text{declclass } \text{new}) \prec_C 1 \text{ superNew}; G \vdash \text{Method } \text{old} \text{ member-of } \text{superNew} \rrbracket$
 $\implies G \vdash (\text{declclass } \text{new}) \prec_C (\text{declclass } \text{old})$
 ⟨proof⟩

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

lemma *member-of-Package*:
assumes $G \vdash m \text{ member-of } C$
and $\text{accmodi } m = \text{Package}$
shows $\text{pid } (\text{declclass } m) = \text{pid } C$
 ⟨proof⟩

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

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

lemma *no-Private-stat-override*:

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

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

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

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

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

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

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

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

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

lemma *member-of-subclseq-declC*:

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

lemma *member-of-inheritance*:

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

lemma *member-of-subcls*:

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

corollary *member-of-overrides-subcls*:

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

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

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

lemma *inherited-field-access*:

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

lemma *accessible-inheritance*:

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

fields and methods

type-synonym

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

translations

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

$$\begin{aligned} imethds &:: prog \Rightarrow qname \Rightarrow (sig, qname \times mhead) \text{ tables } \mathbf{where} \\ imethds \ G \ I &= \\ & \text{iface-rec } G \ I \ (\lambda I \ i \ ts. (Un\text{-}tables \ ts) \oplus \oplus \\ & \quad (set\text{-}option \circ table\text{-}of \ (map \ (\lambda(s,m). (s,I,m)) \ (imethds \ i)))) \end{aligned}$$

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

definition

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

only returns imethds if the interface is accessible

definition

$$\begin{aligned} methd &:: prog \Rightarrow qname \Rightarrow (sig, qname \times methd) \text{ table } \mathbf{where} \\ methd \ G \ C &= \\ & \text{class-rec } G \ C \ Map.empty \\ & \quad (\lambda C \ c \ subcls\text{-}methds. \\ & \quad \quad \text{filter-tab } (\lambda sig \ m. \ G \vdash \ C \ \text{inherits } method \ sig \ m) \\ & \quad \quad \quad subcls\text{-}methds \\ & \quad ++ \\ & \quad \quad \text{table-of } (map \ (\lambda(s,m). (s,C,m)) \ (methods \ c))) \end{aligned}$$

$methd \ G \ C$: 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

$$\begin{aligned} accmethd &:: prog \Rightarrow qname \Rightarrow qname \Rightarrow (sig, qname \times methd) \text{ table } \mathbf{where} \\ accmethd \ G \ S \ C &= \\ & \text{filter-tab } (\lambda sig \ m. \ G \vdash \text{method } sig \ m \ \text{of } C \ \text{accessible-from } S) \ (methd \ G \ C) \end{aligned}$$

$accmethd \ G \ S \ C$: 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

$$\begin{aligned} dynmethd &:: prog \Rightarrow qname \Rightarrow qname \Rightarrow (sig, qname \times methd) \text{ table } \mathbf{where} \\ dynmethd \ G \ statC \ dynC &= \\ & (\lambda sig. \\ & \quad (if \ G \vdash \ dynC \ \preceq_C \ statC \\ & \quad \quad \text{then } (case \ methd \ G \ statC \ sig \ \text{of} \\ & \quad \quad \quad \text{None} \Rightarrow \text{None} \\ & \quad \quad \quad | \ \text{Some } statM \\ & \quad \quad \quad \Rightarrow (class\text{-}rec \ G \ dynC \ Map.empty \\ & \quad \quad \quad \quad (\lambda C \ c \ subcls\text{-}methds. \\ & \quad \quad \quad \quad \quad subcls\text{-}methds \\ & \quad \quad \quad \quad ++ \\ & \quad \quad \quad \quad \text{filter-tab} \\ & \quad \quad \quad \quad \quad (\lambda - \ dynM. \ G, sig \vdash \ dynM \ \text{overrides } statM \vee \ dynM = statM) \\ & \quad \quad \quad \quad \quad \quad (methd \ G \ C) \)) \\ & \quad \quad \quad) \ sig \end{aligned}$$

)
else None))

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

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

definition

dynimethod :: *prog* \Rightarrow *qname* \Rightarrow *qname* \Rightarrow (*sig,qname* \times *method*) *table* **where**
dynimethod *G I dynC* =
 (λ *sig*. if *imethds* *G I sig* \neq {}
 then *method* *G dynC sig*
 else *dynmethod* *G Object dynC sig*)

dynimethod *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 (*method*). Since all interface methods have public access the method can't be coalesced due to some odd visibility effects like in case of *dynmethod*. 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* \Rightarrow *ref-ty* \Rightarrow *qname* \Rightarrow (*sig,qname* \times *method*) *table* **where**
dynlookup *G statT dynC* =
 (*case* *statT* of
 NullT \Rightarrow *Map.empty*
 | *IfaceT I* \Rightarrow *dynimethod* *G I dynC*
 | *ClassT statC* \Rightarrow *dynmethod* *G statC dynC*
 | *ArrayT ty* \Rightarrow *dynmethod* *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* \Rightarrow *qname* \Rightarrow ((*vname* \times *qname*) \times *field*) *list* **where**
fields *G C* =
class-rec *G C* [] (λ *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* \Rightarrow *qname* \Rightarrow *qname* \Rightarrow (*vname*, *qname* \times *field*) *table* **where**
accfield *G S C* =
 (*let* *field-tab* = *table-of*((*map* (λ ((*n,d*),*f*).(*n*,(*d,f*)))) (*fields* *G C*))
 in *filter-tab* (λ *n* (*declC,f*). *G* \vdash (*declC,fdecl* (*n,f*)) of *C* accessible-from *S*)
 field-tab)

accfield *G C S*: fields of a class *C* which are accessible from scope of class *S* with inheritance and hiding, cf. 8.3

note the class component in the accessibility filter (see also *method*). 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

$is_methd :: prog \Rightarrow qname \Rightarrow sig \Rightarrow bool$
where $is_methd\ G = (\lambda C\ sig.\ is_class\ G\ C \wedge methd\ G\ C\ sig \neq None)$

definition

$efname :: ((vname \times qname) \times field) \Rightarrow (vname \times qname)$
where $efname = fst$

lemma $efname_simp[simp]:efname\ (n,f) = n$
 $\langle proof \rangle$

4 imethds

lemma $imethds_rec: \llbracket iface\ G\ I = Some\ i; ws_prog\ G \rrbracket \Longrightarrow$
 $imethds\ G\ I = Un_tables\ ((\lambda J.\ imethds\ G\ J)\ 'set\ (isuperIfs\ i)) \oplus \oplus$
 $(set_option \circ table_of\ (map\ (\lambda(s,mh).\ (s,I,mh))\ (imethods\ i)))$
 $\langle proof \rangle$

lemma $imethds_norec:$

$\llbracket iface\ G\ md = Some\ i; ws_prog\ G; table_of\ (imethods\ i)\ sig = Some\ mh \rrbracket \Longrightarrow$
 $(md, mh) \in imethds\ G\ md\ sig$
 $\langle proof \rangle$

lemma $imethds_declI: \llbracket m \in imethds\ G\ I\ sig; ws_prog\ G; is_iface\ G\ I \rrbracket \Longrightarrow$
 $(\exists i.\ iface\ G\ (decliface\ m) = Some\ i \wedge$
 $table_of\ (imethods\ i)\ sig = Some\ (mthd\ m)) \wedge$
 $(I, decliface\ m) \in (subint1\ G)^* \wedge m \in imethds\ G\ (decliface\ m)\ sig$
 $\langle proof \rangle$

lemma $imethds_cases:$

assumes $im: im \in imethds\ G\ I\ sig$
and $ifI: iface\ G\ I = Some\ i$
and $ws: ws_prog\ G$
obtains $(NewMethod)\ table_of\ (map\ (\lambda(s, mh).\ (s, I, mh))\ (imethods\ i))\ sig = Some\ im$
 $| (InheritedMethod)\ J$ **where** $J \in set\ (isuperIfs\ i)$ **and** $im \in imethds\ G\ J\ sig$
 $\langle proof \rangle$

5 accimethd

lemma $accimethds_simp\ [simp]:$
 $G \vdash Iface\ I\ accessible_in\ pack \Longrightarrow accimethds\ G\ pack\ I = imethds\ G\ I$
 $\langle proof \rangle$

lemma $accimethdsD:$

$im \in accimethds\ G\ pack\ I\ sig$
 $\Longrightarrow im \in imethds\ G\ I\ sig \wedge G \vdash Iface\ I\ accessible_in\ pack$
 $\langle proof \rangle$

lemma $accimethdsI:$

$\llbracket im \in imethds\ G\ I\ sig; G \vdash Iface\ I\ accessible_in\ pack \rrbracket$
 $\Longrightarrow im \in accimethds\ G\ pack\ I\ sig$

$\langle \text{proof} \rangle$

6 method

lemma *method-rec*: $\llbracket \text{class } G \ C = \text{Some } c; \text{ws-prog } G \rrbracket \implies$
 $\text{method } G \ C$
 $= (\text{if } C = \text{Object}$
 $\quad \text{then } \text{Map.empty}$
 $\quad \text{else } \text{filter-tab } (\lambda \text{sig } m. \ G \vdash C \text{ inherits method sig } m)$
 $\quad \quad (\text{method } G \ (\text{super } c)))$
 $++ \text{table-of } (\text{map } (\lambda (s,m). \ (s,C,m)) \ (\text{methods } c))$
 $\langle \text{proof} \rangle$

lemma *method-norec*:
 $\llbracket \text{class } G \ \text{decl}C = \text{Some } c; \text{ws-prog } G; \text{table-of } (\text{methods } c) \ \text{sig} = \text{Some } m \rrbracket$
 $\implies \text{method } G \ \text{decl}C \ \text{sig} = \text{Some } (\text{decl}C, m)$
 $\langle \text{proof} \rangle$

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

lemma *method-inheritedD*:
 $\llbracket \text{class } G \ C = \text{Some } c; \text{ws-prog } G; \text{method } G \ C \ \text{sig} = \text{Some } m \rrbracket$
 $\implies (\text{declclass } m \neq C \longrightarrow G \vdash C \text{ inherits method sig } m)$
 $\langle \text{proof} \rangle$

lemma *method-diff-cls*:
 $\llbracket \text{ws-prog } G; \text{is-class } G \ C; \text{is-class } G \ D;$
 $\text{method } G \ C \ \text{sig} = m; \text{method } G \ D \ \text{sig} = n; m \neq n$
 $\rrbracket \implies C \neq D$
 $\langle \text{proof} \rangle$

lemma *method-declared-inI*:
 $\llbracket \text{table-of } (\text{methods } c) \ \text{sig} = \text{Some } m; \text{class } G \ C = \text{Some } c \rrbracket$
 $\implies G \vdash \text{mdecl } (\text{sig}, m) \ \text{declared-in } C$
 $\langle \text{proof} \rangle$

lemma *method-declared-in-declclass*:
 $\llbracket \text{method } G \ C \ \text{sig} = \text{Some } m; \text{ws-prog } G; \text{is-class } G \ C \rrbracket$
 $\implies G \vdash \text{Methd } \text{sig } m \ \text{declared-in } (\text{declclass } m)$
 $\langle \text{proof} \rangle$

lemma *member-method*:
assumes *member-of*: $G \vdash \text{Methd } \text{sig } m \ \text{member-of } C$ **and**
 $\text{ws: ws-prog } G$
shows $\text{method } G \ C \ \text{sig} = \text{Some } m$
 $\langle \text{proof} \rangle$

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

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

7 accmethd

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

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

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

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

8 dynmethd

lemma *dynmethd-rec*:
 $\llbracket \text{class } G \ \text{dynC} = \text{Some } c; \text{ws-prog } G \rrbracket \implies$
 $\text{dynmethd } G \ \text{statC} \ \text{dynC} \ \text{sig}$
 $= (\text{if } G \vdash \text{dynC} \preceq_C \ \text{statC}$
 $\text{then } (\text{case } \text{methd } G \ \text{statC} \ \text{sig} \text{ of}$
 $\text{None} \Rightarrow \text{None}$
 $\mid \text{Some } \text{statM}$
 $\Rightarrow (\text{case } \text{methd } G \ \text{dynC} \ \text{sig} \text{ of}$
 $\text{None} \Rightarrow \text{dynmethd } G \ \text{statC} \ (\text{super } c) \ \text{sig}$
 $\mid \text{Some } \text{dynM} \Rightarrow$
 $(\text{if } G, \text{sig} \vdash \text{dynM} \text{ overrides } \text{statM} \vee \text{dynM} = \text{statM}$
 $\text{then } \text{Some } \text{dynM}$
 $\text{else } (\text{dynmethd } G \ \text{statC} \ (\text{super } c) \ \text{sig})$
 $)))$
 $\text{else } \text{None})$
 $(\text{is } - \implies - \implies ?\text{Dynmethd-def } \text{dynC} \ \text{sig} = ?\text{Dynmethd-rec } \text{dynC} \ c \ \text{sig})$
 ⟨proof⟩

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

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

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

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

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

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

lemma *methd-Some-dynmethd-Some*:
assumes *statM*: *methd* G *statC* sig = *Some* *statM* **and**
subclseq: $G \vdash \text{dynC} \preceq_C \text{statC}$ **and**

is-cls-statC: *is-class* G *statC* **and**
ws: *ws-prog* G
shows \exists *dynM*. *dynmethd* G *statC* *dynC* *sig* = *Some dynM*
 (**is** ? P *dynC*)
 ⟨*proof*⟩

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

lemma *ws-dynmethd*:
assumes *statM*: *methd* G *statC* *sig* = *Some statM* **and**
subclseq: $G \vdash \text{dynC} \preceq_C \text{statC}$ **and**
is-cls-statC: *is-class* G *statC* **and**
ws: *ws-prog* G
shows
 \exists *dynM*. *dynmethd* G *statC* *dynC* *sig* = *Some dynM* \wedge
is-static *dynM* = *is-static* *statM* \wedge $G \vdash \text{resTy}$ *dynM* \preceq_{resTy} *statM*
 ⟨*proof*⟩

9 dynlookup

lemma *dynlookup-cases*:
assumes *dynlookup* G *statT* *dynC* *sig* = x
obtains (*NullT*) *statT* = *NullT* **and** *Map.empty* *sig* = x
 | (*IfaceT*) I **where** *statT* = *IfaceT* I **and** *dynmethd* G I *dynC* *sig* = x
 | (*ClassT*) *statC* **where** *statT* = *ClassT* *statC* **and** *dynmethd* G *statC* *dynC* *sig* = x
 | (*ArrayT*) *ty* **where** *statT* = *ArrayT* *ty* **and** *dynmethd* G *Object* *dynC* *sig* = x
 ⟨*proof*⟩

10 fields

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

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

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

lemma *fields-declC*:

$$\begin{aligned} & \llbracket \text{table-of } (\text{fields } G \ C) \ \text{efn} = \text{Some } f; \text{ ws-prog } G; \text{ is-class } G \ C \rrbracket \implies \\ & (\exists d. \text{class } G \ (\text{declclassf } \text{efn}) = \text{Some } d \wedge \\ & \quad \text{table-of } (\text{cfields } d) \ (\text{fname } \text{efn}) = \text{Some } f) \wedge \\ & G \vdash C \preceq_C (\text{declclassf } \text{efn}) \wedge \text{table-of } (\text{fields } G \ (\text{declclassf } \text{efn})) \ \text{efn} = \text{Some } f \\ & \langle \text{proof} \rangle \end{aligned}$$

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

lemma *fields-mono-lemma*:

$$\begin{aligned} & \llbracket x \in \text{set } (\text{fields } G \ C); G \vdash D \preceq_C C; \text{ ws-prog } G \rrbracket \\ & \implies x \in \text{set } (\text{fields } G \ D) \\ & \langle \text{proof} \rangle \end{aligned}$$

lemma *ws-unique-fields-lemma*:

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

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

$$\begin{aligned} & \bigwedge C \ c. \llbracket \text{class } G \ C = \text{Some } c \rrbracket \implies \text{unique } (\text{cfields } c) \rrbracket \implies \\ & \text{unique } (\text{fields } G \ C) \\ & \langle \text{proof} \rangle \end{aligned}$$

11 accfield

lemma *accfield-fields*:

$$\begin{aligned} & \text{accfield } G \ S \ C \ \text{fn} = \text{Some } f \\ & \implies \text{table-of } (\text{fields } G \ C) \ (\text{fn}, \text{ declclass } f) = \text{Some } (\text{fld } f) \\ & \langle \text{proof} \rangle \end{aligned}$$

lemma *accfield-declC-is-class*:

$$\begin{aligned} & \llbracket \text{is-class } G \ C; \text{ accfield } G \ S \ C \ \text{en} = \text{Some } (\text{fd}, f); \text{ ws-prog } G \rrbracket \implies \\ & \text{is-class } G \ \text{fd} \\ & \langle \text{proof} \rangle \end{aligned}$$

lemma *accfield-accessibleD*:

$$\text{accfield } G \ S \ C \ \text{fn} = \text{Some } f \implies G \vdash \text{Field } \text{fn } f \ \text{of } C \ \text{accessible-from } S$$
 $\langle \text{proof} \rangle$

12 is methd

lemma *is-methdI*:

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

lemma *is-methdD*:

is-methd G C sig \implies *class G C* \neq *None* \wedge *methd G C sig* \neq *None*
 ⟨proof⟩

lemma *finite-is-methd*:

ws-prog G \implies *finite (Collect (case-prod (is-methd G)))*
 ⟨proof⟩

calculation of the superclasses of a class

definition

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

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

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

⟨proof⟩

lemma *superclasses-mono*:

assumes *clsrel*: $G \vdash C \prec_C D$
and *ws*: *ws-prog G*
and *cls-C*: *class G C* = *Some c*
and *wf*: $\bigwedge C c. \llbracket$ *class G C* = *Some c*; *C* \neq *Object \rrbracket
 $\implies \exists sc. \text{class } G (\text{super } c) = \text{Some } sc$
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* = *Some c*
and *wf*: $\bigwedge C c. \llbracket$ *class G C* = *Some c*; *C* \neq *Object \rrbracket
 $\implies \exists sc. \text{class } G (\text{super } c) = \text{Some } sc$
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 =
  prog:: 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) ( $\lfloor$ prog::prog,cls::qtname,lcl::lenv $\rfloor$ )
(type) env <= (type) ( $\lfloor$ prog::prog,cls::qtname,lcl::lenv,...::'a $\rfloor$ )
```

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

$cmheads :: prog \Rightarrow qname \Rightarrow qname \Rightarrow sig \Rightarrow emhead\ set$
where $cmheads\ G\ S\ C = (\lambda sig. (\lambda (Cls, mthd). (ClassT\ Cls, (mhead\ mthd)))) \text{ ‘ set-option (accmethd\ G\ S\ C\ sig) }$

definition

$Objectmheads :: prog \Rightarrow qname \Rightarrow sig \Rightarrow emhead\ set$ **where**
 $Objectmheads\ G\ S =$
 $(\lambda sig. (\lambda (Cls, mthd). (ClassT\ Cls, (mhead\ mthd))))$
 $\text{ ‘ set-option (filter-tab (\lambda sig\ m. accmodi\ m \neq Private) (accmethd\ G\ S\ Object)\ sig) }$

definition

$accObjectmheads :: prog \Rightarrow qname \Rightarrow ref\text{-}ty \Rightarrow sig \Rightarrow emhead\ set$
where
 $accObjectmheads\ G\ S\ T =$
 $(if\ G \vdash RefT\ T\ accessible\text{-}in\ (pid\ S)$
 $then\ Objectmheads\ G\ S$
 $else\ (\lambda sig. \{\}))$

primrec $mheads :: prog \Rightarrow qname \Rightarrow ref\text{-}ty \Rightarrow sig \Rightarrow emhead\ set$
where

$$\begin{aligned}
& mheads\ G\ S\ \text{NullT} = (\lambda sig. \{\}) \\
| mheads\ G\ S\ (\text{IfaceT}\ I) &= (\lambda sig. (\lambda(I,h).(IfaceT\ I,h)) \\
& \quad 'accimethds\ G\ (pid\ S)\ I\ sig\ \cup \\
& \quad \quad accObjectmheads\ G\ S\ (\text{IfaceT}\ I)\ sig) \\
| mheads\ G\ S\ (\text{ClassT}\ C) &= cmheads\ G\ S\ C \\
| mheads\ G\ S\ (\text{ArrayT}\ T) &= accObjectmheads\ G\ S\ (\text{ArrayT}\ T)
\end{aligned}$$
definition

— applicable methods, cf. 15.11.2.1

$$\begin{aligned}
& appl\text{-}methds :: prog \Rightarrow qname \Rightarrow ref\text{-}ty \Rightarrow sig \Rightarrow (emhead \times ty\ list) \text{ set } \mathbf{where} \\
& appl\text{-}methds\ G\ S\ rt = (\lambda sig. \\
& \quad \{(mh,pTs') \mid mh\ pTs'. mh \in mheads\ G\ S\ rt\ (\!|name=name\ sig,parTs=pTs') \wedge \\
& \quad \quad G \vdash (parTs\ sig) [\preceq] pTs'\})
\end{aligned}$$
definition

— more specific methods, cf. 15.11.2.2

$$\begin{aligned}
& more\text{-}spec :: prog \Rightarrow emhead \times ty\ list \Rightarrow emhead \times ty\ list \Rightarrow bool \mathbf{where} \\
& more\text{-}spec\ G = (\lambda(mh,pTs). \lambda(mh',pTs'). G \vdash pTs [\preceq] pTs')
\end{aligned}$$
definition

— maximally specific methods, cf. 15.11.2.2

$$\begin{aligned}
& max\text{-}spec :: prog \Rightarrow qname \Rightarrow ref\text{-}ty \Rightarrow sig \Rightarrow (emhead \times ty\ list) \text{ set } \mathbf{where} \\
& max\text{-}spec\ G\ S\ rt\ sig = \{m. m \in appl\text{-}methds\ G\ S\ rt\ sig \wedge \\
& \quad (\forall m' \in appl\text{-}methds\ G\ S\ rt\ sig. more\text{-}spec\ G\ m'\ m \longrightarrow m'=m)\}
\end{aligned}$$
lemma *max-spec2appl-meths*:
$$x \in max\text{-}spec\ G\ S\ T\ sig \implies x \in appl\text{-}methds\ G\ S\ T\ sig$$

<proof>

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

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

<proof>

lemma *max-spec2mheads*:
$$\begin{aligned}
& max\text{-}spec\ G\ S\ rt\ (\!|name=mn,parTs=pTs|) = insert\ (mh, pTs')\ A \\
& \implies mh \in mheads\ G\ S\ rt\ (\!|name=mn,parTs=pTs') \wedge G \vdash pTs [\preceq] pTs'
\end{aligned}$$

<proof>

definition

$$\begin{aligned}
& empty\text{-}dt :: dyn\text{-}ty \\
& \mathbf{where}\ empty\text{-}dt = (\lambda a. None)
\end{aligned}$$
definition

$$\begin{aligned}
& invmode :: ('a::type)member\text{-}scheme \Rightarrow expr \Rightarrow inv\text{-}mode \mathbf{where} \\
& invmode\ m\ e = (if\ is\text{-}static\ m \\
& \quad \text{then}\ Static \\
& \quad \text{else}\ if\ e=Super\ \text{then}\ SuperM\ \text{else}\ IntVir)
\end{aligned}$$
lemma *invmode-nonstatic [simp]*:
$$invmode\ (\!|access=a,static=False,..=x|)\ (Acc\ (LVar\ e)) = IntVir$$

<proof>

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

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

lemma *Null-staticD*:

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

Typing for unary operations

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

where

$\text{unop-type } \text{UPlus} = \text{Integer}$
 | $\text{unop-type } \text{UMinus} = \text{Integer}$
 | $\text{unop-type } \text{UBitNot} = \text{Integer}$
 | $\text{unop-type } \text{UNot} = \text{Boolean}$

primrec *wt-unop* :: $\text{unop} \Rightarrow \text{ty} \Rightarrow \text{bool}$

where

$\text{wt-unop } \text{UPlus } t = (t = \text{PrimT Integer})$
 | $\text{wt-unop } \text{UMinus } t = (t = \text{PrimT Integer})$
 | $\text{wt-unop } \text{UBitNot } t = (t = \text{PrimT Integer})$
 | $\text{wt-unop } \text{UNot } t = (t = \text{PrimT Boolean})$

Typing for binary operations

primrec *binop-type* :: $\text{binop} \Rightarrow \text{prim-ty}$

where

$\text{binop-type } \text{Mul} = \text{Integer}$
 | $\text{binop-type } \text{Div} = \text{Integer}$
 | $\text{binop-type } \text{Mod} = \text{Integer}$
 | $\text{binop-type } \text{Plus} = \text{Integer}$
 | $\text{binop-type } \text{Minus} = \text{Integer}$
 | $\text{binop-type } \text{LShift} = \text{Integer}$
 | $\text{binop-type } \text{RShift} = \text{Integer}$
 | $\text{binop-type } \text{RShiftU} = \text{Integer}$
 | $\text{binop-type } \text{Less} = \text{Boolean}$
 | $\text{binop-type } \text{Le} = \text{Boolean}$
 | $\text{binop-type } \text{Greater} = \text{Boolean}$
 | $\text{binop-type } \text{Ge} = \text{Boolean}$
 | $\text{binop-type } \text{Eq} = \text{Boolean}$
 | $\text{binop-type } \text{Neq} = \text{Boolean}$
 | $\text{binop-type } \text{BitAnd} = \text{Integer}$
 | $\text{binop-type } \text{And} = \text{Boolean}$
 | $\text{binop-type } \text{BitXor} = \text{Integer}$
 | $\text{binop-type } \text{Xor} = \text{Boolean}$
 | $\text{binop-type } \text{BitOr} = \text{Integer}$
 | $\text{binop-type } \text{Or} = \text{Boolean}$
 | $\text{binop-type } \text{CondAnd} = \text{Boolean}$
 | $\text{binop-type } \text{CondOr} = \text{Boolean}$

primrec *wt-binop* :: $\text{prog} \Rightarrow \text{binop} \Rightarrow \text{ty} \Rightarrow \text{ty} \Rightarrow \text{bool}$

where

$wt\text{-binop } G \text{ Mul } \quad t1 \ t2 = ((t1 = PrimT Integer) \wedge (t2 = PrimT Integer))$
 $| wt\text{-binop } G \text{ Div } \quad t1 \ t2 = ((t1 = PrimT Integer) \wedge (t2 = PrimT Integer))$
 $| wt\text{-binop } G \text{ Mod } \quad t1 \ t2 = ((t1 = PrimT Integer) \wedge (t2 = PrimT Integer))$
 $| wt\text{-binop } G \text{ Plus } \quad t1 \ t2 = ((t1 = PrimT Integer) \wedge (t2 = PrimT Integer))$
 $| wt\text{-binop } G \text{ Minus } \quad t1 \ t2 = ((t1 = PrimT Integer) \wedge (t2 = PrimT Integer))$
 $| wt\text{-binop } G \text{ LShift } \quad t1 \ t2 = ((t1 = PrimT Integer) \wedge (t2 = PrimT Integer))$
 $| wt\text{-binop } G \text{ RShift } \quad t1 \ t2 = ((t1 = PrimT Integer) \wedge (t2 = PrimT Integer))$
 $| wt\text{-binop } G \text{ RShiftU } \quad t1 \ t2 = ((t1 = PrimT Integer) \wedge (t2 = PrimT Integer))$
 $| wt\text{-binop } G \text{ Less } \quad t1 \ t2 = ((t1 = PrimT Integer) \wedge (t2 = PrimT Integer))$
 $| wt\text{-binop } G \text{ Le } \quad t1 \ t2 = ((t1 = PrimT Integer) \wedge (t2 = PrimT Integer))$
 $| wt\text{-binop } G \text{ Greater } \quad t1 \ t2 = ((t1 = PrimT Integer) \wedge (t2 = PrimT Integer))$
 $| wt\text{-binop } G \text{ Ge } \quad t1 \ t2 = ((t1 = PrimT Integer) \wedge (t2 = PrimT Integer))$
 $| wt\text{-binop } G \text{ Eq } \quad t1 \ t2 = (G \vdash t1 \leq t2 \vee G \vdash t2 \leq t1)$
 $| wt\text{-binop } G \text{ Neq } \quad t1 \ t2 = (G \vdash t1 \leq t2 \vee G \vdash t2 \leq t1)$
 $| wt\text{-binop } G \text{ BitAnd } \quad t1 \ t2 = ((t1 = PrimT Integer) \wedge (t2 = PrimT Integer))$
 $| wt\text{-binop } G \text{ And } \quad t1 \ t2 = ((t1 = PrimT Boolean) \wedge (t2 = PrimT Boolean))$
 $| wt\text{-binop } G \text{ BitXor } \quad t1 \ t2 = ((t1 = PrimT Integer) \wedge (t2 = PrimT Integer))$
 $| wt\text{-binop } G \text{ Xor } \quad t1 \ t2 = ((t1 = PrimT Boolean) \wedge (t2 = PrimT Boolean))$
 $| wt\text{-binop } G \text{ BitOr } \quad t1 \ t2 = ((t1 = PrimT Integer) \wedge (t2 = PrimT Integer))$
 $| wt\text{-binop } G \text{ Or } \quad t1 \ t2 = ((t1 = PrimT Boolean) \wedge (t2 = PrimT Boolean))$
 $| wt\text{-binop } G \text{ CondAnd } \quad t1 \ t2 = ((t1 = PrimT Boolean) \wedge (t2 = PrimT Boolean))$
 $| wt\text{-binop } G \text{ CondOr } \quad t1 \ t2 = ((t1 = PrimT Boolean) \wedge (t2 = PrimT Boolean))$

Typing for terms

type-synonym $tys = ty + ty \text{ list}$

translations

$(type) \ tys \leq (type) \ ty + ty \text{ list}$

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

where

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

— well-typed statements

$| Skip: \quad E, dt \models Skip :: \sqrt{ }$
 $| Expr: \llbracket E, dt \models e :: -T \rrbracket \Longrightarrow \quad E, dt \models Expr \ e :: \sqrt{ }$
— cf. 14.6
 $| Lab: \ E, dt \models c :: \sqrt{ } \Longrightarrow \quad E, dt \models l \cdot c :: \sqrt{ }$
 $| Comp: \llbracket E, dt \models c1 :: \sqrt{ }; \quad E, dt \models c2 :: \sqrt{ } \rrbracket \Longrightarrow \quad E, dt \models c1 ;; c2 :: \sqrt{ }$

— cf. 14.8

- | *If*: $\llbracket E, dt \models e :: -\text{PrimT Boolean};$
 $\quad E, dt \models c1 :: \checkmark;$
 $\quad E, dt \models c2 :: \checkmark \rrbracket \implies$
 $\quad E, dt \models \text{If}(e) \ c1 \ \text{Else} \ c2 :: \checkmark$
- cf. 14.10
- | *Loop*: $\llbracket E, dt \models e :: -\text{PrimT Boolean};$
 $\quad E, dt \models c :: \checkmark \rrbracket \implies$
 $\quad E, dt \models l \cdot \text{While}(e) \ c :: \checkmark$
- cf. 14.13, 14.15, 14.16
- | *Jmp*: $\quad E, dt \models \text{Jmp} \ \text{jump} :: \checkmark$
- cf. 14.16
- | *Throw*: $\llbracket E, dt \models e :: -\text{Class} \ tn;$
 $\quad \text{prg} \ E \vdash \text{tn} \preceq_C \ \text{SXcpt} \ \text{Throwable} \rrbracket \implies$
 $\quad E, dt \models \text{Throw} \ e :: \checkmark$
- cf. 14.18
- | *Try*: $\llbracket E, dt \models c1 :: \checkmark; \text{prg} \ E \vdash \text{tn} \preceq_C \ \text{SXcpt} \ \text{Throwable};$
 $\quad \text{lcl} \ E \ (V\text{Name} \ vn) = \text{None}; E \ (\text{lcl} \ := \ \text{lcl} \ E \ (V\text{Name} \ vn) \rightarrow \text{Class} \ tn) \rrbracket, dt \models c2 :: \checkmark \rrbracket$
 \implies
 $\quad E, dt \models \text{Try} \ c1 \ \text{Catch}(tn \ vn) \ c2 :: \checkmark$
- cf. 14.18
- | *Fin*: $\llbracket E, dt \models c1 :: \checkmark; E, dt \models c2 :: \checkmark \rrbracket \implies$
 $\quad E, dt \models c1 \ \text{Finally} \ c2 :: \checkmark$
- | *Init*: $\llbracket \text{is-class} \ (\text{prg} \ E) \ C \rrbracket \implies$
 $\quad E, dt \models \text{Init} \ C :: \checkmark$
- *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*: $\llbracket \text{is-acc-class} \ (\text{prg} \ E) \ (\text{pkg} \ E) \ C \rrbracket \implies$
 $\quad E, dt \models \text{NewC} \ C :: -\text{Class} \ C$
- cf. 15.9
- | *NewA*: $\llbracket \text{is-acc-type} \ (\text{prg} \ E) \ (\text{pkg} \ E) \ T;$
 $\quad E, dt \models i :: -\text{PrimT} \ \text{Integer} \rrbracket \implies$
 $\quad E, dt \models \text{New} \ T[i] :: -T.[]$
- cf. 15.15
- | *Cast*: $\llbracket E, dt \models e :: -T; \text{is-acc-type} \ (\text{prg} \ E) \ (\text{pkg} \ E) \ T';$
 $\quad \text{prg} \ E \vdash T \preceq^? \ T' \rrbracket \implies$
 $\quad E, dt \models \text{Cast} \ T' \ e :: -T'$
- cf. 15.19.2
- | *Inst*: $\llbracket E, dt \models e :: -\text{RefT} \ T; \text{is-acc-type} \ (\text{prg} \ E) \ (\text{pkg} \ E) \ (\text{RefT} \ T');$
 $\quad \text{prg} \ E \vdash \text{RefT} \ T \preceq^? \ \text{RefT} \ T' \rrbracket \implies$
 $\quad E, dt \models e \ \text{InstOf} \ T' :: -\text{PrimT} \ \text{Boolean}$
- cf. 15.7.1
- | *Lit*: $\llbracket \text{typeof} \ dt \ x = \text{Some} \ T \rrbracket \implies$
 $\quad E, dt \models \text{Lit} \ x :: -T$
- | *UnOp*: $\llbracket E, dt \models e :: -T_e; \text{wt-unop} \ \text{unop} \ T_e; T = \text{PrimT} \ (\text{unop-type} \ \text{unop}) \rrbracket$
 \implies
 $\quad E, dt \models \text{UnOp} \ \text{unop} \ e :: -T$

| *BinOp*: $\llbracket E, dt \models e1 :: -T1; E, dt \models e2 :: -T2; wt\text{-binop } (prg\ E)\ binop\ T1\ T2;$
 $T = PrimT\ (binop\text{-type}\ binop) \rrbracket$
 \implies
 $E, dt \models BinOp\ binop\ e1\ e2 :: -T$

— cf. 15.10.2, 15.11.1

| *Super*: $\llbracket lcl\ E\ This = Some\ (Class\ C); C \neq Object;$
 $class\ (prg\ E)\ C = Some\ c \rrbracket \implies$
 $E, dt \models Super :: -Class\ (super\ c)$

— cf. 15.13.1, 15.10.1, 15.12

| *Acc*: $\llbracket E, dt \models va :: =T \rrbracket \implies$
 $E, dt \models Acc\ va :: -T$

— cf. 15.25, 15.25.1

| *Ass*: $\llbracket E, dt \models va :: =T; va \neq LVar\ This;$
 $E, dt \models v :: -T';$
 $prg\ E \vdash T' \preceq T \rrbracket \implies$
 $E, dt \models va = v :: -T'$

— cf. 15.24

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

— cf. 15.11.1, 15.11.2, 15.11.3

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

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

— 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 (enshured indirectly by *Call*). Note that l is just a dummy value. It is only used in the smallstep semantics. To proof typesafety directly for the smallstep semantics we would have to assume conformance of l here!

| *Body*: $\llbracket is\text{-class}\ (prg\ E)\ D;$
 $E, dt \models blk :: \surd;$
 $(lcl\ E)\ Result = Some\ T;$
 $is\text{-type}\ (prg\ E)\ T \rrbracket \implies$
 $E, dt \models Body\ D\ blk :: -T$

— The class D implementing the method must not directly be accessible from the current package $pkg\ E$, but can also be indirectly accessible due to inheritance (enshured in *Call*) The result type hasn't got to be accessible in Java! (If it is not accessible you can only assign it to Object). For dummy value l see rule *Methd*.

— well-typed variables

— cf. 15.13.1

| *LVar*: $\llbracket lcl\ E\ vn = Some\ T; is\text{-acc}\text{-type}\ (prg\ E)\ (pkg\ E)\ T \rrbracket \implies$
 $E, dt \models LVar\ vn :: =T$

— cf. 15.10.1

| *FVar*: $\llbracket E, dt \models e :: -Class\ C;$

$accfield (prg E) (cls E) C fn = Some (statDeclC, f) \implies$
 $E, dt \models \{cls E, statDeclC, is-static f\} e..fn ::= (type f)$
 — cf. 15.12
 | $AVar: \llbracket E, dt \models e :: - T. \rrbracket;$
 $E, dt \models i :: - PrimT Integer \implies$
 $E, dt \models e.[i] :: T$

— well-typed expression lists

— cf. 15.11.???
 | $Nil: E, dt \models [] :: \doteq []$
 — cf. 15.11.???
 | $Cons: \llbracket E, dt \models e :: - T;$
 $E, dt \models es :: \doteq Ts \rrbracket \implies$
 $E, dt \models e \# es :: \doteq T \# Ts$

abbreviation

$wt-syntax :: env \Rightarrow [term, tys] \Rightarrow bool (-|-:- [51, 51, 51] 50)$
where $E \vdash t :: T == E, empty-dt \models t :: T$

abbreviation

$wt-stmt-syntax :: env \Rightarrow stmt \Rightarrow bool (-|-|\sqrt [51, 51] 50)$
where $E \vdash s :: \sqrt == E \vdash In1 s :: In1 (PrimT Void)$

abbreviation

$ty-expr-syntax :: env \Rightarrow [expr, ty] \Rightarrow bool (-|-:- [51, 51, 51] 50)$
where $E \vdash e :: - T == E \vdash In1 e :: In1 T$

abbreviation

$ty-var-syntax :: env \Rightarrow [var, ty] \Rightarrow bool (-|-:=- [51, 51, 51] 50)$
where $E \vdash e :: T == E \vdash In2 e :: In1 T$

abbreviation

$ty-exprs-syntax :: env \Rightarrow [expr list, ty list] \Rightarrow bool (-|-:;\doteq [51, 51, 51] 50)$
where $E \vdash e :: \doteq T == E \vdash In3 e :: Inr T$

notation (ASCII)

$wt-syntax (-|-:- [51, 51, 51] 50)$ **and**
 $wt-stmt-syntax (-|-:;< [51, 51] 50)$ **and**
 $ty-expr-syntax (-|-:- [51, 51, 51] 50)$ **and**
 $ty-var-syntax (-|-:=- [51, 51, 51] 50)$ **and**
 $ty-exprs-syntax (-|-:;\#- [51, 51, 51] 50)$

declare *not-None-eq* [*simp del*]

declare *if-split* [*split del*] *if-split-asm* [*split del*]

declare *split-paired-All* [*simp del*] *split-paired-Ex* [*simp del*]

$\langle ML \rangle$

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

$E, dt \models In2 (LVar vn) \quad \quad \quad :: T$
 $E, dt \models In2 (\{accC, statDeclC, s\} e..fn) :: T$
 $E, dt \models In2 (e.[i]) \quad \quad \quad \quad \quad :: T$
 $E, dt \models In1 (NewC C) \quad \quad \quad \quad \quad :: T$
 $E, dt \models In1 (New T'[i]) \quad \quad \quad \quad \quad :: T$
 $E, dt \models In1 (Cast T' e) \quad \quad \quad \quad \quad :: T$

$$\begin{array}{ll}
E, dt \models_{In1l} (e \text{ InstOf } T') & :: T \\
E, dt \models_{In1l} (Lit \ x) & :: T \\
E, dt \models_{In1l} (UnOp \ unop \ e) & :: T \\
E, dt \models_{In1l} (BinOp \ binop \ e1 \ e2) & :: T \\
E, dt \models_{In1l} (Super) & :: T \\
E, dt \models_{In1l} (Acc \ va) & :: T \\
E, dt \models_{In1l} (Ass \ va \ v) & :: T \\
E, dt \models_{In1l} (e0 \ ? \ e1 \ : \ e2) & :: T \\
E, dt \models_{In1l} (\{accC, statT, mode\}e \cdot mn(\{pT'\}p)) & :: T \\
E, dt \models_{In1l} (Methd \ C \ sig) & :: T \\
E, dt \models_{In1l} (Body \ D \ blk) & :: T \\
E, dt \models_{In3} (\[]) & :: Ts \\
E, dt \models_{In3} (e \# \ es) & :: Ts \\
E, dt \models_{In1r} Skip & :: x \\
E, dt \models_{In1r} (Expr \ e) & :: x \\
E, dt \models_{In1r} (c1 \ ; \ ; \ c2) & :: x \\
E, dt \models_{In1r} (l \cdot \ c) & :: x \\
E, dt \models_{In1r} (If(e) \ c1 \ Else \ c2) & :: x \\
E, dt \models_{In1r} (l \cdot \ While(e) \ c) & :: x \\
E, dt \models_{In1r} (Jmp \ jump) & :: x \\
E, dt \models_{In1r} (Throw \ e) & :: x \\
E, dt \models_{In1r} (Try \ c1 \ Catch(tn \ vn) \ c2) & :: x \\
E, dt \models_{In1r} (c1 \ Finally \ c2) & :: x \\
E, dt \models_{In1r} (Init \ C) & :: x
\end{array}$$

declare *not-None-eq* [*simp*]
declare *if-split* [*split*] *if-split-asm* [*split*]
declare *split-paired-All* [*simp*] *split-paired-Ex* [*simp*]
 <ML>

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

lemma *is-acc-iface-is-iface*: *is-acc-iface* $G \ P \ I \implies is-iface \ G \ I$
 <proof>

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

lemma *is-acc-type-is-type*: *is-acc-type* $G \ P \ T \implies is-type \ G \ T$
 <proof>

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

lemma *wt-Methd-is-methd*:
 $E \vdash_{In1l} (\text{Methd } C \ sig) :: T \implies is-methd \ (prg \ E) \ C \ sig$
 <proof>

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

lemma *wt-Call*:

$$\begin{aligned} & \llbracket E, dt \models e :: -\text{RefT } \text{statT}; E, dt \models ps :: \doteq pTs; \\ & \quad \text{max-spec } (\text{prg } E) (\text{cls } E) \text{ statT } (\downarrow \text{name} = mn, \text{parTs} = pTs) \\ & \quad = \{((\text{statDeclC}, m), pTs')\}; rT = (\text{resTy } m); \text{accC} = \text{cls } E; \\ & \quad \text{mode} = \text{invmode } m \ e \rrbracket \implies E, dt \models \{\text{accC}, \text{statT}, \text{mode}\} e \cdot mn(\{pTs'\}ps) :: -rT \\ & \langle \text{proof} \rangle \end{aligned}$$

lemma *invocationTypeExpr-noClassD*:

$$\begin{aligned} & \llbracket E \vdash e :: -\text{RefT } \text{statT} \rrbracket \\ & \implies (\forall \text{ statC}. \text{statT} \neq \text{ClassT } \text{statC}) \longrightarrow \text{invmode } m \ e \neq \text{SuperM} \\ & \langle \text{proof} \rangle \end{aligned}$$

lemma *wt-Super*:

$$\begin{aligned} & \llbracket \text{lcl } E \text{ This} = \text{Some } (\text{Class } C); C \neq \text{Object}; \text{class } (\text{prg } E) \ C = \text{Some } c; D = \text{super } c \rrbracket \\ & \implies E, dt \models \text{Super} :: -\text{Class } D \\ & \langle \text{proof} \rangle \end{aligned}$$

lemma *wt-FVar*:

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

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*:

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

$\langle \text{proof} \rangle$

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*: $E, dt \models \text{In3 } t :: U = (\exists Ts. U = \text{Inr } Ts \wedge E, dt \models t :: \doteq Ts)$

<proof>

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

<ML>

lemma *wt-elim-BinOp*:

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

lemma *Inj-eq-lemma* [*simp*]:

$(\forall T. (\exists T'. T = \text{Inj } T' \wedge P \ T') \longrightarrow Q \ T) = (\forall T'. P \ T' \longrightarrow Q \ (\text{Inj } T'))$
<proof>

lemma *single-valued-tys-lemma* [*rule-format* (*no-asm*)]:

$\forall S \ T. G \vdash S \leq T \longrightarrow G \vdash T \leq S \longrightarrow S = T \implies E, dt \models t :: T \implies$
 $G = \text{prg } E \longrightarrow (\forall T'. E, dt \models t :: T' \longrightarrow T = T')$
<proof>

lemma *single-valued-tys*:

ws-prog (*prg* E) \implies *single-valued* $\{(t, T). E, dt \models t :: T\}$
<proof>

lemma *typeof-empty-is-type*: *typeof* $(\lambda a. \text{None}) \ v = \text{Some } T \implies$ *is-type* $G \ T$
<proof>

lemma *typeof-is-type*: $(\forall a. v \neq \text{Addr } a) \implies \exists T. \text{typeof } dt \ v = \text{Some } T \wedge$ *is-type* $G \ T$
<proof>

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

$jumpNestingOkS\ jmps\ (Skip) = True$
 $jumpNestingOkS\ jmps\ (Expr\ e) = True$
 $jumpNestingOkS\ jmps\ (j \cdot s) = jumpNestingOkS\ (\{j\} \cup jmps)\ s$
 $jumpNestingOkS\ jmps\ (c1;;c2) = (jumpNestingOkS\ jmps\ c1 \wedge jumpNestingOkS\ jmps\ c2)$
 $jumpNestingOkS\ jmps\ (If\ (e)\ c1\ Else\ c2) = (jumpNestingOkS\ jmps\ c1 \wedge jumpNestingOkS\ jmps\ c2)$
 $jumpNestingOkS\ jmps\ (l \cdot While(e)\ c) = jumpNestingOkS\ (\{Cont\ l\} \cup jmps)\ c$
 — The label of the while loop only handles continue jumps. Breaks are only handled by *Lab*
 $jumpNestingOkS\ jmps\ (Jmp\ j) = (j \in jmps)$
 $jumpNestingOkS\ jmps\ (Throw\ e) = True$
 $jumpNestingOkS\ jmps\ (Try\ c1\ Catch\ (C\ vn)\ c2) = (jumpNestingOkS\ jmps\ c1 \wedge jumpNestingOkS\ jmps\ c2)$
 $jumpNestingOkS\ jmps\ (c1\ Finally\ c2) = (jumpNestingOkS\ jmps\ c1 \wedge jumpNestingOkS\ jmps\ c2)$
 $jumpNestingOkS\ jmps\ (Init\ C) = True$
 — wellformedness of the program must ensure that for all initializers $jumpNestingOkS$ holds
 — Dummy analysis for intermediate smallest term *FinA*
 $jumpNestingOkS\ jmps\ (FinA\ a\ c) = False$

definition $jumpNestingOk :: jump\ set \Rightarrow term \Rightarrow bool$ **where**

$jumpNestingOk\ jmps\ t = (case\ t\ of$
 $\quad In1\ se \Rightarrow (case\ se\ of$
 $\quad\quad Inl\ e \Rightarrow True$
 $\quad\quad | Inr\ s \Rightarrow jumpNestingOkS\ jmps\ s)$
 $\quad | In2\ v \Rightarrow True$
 $\quad | In3\ es \Rightarrow True)$

lemma $jumpNestingOk\ expr\ simp\ [simp]: jumpNestingOk\ jmps\ (In1l\ e) = True$
 $\langle proof \rangle$

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

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

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

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

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

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

lemma *jumpNestingOk-expr-list-simp1* [simp]:
jumpNestingOk jmps (es::expr list) = True
 ⟨proof⟩

Calculation of assigned variables for boolean expressions

2 Very restricted calculation fallback calculation

primrec *the-LVar-name* :: *var* ⇒ *lname*
 where *the-LVar-name* (LVar *n*) = *n*

primrec *assignsE* :: *expr* ⇒ *lname set*
and *assignsV* :: *var* ⇒ *lname set*
and *assignsEs*:: *expr list* ⇒ *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* ∨ *binop=CondOr*
 then (*assignsE e1*)
 else (*assignsE e1*) ∪ (*assignsE e2*))
 | *assignsE* (Super) = {}
 | *assignsE* (Acc *v*) = *assignsV v*
 | *assignsE* (*v:=e*) = (*assignsV v*) ∪ (*assignsE e*) ∪
 (if ∃ *n. v=(LVar n)* then {*the-LVar-name v*}
 else {})
 | *assignsE* (*b? e1 : e2*) = (*assignsE b*) ∪ ((*assignsE e1*) ∩ (*assignsE e2*))
 | *assignsE* ({*accC,statT,mode*}objRef·mn({*pTs*}args))
 = (*assignsE objRef*) ∪ (*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* ∪ *assignsE e2*

| *assignsEs* [] = {}
 | *assignsEs* (*e#es*) = *assignsE e* ∪ *assignsEs es*

definition *assigns* :: *term* ⇒ *lname set* where

assigns t = (case *t* of
 In1 *se* ⇒ (case *se* of
 Inl *e* ⇒ *assignsE e*
 | Inr *s* ⇒ {})
 | In2 *v* ⇒ *assignsV v*
 | In3 *es* ⇒ *assignsEs es*)

lemma *assigns-expr-simp* [simp]: *assigns (In1l e) = assignsE e*
 ⟨proof⟩

lemma *assigns-expr-simp1* [simp]: *assigns* ($\langle e \rangle$) = *assignsE* *e*
 \langle proof \rangle

lemma *assigns-stmt-simp* [simp]: *assigns* (*In1r* *s*) = {}
 \langle proof \rangle

lemma *assigns-stmt-simp1* [simp]: *assigns* ($\langle s::\text{stmt} \rangle$) = {}
 \langle proof \rangle

lemma *assigns-var-simp* [simp]: *assigns* (*In2* *v*) = *assignsV* *v*
 \langle proof \rangle

lemma *assigns-var-simp1* [simp]: *assigns* ($\langle v \rangle$) = *assignsV* *v*
 \langle proof \rangle

lemma *assigns-expr-list-simp* [simp]: *assigns* (*In3* *es*) = *assignsEs* *es*
 \langle proof \rangle

lemma *assigns-expr-list-simp1* [simp]: *assigns* ($\langle es \rangle$) = *assignsEs* *es*
 \langle proof \rangle

3 Analysis of constant expressions

primrec *constVal* :: *expr* \Rightarrow *val option*

where

$\text{constVal } (\text{NewC } c) = \text{None}$
 $\text{constVal } (\text{NewA } t e) = \text{None}$
 $\text{constVal } (\text{Cast } t e) = \text{None}$
 $\text{constVal } (\text{Inst } e r) = \text{None}$
 $\text{constVal } (\text{Lit } val) = \text{Some } val$
 $\text{constVal } (\text{UnOp } unop e) = (\text{case } (\text{constVal } e) \text{ of}$
 $\text{None} \Rightarrow \text{None}$
 $\text{Some } v \Rightarrow \text{Some } (\text{eval-unop } unop v))$
 $\text{constVal } (\text{BinOp } binop e1 e2) = (\text{case } (\text{constVal } e1) \text{ of}$
 $\text{None} \Rightarrow \text{None}$
 $\text{Some } v1 \Rightarrow (\text{case } (\text{constVal } e2) \text{ of}$
 $\text{None} \Rightarrow \text{None}$
 $\text{Some } v2 \Rightarrow \text{Some } (\text{eval-binop } binop v1 v2)))$
 $\text{constVal } (\text{Super}) = \text{None}$
 $\text{constVal } (\text{Acc } v) = \text{None}$
 $\text{constVal } (\text{Ass } v e) = \text{None}$
 $\text{constVal } (\text{Cond } b e1 e2) = (\text{case } (\text{constVal } b) \text{ of}$
 $\text{None} \Rightarrow \text{None}$
 $\text{Some } bv \Rightarrow (\text{case the-Bool } bv \text{ of}$
 $\text{True} \Rightarrow (\text{case } (\text{constVal } e2) \text{ of}$
 $\text{None} \Rightarrow \text{None}$
 $\text{Some } v \Rightarrow \text{constVal } e1)$
 $\text{False} \Rightarrow (\text{case } (\text{constVal } e1) \text{ of}$
 $\text{None} \Rightarrow \text{None}$
 $\text{Some } v \Rightarrow \text{constVal } e2)))$

— 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*: $\text{constVal } e = \text{Some } v$ **and**

hyp-Lit: $\bigwedge v. P (\text{Lit } v)$ **and**

hyp-UnOp: $\bigwedge \text{unop } e'. P e' \implies P (\text{UnOp unop } e')$ **and**

hyp-BinOp: $\bigwedge \text{binop } e1 e2. \llbracket P e1; P e2 \rrbracket \implies P (\text{BinOp binop } e1 e2)$ **and**

hyp-CondL: $\bigwedge b \text{ bv } e1 e2. \llbracket \text{constVal } b = \text{Some } \text{bv}; \text{the-Bool } \text{bv}; P b; P e1 \rrbracket$
 $\implies P (b? e1 : e2)$ **and**

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

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

```

  assigns-if b (NewC c)           = UNIV — can never evaluate to Boolean
| assigns-if b (NewA t e)       = UNIV — can never evaluate to Boolean
| assigns-if b (Cast t e)       = assigns-if b e
| assigns-if b (Inst e r)       = assignsE e — Inst has type Boolean but e is a reference type
| assigns-if b (Lit val)        = (if val=Bool b then {} else UNIV)
| assigns-if b (UnOp unop e)    = (case constVal (UnOp unop e) of
  None  => (if unop = UNot
            then assigns-if (¬b) e
            else UNIV)
  | Some v => (if v=Bool b
              then {}
              else UNIV))
| assigns-if b (BinOp binop e1 e2)
  = (case constVal (BinOp binop e1 e2) of
    None => (if binop=CondAnd then
              (case b of
                True  => assigns-if True e1 ∪ assigns-if True e2
                | False => assigns-if False e1 ∩
                          (assigns-if True e1 ∪ assigns-if False e2))
            else
              (if binop=CondOr then
                (case b of
                  True  => assigns-if True e1 ∩
                          (assigns-if False e1 ∪ assigns-if True e2)
                  | False => assigns-if False e1 ∪ assigns-if False e2)
                else assignsE e1 ∪ assignsE e2))
    | Some v => (if v=Bool b then {} else UNIV))

```

$| \text{assigns-if } b \text{ (Super)} = \text{UNIV} \text{ — can never evaluate to Boolean}$
 $| \text{assigns-if } b \text{ (Acc } v) = (\text{assignsV } v)$
 $| \text{assigns-if } b \text{ (} v := e) = (\text{assignsE } (\text{Ass } v \ e))$
 $| \text{assigns-if } b \text{ (} c? \ e1 : \ e2) = (\text{assignsE } c) \cup$
 $\quad (\text{case } (\text{constVal } c) \text{ of}$
 $\quad \quad \text{None} \Rightarrow (\text{assigns-if } b \ e1) \cap$
 $\quad \quad (\text{assigns-if } b \ e2)$
 $\quad | \text{Some } bv \Rightarrow (\text{case the-Bool } bv \text{ of}$
 $\quad \quad \text{True} \Rightarrow \text{assigns-if } b \ e1$
 $\quad \quad | \text{False} \Rightarrow \text{assigns-if } b \ e2))$
 $| \text{assigns-if } b \text{ (}\{\text{accC, statT, mode}\} \text{objRef.mn}(\{pTs\} \text{args}))$
 $\quad = \text{assignsE } (\{\text{accC, statT, mode}\} \text{objRef.mn}(\{pTs\} \text{args}))$
— Only dummy analysis for intermediate expressions *Method*, *Body*, *InsInitE* and *Callee*
 $| \text{assigns-if } b \text{ (Method } C \ \text{sig}) = \{\}$
 $| \text{assigns-if } b \text{ (Body } C \ s) = \{\}$
 $| \text{assigns-if } b \text{ (InsInitE } s \ e) = \{\}$
 $| \text{assigns-if } b \text{ (Callee } l \ e) = \{\}$

lemma *assigns-if-const-b-simp*:

assumes *boolConst*: $\text{constVal } e = \text{Some } (\text{Bool } b)$ (**is** *?Const* $b \ e$)

shows $\text{assigns-if } b \ e = \{\}$ (**is** *?Ass* $b \ e$)

<proof>

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

assumes *boolConst*: $\text{constVal } e = \text{Some } (\text{Bool } b)$ (**is** *?Const* $b \ e$)

shows $\text{assigns-if } (\neg b) \ e = \text{UNIV}$ (**is** *?Ass* $b \ e$)

<proof>

5 Lifting set operations to range of tables (map to a set)

definition

union-ts :: $('a, 'b) \text{ tables} \Rightarrow ('a, 'b) \text{ tables} \Rightarrow ('a, 'b) \text{ tables}$ ($- \Rightarrow \cup -$ [67,67] 65)

where $A \Rightarrow \cup B = (\lambda k. A \ k \cup B \ k)$

definition

intersect-ts :: $('a, 'b) \text{ tables} \Rightarrow ('a, 'b) \text{ tables} \Rightarrow ('a, 'b) \text{ tables}$ ($- \Rightarrow \cap -$ [72,72] 71)

where $A \Rightarrow \cap B = (\lambda k. A \ k \cap B \ k)$

definition

all-union-ts :: $('a, 'b) \text{ tables} \Rightarrow 'b \ \text{set} \Rightarrow ('a, 'b) \text{ tables}$ (**infixl** $\Rightarrow \cup_{\forall}$ 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!*]: $(c \notin B \ k \implies c \in A \ k) \implies c \in (A \Rightarrow \cup B) \ k$
 ⟨*proof*⟩

lemma *union-tsE* [*elim!*]:
 $\llbracket c \in (A \Rightarrow \cup B) \ k; (c \in A \ k \implies P); (c \in B \ k \implies P) \rrbracket \implies P$
 ⟨*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!*]: $\llbracket c \in A \ k; c \in B \ k \rrbracket \implies c \in (A \Rightarrow \cap B) \ k$
 ⟨*proof*⟩

lemma *intersect-tsD1*: $c \in (A \Rightarrow \cap B) \ k \implies c \in A \ k$
 ⟨*proof*⟩

lemma *intersect-tsD2*: $c \in (A \Rightarrow \cap B) \ k \implies c \in B \ k$
 ⟨*proof*⟩

lemma *intersect-tsE* [*elim!*]:
 $\llbracket c \in (A \Rightarrow \cap B) \ k; \llbracket c \in A \ k; c \in B \ k \rrbracket \implies P \rrbracket \implies P$
 ⟨*proof*⟩

All-Union of tables and set

lemma *all-union-ts-iff* [*simp*]: $(c \in (A \Rightarrow \cup_{\forall} B) \ k) = (c \in A \ k \vee c \in B)$
 ⟨*proof*⟩

lemma *all-union-tsI1* [*elim?*]: $c \in A \ k \implies c \in (A \Rightarrow \cup_{\forall} B) \ k$
 ⟨*proof*⟩

lemma *all-union-tsI2* [*elim?*]: $c \in B \implies c \in (A \Rightarrow \cup_{\forall} B) \ k$
 ⟨*proof*⟩

lemma *all-union-tsCI* [*intro!*]: $(c \notin B \implies c \in A \ k) \implies c \in (A \Rightarrow \cup_{\forall} B) \ k$
 ⟨*proof*⟩

lemma *all-union-tsE* [*elim!*]:
 $\llbracket c \in (A \Rightarrow \cup_{\forall} B) \ k; (c \in A \ k \implies P); (c \in B \implies P) \rrbracket \implies P$
 ⟨*proof*⟩

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

definition

$range\text{-}inter\text{-}ts :: ('a, 'b)\ tables \Rightarrow 'b\ set (\Rightarrow \cap - 80)$
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

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

where

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

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

\Rightarrow

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

| $Lab: \llbracket Env \vdash B \gg \langle c \rangle \gg C; nrm\ A = nrm\ C \cap (brk\ C)\ l; brk\ A = rmlab\ l\ (brk\ C) \rrbracket$

\Rightarrow

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

| $Comp: \llbracket Env \vdash B \gg \langle c1 \rangle \gg C1; Env \vdash nrm\ C1 \gg \langle c2 \rangle \gg C2;$

$nrm\ A = nrm\ C2; brk\ A = (brk\ C1) \Rightarrow \cap (brk\ C2) \rrbracket$

\Rightarrow

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

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

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

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

$nrm\ A = nrm\ C1 \cap nrm\ C2;$

$brk\ A = brk\ C1 \Rightarrow \cap brk\ C2 \rrbracket$

\Rightarrow

$Env \vdash B \gg \langle If(e)\ c1\ Else\ c2 \rangle \gg A$

— Note that E is not further used, because we take the specialized sets that also consider if the expression evaluates to true or false. Inside of e there is no **break** or **finally**, so the break map of E will be the trivial one. So $Env \vdash B \gg \langle e \rangle \gg E$ is just used to ensure the definite assignment in expression e . Notice the implicit analysis of a constant boolean expression e in this rule. For example, if e is constantly *True* then *assigns-if False e* = *UNIV* and therefor $nrm\ C2 = UNIV$. So finally $nrm\ A = nrm\ C1$. For the break maps this trick workd too, because the trival break map will map all labels to *UNIV*. In the example, if no break occurs in $c2$ the break maps will trivially map to *UNIV* and if a break occurs it will map to *UNIV* too, because *assigns-if False e* = *UNIV*. So in the intersection of the break maps the path $c2$ will have no contribution.

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

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

$nrm\ A = nrm\ C \cap (B \cup \text{assigns-if False } e);$

$brk\ A = brk\ C \rrbracket$

\Rightarrow

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

— The *Loop* rule resembles some of the ideas of the *If* rule. For the $nrm\ A$ 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.

| *Jmp*: $\llbracket \text{jump} = \text{Ret} \longrightarrow \text{Result} \in B; \\ \text{nrm } A = \text{UNIV}; \\ \text{brk } A = (\text{case jump of} \\ \quad \text{Break } l \Rightarrow \lambda k. \text{ if } k=l \text{ then } B \text{ else } \text{UNIV} \\ \quad | \text{Cont } l \Rightarrow \lambda k. \text{UNIV} \\ \quad | \text{Ret} \Rightarrow \lambda k. \text{UNIV}) \rrbracket$

\Longrightarrow

$\text{Env} \vdash B \gg \langle \text{Jmp jump} \rangle \gg A$

— 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 ensure that the result value is assigned.

| *Throw*: $\llbracket \text{Env} \vdash B \gg \langle e \rangle \gg E; \text{nrm } A = \text{UNIV}; \text{brk } A = (\lambda l. \text{UNIV}) \rrbracket \\ \Longrightarrow \text{Env} \vdash B \gg \langle \text{Throw } e \rangle \gg A$

| *Try*: $\llbracket \text{Env} \vdash B \gg \langle c1 \rangle \gg C1; \\ \text{Env}(\text{lcl} := \text{lcl } \text{Env}(\text{VName } vn \mapsto \text{Class } C)) \vdash (B \cup \{\text{VName } vn\}) \gg \langle c2 \rangle \gg C2; \\ \text{nrm } A = \text{nrm } C1 \cap \text{nrm } C2; \\ \text{brk } A = \text{brk } C1 \Rightarrow \cap \text{brk } C2 \rrbracket \\ \Longrightarrow \text{Env} \vdash B \gg \langle \text{Try } c1 \text{ Catch}(C \text{ } vn) \text{ } c2 \rangle \gg A$

| *Fin*: $\llbracket \text{Env} \vdash B \gg \langle c1 \rangle \gg C1; \\ \text{Env} \vdash B \gg \langle c2 \rangle \gg C2; \\ \text{nrm } A = \text{nrm } C1 \cup \text{nrm } C2; \\ \text{brk } A = ((\text{brk } C1) \Rightarrow \cup_{\vee} (\text{nrm } C2)) \Rightarrow \cap (\text{brk } C2) \rrbracket \\ \Longrightarrow \\ \text{Env} \vdash B \gg \langle c1 \text{ Finally } c2 \rangle \gg A$

— 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* ($\Rightarrow \cup_{\vee}$). If *c2* exits with a break this break will appear in the overall result state. We don't know if *c1* completed normally or abruptly (maybe with an exception not only a break) so *c1* has no contribution to the break map following this path.

— Evaluation of expressions and the break sets of definite assignment: Thinking of a Java expression we assume that we can never have a break statement inside of an expression. So for all expressions the break sets could be set to the trivial one: $\lambda l. \text{UNIV}$. But we can't trivially prove, that evaluating an expression will never result in a break, although Java expressions already syntactically don't allow nested statements in them. The reason are the nested class initialization statements which are inserted by the evaluation rules. So to prove the absence of a break we need to ensure, that the initialization statements will never end up in a break. In a wellformed initialization statement, of course, were 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*

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

— Wellformedness of a program will ensure, that every static initialiser is definitely assigned and the jumps are nested correctly. The case here for *Init* is just for convenience, to get a proper precondition for the induction hypothesis in various proofs, so that we don't have to expand the initialisation on every point where it is

triggered by the evaluation rules.

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

| *NewA*: $Env \vdash B \gg \langle e \rangle \gg A$
 \implies
 $Env \vdash B \gg \langle New\ T[e] \rangle \gg A$

| *Cast*: $Env \vdash B \gg \langle e \rangle \gg A$
 \implies
 $Env \vdash B \gg \langle Cast\ T\ e \rangle \gg A$

| *Inst*: $Env \vdash B \gg \langle e \rangle \gg A$
 \implies
 $Env \vdash B \gg \langle e\ InstOf\ T \rangle \gg A$

| *Lit*: $Env \vdash B \gg \langle Lit\ v \rangle \gg (\uparrow nrm=B, brk=\lambda l. UNIV)$

| *UnOp*: $Env \vdash B \gg \langle e \rangle \gg A$
 \implies
 $Env \vdash B \gg \langle UnOp\ unop\ e \rangle \gg A$

| *CondAnd*: $\llbracket Env \vdash B \gg \langle e1 \rangle \gg E1; Env \vdash (B \cup assigns\text{-}if\ True\ e1) \gg \langle e2 \rangle \gg E2;$
 $nrm\ A = B \cup (assigns\text{-}if\ True\ (BinOp\ CondAnd\ e1\ e2) \cap$
 $assigns\text{-}if\ False\ (BinOp\ CondAnd\ e1\ e2));$
 $brk\ A = (\lambda l. UNIV) \rrbracket$
 \implies
 $Env \vdash B \gg \langle BinOp\ CondAnd\ e1\ e2 \rangle \gg A$

| *CondOr*: $\llbracket Env \vdash B \gg \langle e1 \rangle \gg E1; Env \vdash (B \cup assigns\text{-}if\ False\ e1) \gg \langle e2 \rangle \gg E2;$
 $nrm\ A = B \cup (assigns\text{-}if\ True\ (BinOp\ CondOr\ e1\ e2) \cap$
 $assigns\text{-}if\ False\ (BinOp\ CondOr\ e1\ e2));$
 $brk\ A = (\lambda l. UNIV) \rrbracket$
 \implies
 $Env \vdash B \gg \langle BinOp\ CondOr\ e1\ e2 \rangle \gg A$

| *BinOp*: $\llbracket Env \vdash B \gg \langle e1 \rangle \gg E1; Env \vdash nrm\ E1 \gg \langle e2 \rangle \gg A;$
 $binop \neq CondAnd; binop \neq CondOr \rrbracket$
 \implies
 $Env \vdash B \gg \langle BinOp\ binop\ e1\ e2 \rangle \gg A$

| *Super*: $This \in B$
 \implies
 $Env \vdash B \gg \langle Super \rangle \gg (\uparrow nrm=B, brk=\lambda l. UNIV)$

| *AccLVar*: $\llbracket vn \in B;$
 $nrm\ A = B; brk\ A = (\lambda k. UNIV) \rrbracket$
 \implies
 $Env \vdash B \gg \langle Acc\ (LVar\ vn) \rangle \gg A$

— To properly access a local variable we have to test the definite assignment here. The variable must occur in the set B

| *Acc*: $\llbracket \forall vn. v \neq LVar\ vn;$
 $Env \vdash B \gg \langle v \rangle \gg A \rrbracket$
 \implies
 $Env \vdash B \gg \langle Acc\ v \rangle \gg A$

| *AssLVar*: $\llbracket Env \vdash B \gg \langle e \rangle \gg E; nrm\ A = nrm\ E \cup \{vn\}; brk\ A = brk\ E \rrbracket$
 \implies
 $Env \vdash B \gg \langle (LVar\ vn) := e \rangle \gg A$

$$\begin{array}{l} | \text{Ass: } \llbracket \forall vn. v \neq \text{LVar } vn; \text{Env} \vdash B \gg \langle v \rangle \gg V; \text{Env} \vdash \text{nrm } V \gg \langle e \rangle \gg A \rrbracket \\ \implies \\ \text{Env} \vdash B \gg \langle v := e \rangle \gg A \end{array}$$

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

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

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

— 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 *Method* and then further to *Body* during evaluation to establish the definite assignment of *Method* during evaluation of *Call*, and of *Body* during evaluation of *Method*. This is necessary since definite assignment will be a precondition for each induction hypothesis coming out of the evaluation rules, and therefore we have to establish the definite assignment of the sub-evaluation during the type-safety proof. Note that well-typedness is also a precondition for type-safety and so we can omit some assertion that are already ensured by well-typedness.

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

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

— Note that A is not correlated to C . If the body statement returns abruptly with return, evaluation of *Body* will absorb this return and complete normally. So we cannot trivially get the assigned variables of the body statement since it has not completed normally or with a break. If the body completes normally we guarantee that the result variable is set with this rule. But if the body completes abruptly with a return we can't guarantee that the result variable is set here, since definite assignment only talks about normal completion and breaks. So for a return the *Jump* rule ensures that the result variable is set and then this information must be carried over to the *Body* rule by the conformance predicate of the state.

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

$$\begin{array}{l} | \text{FVar: } \text{Env} \vdash B \gg \langle e \rangle \gg A \\ \implies \end{array}$$

$$\begin{aligned} & \text{Env} \vdash B \gg \langle \{ \text{acc} C, \text{statDecl} C, \text{stat} \} e..fn \rangle \gg A \\ | \text{AVar}: & \llbracket \text{Env} \vdash B \gg \langle e1 \rangle \gg E1; \text{Env} \vdash \text{nrm } E1 \gg \langle e2 \rangle \gg A \rrbracket \\ & \implies \\ & \text{Env} \vdash B \gg \langle e1.[e2] \rangle \gg A \\ | \text{Nil}: & \text{Env} \vdash B \gg \langle []::\text{expr list} \rangle \gg (\text{nrm}=B, \text{brk}=\lambda l. \text{UNIV}) \\ | \text{Cons}: & \llbracket \text{Env} \vdash B \gg \langle e::\text{expr} \rangle \gg E; \text{Env} \vdash \text{nrm } E \gg \langle es \rangle \gg A \rrbracket \\ & \implies \\ & \text{Env} \vdash B \gg \langle e\#es \rangle \gg A \end{aligned}$$

declare *inj-term-sym-simps* [simp]
declare *assigns-if.simps* [simp del]
declare *split-paired-All* [simp del] *split-paired-Ex* [simp del]
 $\langle ML \rangle$

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

$$\begin{aligned} & \text{Env} \vdash B \gg \langle \text{Skip} \rangle \gg A \\ & \text{Env} \vdash B \gg \langle \text{In1r Skip} \rangle \gg A \\ & \text{Env} \vdash B \gg \langle \text{Expr } e \rangle \gg A \\ & \text{Env} \vdash B \gg \langle \text{In1r } (\text{Expr } e) \rangle \gg A \\ & \text{Env} \vdash B \gg \langle l \cdot c \rangle \gg A \\ & \text{Env} \vdash B \gg \langle \text{In1r } (l \cdot c) \rangle \gg A \\ & \text{Env} \vdash B \gg \langle c1;; c2 \rangle \gg A \\ & \text{Env} \vdash B \gg \langle \text{In1r } (c1;; c2) \rangle \gg A \\ & \text{Env} \vdash B \gg \langle \text{If } (e) \text{ } c1 \text{ Else } c2 \rangle \gg A \\ & \text{Env} \vdash B \gg \langle \text{In1r } (\text{If } (e) \text{ } c1 \text{ Else } c2) \rangle \gg A \\ & \text{Env} \vdash B \gg \langle l \cdot \text{While}(e) \text{ } c \rangle \gg A \\ & \text{Env} \vdash B \gg \langle \text{In1r } (l \cdot \text{While}(e) \text{ } c) \rangle \gg A \\ & \text{Env} \vdash B \gg \langle \text{Jmp jump} \rangle \gg A \\ & \text{Env} \vdash B \gg \langle \text{In1r } (\text{Jmp jump}) \rangle \gg A \\ & \text{Env} \vdash B \gg \langle \text{Throw } e \rangle \gg A \\ & \text{Env} \vdash B \gg \langle \text{In1r } (\text{Throw } e) \rangle \gg A \\ & \text{Env} \vdash B \gg \langle \text{Try } c1 \text{ Catch } (C \text{ } vn) \text{ } c2 \rangle \gg A \\ & \text{Env} \vdash B \gg \langle \text{In1r } (\text{Try } c1 \text{ Catch } (C \text{ } vn) \text{ } c2) \rangle \gg A \\ & \text{Env} \vdash B \gg \langle c1 \text{ Finally } c2 \rangle \gg A \\ & \text{Env} \vdash B \gg \langle \text{In1r } (c1 \text{ Finally } c2) \rangle \gg A \\ & \text{Env} \vdash B \gg \langle \text{Init } C \rangle \gg A \\ & \text{Env} \vdash B \gg \langle \text{In1r } (\text{Init } C) \rangle \gg A \\ & \text{Env} \vdash B \gg \langle \text{NewC } C \rangle \gg A \\ & \text{Env} \vdash B \gg \langle \text{In1l } (\text{NewC } C) \rangle \gg A \\ & \text{Env} \vdash B \gg \langle \text{New } T[e] \rangle \gg A \\ & \text{Env} \vdash B \gg \langle \text{In1l } (\text{New } T[e]) \rangle \gg A \\ & \text{Env} \vdash B \gg \langle \text{Cast } T \text{ } e \rangle \gg A \\ & \text{Env} \vdash B \gg \langle \text{In1l } (\text{Cast } T \text{ } e) \rangle \gg A \\ & \text{Env} \vdash B \gg \langle e \text{ InstOf } T \rangle \gg A \\ & \text{Env} \vdash B \gg \langle \text{In1l } (e \text{ InstOf } T) \rangle \gg A \\ & \text{Env} \vdash B \gg \langle \text{Lit } v \rangle \gg A \\ & \text{Env} \vdash B \gg \langle \text{In1l } (\text{Lit } v) \rangle \gg A \\ & \text{Env} \vdash B \gg \langle \text{UnOp unop } e \rangle \gg A \\ & \text{Env} \vdash B \gg \langle \text{In1l } (\text{UnOp unop } e) \rangle \gg A \\ & \text{Env} \vdash B \gg \langle \text{BinOp binop } e1 \text{ } e2 \rangle \gg A \\ & \text{Env} \vdash B \gg \langle \text{In1l } (\text{BinOp binop } e1 \text{ } e2) \rangle \gg A \\ & \text{Env} \vdash B \gg \langle \text{Super} \rangle \gg A \\ & \text{Env} \vdash B \gg \langle \text{In1l } (\text{Super}) \rangle \gg A \\ & \text{Env} \vdash B \gg \langle \text{Acc } v \rangle \gg A \\ & \text{Env} \vdash B \gg \langle \text{In1l } (\text{Acc } v) \rangle \gg A \end{aligned}$$

$Env \vdash B \gg \langle v := e \rangle \gg A$
 $Env \vdash B \gg In1l (v := e) \gg A$
 $Env \vdash B \gg \langle c ? e1 : e2 \rangle \gg A$
 $Env \vdash B \gg In1l (c ? e1 : e2) \gg A$
 $Env \vdash B \gg \langle \{accC, statT, mode\} e \cdot mn(\{pTs\} args) \rangle \gg A$
 $Env \vdash B \gg In1l (\{accC, statT, mode\} e \cdot mn(\{pTs\} args)) \gg A$
 $Env \vdash B \gg \langle Methd C sig \rangle \gg A$
 $Env \vdash B \gg In1l (Methd C sig) \gg A$
 $Env \vdash B \gg \langle Body D c \rangle \gg A$
 $Env \vdash B \gg In1l (Body D c) \gg A$
 $Env \vdash B \gg \langle LVar vn \rangle \gg A$
 $Env \vdash B \gg In2 (LVar vn) \gg A$
 $Env \vdash B \gg \langle \{accC, statDeclC, stat\} e \cdot fn \rangle \gg A$
 $Env \vdash B \gg In2 (\{accC, statDeclC, stat\} e \cdot fn) \gg A$
 $Env \vdash B \gg \langle e1.[e2] \rangle \gg A$
 $Env \vdash B \gg In2 (e1.[e2]) \gg A$
 $Env \vdash B \gg \langle [] :: expr list \rangle \gg A$
 $Env \vdash B \gg In3 ([] :: expr list) \gg A$
 $Env \vdash B \gg \langle e \# es \rangle \gg A$
 $Env \vdash B \gg In3 (e \# es) \gg A$
declare inj-term-sym-simps [simp del]
declare assigns-if.simps [simp]
declare split-paired-All [simp] split-paired-Ex [simp]
 $\langle ML \rangle$

lemma da-Skip: $A = \langle nrm=B, brk=\lambda l. UNIV \rangle \implies Env \vdash B \gg \langle Skip \rangle \gg A$
 $\langle proof \rangle$

lemma da-NewC: $A = \langle nrm=B, brk=\lambda l. UNIV \rangle \implies Env \vdash B \gg \langle NewC C \rangle \gg A$
 $\langle proof \rangle$

lemma da-Lit: $A = \langle nrm=B, brk=\lambda l. UNIV \rangle \implies Env \vdash B \gg \langle Lit v \rangle \gg A$
 $\langle proof \rangle$

lemma da-Super: $\llbracket This \in B; A = \langle nrm=B, brk=\lambda l. UNIV \rangle \rrbracket \implies Env \vdash B \gg \langle Super \rangle \gg A$
 $\langle proof \rangle$

lemma da-Init: $A = \langle nrm=B, brk=\lambda l. UNIV \rangle \implies Env \vdash B \gg \langle Init C \rangle \gg A$
 $\langle proof \rangle$

lemma assignsE-subseteq-assigns-ifs:

assumes boolEx: $E \vdash e :: \text{--PrimT Boolean (is ?Boolean e)}$

shows assignsE $e \subseteq$ assigns-if True $e \cap$ assigns-if False e (is ?Incl e)

$\langle proof \rangle$

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

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

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

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

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

lemma *da-monotone*:

assumes *da*: $Env \vdash B \gg t \gg A$ **and**

$B \subseteq B'$ **and**

da': $Env \vdash B' \gg t \gg A'$

shows $(nrm\ A \subseteq nrm\ A') \wedge (\forall l. (brk\ A\ l \subseteq brk\ A'\ l))$

⟨*proof*⟩

lemma *da-weaken*:

assumes *da*: $Env \vdash B \gg t \gg A$ **and** $B \subseteq B'$

shows $\exists A'. Env \vdash B' \gg t \gg A'$

⟨*proof*⟩

corollary *da-weakenE* [*consumes 2*]:

assumes $da: Env \vdash B \gg t \gg A$ **and**

$B': B \subseteq B'$ **and**

ex-mono: $\bigwedge A'. \llbracket Env \vdash B' \gg t \gg A'; nrm\ A \subseteq nrm\ A';$
 $\bigwedge l. brk\ A\ l \subseteq brk\ A'\ l \rrbracket \implies P$

shows *P*

⟨*proof*⟩

end

Chapter 13

WellForm

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

$wf\text{-}fdecl :: prog \Rightarrow pname \Rightarrow fdecl \Rightarrow bool$
where $wf\text{-}fdecl\ G\ P = (\lambda(fn,f). is\text{-}acc\text{-}type\ G\ P\ (type\ f))$

lemma $wf\text{-}fdecl\text{-}def2: \bigwedge fd. wf\text{-}fdecl\ G\ P\ fd = is\text{-}acc\text{-}type\ G\ P\ (type\ (snd\ fd))$
<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

$wf\text{-mhead} :: prog \Rightarrow pname \Rightarrow sig \Rightarrow mhead \Rightarrow bool$ **where**
 $wf\text{-mhead } G P = (\lambda sig mh. length (parTs sig) = length (pars mh) \wedge$
 $(\forall T \in set (parTs sig). is\text{-acc-type } G P T) \wedge$
 $is\text{-acc-type } G P (resTy mh) \wedge$
 $distinct (pars mh))$

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

$callee\text{-lcl} :: qname \Rightarrow sig \Rightarrow methd \Rightarrow lenv$ **where**
 $callee\text{-lcl } C sig m =$
 $(\lambda k. (case k of$
 $EName e$
 $\Rightarrow (case e of$
 $VNam v$
 $\Rightarrow (table\text{-of } (lcls (mbody m)) ((pars m) [\mapsto] (parTs sig))) v$
 $| Res \Rightarrow Some (resTy m)$
 $| This \Rightarrow if is\text{-static } m then None else Some (Class C)))$

definition

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

definition

$wf\text{-mdecl} :: prog \Rightarrow qname \Rightarrow mdecl \Rightarrow bool$ **where**
 $wf\text{-mdecl } G C =$
 $(\lambda (sig, m).$
 $wf\text{-mhead } G (pid C) sig (mhead m) \wedge$
 $unique (lcls (mbody m)) \wedge$
 $(\forall (vn, T) \in set (lcls (mbody m)). is\text{-acc-type } G (pid C) T) \wedge$
 $(\forall pn \in set (pars m). table\text{-of } (lcls (mbody m)) pn = None) \wedge$
 $jumpNestingOkS \{Ret\} (stmt (mbody m)) \wedge$
 $is\text{-class } G C \wedge$
 $(\downarrow prg = G, cls = C, lcl = callee\text{-lcl } C sig m) \vdash (stmt (mbody m)) :: \checkmark \wedge$
 $(\exists A. (\downarrow prg = G, cls = C, lcl = callee\text{-lcl } C sig m)$
 $\vdash parameters m \gg (stmt (mbody m)) \gg A$
 $\wedge Result \in nrm A))$

lemma *callee-lcl-VNam-simp* [simp]:

$callee\text{-lcl } C sig m (EName (VNam v))$
 $= (table\text{-of } (lcls (mbody m)) ((pars m) [\mapsto] (parTs sig))) v$

$\langle \text{proof} \rangle$

lemma *callee-lcl-Res-simp* [simp]:

$\text{callee-lcl } C \text{ sig } m \text{ (EName Res)} = \text{Some (resTy } m)$

$\langle \text{proof} \rangle$

lemma *callee-lcl-This-simp* [simp]:

$\text{callee-lcl } C \text{ sig } m \text{ (This)} = (\text{if is-static } m \text{ then None else Some (Class } C))$

$\langle \text{proof} \rangle$

lemma *callee-lcl-This-static-simp*:

$\text{is-static } m \implies \text{callee-lcl } C \text{ sig } m \text{ (This)} = \text{None}$

$\langle \text{proof} \rangle$

lemma *callee-lcl-This-not-static-simp*:

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

$\langle \text{proof} \rangle$

lemma *wf-mheadI*:

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

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

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

$\langle \text{proof} \rangle$

lemma *wf-mdeclI*: \llbracket

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

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

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

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

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

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

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

$\wedge \text{Result} \in \text{nrm } A)$

$\rrbracket \implies$

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

$\langle \text{proof} \rangle$

lemma *wf-mdeclE* [consumes 1]:

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

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

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

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

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

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

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

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

$\wedge \text{Result} \in \text{nrm } A)$

$\rrbracket \implies P$

$\rrbracket \implies P$

$\langle \text{proof} \rangle$

lemma *wf-mdeclD1*:

$$\begin{aligned} wf\text{-}mdecl\ G\ C\ (sig,m) \implies \\ & wf\text{-}mhead\ G\ (pid\ C)\ sig\ (mhead\ m) \wedge unique\ (lcls\ (mbody\ m)) \wedge \\ & (\forall pn \in set\ (pars\ m). table\text{-}of\ (lcls\ (mbody\ m))\ pn = None) \wedge \\ & (\forall (vn,T) \in set\ (lcls\ (mbody\ m)). is\text{-}acc\text{-}type\ G\ (pid\ C)\ T) \\ \langle proof \rangle \end{aligned}$$

lemma *wf-mdecl-bodyD*:

$$\begin{aligned} wf\text{-}mdecl\ G\ C\ (sig,m) \implies \\ & (\exists T. (\{prg=G,cls=C,lcl=callee\text{-}lcl\ C\ sig\ m\}) \vdash Body\ C\ (stmt\ (mbody\ m)) :: -T \wedge \\ & \quad G \vdash T \preceq (resTy\ m)) \\ \langle proof \rangle \end{aligned}$$

lemma *rT-is-acc-type*:

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

<proof>

well-formed interface declarations

A interface declaration is wellformed if:

- the interface hierarchy is wellstructured
- there is no class with the same name
- the method heads are wellformed and not static and have Public access
- the methods are uniquely named
- all superinterfaces are accessible
- the result type of a method overriding a method of Object widens to the result type of the overridden method. Shadowing static methods is forbidden.
- the result type of a method overriding a set of methods defined in the superinterfaces widens to each of the corresponding result types

definition

$$\begin{aligned} wf\text{-}idecl\ ::\ prog \Rightarrow idecl \Rightarrow bool \textbf{ where} \\ wf\text{-}idecl\ G = \\ & (\lambda(I,i). \\ & \quad ws\text{-}idecl\ G\ I\ (isuperIfs\ i) \wedge \\ & \quad \neg is\text{-}class\ G\ I \wedge \\ & \quad (\forall (sig,mh) \in set\ (imethods\ i). wf\text{-}mhead\ G\ (pid\ I)\ sig\ mh \wedge \\ & \quad \quad \neg is\text{-}static\ mh \wedge \\ & \quad \quad accmodi\ mh = Public) \wedge \\ & \quad unique\ (imethods\ i) \wedge \\ & \quad (\forall J \in set\ (isuperIfs\ i). is\text{-}acc\text{-}iface\ G\ (pid\ I)\ J) \wedge \\ & \quad (table\text{-}of\ (imethods\ i) \\ & \quad \quad hiding\ (methd\ G\ Object) \\ & \quad \quad under\ (\lambda new\ old. accmodi\ old \neq Private) \\ & \quad \quad entails\ (\lambda new\ old. G \vdash resTy\ new \preceq resTy\ old \wedge \\ & \quad \quad \quad is\text{-}static\ new = is\text{-}static\ old)) \wedge \end{aligned}$$

$$\begin{aligned} & (\text{set-option} \circ \text{table-of} \ (\text{imethods } i) \\ & \quad \text{hidings } \text{Un-tables}((\lambda J. (\text{imethds } G \ J)) \text{'set } (\text{isuperIfs } i)) \\ & \quad \text{entails } (\lambda \text{new old. } G \vdash \text{resTy } \text{new} \preceq \text{resTy } \text{old})) \end{aligned}$$

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

lemma *wf-idecl-hidings*:
 $\text{wf-idecl } G \ (I, i) \implies$
 $(\lambda s. \text{set-option} \ (\text{table-of} \ (\text{imethods } i) \ s))$
 $\text{hidings } \text{Un-tables} \ ((\lambda J. \text{imethds } G \ J) \ \text{'set} \ (\text{isuperIfs } i))$
 $\text{entails } \lambda \text{new old. } G \vdash \text{resTy } \text{new} \preceq \text{resTy } \text{old}$
 ⟨proof⟩

lemma *wf-idecl-hiding*:
 $\text{wf-idecl } G \ (I, i) \implies$
 $(\text{table-of} \ (\text{imethods } i)$
 $\quad \text{hiding} \ (\text{methd } G \ \text{Object})$
 $\quad \text{under} \ (\lambda \text{new old. } \text{accmodi } \text{old} \neq \text{Private})$
 $\quad \text{entails } (\lambda \text{new old. } G \vdash \text{resTy } \text{new} \preceq \text{resTy } \text{old} \wedge$
 $\quad \quad \text{is-static } \text{new} = \text{is-static } \text{old}))$
 ⟨proof⟩

lemma *wf-idecl-supD*:
 $\llbracket \text{wf-idecl } G \ (I, i); J \in \text{set} \ (\text{isuperIfs } i) \rrbracket$
 $\implies \text{is-acc-iface } G \ (\text{pid } I) \ J \wedge (J, I) \notin (\text{subint1 } G)^+$
 ⟨proof⟩

well-formed class declarations

A class declaration is wellformed if:

- there is no interface with the same name
- all superinterfaces are accessible and for all methods implementing an interface method the result type widens to the result type of the interface method, the method is not static and offers at least as much access (this actually means that the method has Public access, since all interface methods have public access)
- all field declarations are wellformed and the field names are unique
- all method declarations are wellformed and the method names are unique
- the initialization statement is welltyped
- the classhierarchy is wellstructured
- Unless the class is Object:
 - the superclass is accessible
 - for all methods overriding another method (of a superclass) the result type widens to the result type of the overridden method, the access modifier of the new method provides at least as much access as the overwritten one.

- for all methods hiding a method (of a superclass) the hidden method must be static and offer at least as much access rights. Remark: In contrast to the Java Language Specification we don't restrict the result types of the method (as in case of overriding), because there seems to be no reason, since there is no dynamic binding of static methods. (cf. 8.4.6.3 vs. 15.12.1). Stricly speaking the restrictions on the access rights aren't necessary to, since the static type and the access rights together determine which method is to be called statically. But if a class gains more then one static method with the same signature due to inheritance, it is confusing when the method selection depends on the access rights only: e.g. Class C declares static public method foo(). Class D is subclass of C and declares static method foo() with default package access. D.foo() ? if this call is in the same package as D then foo of class D is called, otherwise foo of class C.

definition

$entails :: ('a, 'b) table \Rightarrow ('b \Rightarrow bool) \Rightarrow bool$ (- entails - 20)
where $(t entails P) = (\forall k. \forall x \in t k: P x)$

lemma entailsD:

$\llbracket t entails P; t k = Some x \rrbracket \Longrightarrow P x$
 <proof>

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

<proof>

definition

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

lemma wf-cdeclE [consumes 1]:

$\llbracket wf-cdecl G (C, c);$

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

lemma *wf-cdecl-unique:*

$\text{wf-cdecl } G \ (C, c) \implies \text{unique } (\text{cfields } c) \wedge \text{unique } (\text{methods } c)$
 $\langle \text{proof} \rangle$

lemma *wf-cdecl-fdecl:*

$\llbracket \text{wf-cdecl } G \ (C, c); f \in \text{set } (\text{cfields } c) \rrbracket \implies \text{wf-fdecl } G \ (\text{pid } C) \ f$
 $\langle \text{proof} \rangle$

lemma *wf-cdecl-mdecl:*

$\llbracket \text{wf-cdecl } G \ (C, c); m \in \text{set } (\text{methods } c) \rrbracket \implies \text{wf-mdecl } G \ C \ m$
 $\langle \text{proof} \rangle$

lemma *wf-cdecl-impD:*

$\llbracket \text{wf-cdecl } G \ (C, c); I \in \text{set } (\text{superIfs } c) \rrbracket$
 $\implies \text{is-acc-iface } G \ (\text{pid } C) \ I \wedge$
 $(\forall s. \forall im \in \text{imethds } G \ I \ s.$
 $(\exists cm \in \text{methd } G \ C \ s: G \vdash \text{resTy } cm \preceq \text{resTy } im \wedge \neg \text{is-static } cm \wedge$
 $\quad \text{accmodi } im \leq \text{accmodi } cm))$
 $\langle \text{proof} \rangle$

lemma *wf-cdecl-supD:*

$\llbracket \text{wf-cdecl } G \ (C, c); C \neq \text{Object} \rrbracket \implies$
 $\text{is-acc-class } G \ (\text{pid } C) \ (\text{super } c) \wedge (\text{super } c, C) \notin (\text{subcls1 } G)^+ \wedge$
 $(\text{table-of } (\text{map } (\lambda (s,m). (s,C,m)) \ (\text{methods } c))$
 $\text{entails } (\lambda \text{new}. \forall \text{old sig.}$
 $(G, \text{sig} \vdash \text{new overrides}_S \text{old}$
 $\longrightarrow (G \vdash \text{resTy } \text{new} \preceq \text{resTy } \text{old} \wedge$

$$\begin{aligned}
& \text{accmodi old} \leq \text{accmodi new} \wedge \\
& \neg \text{is-static old}))) \wedge \\
& (G, \text{sig} \vdash \text{new hides old} \\
& \rightarrow (\text{accmodi old} \leq \text{accmodi new} \wedge \\
& \text{is-static old})))
\end{aligned}$$

$\langle \text{proof} \rangle$

lemma *wf-cdecl-overrides-SomeD*:

$$\begin{aligned}
& \llbracket \text{wf-cdecl } G (C, c); C \neq \text{Object}; \text{table-of (methods } c) \text{ sig} = \text{Some newM}; \\
& G, \text{sig} \vdash (C, \text{newM}) \text{ overrides}_S \text{ old} \\
& \rrbracket \implies G \vdash \text{resTy newM} \preceq \text{resTy old} \wedge \\
& \text{accmodi old} \leq \text{accmodi newM} \wedge \\
& \neg \text{is-static old}
\end{aligned}$$

$\langle \text{proof} \rangle$

lemma *wf-cdecl-hides-SomeD*:

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

$\langle \text{proof} \rangle$

lemma *wf-cdecl-wt-init*:

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

$\langle \text{proof} \rangle$

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

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

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

$\langle \text{proof} \rangle$

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

lemma *wf-prog-Object-mdecls*:
 $\text{wf-prog } G \implies (\forall m \in \text{set } \text{Object-mdecls}. \text{accmodi } m \neq \text{Package})$
 ⟨proof⟩

lemma *wf-prog-acc-superD*:
 $\llbracket \text{wf-prog } G; \text{class } G \ C = \text{Some } c; C \neq \text{Object} \rrbracket$
 $\implies \text{is-acc-class } G \ (\text{pid } C) \ (\text{super } c)$
 ⟨proof⟩

lemma *wf-ws-prog [elim!,simp]*: $\text{wf-prog } G \implies \text{ws-prog } G$
 ⟨proof⟩

lemma *class-Object [simp]*:
 $\text{wf-prog } G \implies$
 $\text{class } G \ \text{Object} = \text{Some } (\llbracket \text{access}=\text{Public}, \text{cfields}=[], \text{methods}=\text{Object-mdecls},$
 $\text{init}=\text{Skip}, \text{super}=\text{undefined}, \text{superIfs}=[] \rrbracket)$
 ⟨proof⟩

lemma *methd-Object[simp]*: $\text{wf-prog } G \implies \text{methd } G \ \text{Object} =$
 $\text{table-of } (\text{map } (\lambda(s, m). (s, \text{Object}, m)) \ \text{Object-mdecls})$
 ⟨proof⟩

lemma *wf-prog-Object-methd*:
 $\llbracket \text{wf-prog } G; \text{methd } G \ \text{Object } \text{sig} = \text{Some } m \rrbracket \implies \text{accmodi } m \neq \text{Package}$
 ⟨proof⟩

lemma *wf-prog-Object-is-public[intro]*:
 $\text{wf-prog } G \implies \text{is-public } G \ \text{Object}$
 ⟨proof⟩

lemma *class-SXcpt [simp]*:
 $\text{wf-prog } G \implies$
 $\text{class } G \ (\text{SXcpt } xn) = \text{Some } (\llbracket \text{access}=\text{Public}, \text{cfields}=[], \text{methods}=\text{SXcpt-mdecls},$
 $\text{init}=\text{Skip},$
 $\text{super}=\text{if } xn = \text{Throwable then } \text{Object}$
 $\text{else } \text{SXcpt } \text{Throwable},$
 $\text{superIfs}=[] \rrbracket)$
 ⟨proof⟩

lemma *wf-ObjectC [simp]*:
 $\text{wf-cdecl } G \ \text{ObjectC} = (\neg \text{is-iface } G \ \text{Object} \wedge \text{Ball } (\text{set } \text{Object-mdecls})$
 $(\text{wf-mdecl } G \ \text{Object}) \wedge \text{unique } \text{Object-mdecls})$
 ⟨proof⟩

lemma *Object-is-class* [simp,elim!]: $wf\text{-prog } G \implies is\text{-class } G \text{ Object}$
 ⟨proof⟩

lemma *Object-is-acc-class* [simp,elim!]: $wf\text{-prog } G \implies is\text{-acc-class } G \text{ S Object}$
 ⟨proof⟩

lemma *SXcpt-is-class* [simp,elim!]: $wf\text{-prog } G \implies is\text{-class } G \text{ (SXcpt } xn)$
 ⟨proof⟩

lemma *SXcpt-is-acc-class* [simp,elim!]:
 $wf\text{-prog } G \implies is\text{-acc-class } G \text{ S (SXcpt } xn)$
 ⟨proof⟩

lemma *fields-Object* [simp]: $wf\text{-prog } G \implies DeclConcepts.fields \ G \ \text{Object} = []$
 ⟨proof⟩

lemma *accfield-Object* [simp]:
 $wf\text{-prog } G \implies accfield \ G \ \text{S Object} = Map.empty$
 ⟨proof⟩

lemma *fields-Throwable* [simp]:
 $wf\text{-prog } G \implies DeclConcepts.fields \ G \ \text{(SXcpt Throwable)} = []$
 ⟨proof⟩

lemma *fields-SXcpt* [simp]: $wf\text{-prog } G \implies DeclConcepts.fields \ G \ \text{(SXcpt } xn) = []$
 ⟨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]:
 $wf\text{-prog } G \implies G \vdash SXcpt \ xn \preceq_C \ SXcpt \ \text{Throwable}$
 ⟨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*:
 $\llbracket table\text{-of } (DeclConcepts.fields \ G \ C) \ fn = Some \ f; G \vdash D \preceq_C \ C;$
 $is\text{-class } G \ D; wf\text{-prog } G \rrbracket$
 $\implies table\text{-of } (DeclConcepts.fields \ G \ D) \ fn = Some \ f$
 ⟨proof⟩

lemma *fields-is-type* [elim]:

$\llbracket \text{table-of } (DeclConcepts.fields\ G\ C)\ m = \text{Some } f; \text{ wf-prog } G; \text{ is-class } G\ C \rrbracket \implies$
 $\text{is-type } G\ (\text{type } f)$
 ⟨proof⟩

lemma *imethds-wf-mhead* [rule-format (no-asm)]:
 $\llbracket m \in \text{imethds } G\ I\ \text{sig}; \text{ wf-prog } G; \text{ is-iface } G\ I \rrbracket \implies$
 $\text{wf-mhead } G\ (\text{pid } (\text{decliface } m))\ \text{sig } (\text{mthd } m) \wedge$
 $\neg \text{is-static } m \wedge \text{accmodi } m = \text{Public}$
 ⟨proof⟩

lemma *methd-wf-mdecl*:
 $\llbracket \text{methd } G\ C\ \text{sig} = \text{Some } m; \text{ wf-prog } G; \text{ class } G\ C = \text{Some } y \rrbracket \implies$
 $G \vdash C \preceq_C (\text{declclass } m) \wedge \text{is-class } G\ (\text{declclass } m) \wedge$
 $\text{wf-mdecl } G\ (\text{declclass } m)\ (\text{sig}, (\text{mthd } m))$
 ⟨proof⟩

lemma *methd-rT-is-type*:
 $\llbracket \text{wf-prog } G; \text{methd } G\ C\ \text{sig} = \text{Some } m;$
 $\text{class } G\ C = \text{Some } y \rrbracket$
 $\implies \text{is-type } G\ (\text{resTy } m)$
 ⟨proof⟩

lemma *accmethd-rT-is-type*:
 $\llbracket \text{wf-prog } G; \text{accmethd } G\ S\ C\ \text{sig} = \text{Some } m;$
 $\text{class } G\ C = \text{Some } y \rrbracket$
 $\implies \text{is-type } G\ (\text{resTy } m)$
 ⟨proof⟩

lemma *methd-Object-SomeD*:
 $\llbracket \text{wf-prog } G; \text{methd } G\ \text{Object}\ \text{sig} = \text{Some } m \rrbracket$
 $\implies \text{declclass } m = \text{Object}$
 ⟨proof⟩

lemmas *iface-rec-induct'* = *iface-rec.induct* [of %x y z. P x y] for P

lemma *wf-imethdsD*:
 $\llbracket im \in \text{imethds } G\ I\ \text{sig}; \text{ wf-prog } G; \text{ is-iface } G\ I \rrbracket$
 $\implies \neg \text{is-static } im \wedge \text{accmodi } im = \text{Public}$
 ⟨proof⟩

lemma *wf-prog-hidesD*:
assumes *hides*: $G \vdash \text{new hides old}$ **and** *wf*: $\text{wf-prog } G$
shows
 $\text{accmodi } old \leq \text{accmodi } new \wedge$
 $\text{is-static } old$
 ⟨proof⟩

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 confrom result types as precondition. But with some efford we can guarantee the access modifier restriction for dynamic overriding, too. See lemma *wf-prog-dyn-override-prop*.

lemma *wf-prog-stat-overridesD*:

assumes *stat-override*: $G \vdash \text{new overrides}_S \text{ old}$ **and** *wf*: *wf-prog* G

shows

$G \vdash \text{resTy new} \preceq \text{resTy old} \wedge$
 $\text{accmodi old} \leq \text{accmodi new} \wedge$
 $\neg \text{is-static old}$

<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**
accommodi-new: $\text{accommodi new} = \text{Package}$ **and**
wf: *wf-prog* G
shows $\text{accommodi old} = \text{Package}$
 $\langle \text{proof} \rangle$

lemma *dyn-override-Package*:

assumes *dyn-override*: $G \vdash \text{new overrides old}$ **and**
accommodi-old: $\text{accommodi old} = \text{Package}$ **and**
accommodi-new: $\text{accommodi new} = \text{Package}$ **and**
wf: *wf-prog* G
shows $\text{pid}(\text{declclass old}) = \text{pid}(\text{declclass new})$
 $\langle \text{proof} \rangle$

lemma *dyn-override-Package-escape*:

assumes *dyn-override*: $G \vdash \text{new overrides old}$ **and**
accommodi-old: $\text{accommodi old} = \text{Package}$ **and**
outside-pack: $\text{pid}(\text{declclass old}) \neq \text{pid}(\text{declclass new})$ **and**
wf: *wf-prog* G
shows $\exists \text{inter}. G \vdash \text{new overrides inter} \wedge G \vdash \text{inter overrides old} \wedge$
 $\text{pid}(\text{declclass old}) = \text{pid}(\text{declclass inter}) \wedge$
 $\text{Protected} \leq \text{accommodi inter}$
 $\langle \text{proof} \rangle$

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
 $\longrightarrow (\forall c m. \text{class } G C = \text{Some } c \longrightarrow \text{methd } G C \text{ sig} = \text{Some } m$
 $\longrightarrow G \vdash C \preceq_C \text{declclass } m) \text{ (is } ?P G C)$
 $\langle \text{proof} \rangle$

lemma *declclass-methd-Object*:

$\llbracket \text{wf-prog } G; \text{methd } G \text{ Object sig} = \text{Some } m \rrbracket \Longrightarrow \text{declclass } m = \text{Object}$
 $\langle \text{proof} \rangle$

lemma *methd-declaredD*:

$\llbracket \text{wf-prog } G; \text{is-class } G C; \text{methd } G C \text{ sig} = \text{Some } m \rrbracket$
 $\Longrightarrow G \vdash (\text{mdecl}(\text{sig}, \text{methd } m)) \text{ declared-in } (\text{declclass } m)$
 $\langle \text{proof} \rangle$

lemma *methd-rec-Some-cases*:

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*) $G \vdash C \text{ inherits } (\text{method } \text{sig } m)$ **and** $\text{methd } G (\text{super } c) \text{ sig} = \text{Some } m$
 $\langle \text{proof} \rangle$

lemma *methd-member-of*:

assumes *wf*: *wf-prog G*

shows

$\llbracket \text{is-class } G \ C; \text{methd } G \ C \ \text{sig} = \text{Some } m \rrbracket \implies G \vdash \text{Methd } \text{sig } m \ \text{member-of } C$

(**is** *?Class C* \implies *?Method C* \implies *?MemberOf C*)

$\langle \text{proof} \rangle$

lemma *current-methd*:

$\llbracket \text{table-of } (\text{methods } c) \ \text{sig} = \text{Some } \text{new};$

ws-prog G; *class G C = Some c*; *C* \neq *Object*;

methd G (super c) sig = Some old \rrbracket

$\implies \text{methd } G \ C \ \text{sig} = \text{Some } (C, \text{new})$

$\langle \text{proof} \rangle$

lemma *wf-prog-staticD*:

assumes *wf*: *wf-prog G* **and**

clsC: *class G C = Some c* **and**

neq-C-Obj: *C* \neq *Object* **and**

old: *methd G (super c) sig = Some old* **and**

accommodi-old: *Protected* \leq *accommodi old* **and**

new: *table-of (methods c) sig = Some new*

shows *is-static new = is-static old*

$\langle \text{proof} \rangle$

lemma *inheritable-instance-methd*:

assumes *subclseq-C-D*: $G \vdash C \preceq_C D$ **and**

is-cls-D: *is-class G D* **and**

wf: *wf-prog G* **and**

old: *methd G D sig = Some old* **and**

accommodi-old: *Protected* \leq *accommodi old* **and**

not-static-old: \neg *is-static old*

shows

$\exists \text{new. methd } G \ C \ \text{sig} = \text{Some } \text{new} \wedge$

$(\text{new} = \text{old} \vee G, \text{sig} \vdash \text{new overrides}_S \ \text{old})$

(**is** $(\exists \text{new. } (?Constraint \ C \ \text{new} \ \text{old}))$)

$\langle \text{proof} \rangle$

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**

accommodi-old: *Protected* \leq *accommodi old* **and**

not-static-old: \neg *is-static old*

obtains (*Inheritance*) *methd G C sig = Some old*

| (*Overriding*) *new* **where** *methd G C sig = Some new* **and** $G, \text{sig} \vdash \text{new overrides}_S \ \text{old}$

$\langle \text{proof} \rangle$

lemma *inheritable-instance-methd-props*:

assumes *subclseq-C-D*: $G \vdash C \preceq_C D$ **and**

is-cls-D: *is-class G D* **and**

wf: *wf-prog G* **and**

old: *methd G D sig = Some old* **and**

accommodi-old: *Protected* \leq *accommodi old* **and**

not-static-old: \neg *is-static old*

shows

\exists *new*. *methd G C sig = Some new* \wedge

\neg *is-static new* \wedge $G \vdash \text{resTy new} \preceq \text{resTy old} \wedge \text{accmodi old} \leq \text{accmodi new}$

(**is** (\exists *new*. (?Constraint C new old)))

\langle proof \rangle

lemma *boxI'*: $x \in A \implies P x \implies \exists x \in A. P x$ \langle proof \rangle

lemma *ballE'*: $\forall x \in A. P x \implies (x \notin A \implies Q) \implies (P x \implies Q) \implies Q$ \langle proof \rangle

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*: $jm \in \text{imethds } G J \text{ sig}$

shows $\exists im \in \text{imethds } G I \text{ sig. is-static } im = \text{is-static } jm \wedge$

$\text{accmodi } im = \text{accmodi } jm \wedge$

$G \vdash \text{resTy } im \preceq \text{resTy } jm$

\langle proof \rangle

lemma *implmt1-methd*:

$\bigwedge \text{sig. } \llbracket G \vdash C \rightsquigarrow I I; \text{wf-prog } G; im \in \text{imethds } G I \text{ sig} \rrbracket \implies$

$\exists cm \in \text{methd } G C \text{ sig: } \neg \text{is-static } cm \wedge \neg \text{is-static } im \wedge$

$G \vdash \text{resTy } cm \preceq \text{resTy } im \wedge$

$\text{accmodi } im = \text{Public} \wedge \text{accmodi } cm = \text{Public}$

\langle proof \rangle

lemma *implmt-methd* [*rule-format (no-asm)*]:

$\llbracket \text{wf-prog } G; G \vdash C \rightsquigarrow I \rrbracket \implies \text{is-iface } G I \longrightarrow$

$(\forall im \in \text{imethds } G I \text{ sig.}$

$\exists cm \in \text{methd } G C \text{ sig: } \neg \text{is-static } cm \wedge \neg \text{is-static } im \wedge$

$G \vdash \text{resTy } cm \preceq \text{resTy } im \wedge$

$\text{accmodi } im = \text{Public} \wedge \text{accmodi } cm = \text{Public})$

\langle proof \rangle

lemma *mheadsD* [*rule-format (no-asm)*]:

$emh \in \text{mheads } G S t \text{ sig} \longrightarrow \text{wf-prog } G \longrightarrow$

$(\exists C D m. t = \text{ClassT } C \wedge \text{declrefT } emh = \text{ClassT } D \wedge$

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

$(\text{declclass } m = D) \wedge \text{mhead } (\text{mthd } m) = (\text{mhd } emh)) \vee$

$(\exists I. t = \text{IfaceT } I \wedge ((\exists im. im \in \text{accimethds } G (\text{pid } S) I \text{ sig} \wedge$

$\text{mthd } im = \text{mhd } emh)) \vee$

$(\exists m. G \vdash \text{Iface } I \text{ accessible-in } (\text{pid } S) \wedge \text{accmethd } G S \text{ Object sig} = \text{Some } m \wedge$

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

$\text{declrefT } emh = \text{ClassT } \text{Object} \wedge \text{mhead } (\text{mthd } m) = \text{mhd } emh))) \vee$

$(\exists T m. t = \text{Array}T T \wedge G \vdash \text{Array} T \text{ accessible-in } (pid S) \wedge$
 $\text{accmethd } G S \text{ Object sig} = \text{Some } m \wedge \text{accmodi } m \neq \text{Private} \wedge$
 $\text{declref}T \text{ emh} = \text{Class}T \text{ Object} \wedge \text{mhead } (mthd m) = \text{mhd } emh)$
 ⟨proof⟩

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**

$t = \text{Class}T C \text{ declref}T \text{ emh} = \text{Class}T D \text{ accmethd } G S C \text{ sig} = \text{Some } m$
 $\text{declclass } m = D \text{ mhead } (mthd m) = \text{mhd } emh$

| $(\text{Iface-methd}) I im$ **where** $t = \text{Iface}T I$

$im \in \text{accimethds } G (pid S) I \text{ sig}$ $mthd im = \text{mhd } emh$

| $(\text{Iface-Object-methd}) I m$ **where**

$t = \text{Iface}T I G \vdash \text{Iface } I \text{ accessible-in } (pid S)$

$\text{accmethd } G S \text{ Object sig} = \text{Some } m \text{ accmodi } m \neq \text{Private}$

$\text{declref}T \text{ emh} = \text{Class}T \text{ Object mhead } (mthd m) = \text{mhd } emh$

| $(\text{Array-Object-methd}) T m$ **where**

$t = \text{Array}T T G \vdash \text{Array} T \text{ accessible-in } (pid S)$

$\text{accmethd } G S \text{ Object sig} = \text{Some } m \text{ accmodi } m \neq \text{Private}$

$\text{declref}T \text{ emh} = \text{Class}T \text{ Object mhead } (mthd m) = \text{mhd } emh$

⟨proof⟩

lemma *declclassD*[rule-format]:

$\llbracket \text{wf-prog } G; \text{class } G C = \text{Some } c; \text{methd } G C \text{ sig} = \text{Some } m;$

$\text{class } G (\text{declclass } m) = \text{Some } d \rrbracket$

$\implies \text{table-of } (\text{methods } d) \text{ sig} = \text{Some } (mthd m)$

⟨proof⟩

lemma *dynmethd-Object*:

assumes $\text{stat}M: \text{methd } G \text{ Object sig} = \text{Some } \text{stat}M$ **and**

$\text{private}: \text{accmodi } \text{stat}M = \text{Private}$ **and**

$\text{is-cls-}C: \text{is-class } G C$ **and**

$\text{wf}: \text{wf-prog } G$

shows $\text{dynmethd } G \text{ Object } C \text{ sig} = \text{Some } \text{stat}M$

⟨proof⟩

lemma *wf-imethds-hiding-objmethdsD*:

assumes $\text{old}: \text{methd } G \text{ Object sig} = \text{Some } \text{old}$ **and**

$\text{is-if-}I: \text{is-iface } G I$ **and**

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

$\text{not-private}: \text{accmodi } \text{old} \neq \text{Private}$ **and**

$\text{new}: \text{new} \in \text{imethds } G I \text{ sig}$

shows $G \vdash \text{resTy } \text{new} \preceq \text{resTy } \text{old} \wedge \text{is-static } \text{new} = \text{is-static } \text{old}$ (**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 $\text{stat}T$ and a dynamic class $\text{dyn}C$. Between both of these types the widening relation holds $G \vdash \text{Class } \text{dyn}C \preceq \text{stat}T$. 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* \Rightarrow *ref-ty* \Rightarrow *qname* \Rightarrow *bool* \Rightarrow *bool*
 (*-*, *-* \vdash *- valid'-lookup'-cls'-for - [61,61,61,61] 60*)

where

G, *NullT* \vdash *dynC valid-lookup-cls-for static-membr* = *False*
 | *G*, *IfaceT I* \vdash *dynC valid-lookup-cls-for static-membr*
 = (*if static-membr*
 then dynC=Object
 else G \vdash *Class dynC* \preceq *Iface I*)
 | *G*, *ClassT C* \vdash *dynC valid-lookup-cls-for static-membr* = *G* \vdash *Class dynC* \preceq *Class C*
 | *G*, *ArrayT T* \vdash *dynC valid-lookup-cls-for static-membr* = (*dynC=Object*)

lemma *valid-lookup-cls-is-class*:

assumes *dynC*: *G*, *statT* \vdash *dynC valid-lookup-cls-for static-membr* **and**
ty-statT: *isrtype G statT* **and**
wf: *wf-prog G*
shows *is-class G dynC*
 <proof>

declare *split-paired-All* [*simp del*] *split-paired-Ex* [*simp del*]
 <ML>

lemma *dynamic-mheadsD*:

\llbracket *emh* \in *mheads G S statT sig*;
G, *statT* \vdash *dynC valid-lookup-cls-for (is-static emh)*;
isrtype G statT; *wf-prog G*
 $\rrbracket \Longrightarrow \exists m \in$ *dynlookup G statT dynC sig*:
is-static m=is-static emh \wedge *G* \vdash *resTy m* \preceq *resTy emh*
 <proof>

declare *split-paired-All* [*simp*] *split-paired-Ex* [*simp*]
 <ML>

lemma *methd-declclass*:

\llbracket *class G C = Some c*; *wf-prog G*; *methd G C sig = Some m* \rrbracket
 \Longrightarrow *methd G (declclass m) sig = Some m*
 <proof>

lemma *dynmethd-declclass*:

$$\begin{aligned} & \llbracket \text{dynmethd } G \text{ statC dynC sig} = \text{Some } m; \\ & \quad \text{wf-prog } G; \text{ is-class } G \text{ statC} \\ & \rrbracket \implies \text{methd } G \text{ (declclass } m) \text{ sig} = \text{Some } m \\ & \langle \text{proof} \rangle \end{aligned}$$

lemma *dynlookup-declC*:

$$\begin{aligned} & \llbracket \text{dynlookup } G \text{ statT dynC sig} = \text{Some } m; \text{ wf-prog } G; \\ & \quad \text{is-class } G \text{ dynC}; \text{ isrtype } G \text{ statT} \\ & \rrbracket \implies G \vdash \text{dynC} \preceq_C (\text{declclass } m) \wedge \text{is-class } G \text{ (declclass } m) \\ & \langle \text{proof} \rangle \end{aligned}$$

lemma *dynlookup-Array-declclassD* [simp]:

$$\begin{aligned} & \llbracket \text{dynlookup } G \text{ (ArrayT } T) \text{ Object sig} = \text{Some } dm; \text{ wf-prog } G \rrbracket \\ & \implies \text{declclass } dm = \text{Object} \\ & \langle \text{proof} \rangle \end{aligned}$$

declare *split-paired-All* [simp del] *split-paired-Ex* [simp del]
 <ML>

lemma *wt-is-type*: $E, dt \models v :: T \implies \text{wf-prog} (\text{prg } E) \longrightarrow$

$$\begin{aligned} & dt = \text{empty-dt} \longrightarrow (\text{case } T \text{ of} \\ & \quad \text{Inl } T \Rightarrow \text{is-type} (\text{prg } E) T \\ & \quad | \text{Inr } Ts \Rightarrow \text{Ball} (\text{set } Ts) (\text{is-type} (\text{prg } E))) \end{aligned}$$

<proof>

declare *split-paired-All* [simp] *split-paired-Ex* [simp]
 <ML>

lemma *ty-expr-is-type*:

$$\llbracket E \vdash e :: -T; \text{ wf-prog} (\text{prg } E) \rrbracket \implies \text{is-type} (\text{prg } E) T$$

<proof>

lemma *ty-var-is-type*:

$$\llbracket E \vdash v :: =T; \text{ wf-prog} (\text{prg } E) \rrbracket \implies \text{is-type} (\text{prg } E) T$$

<proof>

lemma *ty-exprs-is-type*:

$$\llbracket E \vdash es :: =Ts; \text{ wf-prog} (\text{prg } E) \rrbracket \implies \text{Ball} (\text{set } Ts) (\text{is-type} (\text{prg } E))$$

<proof>

lemma *static-mheadsD*:

$$\begin{aligned} & \llbracket \text{emh} \in \text{mheads } G \text{ S } t \text{ sig}; \text{ wf-prog } G; E \vdash e :: -\text{RefT } t; \text{ prg } E = G ; \\ & \quad \text{invmode} (\text{mhd } \text{emh}) e \neq \text{IntVir} \\ & \rrbracket \implies \exists m. ((\exists C. t = \text{ClassT } C \wedge \text{accmethd } G \text{ S } C \text{ sig} = \text{Some } m) \\ & \quad \vee (\forall C. t \neq \text{ClassT } C \wedge \text{accmethd } G \text{ S } \text{Object sig} = \text{Some } m)) \wedge \\ & \quad \text{declrefT } \text{emh} = \text{ClassT} (\text{declclass } m) \wedge \text{mhead} (\text{mthd } m) = (\text{mhd } \text{emh}) \\ & \langle \text{proof} \rangle \end{aligned}$$

lemma *wt-MethodI*:

$$\llbracket \text{methd } G \text{ C sig} = \text{Some } m; \text{ wf-prog } G; \rrbracket$$

$class\ G\ C = Some\ c \implies$
 $\exists T. (\text{prg}=G, \text{cls}=(\text{declclass } m),$
 $\quad \text{lcl}=\text{callee-lcl } (\text{declclass } m)\ \text{sig } (m\text{thd } m)) \Vdash \text{Methd } C\ \text{sig}::-T \wedge G \vdash T \preceq_{resTy} m$
 $\langle \text{proof} \rangle$

2 accessibility concerns

lemma *mheads-type-accessible*:

$\llbracket emh \in mheads\ G\ S\ T\ sig; wf\text{-prog } G \rrbracket$
 $\implies G \vdash RefT\ T\ \text{accessible-in } (pid\ S)$
 $\langle \text{proof} \rangle$

lemma *static-to-dynamic-accessible-from-aux*:

$\llbracket G \vdash m\ \text{of } C\ \text{accessible-from } accC; wf\text{-prog } G \rrbracket$
 $\implies G \vdash m\ \text{in } C\ \text{dyn-accessible-from } accC$
 $\langle \text{proof} \rangle$

lemma *static-to-dynamic-accessible-from*:

assumes *stat-acc*: $G \vdash m\ \text{of } statC\ \text{accessible-from } accC$ **and**
 $subclseq: G \vdash dynC \preceq_C\ statC$ **and**
 $wf: wf\text{-prog } G$
shows $G \vdash m\ \text{in } dynC\ \text{dyn-accessible-from } accC$
 $\langle \text{proof} \rangle$

lemma *static-to-dynamic-accessible-from-static*:

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

lemma *dynmethd-member-in*:

assumes $m: dynmethd\ G\ statC\ dynC\ sig = Some\ m$ **and**
 $iscls\text{-statC}: is\text{-class } G\ statC$ **and**
 $wf: wf\text{-prog } G$
shows $G \vdash Methd\ sig\ m\ \text{member-in } dynC$
 $\langle \text{proof} \rangle$

lemma *dynmethd-access-prop*:

assumes *statM*: $methd\ G\ statC\ sig = Some\ statM$ **and**
 $stat\text{-acc}: G \vdash Methd\ sig\ statM\ \text{of } statC\ \text{accessible-from } accC$ **and**
 $dynM: dynmethd\ G\ statC\ dynC\ sig = Some\ dynM$ **and**
 $wf: wf\text{-prog } G$
shows $G \vdash Methd\ sig\ dynM\ \text{in } dynC\ \text{dyn-accessible-from } accC$
 $\langle \text{proof} \rangle$

lemma *implmt-methd-access*:

fixes $accC::qname$
assumes *iface-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 $\exists dynM. methd\ G\ dynC\ sig = Some\ dynM \wedge$

$G \vdash \text{Methd sig dynM in dynC dyn-accessible-from accC}$
 ⟨proof⟩

corollary *implmt-dynimethd-access:*

fixes $\text{accC}::\text{qname}$

assumes *iface-methd*: $\text{imethds } G \ I \ \text{sig} \neq \{\}$ **and**

implements: $G \vdash \text{dynC} \rightsquigarrow I$ **and**

isif-I: *is-iface* $G \ I$ **and**

wf: *wf-prog* G

shows $\exists \text{ dynM}. \text{dynimethd } G \ I \ \text{dynC} \ \text{sig} = \text{Some } \text{dynM} \wedge$

$G \vdash \text{Methd sig dynM in dynC dyn-accessible-from accC}$

⟨proof⟩

lemma *dynlookup-access-prop:*

assumes *emh*: $\text{emh} \in \text{mheads } G \ \text{accC} \ \text{statT} \ \text{sig}$ **and**

dynM: $\text{dynlookup } G \ \text{statT} \ \text{dynC} \ \text{sig} = \text{Some } \text{dynM}$ **and**

dynC-prop: $G, \text{statT} \vdash \text{dynC} \ \text{valid-lookup-cls-for } \text{is-static } \text{emh}$ **and**

isT-statT: *isrtype* $G \ \text{statT}$ **and**

wf: *wf-prog* G

shows $G \vdash \text{Methd sig dynM in dynC dyn-accessible-from accC}$

⟨proof⟩

lemma *dynlookup-access:*

assumes *emh*: $\text{emh} \in \text{mheads } G \ \text{accC} \ \text{statT} \ \text{sig}$ **and**

dynC-prop: $G, \text{statT} \vdash \text{dynC} \ \text{valid-lookup-cls-for } (\text{is-static } \text{emh})$ **and**

isT-statT: *isrtype* $G \ \text{statT}$ **and**

wf: *wf-prog* G

shows $\exists \text{ dynM}. \text{dynlookup } G \ \text{statT} \ \text{dynC} \ \text{sig} = \text{Some } \text{dynM} \wedge$

$G \vdash \text{Methd sig dynM in dynC dyn-accessible-from accC}$

⟨proof⟩

lemma *stat-overrides-Package-old:*

assumes *stat-override*: $G \vdash \text{new overrides}_s \ \text{old}$ **and**

accmodi-new: $\text{accmodi } \text{new} = \text{Package}$ **and**

wf: *wf-prog* G

shows $\text{accmodi } \text{old} = \text{Package}$

⟨proof⟩

Properties of dynamic accessibility

lemma *dyn-accessible-Private:*

assumes *dyn-acc*: $G \vdash m \ \text{in } C \ \text{dyn-accessible-from } \text{accC}$ **and**

priv: $\text{accmodi } m = \text{Private}$

shows $\text{accC} = \text{declclass } m$

⟨proof⟩

dyn-accessible-Package only works with the *wf-prog* assumption. Without it, it is easy to leaf the Package!

lemma *dyn-accessible-Package:*

$\llbracket G \vdash m \ \text{in } C \ \text{dyn-accessible-from } \text{accC}; \ \text{accmodi } m = \text{Package};$

$\text{wf-prog } G \rrbracket$

$\implies \text{pid } \text{accC} = \text{pid } (\text{declclass } m)$

⟨proof⟩

For fields we don't need the wellformedness of the program, since there is no overriding

lemma *dyn-accessible-field-Package:*

assumes *dyn-acc*: $G \vdash f$ in C *dyn-accessible-from* *accC* **and**
 pack: *accmodi* $f = \text{Package}$ **and**
 field: *is-field* f
shows *pid* *accC* = *pid* (*declclass* f)
 ⟨*proof*⟩

dyn-accessible-instance-field-Protected only works for fields since methods can break the package bounds due to overriding

lemma *dyn-accessible-instance-field-Protected*:
assumes *dyn-acc*: $G \vdash f$ in C *dyn-accessible-from* *accC* **and**
 prot: *accmodi* $f = \text{Protected}$ **and**
 field: *is-field* f **and**
 instance-field: \neg *is-static* f **and**
 outside: *pid* (*declclass* f) \neq *pid* *accC*
shows $G \vdash C \preceq_C \text{accC}$
 ⟨*proof*⟩

lemma *dyn-accessible-static-field-Protected*:
assumes *dyn-acc*: $G \vdash f$ in C *dyn-accessible-from* *accC* **and**
 prot: *accmodi* $f = \text{Protected}$ **and**
 field: *is-field* f **and**
 static-field: *is-static* f **and**
 outside: *pid* (*declclass* f) \neq *pid* *accC*
shows $G \vdash \text{accC} \preceq_C \text{declclass } f \wedge G \vdash C \preceq_C \text{declclass } f$
 ⟨*proof*⟩

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* × (*qname*, *obj*) *table* $\sim =$ (*loc* + *qname*, *obj*) *table*)

objects

```
datatype obj-tag =
  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]:
```

$\llbracket \text{tag } \text{obj} = \text{Arr } T \ k \rrbracket \implies \text{the-Arr } (\text{Some } \text{obj}) = (T, k, \text{values } \text{obj})$
 $\langle \text{proof} \rangle$

definition

$\text{upd-obj} :: \text{vn} \Rightarrow \text{val} \Rightarrow \text{obj} \Rightarrow \text{obj}$
where $\text{upd-obj } n \ v = (\lambda \text{obj}. \text{obj } (\!| \text{values} := (\text{values } \text{obj})(n \mapsto v) \!|))$

lemma upd-obj-def2 [*simp*]:

$\text{upd-obj } n \ v \ \text{obj} = \text{obj } (\!| \text{values} := (\text{values } \text{obj})(n \mapsto v) \!|)$
 $\langle \text{proof} \rangle$

definition

$\text{obj-ty} :: \text{obj} \Rightarrow \text{ty}$ **where**
 $\text{obj-ty } \text{obj} = (\text{case } \text{tag } \text{obj} \ \text{of}$
 $\quad \text{CInst } C \Rightarrow \text{Class } C$
 $\quad | \ \text{Arr } T \ k \Rightarrow T.\[])$

lemma obj-ty-eq [*intro!*]: $\text{obj-ty } (\!| \text{tag} = \text{oi}, \text{values} = x \!|) = \text{obj-ty } (\!| \text{tag} = \text{oi}, \text{values} = y \!|)$
 $\langle \text{proof} \rangle$

lemma obj-ty-eq1 [*intro!, dest*]:

$\text{tag } \text{obj} = \text{tag } \text{obj}' \implies \text{obj-ty } \text{obj} = \text{obj-ty } \text{obj}'$
 $\langle \text{proof} \rangle$

lemma obj-ty-cong [*simp*]:

$\text{obj-ty } (\text{obj } (\!| \text{values} := \text{vs} \!|)) = \text{obj-ty } \text{obj}$
 $\langle \text{proof} \rangle$

lemma obj-ty-CInst [*simp*]:

$\text{obj-ty } (\!| \text{tag} = \text{CInst } C, \text{values} = \text{vs} \!|) = \text{Class } C$
 $\langle \text{proof} \rangle$

lemma obj-ty-CInst1 [*simp, intro!, dest*]:

$\llbracket \text{tag } \text{obj} = \text{CInst } C \rrbracket \implies \text{obj-ty } \text{obj} = \text{Class } C$
 $\langle \text{proof} \rangle$

lemma obj-ty-Arr [*simp*]:

$\text{obj-ty } (\!| \text{tag} = \text{Arr } T \ i, \text{values} = \text{vs} \!|) = T.\[]$
 $\langle \text{proof} \rangle$

lemma obj-ty-Arr1 [*simp, intro!, dest*]:

$\llbracket \text{tag } \text{obj} = \text{Arr } T \ i \rrbracket \implies \text{obj-ty } \text{obj} = T.\[]$
 $\langle \text{proof} \rangle$

lemma obj-ty-widenD :

$G \vdash \text{obj-ty } \text{obj} \preceq \text{RefT } t \implies (\exists C. \text{tag } \text{obj} = \text{CInst } C) \vee (\exists T \ k. \text{tag } \text{obj} = \text{Arr } T \ k)$
 $\langle \text{proof} \rangle$

definition


```

obj-class :: obj  $\Rightarrow$  qname where
obj-class obj = (case tag obj of
  CInst C  $\Rightarrow$  C
  | Arr T k  $\Rightarrow$  Object)

```

lemma *obj-class-CInst* [simp]: *obj-class* ($\{tag=CInst\ C, values=vs\}$) = *C*
 <proof>

lemma *obj-class-CInst1* [simp,intro!,dest]:
 tag obj = CInst C \implies *obj-class* obj = *C*
 <proof>

lemma *obj-class-Arr* [simp]: *obj-class* ($\{tag=Arr\ T\ k, values=vs\}$) = *Object*
 <proof>

lemma *obj-class-Arr1* [simp,intro!,dest]:
 tag obj = Arr T k \implies *obj-class* obj = *Object*
 <proof>

lemma *obj-ty-obj-class*: $G \vdash$ obj-ty obj \preceq Class statC = $G \vdash$ *obj-class* obj \preceq_C statC
 <proof>

object references

type-synonym *oref* = loc + qname — generalized object reference

syntax

```

Heap :: loc  $\Rightarrow$  oref
Stat :: qname  $\Rightarrow$  oref

```

translations

```

Heap => CONST Inl
Stat => CONST Inr
(type) oref <= (type) loc + qname

```

definition

```

fields-table :: prog  $\Rightarrow$  qname  $\Rightarrow$  (fspec  $\Rightarrow$  field  $\Rightarrow$  bool)  $\Rightarrow$  (fspec, ty) table where
fields-table G C P =
  map-option type  $\circ$  table-of (filter (case-prod P) (DeclConcepts.fields G C))

```

lemma *fields-table-SomeI*:
 \llbracket table-of (DeclConcepts.fields G C) $n =$ Some f ; $P\ n\ f$ \rrbracket
 \implies *fields-table* G C P $n =$ Some (type f)
 <proof>

lemma *fields-table-SomeD'*: *fields-table* G C P $fn =$ Some $T \implies$
 $\exists f. (fn, f) \in \text{set}(DeclConcepts.fields\ G\ C) \wedge \text{type}\ f = T$
 <proof>

lemma *fields-table-SomeD*:

$\llbracket \text{fields-table } G \ C \ P \ fn = \text{Some } T; \text{unique } (\text{DeclConcepts.fields } G \ C) \rrbracket \implies$
 $\exists f. \text{table-of } (\text{DeclConcepts.fields } G \ C) \ fn = \text{Some } f \wedge \text{type } f = T$
 ⟨proof⟩

definition

$\text{in-bounds} :: \text{int} \Rightarrow \text{int} \Rightarrow \text{bool} \ ((-/ \text{in}'\text{-bounds } -) [50, 51] 50)$
where $i \text{ in-bounds } k = (0 \leq i \wedge i < k)$

definition

$\text{arr-comps} :: 'a \Rightarrow \text{int} \Rightarrow \text{int} \Rightarrow 'a \text{ option}$
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

$\text{var-tys} :: \text{prog} \Rightarrow \text{obj-tag} \Rightarrow \text{oref} \Rightarrow (\text{vn}, \text{ty}) \text{ table}$ **where**
 $\text{var-tys } G \ oi \ r =$
 (case r of
 $\text{Heap } a \Rightarrow (\text{case } oi \text{ of}$
 $\text{CInst } C \Rightarrow \text{fields-table } G \ C \ (\lambda n \ f. \neg \text{static } f) \ (+) \ \text{Map.empty}$
 $| \text{Arr } T \ k \Rightarrow \text{Map.empty } \ (+) \ \text{arr-comps } T \ k)$
 $| \text{Stat } C \Rightarrow \text{fields-table } G \ C \ (\lambda fn \ f. \text{declclassf } fn = C \wedge \text{static } f)$
 $(+) \ \text{Map.empty})$

lemma *var-tys-Some-eq*:

$\text{var-tys } G \ oi \ r \ n = \text{Some } T$
 = (case r of
 $\text{Inl } a \Rightarrow (\text{case } oi \text{ of}$
 $\text{CInst } C \Rightarrow (\exists nt. n = \text{Inl } nt \wedge \text{fields-table } G \ C \ (\lambda n \ f. \neg \text{static } f) \ nt = \text{Some } T)$
 $| \text{Arr } t \ k \Rightarrow (\exists i. n = \text{Inr } i \wedge i \text{ in-bounds } k \wedge t = T))$
 $| \text{Inr } C \Rightarrow (\exists nt. n = \text{Inl } nt \wedge$
 $\text{fields-table } G \ C \ (\lambda fn \ f. \text{declclassf } fn = C \wedge \text{static } f) \ nt$
 $= \text{Some } T))$

⟨proof⟩

stores

type-synonym *globals* — global variables: heap and static variables

= (*oref* , *obj*) table

type-synonym *heap*

= (*loc* , *obj*) table

translations

(*type*) *globals* <= (*type*) (*oref* , *obj*) table

(*type*) *heap* <= (*type*) (*loc* , *obj*) table

datatype *st* =

st *globals* *locals*

2 access**definition**

$\text{globals} :: \text{st} \Rightarrow \text{globals}$
where $\text{globals} = \text{case-st } (\lambda g \ l. g)$

definition

$\text{locals} :: \text{st} \Rightarrow \text{locals}$

where $locals = case-st (\lambda g l. l)$

definition $heap :: st \Rightarrow heap$ **where**
 $heap\ s = globs\ s \circ Heap$

lemma $globs-def2$ [simp]: $globs (st\ g\ l) = g$
 ⟨proof⟩

lemma $locals-def2$ [simp]: $locals (st\ g\ l) = l$
 ⟨proof⟩

lemma $heap-def2$ [simp]: $heap\ s\ a = globs\ s (Heap\ a)$
 ⟨proof⟩

abbreviation $val-this :: st \Rightarrow val$
where $val-this\ s == the (locals\ s\ This)$

abbreviation $lookup-obj :: st \Rightarrow val \Rightarrow obj$
where $lookup-obj\ s\ a' == the (heap\ s (the-Addr\ a'))$

3 memory allocation

definition
 $new-Addr :: heap \Rightarrow loc\ option$ **where**
 $new-Addr\ h = (if (\forall a. h\ a \neq None) then None else Some (SOME\ a. h\ a = None))$

lemma $new-AddrD$: $new-Addr\ h = Some\ a \implies h\ a = None$
 ⟨proof⟩

lemma $new-AddrD2$: $new-Addr\ h = Some\ a \implies \forall b. h\ b \neq None \longrightarrow b \neq a$
 ⟨proof⟩

lemma $new-Addr-SomeI$: $h\ a = None \implies \exists b. new-Addr\ h = Some\ b \wedge h\ b = None$
 ⟨proof⟩

4 initialization

abbreviation $init-vals :: ('a, ty)\ table \Rightarrow ('a, val)\ table$
where $init-vals\ vs == map-option\ default-val \circ vs$

lemma $init-arr-comps-base$ [simp]: $init-vals (arr-comps\ T\ 0) = Map.empty$
 ⟨proof⟩

lemma $init-arr-comps-step$ [simp]:
 $0 < j \implies init-vals (arr-comps\ T\ j) =$
 $init-vals (arr-comps\ T\ (j - 1))(j - 1 \mapsto default-val\ T)$
 ⟨proof⟩

5 update

definition

$gupd :: oref \Rightarrow obj \Rightarrow st \Rightarrow st$ ($gupd'(-\mapsto-)$ [10, 10] 1000)
where $gupd\ r\ obj = case-st\ (\lambda g\ l.\ st\ (g(r\mapsto obj)))\ l$

definition

$lupd :: lname \Rightarrow val \Rightarrow st \Rightarrow st$ ($lupd'(-\mapsto-)$ [10, 10] 1000)
where $lupd\ vn\ v = case-st\ (\lambda g\ l.\ st\ g\ (l(vn\mapsto v)))$

definition

$upd-gobj :: oref \Rightarrow vn \Rightarrow val \Rightarrow st \Rightarrow st$
where $upd-gobj\ r\ n\ v = case-st\ (\lambda g\ l.\ st\ (chg-map\ (upd-obj\ n\ v)\ r\ g)\ l)$

definition

$set-locals :: locals \Rightarrow st \Rightarrow st$
where $set-locals\ l = case-st\ (\lambda g\ l'.\ st\ g\ l)$

definition

$init-obj :: prog \Rightarrow obj-tag \Rightarrow oref \Rightarrow st \Rightarrow st$
where $init-obj\ G\ oi\ r = gupd(r\mapsto(\{tag=oi, values=init-vals\ (var-tys\ G\ oi\ r)\}))$

abbreviation

$init-class-obj :: prog \Rightarrow qname \Rightarrow st \Rightarrow st$
where $init-class-obj\ G\ C == init-obj\ G\ undefined\ (Inr\ C)$

lemma $gupd-def2$ [simp]: $gupd(r\mapsto obj)\ (st\ g\ l) = st\ (g(r\mapsto obj))\ l$
 ⟨proof⟩

lemma $lupd-def2$ [simp]: $lupd(vn\mapsto v)\ (st\ g\ l) = st\ g\ (l(vn\mapsto v))$
 ⟨proof⟩

lemma $globs-gupd$ [simp]: $globs\ (gupd(r\mapsto obj)\ s) = globs\ s(r\mapsto obj)$
 ⟨proof⟩

lemma $globs-lupd$ [simp]: $globs\ (lupd(vn\mapsto v)\ s) = globs\ s$
 ⟨proof⟩

lemma $locals-gupd$ [simp]: $locals\ (gupd(r\mapsto obj)\ s) = locals\ s$
 ⟨proof⟩

lemma $locals-lupd$ [simp]: $locals\ (lupd(vn\mapsto v)\ s) = locals\ s(vn\mapsto v)$
 ⟨proof⟩

lemma $globs-upd-gobj-new$ [rule-format (no-asm), simp]:
 $globs\ s\ r = None \longrightarrow globs\ (upd-gobj\ r\ n\ v\ s) = globs\ s$
 ⟨proof⟩

lemma $globs-upd-gobj-upd$ [rule-format (no-asm), simp]:
 $globs\ s\ r = Some\ obj \longrightarrow globs\ (upd-gobj\ r\ n\ v\ s) = globs\ s(r\mapsto upd-obj\ n\ v\ obj)$
 ⟨proof⟩

lemma *locals-upd-gobj* [simp]: $locals (upd-gobj\ r\ n\ v\ s) = locals\ s$
 ⟨proof⟩

lemma *globs-init-obj* [simp]: $globs (init-obj\ G\ oi\ r\ s)\ t =$
 (if $t=r$ then $Some (\tag=oi,values=init-vals (var-tys\ G\ oi\ r))$) else $globs\ s\ t$)
 ⟨proof⟩

lemma *locals-init-obj* [simp]: $locals (init-obj\ G\ oi\ r\ s) = locals\ s$
 ⟨proof⟩

lemma *surjective-st* [simp]: $st (globs\ s) (locals\ s) = s$
 ⟨proof⟩

lemma *surjective-st-init-obj*:
 $st (globs (init-obj\ G\ oi\ r\ s)) (locals\ s) = init-obj\ G\ oi\ r\ s$
 ⟨proof⟩

lemma *heap-heap-upd* [simp]:
 $heap (st (g(Inl\ a\mapsto\ obj)))\ l = heap (st\ g\ l)(a\mapsto\ obj)$
 ⟨proof⟩

lemma *heap-stat-upd* [simp]: $heap (st (g(Inr\ C\mapsto\ obj)))\ l = heap (st\ g\ l)$
 ⟨proof⟩

lemma *heap-local-upd* [simp]: $heap (st\ g (l(vn\mapsto\ v))) = heap (st\ g\ l)$
 ⟨proof⟩

lemma *heap-gupd-Heap* [simp]: $heap (gupd(Heap\ a\mapsto\ obj)\ s) = heap\ s(a\mapsto\ obj)$
 ⟨proof⟩

lemma *heap-gupd-Stat* [simp]: $heap (gupd(Stat\ C\mapsto\ obj)\ s) = heap\ s$
 ⟨proof⟩

lemma *heap-lupd* [simp]: $heap (lupd(vn\mapsto\ v)\ s) = heap\ s$
 ⟨proof⟩

lemma *heap-upd-gobj-Stat* [simp]: $heap (upd-gobj (Stat\ C)\ n\ v\ s) = heap\ s$
 ⟨proof⟩

lemma *set-locals-def2* [simp]: $set-locals\ l (st\ g\ l') = st\ g\ l$
 ⟨proof⟩

lemma *set-locals-id* [simp]: $set-locals (locals\ s)\ s = s$
 ⟨proof⟩

lemma *set-set-locals* [simp]: $set-locals\ l (set-locals\ l'\ s) = set-locals\ l\ s$

<proof>

lemma *locals-set-locals* [simp]: *locals (set-locals l s) = l*
<proof>

lemma *globs-set-locals* [simp]: *globs (set-locals l s) = globs s*
<proof>

lemma *heap-set-locals* [simp]: *heap (set-locals l s) = heap s*
<proof>

abrupt completion

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* \Longrightarrow *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* \Longrightarrow *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* == *abrupt-if* *c* (*Some* (*Xcpt* (*Std xn*)))

abbreviation *np* :: *val* \Rightarrow *abopt* \Rightarrow *abopt*
where *np* *v* == *raise-if* (*v* = *Null*) *NullPointer*

abbreviation *check-neg* :: *val* \Rightarrow *abopt* \Rightarrow *abopt*
where *check-neg* *i'* == *raise-if* (*the-Intg* *i'* < 0) *NegArrSize*

abbreviation *error-if* :: *bool* \Rightarrow *error* \Rightarrow *abopt* \Rightarrow *abopt*
where *error-if* *c e* == *abrupt-if* *c* (*Some* (*Error* *e*))

lemma *raise-if-None* [*simp*]: (*raise-if* *c x y* = *None*) = (\neg *c* \wedge *y* = *None*)
 <proof>
declare *raise-if-None* [*THEN iffD1, dest!*]

lemma *if-raise-if-None* [*simp*]:
 ((*if* *b* *then* *y* *else* *raise-if* *c x y*) = *None*) = ((*c* \longrightarrow *b*) \wedge *y* = *None*)
 <proof>

lemma *raise-if-SomeD* [*dest!*]:
raise-if *c x y* = *Some* *z* \Longrightarrow *c* \wedge *z* = (*Xcpt* (*Std x*)) \wedge *y* = *None* \vee (*y* = *Some* *z*)
 <proof>

lemma *error-if-None* [*simp*]: (*error-if* *c e y* = *None*) = (\neg *c* \wedge *y* = *None*)
 <proof>
declare *error-if-None* [*THEN iffD1, dest!*]

lemma *if-error-if-None* [*simp*]:
 ((*if* *b* *then* *y* *else* *error-if* *c e y*) = *None*) = ((*c* \longrightarrow *b*) \wedge *y* = *None*)
 <proof>

lemma *error-if-SomeD* [*dest!*]:
error-if *c e y* = *Some* *z* \Longrightarrow *c* \wedge *z* = (*Error* *e*) \wedge *y* = *None* \vee (*y* = *Some* *z*)
 <proof>

definition
absorb :: *jump* \Rightarrow *abopt* \Rightarrow *abopt*
where *absorb* *j a* = (*if* *a* = *Some* (*Jump* *j*) *then* *None* *else* *a*)

lemma *absorb-SomeD* [*dest!*]: *absorb* *j a* = *Some* *x* \Longrightarrow *a* = *Some* *x*
 <proof>

lemma *absorb-same* [*simp*]: *absorb* *j* (*Some* (*Jump* *j*)) = *None*
 <proof>

lemma *absorb-other* [*simp*]: *a* \neq *Some* (*Jump* *j*) \Longrightarrow *absorb* *j a* = *a*
 <proof>

lemma *absorb-Some-NoneD*: $\text{absorb } j \text{ (Some } abr) = \text{None} \implies abr = \text{Jump } j$
 ⟨*proof*⟩

lemma *absorb-Some-JumpD*: $\text{absorb } j \text{ } s = \text{Some (Jump } j') \implies j' \neq j$
 ⟨*proof*⟩

full program state

type-synonym

$state = \text{abopt} \times st$ — state including abruption 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 $Norm \text{ } s == (\text{None}, s)$

abbreviation (*input*)

$abrupt :: state \Rightarrow \text{abopt}$

where $abrupt == fst$

abbreviation (*input*)

$store :: state \Rightarrow st$

where $store == snd$

lemma *single-stateE*: $\forall Z. Z = (s::state) \implies \text{False}$
 ⟨*proof*⟩

lemma *state-not-single*: $All ((=) (x::state)) \implies R$
 ⟨*proof*⟩

definition

$normal :: state \Rightarrow bool$

where $normal = (\lambda s. abrupt \text{ } s = \text{None})$

lemma *normal-def2* [*simp*]: $normal \text{ } s = (abrupt \text{ } s = \text{None})$
 ⟨*proof*⟩

definition

$heap\text{-free} :: nat \Rightarrow state \Rightarrow bool$

where $heap\text{-free} \text{ } n = (\lambda s. atleast\text{-free} (heap (store \text{ } s)) \text{ } n)$

lemma *heap-free-def2* [*simp*]: $heap\text{-free} \text{ } n \text{ } s = atleast\text{-free} (heap (store \text{ } s)) \text{ } n$
 ⟨*proof*⟩

6 update

definition

$abupd :: (\text{abopt} \Rightarrow \text{abopt}) \Rightarrow state \Rightarrow state$

where $abupd \text{ } f = map\text{-prod } f \text{ } id$

definition

$supd :: (st \Rightarrow st) \Rightarrow state \Rightarrow state$
where $supd = map\text{-}prod\ id$

lemma $abupd\text{-}def2$ [simp]: $abupd\ f\ (x,s) = (f\ x,s)$
 $\langle proof \rangle$

lemma $abupd\text{-}abrupt\text{-}if\text{-}False$ [simp]: $\bigwedge s. abupd\ (abrupt\text{-}if\ False\ xo)\ s = s$
 $\langle proof \rangle$

lemma $supd\text{-}def2$ [simp]: $supd\ f\ (x,s) = (x,f\ s)$
 $\langle proof \rangle$

lemma $supd\text{-}lupd$ [simp]:
 $\bigwedge s. supd\ (lupd\ vn\ v)\ s = (abrupt\ s, lupd\ vn\ v\ (store\ s))$
 $\langle proof \rangle$

lemma $supd\text{-}gupd$ [simp]:
 $\bigwedge s. supd\ (gupd\ r\ obj)\ s = (abrupt\ s, gupd\ r\ obj\ (store\ s))$
 $\langle proof \rangle$

lemma $supd\text{-}init\text{-}obj$ [simp]:
 $supd\ (init\text{-}obj\ G\ oi\ r)\ s = (abrupt\ s, init\text{-}obj\ G\ oi\ r\ (store\ s))$
 $\langle proof \rangle$

lemma $abupd\text{-}store\text{-}invariant$ [simp]: $store\ (abupd\ f\ s) = store\ s$
 $\langle proof \rangle$

lemma $supd\text{-}abrupt\text{-}invariant$ [simp]: $abrupt\ (supd\ f\ s) = abrupt\ s$
 $\langle proof \rangle$

abbreviation $set\text{-}lvars :: locals \Rightarrow state \Rightarrow state$
where $set\text{-}lvars\ l == supd\ (set\text{-}locals\ l)$

abbreviation $restore\text{-}lvars :: state \Rightarrow state \Rightarrow state$
where $restore\text{-}lvars\ s'\ s == set\text{-}lvars\ (locals\ (store\ s'))\ s$

lemma $set\text{-}set\text{-}lvars$ [simp]: $\bigwedge s. set\text{-}lvars\ l\ (set\text{-}lvars\ l'\ s) = set\text{-}lvars\ l\ s$
 $\langle proof \rangle$

lemma $set\text{-}lvars\text{-}id$ [simp]: $\bigwedge s. set\text{-}lvars\ (locals\ (store\ s))\ s = s$
 $\langle proof \rangle$

initialisation test**definition**

$inited :: qname \Rightarrow globs \Rightarrow bool$
where $inited\ C\ g = (g\ (Stat\ C) \neq None)$

definition

initd :: *qname* \Rightarrow *state* \Rightarrow *bool*
where *initd* *C* = *initd* *C* \circ *globs* \circ *store*

lemma *not-initd-empty* [*simp*]: \neg *initd* *C* *Map.empty*
 ⟨*proof*⟩

lemma *initd-gupdate* [*simp*]: *initd* *C* (*g*(*r* \mapsto *obj*)) = (*initd* *C* *g* \vee *r* = *Stat* *C*)
 ⟨*proof*⟩

lemma *initd-init-class-obj* [*intro!*]: *initd* *C* (*globs* (*init-class-obj* *G* *C* *s*))
 ⟨*proof*⟩

lemma *not-initdD*: \neg *initd* *C* *g* \implies *g* (*Stat* *C*) = *None*
 ⟨*proof*⟩

lemma *initdD*: *initd* *C* *g* \implies \exists *obj*. *g* (*Stat* *C*) = *Some obj*
 ⟨*proof*⟩

lemma *initd-def2* [*simp*]: *initd* *C* *s* = *initd* *C* (*globs* (*store* *s*))
 ⟨*proof*⟩

error-free

definition

error-free :: *state* \Rightarrow *bool*
where *error-free* *s* = (\neg (\exists *err*. *abrupt* *s* = *Some* (*Error* *err*)))

lemma *error-free-Norm* [*simp,intro*]: *error-free* (*Norm* *s*)
 ⟨*proof*⟩

lemma *error-free-normal* [*simp,intro*]: *normal* *s* \implies *error-free* *s*
 ⟨*proof*⟩

lemma *error-free-Xcpt* [*simp*]: *error-free* (*Some* (*Xcpt* *x*),*s*)
 ⟨*proof*⟩

lemma *error-free-Jump* [*simp,intro*]: *error-free* (*Some* (*Jump* *j*),*s*)
 ⟨*proof*⟩

lemma *error-free-Error* [*simp*]: *error-free* (*Some* (*Error* *e*),*s*) = *False*
 ⟨*proof*⟩

lemma *error-free-Some* [*simp,intro*]:
 \neg (\exists *err*. *x*=*Error* *err*) \implies *error-free* ((*Some* *x*),*s*)
 ⟨*proof*⟩

lemma *error-free-abupd-absorb* [*simp,intro*]:
 $error\text{-}free\ s \implies error\text{-}free\ (abupd\ (absorb\ j)\ s)$
 ⟨*proof*⟩

lemma *error-free-absorb* [*simp,intro*]:
 $error\text{-}free\ (a,s) \implies error\text{-}free\ (absorb\ j\ a,\ s)$
 ⟨*proof*⟩

lemma *error-free-abrupt-if* [*simp,intro*]:
 $\llbracket error\text{-}free\ s; \neg (\exists\ err.\ x=Error\ err) \rrbracket$
 $\implies error\text{-}free\ (abupd\ (abrupt\text{-}if\ p\ (Some\ x))\ s)$
 ⟨*proof*⟩

lemma *error-free-abrupt-if1* [*simp,intro*]:
 $\llbracket error\text{-}free\ (a,s); \neg (\exists\ err.\ x=Error\ err) \rrbracket$
 $\implies error\text{-}free\ (abrupt\text{-}if\ p\ (Some\ x)\ a,\ s)$
 ⟨*proof*⟩

lemma *error-free-abrupt-if-Xcpt* [*simp,intro*]:
 $error\text{-}free\ s$
 $\implies error\text{-}free\ (abupd\ (abrupt\text{-}if\ p\ (Some\ (Xcpt\ x)))\ s)$
 ⟨*proof*⟩

lemma *error-free-abrupt-if-Xcpt1* [*simp,intro*]:
 $error\text{-}free\ (a,s)$
 $\implies error\text{-}free\ (abrupt\text{-}if\ p\ (Some\ (Xcpt\ x))\ a,\ s)$
 ⟨*proof*⟩

lemma *error-free-abrupt-if-Jump* [*simp,intro*]:
 $error\text{-}free\ s$
 $\implies error\text{-}free\ (abupd\ (abrupt\text{-}if\ p\ (Some\ (Jump\ j)))\ s)$
 ⟨*proof*⟩

lemma *error-free-abrupt-if-Jump1* [*simp,intro*]:
 $error\text{-}free\ (a,s)$
 $\implies error\text{-}free\ (abrupt\text{-}if\ p\ (Some\ (Jump\ j))\ a,\ s)$
 ⟨*proof*⟩

lemma *error-free-raise-if* [*simp,intro*]:
 $error\text{-}free\ s \implies error\text{-}free\ (abupd\ (raise\text{-}if\ p\ x)\ s)$
 ⟨*proof*⟩

lemma *error-free-raise-if1* [*simp,intro*]:
 $error\text{-}free\ (a,s) \implies error\text{-}free\ ((raise\text{-}if\ p\ x)\ a,\ s)$
 ⟨*proof*⟩

lemma *error-free-supd* [*simp,intro*]:

$error\text{-}free\ s \implies error\text{-}free\ (supd\ f\ s)$
 $\langle proof \rangle$

lemma *error-free-supd1* [*simp,intro*]:
 $error\text{-}free\ (a,s) \implies error\text{-}free\ (a,f\ s)$
 $\langle proof \rangle$

lemma *error-free-set-lvars* [*simp,intro*]:
 $error\text{-}free\ s \implies error\text{-}free\ ((set\text{-}lvars\ l)\ s)$
 $\langle proof \rangle$

lemma *error-free-set-locals* [*simp,intro*]:
 $error\text{-}free\ (x, s)$
 $\implies error\text{-}free\ (x, set\text{-}locals\ l\ s')$
 $\langle proof \rangle$

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 $vals = (val, vvar, val\ list) \text{ sum}3$

translations

$(type) vvar \leq (type) val \times (val \Rightarrow state \Rightarrow state)$

$(type) vals \leq (type) (val, vvar, val\ list) \text{ sum}3$

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* (\diamond)
where $\diamond == \text{In1 Unit}$

abbreviation (*input*)

val-inj-vals ($[-]_e$ 1000)
where $[e]_e == \text{In1 } e$

abbreviation (*input*)

var-inj-vals ($[-]_v$ 1000)
where $[v]_v == \text{In2 } v$

abbreviation (*input*)

lst-inj-vals ($[-]_l$ 1000)
where $[es]_l == \text{In3 } es$

definition *undefined3* :: ('a1 + 'ar, 'b, 'c) *sum3* \Rightarrow *vals* **where**

undefined3 = *case-sum3* (*In1* \circ *case-sum* ($\lambda x. \text{undefined}$) ($\lambda x. \text{Unit}$))
 $(\lambda x. \text{In2 } \text{undefined}) (\lambda x. \text{In3 } \text{undefined})$

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* ($[-, + - \text{fits } [61, 61, 61, 61] 60$)
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]:

$G, s \vdash \text{Addr } a \text{ fits } \text{RefT } t = G \vdash \text{obj-ty } (\text{the } (\text{heap } s \ a)) \preceq \text{RefT } t$
 ⟨proof⟩

lemma fitsD: $\bigwedge X. G, s \vdash a' \text{ fits } T \implies (\exists pt. T = \text{PrimT } pt) \vee$

$(\exists t. T = \text{RefT } t) \wedge a' = \text{Null} \vee$

$(\exists t. T = \text{RefT } t) \wedge a' \neq \text{Null} \wedge G \vdash \text{obj-ty } (\text{lookup-obj } s \ a') \preceq T$

⟨proof⟩

definition

$\text{catch} :: \text{prog} \Rightarrow \text{state} \Rightarrow \text{qname} \Rightarrow \text{bool } (-, \vdash \text{catch } -[61,61,61]60)$ **where**

$G, s \vdash \text{catch } C = (\exists xc. \text{abrupt } s = \text{Some } (Xcpt \ xc) \wedge$

$G, \text{store } s \vdash \text{Addr } (\text{the-Loc } xc) \text{ fits } \text{Class } C)$

lemma catch-Norm [simp]: $\neg G, \text{Norm } s \vdash \text{catch } tn$

⟨proof⟩

lemma catch-XcptLoc [simp]:

$G, (\text{Some } (Xcpt \ (\text{Loc } a)), s) \vdash \text{catch } C = G, s \vdash \text{Addr } a \text{ fits } \text{Class } C$

⟨proof⟩

lemma catch-Jump [simp]: $\neg G, (\text{Some } (\text{Jump } j), s) \vdash \text{catch } tn$

⟨proof⟩

lemma catch-Error [simp]: $\neg G, (\text{Some } (\text{Error } e), s) \vdash \text{catch } tn$

⟨proof⟩

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 [simp]:

$\text{new-xcpt-var } vn \ (x, s) =$

$\text{Norm } (\text{lupd}(\text{VName } vn \mapsto \text{Addr } (\text{the-Loc } (\text{the-Xcpt } (\text{the } x)))) s)$

⟨proof⟩

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 [simp]:

$f \ v \ (\text{Norm } s) = \text{Norm } s' \implies \text{assign } f \ v \ (\text{Norm } s) = \text{Norm } s'$

⟨proof⟩

lemma *assign-Norm-Some* [simp]:
 $\llbracket \text{abrupt } (f \ v \ (\text{Norm } s)) = \text{Some } y \rrbracket$
 $\implies \text{assign } f \ v \ (\text{Norm } s) = (\text{Some } y, s)$
 ⟨proof⟩

lemma *assign-Some* [simp]:
 $\text{assign } f \ v \ (\text{Some } x, s) = (\text{Some } x, s)$
 ⟨proof⟩

lemma *assign-Some1* [simp]: $\neg \text{normal } s \implies \text{assign } f \ v \ s = s$
 ⟨proof⟩

lemma *assign-supd* [simp]:
 $\text{assign } (\lambda v. \text{supd } (f \ v)) \ v \ (x, s)$
 $= (x, \text{if } x = \text{None} \text{ then } f \ v \ s \ \text{else } s)$
 ⟨proof⟩

lemma *assign-raise-if* [simp]:
 $\text{assign } (\lambda v \ (x, s). \ (\text{raise-if } (b \ s \ v) \ \text{xcpt } x, \ f \ v \ s)) \ v \ (x, s) =$
 $(\text{raise-if } (b \ s \ v) \ \text{xcpt } x, \ \text{if } x = \text{None} \wedge \neg b \ s \ v \ \text{then } f \ v \ s \ \text{else } s)$
 ⟨proof⟩

definition

init-comp-ty :: $ty \Rightarrow \text{stmt}$
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]: $\text{init-comp-ty } (\text{PrimT } pt) = \text{Skip}$
 ⟨proof⟩

definition

invocation-class :: $\text{inv-mode} \Rightarrow \text{st} \Rightarrow \text{val} \Rightarrow \text{ref-ty} \Rightarrow \text{qname}$ **where**
invocation-class $m \ s \ a' \ \text{statT} =$
 (case m of
 $\text{Static} \Rightarrow \text{if } (\exists \text{statC}. \ \text{statT} = \text{ClassT } \text{statC})$
 then $\text{the-Class } (\text{RefT } \text{statT})$
 else Object
 $\text{SuperM} \Rightarrow \text{the-Class } (\text{RefT } \text{statT})$
 $\text{IntVir} \Rightarrow \text{obj-class } (\text{lookup-obj } s \ a')$)

definition

invocation-declclass :: $\text{prog} \Rightarrow \text{inv-mode} \Rightarrow \text{st} \Rightarrow \text{val} \Rightarrow \text{ref-ty} \Rightarrow \text{sig} \Rightarrow \text{qname}$ **where**
invocation-declclass $G \ m \ s \ a' \ \text{statT} \ \text{sig} =$
 $\text{declclass } (\text{the } (\text{dynlookup } G \ \text{statT}$
 $(\text{invocation-class } m \ s \ a' \ \text{statT})$
 $\text{sig}))$

lemma *invocation-class-IntVir* [simp]:

invocation-class IntVir $s\ a'\ statT = obj\text{-}class\ (lookup\text{-}obj\ s\ a')$
 ⟨proof⟩

lemma *dynclass-SuperM* [simp]:
invocation-class SuperM $s\ a'\ statT = the\text{-}Class\ (RefT\ statT)$
 ⟨proof⟩

lemma *invocation-class-Static* [simp]:
invocation-class Static $s\ a'\ statT = (if\ (\exists\ statC.\ statT = ClassT\ statC)$
 $then\ the\text{-}Class\ (RefT\ statT)$
 $else\ Object)$
 ⟨proof⟩

definition

init-lvars :: $prog \Rightarrow qname \Rightarrow sig \Rightarrow inv\text{-}mode \Rightarrow val \Rightarrow val\ list \Rightarrow state \Rightarrow state$

where

init-lvars $G\ C\ sig\ mode\ a'\ pvs =$
 $(\lambda(x,s).$
 $let\ m = methd\ (the\ (methd\ G\ C\ sig));$
 $l = \lambda\ k.$
 $(case\ k\ of$
 $ENam\ e$
 $\Rightarrow (case\ e\ of$
 $VNam\ v \Rightarrow (Map.empty\ ((pars\ m)[\mapsto]pvs))\ v$
 $| Res \Rightarrow None)$
 $| This$
 $\Rightarrow (if\ mode=Static\ then\ None\ else\ Some\ a^'))$
 $in\ set\text{-}lvars\ l\ (if\ mode = Static\ then\ x\ else\ np\ a'\ x,s))$

lemma *init-lvars-def2*: — better suited for simplification

init-lvars $G\ C\ sig\ mode\ a'\ pvs\ (x,s) =$
set-lvars
 $(\lambda\ k.$
 $(case\ k\ of$
 $ENam\ e$
 $\Rightarrow (case\ e\ of$
 $VNam\ v$
 $\Rightarrow (Map.empty\ ((pars\ (methd\ (the\ (methd\ G\ C\ sig))))[\mapsto]pvs))\ v$
 $| Res \Rightarrow None)$
 $| This$
 $\Rightarrow (if\ mode=Static\ then\ None\ else\ Some\ a^'))$
 $(if\ mode = Static\ then\ x\ else\ np\ a'\ x,s)$
 ⟨proof⟩

definition

body :: $prog \Rightarrow qname \Rightarrow sig \Rightarrow expr$ **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

$lvar :: lname \Rightarrow st \Rightarrow vvar$
where $lvar\ vn\ s = (the\ (locals\ s\ vn), \lambda v. supd\ (lupd(vn \mapsto v)))$

definition

$fvar :: qname \Rightarrow bool \Rightarrow vname \Rightarrow val \Rightarrow state \Rightarrow vvar \times state$ **where**
 $fvar\ C\ stat\ fn\ a'\ s =$
 $(let\ (oref,xf) = if\ stat\ then\ (Stat\ C, id)$
 $\quad\quad\quad else\ (Heap\ (the-Addr\ a'), np\ a');$
 $\quad n = Inl\ (fn, C);$
 $\quad f = (\lambda v. supd\ (upd-gobj\ oref\ n\ v))$
 $in\ ((the\ (values\ (the\ (globs\ (store\ s)\ oref))\ n), f), abupd\ xf\ s))$

definition

$avar :: prog \Rightarrow val \Rightarrow val \Rightarrow state \Rightarrow vvar \times state$ **where**
 $avar\ G\ i'\ a'\ s =$
 $(let\ oref = Heap\ (the-Addr\ a');$
 $\quad i = the-Intg\ i';$
 $\quad n = Inr\ i;$
 $\quad (T, k, cs) = the-Arr\ (globs\ (store\ s)\ oref);$
 $\quad f = (\lambda v\ (x, s). (raise-if\ (\neg G, s \vdash v\ fits\ T)$
 $\quad\quad\quad ArrStore\ x$
 $\quad\quad\quad , upd-gobj\ oref\ n\ v\ s))$
 $in\ ((the\ (cs\ n), f), abupd\ (raise-if\ (\neg i\ in-bounds\ k)\ IndOutBound\ \circ\ np\ a')\ s))$

lemma *fvar-def2*: — better suited for simplification

$fvar\ C\ stat\ fn\ a'\ s =$
 $((the$
 $\quad (values$
 $\quad\quad (the\ (globs\ (store\ s)\ (if\ stat\ then\ Stat\ C\ else\ Heap\ (the-Addr\ a'))))$
 $\quad\quad (Inl\ (fn, C)))$
 $\quad , (\lambda v. supd\ (upd-gobj\ (if\ stat\ then\ Stat\ C\ else\ Heap\ (the-Addr\ a')$
 $\quad\quad\quad (Inl\ (fn, C))$
 $\quad\quad\quad v)))$
 $\quad , abupd\ (if\ stat\ then\ id\ else\ np\ a')\ s)$

<proof>

lemma *avar-def2*: — better suited for simplification

$avar\ G\ i'\ a'\ s =$
 $((the\ ((snd(snd(the-Arr\ (globs\ (store\ s)\ (Heap\ (the-Addr\ a')))))$
 $\quad (Inr\ (the-Intg\ i'))$
 $\quad , (\lambda v\ (x, s'). (raise-if\ (\neg G, s \vdash v\ fits\ (fst(the-Arr\ (globs\ (store\ s)$
 $\quad\quad\quad (Heap\ (the-Addr\ a'))))))$
 $\quad\quad\quad ArrStore\ x$
 $\quad\quad\quad , upd-gobj\ (Heap\ (the-Addr\ a'))$
 $\quad\quad\quad (Inr\ (the-Intg\ i')\ v\ s'))$
 $\quad , abupd\ (raise-if\ (\neg (the-Intg\ i')\ in-bounds\ (fst(snd(the-Arr\ (globs\ (store\ s)$
 $\quad\quad\quad (Heap\ (the-Addr\ a'))))))\ IndOutBound\ \circ\ np\ a')$
 $\quad s)$

<proof>

definition

check-field-access :: *prog* \Rightarrow *qname* \Rightarrow *qname* \Rightarrow *vname* \Rightarrow *bool* \Rightarrow *val* \Rightarrow *state* \Rightarrow *state* **where**
check-field-access *G accC statDeclC fn stat a' s* =
 (let *oref* = if *stat* then *Stat statDeclC*
 else *Heap (the-Addr a')*;
 dynC = case *oref* of
 Heap a \Rightarrow *obj-class (the (globs (store s) oref))*
 | *Stat C* \Rightarrow *C*;
 f = (the (table-of (*DeclConcepts.fields G dynC*) (*fn,statDeclC*)))
 in *abupd*
 (error-if ($\neg G \vdash \text{Field } fn (statDeclC, f) \text{ in } dynC \text{ dyn-accessible-from } accC$)
 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 accC statT mode sig a' s* =
 (let *invC* = *invocation-class mode (store s) a' statT*;
 dynM = the (*dynlookup G statT invC sig*)
 in *abupd*
 (error-if ($\neg G \vdash \text{Methd } sig \text{ dynM in } invC \text{ dyn-accessible-from } accC$)
 AccessViolation)
 s)

evaluation judgments**inductive**

halloc :: [*prog, state, obj-tag, loc, state*] \Rightarrow *bool* (\vdash - *halloc* \rightarrow - [61,61,61,61,61]60) **for** *G::prog*
where — allocating objects on the heap, cf. 12.5

Abrupt:

$G \vdash (\text{Some } x, s) \text{ -halloc } oi \succ \text{undefined} \rightarrow (\text{Some } x, s)$

| *New*: $\llbracket \text{new-Addr (heap s) = Some } a;$
 $(x, oi) = (\text{if atleast-free (heap s) (Suc (Suc 0)) then (None, oi)}$
 $\text{else (Some (Xcpt (Loc a)), CInst (SXcpt OutOfMemory))} \rrbracket$
 \Longrightarrow
 $G \vdash \text{Norm } s \text{ -halloc } oi \succ a \rightarrow (x, \text{init-obj } G \text{ } oi' (\text{Heap } a) s)$

inductive *sxalloc* :: [*prog, state, state*] \Rightarrow *bool* (\vdash - *sxalloc* \rightarrow - [61,61,61]60) **for** *G::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*: $G \vdash (\text{Some (Jump } j), s) \text{ -sxalloc} \rightarrow (\text{Some (Jump } j), s)$

| *Error*: $G \vdash (\text{Some (Error } e), s) \text{ -sxalloc} \rightarrow (\text{Some (Error } e), s)$

| *XcptL*: $G \vdash (\text{Some (Xcpt (Loc } a), s) \text{ -sxalloc} \rightarrow (\text{Some (Xcpt (Loc } a), s)$

| *SXcpt*: $\llbracket G \vdash \text{Norm } s0 \text{ -halloc (CInst (SXcpt } xn)) \succ a \rightarrow (x, s1) \rrbracket \Longrightarrow$
 $G \vdash (\text{Some (Xcpt (Std } xn), s0) \text{ -sxalloc} \rightarrow (\text{Some (Xcpt (Loc } a), s1)$

inductive

eval :: [*prog, state, term, vals, state*] \Rightarrow *bool* (\vdash - \rightarrow '(-, -)' [61,61,80,0,0]60)
and *exec* :: [*prog, state, stmt, state*] \Rightarrow *bool* (\vdash - \rightarrow - [61,61,65, 61]60)
and *evar* :: [*prog, state, var, vvar, state*] \Rightarrow *bool* (\vdash - \rightarrow - [61,61,90,61,61]60)
and *eval'* :: [*prog, state, expr, val, state*] \Rightarrow *bool* (\vdash - \rightarrow - [61,61,80,61,61]60)
and *evals* :: [*prog, state, expr list* ,

$val\ list\ ,state] \Rightarrow bool(+ - - - \dot{\succ} \dashrightarrow -[61,61,61,61,61]60)$

for $G::prog$
where

$G \vdash s - c \rightarrow s' \equiv G \vdash s - In1r\ c \succ \rightarrow (\diamond, s')$
 $| G \vdash s - e - \succ v \rightarrow s' \equiv G \vdash s - In1l\ e \succ \rightarrow (In1\ v, s')$
 $| G \vdash s - e \equiv \succ vf \rightarrow s' \equiv G \vdash s - In2\ e \succ \rightarrow (In2\ vf, s')$
 $| G \vdash s - e \dot{\succ} v \rightarrow s' \equiv G \vdash s - In3\ e \succ \rightarrow (In3\ v, s')$

— propagation of abrupt completion

— cf. 14.1, 15.5

$| Abrupt:$

$G \vdash (Some\ xc, s) - t \succ \rightarrow (undefined3\ t, (Some\ xc, s))$

— execution of statements

— cf. 14.5

$| Skip:$ $G \vdash Norm\ s - Skip \rightarrow Norm\ s$

— cf. 14.7

$| Expr:$ $\llbracket G \vdash Norm\ s0 - e - \succ v \rightarrow s1 \rrbracket \Longrightarrow$
 $G \vdash Norm\ s0 - Expr\ e \rightarrow s1$

$| Lab:$ $\llbracket G \vdash Norm\ s0 - c \rightarrow s1 \rrbracket \Longrightarrow$
 $G \vdash Norm\ s0 - l \cdot c \rightarrow abupd\ (absorb\ l)\ s1$

— cf. 14.2

$| Comp:$ $\llbracket G \vdash Norm\ s0 - c1 \rightarrow s1;$
 $G \vdash s1 - c2 \rightarrow s2 \rrbracket \Longrightarrow$
 $G \vdash Norm\ s0 - c1;;\ c2 \rightarrow s2$

— cf. 14.8.2

$| If:$ $\llbracket G \vdash Norm\ s0 - e - \succ b \rightarrow s1;$
 $G \vdash s1 - (if\ the\ Bool\ b\ then\ c1\ else\ c2) \rightarrow s2 \rrbracket \Longrightarrow$
 $G \vdash Norm\ s0 - If(e)\ c1\ Else\ c2 \rightarrow s2$

— 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*...).

$| Loop:$ $\llbracket G \vdash Norm\ s0 - e - \succ b \rightarrow s1;$
 $if\ the\ Bool\ b$
 $then\ (G \vdash s1 - c \rightarrow s2 \wedge$
 $G \vdash (abupd\ (absorb\ (Cont\ l))\ s2) - l \cdot While(e)\ c \rightarrow s3)$
 $else\ s3 = s1 \rrbracket \Longrightarrow$
 $G \vdash Norm\ s0 - l \cdot While(e)\ c \rightarrow s3$

$| Jmp:$ $G \vdash Norm\ s - Jmp\ j \rightarrow (Some\ (Jump\ j), s)$

— cf. 14.16

$| Throw:$ $\llbracket G \vdash Norm\ s0 - e - \succ a' \rightarrow s1 \rrbracket \Longrightarrow$
 $G \vdash Norm\ s0 - Throw\ e \rightarrow abupd\ (throw\ a')\ s1$

— cf. 14.18.1

$| Try:$ $\llbracket G \vdash Norm\ s0 - c1 \rightarrow s1; G \vdash s1 - s\!x\!alloc \rightarrow s2;$
 $if\ G, s2 \vdash catch\ C\ then\ G \vdash new\ x\!c\!pt\!var\ vn\ s2 - c2 \rightarrow s3\ else\ s3 = s2 \rrbracket \Longrightarrow$
 $G \vdash Norm\ s0 - Try\ c1\ Catch(C\ vn)\ c2 \rightarrow s3$

- cf. 14.18.2
- | *Fin*: $\llbracket G \vdash \text{Norm } s0 \text{ } -c1 \rightarrow (x1, s1);$
 $G \vdash \text{Norm } s1 \text{ } -c2 \rightarrow s2;$
 $s3 = (\text{if } (\exists \text{ err. } x1 = \text{Some } (\text{Error } \text{err}))$
 $\text{then } (x1, s1)$
 $\text{else } \text{abupd } (\text{abrupt-if } (x1 \neq \text{None}) x1) s2) \rrbracket$
 \implies
 $G \vdash \text{Norm } s0 \text{ } -c1 \text{ Finally } c2 \rightarrow s3$
- cf. 12.4.2, 8.5
- | *Init*: $\llbracket \text{the } (\text{class } G \ C) = c;$
 $\text{if } \text{inited } C \ (\text{globs } s0) \text{ then } s3 = \text{Norm } s0$
 $\text{else } (G \vdash \text{Norm } (\text{init-class-obj } G \ C \ s0)$
 $\text{ } -(\text{if } C = \text{Object then Skip else Init } (\text{super } c)) \rightarrow s1 \wedge$
 $G \vdash \text{set-lvars } \text{Map.empty } s1 \text{ } -\text{init } c \rightarrow s2 \wedge s3 = \text{restore-lvars } s1 \ s2) \rrbracket$
 \implies
 $G \vdash \text{Norm } s0 \text{ } -\text{Init } C \rightarrow s3$
- This class initialisation rule is a little bit inaccurate. Look at the exact sequence: (1) The current class object (the static fields) are initialised (*init-class-obj*), (2) the superclasses are initialised, (3) the static initialiser of the current class is invoked. More precisely we should expect another ordering, namely 2 1 3. But we can't just naively toggle 1 and 2. By calling *init-class-obj* before initialising the superclasses, we also implicitly record that we have started to initialise the current class (by setting an value for the class object). This becomes crucial for the completeness proof of the axiomatic semantics *AxCompl.thy*. Static initialisation requires an induction on the number of classes not yet initialised (or to be more precise, classes were the initialisation has not yet begun). So we could first assign a dummy value to the class before superclass initialisation and afterwards set the correct values. But as long as we don't take memory overflow into account when allocating class objects, we can leave things as they are for convenience.
- evaluation of expressions
- cf. 15.8.1, 12.4.1
- | *NewC*: $\llbracket G \vdash \text{Norm } s0 \text{ } -\text{Init } C \rightarrow s1;$
 $G \vdash \quad s1 \text{ } -\text{halloc } (C \text{Inst } C) \succ a \rightarrow s2 \rrbracket \implies$
 $G \vdash \text{Norm } s0 \text{ } -\text{NewC } C \text{ } -\succ \text{Addr } a \rightarrow s2$
- cf. 15.9.1, 12.4.1
- | *NewA*: $\llbracket G \vdash \text{Norm } s0 \text{ } -\text{init-comp-ty } T \rightarrow s1; G \vdash s1 \text{ } -e \text{ } -\succ i' \rightarrow s2;$
 $G \vdash \text{abupd } (\text{check-neg } i') s2 \text{ } -\text{halloc } (\text{Arr } T \ (\text{the-Intg } i')) \succ a \rightarrow s3 \rrbracket \implies$
 $G \vdash \text{Norm } s0 \text{ } -\text{New } T[e] \text{ } -\succ \text{Addr } a \rightarrow s3$
- cf. 15.15
- | *Cast*: $\llbracket G \vdash \text{Norm } s0 \text{ } -e \text{ } -\succ v \rightarrow s1;$
 $s2 = \text{abupd } (\text{raise-if } (\neg G, \text{store } s1 \vdash v \text{ fits } T) \ \text{ClassCast}) s1 \rrbracket \implies$
 $G \vdash \text{Norm } s0 \text{ } -\text{Cast } T \ e \text{ } -\succ v \rightarrow s2$
- cf. 15.19.2
- | *Inst*: $\llbracket G \vdash \text{Norm } s0 \text{ } -e \text{ } -\succ v \rightarrow s1;$
 $b = (v \neq \text{Null} \wedge G, \text{store } s1 \vdash v \text{ fits } \text{RefT } T) \rrbracket \implies$
 $G \vdash \text{Norm } s0 \text{ } -e \ \text{InstOf } T \text{ } -\succ \text{Bool } b \rightarrow s1$
- cf. 15.7.1
- | *Lit*: $G \vdash \text{Norm } s \text{ } -\text{Lit } v \text{ } -\succ v \rightarrow \text{Norm } s$
- | *UnOp*: $\llbracket G \vdash \text{Norm } s0 \text{ } -e \text{ } -\succ v \rightarrow s1 \rrbracket$
 $\implies G \vdash \text{Norm } s0 \text{ } -\text{UnOp } \text{unop } e \text{ } -\succ (\text{eval-unop } \text{unop } v) \rightarrow s1$
- | *BinOp*: $\llbracket G \vdash \text{Norm } s0 \text{ } -e1 \text{ } -\succ v1 \rightarrow s1;$
 $G \vdash s1 \text{ } -(\text{if } \text{need-second-arg } \text{binop } v1 \text{ then } (\text{In1l } e2) \text{ else } (\text{In1r } \text{Skip}))$
 $\succ \rightarrow (\text{In1 } v2, s2)$
 \rrbracket

$$\implies G \vdash \text{Norm } s0 \text{ -BinOp } binop \ e1 \ e2 \text{-}\succ (eval\text{-}binop \ binop \ v1 \ v2) \rightarrow s2$$

— cf. 15.10.2

$$| \text{Super: } G \vdash \text{Norm } s \text{ -Super-}\succ \text{val-this } s \rightarrow \text{Norm } s$$

— cf. 15.2

$$| \text{Acc: } \llbracket G \vdash \text{Norm } s0 \text{ -va=}\succ (v,f) \rightarrow s1 \rrbracket \implies \\ G \vdash \text{Norm } s0 \text{ -Acc } va \text{-}\succ v \rightarrow s1$$

— cf. 15.25.1

$$| \text{Ass: } \llbracket G \vdash \text{Norm } s0 \text{ -va=}\succ (w,f) \rightarrow s1; \\ G \vdash s1 \text{ -e-}\succ v \rightarrow s2 \rrbracket \implies \\ G \vdash \text{Norm } s0 \text{ -va:=e-}\succ v \rightarrow assign \ f \ v \ s2$$

— cf. 15.24

$$| \text{Cond: } \llbracket G \vdash \text{Norm } s0 \text{ -e0-}\succ b \rightarrow s1; \\ G \vdash s1 \text{ -(if the-Bool } b \text{ then } e1 \text{ else } e2)\text{-}\succ v \rightarrow s2 \rrbracket \implies \\ G \vdash \text{Norm } s0 \text{ -e0 ? } e1 : e2 \text{-}\succ v \rightarrow s2$$

— 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*:

$$\llbracket G \vdash \text{Norm } s0 \text{ -e-}\succ a' \rightarrow s1; G \vdash s1 \text{ -args=}\succ vs \rightarrow s2; \\ D = \text{invocation-declclass } G \text{ mode } (store \ s2) \ a' \ \text{statT } (\{name=mn, parTs=pTs\}); \\ s3 = \text{init-lvars } G \ D \ (\{name=mn, parTs=pTs\}) \ \text{mode } a' \ vs \ s2; \\ s3' = \text{check-method-access } G \ \text{accC } \ \text{statT } \ \text{mode } (\{name=mn, parTs=pTs\}) \ a' \ s3; \\ G \vdash s3' \text{ -Method } D \ (\{name=mn, parTs=pTs\}) \text{-}\succ v \rightarrow s4 \rrbracket$$

\implies

$$G \vdash \text{Norm } s0 \text{ -}\{accC, statT, mode\}e.mn(\{pTs\}args)\text{-}\succ v \rightarrow (restore\text{-}lvars \ s2 \ s4)$$

— The accessibility check is after *init-lvars*, to keep it simple. *init-lvars* already tests for the absence of a null-pointer reference in case of an instance method invocation.

$$| \text{Method: } \llbracket G \vdash \text{Norm } s0 \text{ -body } G \ D \ sig \text{-}\succ v \rightarrow s1 \rrbracket \implies \\ G \vdash \text{Norm } s0 \text{ -Method } D \ sig \text{-}\succ v \rightarrow s1$$

$$| \text{Body: } \llbracket G \vdash \text{Norm } s0 \text{ -Init } D \rightarrow s1; G \vdash s1 \text{ -c} \rightarrow s2; \\ s3 = (\text{if } (\exists \ l. \ \text{abrupt } s2 = \text{Some } (\text{Jump } (\text{Break } l))) \vee \\ \text{abrupt } s2 = \text{Some } (\text{Jump } (\text{Cont } l))) \\ \text{then } \text{abupd } (\lambda \ x. \ \text{Some } (\text{Error } \text{CrossMethodJump})) \ s2 \\ \text{else } s2 \rrbracket \implies \\ G \vdash \text{Norm } s0 \text{ -Body } D \ c \text{-}\succ \text{the } (locals \ (store \ s2) \ \text{Result}) \\ \rightarrow \text{abupd } (absorb \ Ret) \ s3$$

— cf. 14.15, 12.4.1

— We filter out a break/continue in *s2*, so that we can proof definite assignment correct, without the need of conformance of the state. By this the different parts of the typesafety proof can be disentangled a little.

— evaluation of variables

— cf. 15.13.1, 15.7.2

| *LVar*: $G \vdash \text{Norm } s \text{ -LVar } vn \Rightarrow \text{lvar } vn \text{ s} \rightarrow \text{Norm } s$

— cf. 15.10.1, 12.4.1

| *FVar*: $\llbracket G \vdash \text{Norm } s0 \text{ -Init } \text{statDeclC} \rightarrow s1; G \vdash s1 \text{ -e} \rightarrow a \rightarrow s2;$
 $(v, s2') = \text{fvar } \text{statDeclC} \text{ stat } \text{fn } a \text{ s2};$
 $s3 = \text{check-field-access } G \text{ accC } \text{statDeclC} \text{ fn } \text{stat } a \text{ s2}' \rrbracket \implies$
 $G \vdash \text{Norm } s0 \text{ -}\{\text{accC}, \text{statDeclC}, \text{stat}\}e.. \text{fn} \Rightarrow v \rightarrow s3$

— The accessibility check is after *fvar*, to keep it simple. *fvar* already tests for the absence of a null-pointer reference in case of an instance field

— cf. 15.12.1, 15.25.1

| *AVar*: $\llbracket G \vdash \text{Norm } s0 \text{ -e1} \rightarrow a \rightarrow s1; G \vdash s1 \text{ -e2} \rightarrow i \rightarrow s2;$
 $(v, s2') = \text{avar } G \text{ i } a \text{ s2} \rrbracket \implies$
 $G \vdash \text{Norm } s0 \text{ -e1}.[e2] \Rightarrow v \rightarrow s2'$

— evaluation of expression lists

— cf. 15.11.4.2

| *Nil*:

$$G \vdash \text{Norm } s0 \text{ -}[\] \Rightarrow [\] \rightarrow \text{Norm } s0$$

— cf. 15.6.4

| *Cons*: $\llbracket G \vdash \text{Norm } s0 \text{ -e} \rightarrow v \rightarrow s1;$
 $G \vdash s1 \text{ -es} \Rightarrow vs \rightarrow s2 \rrbracket \implies$
 $G \vdash \text{Norm } s0 \text{ -e}\#es \Rightarrow v\#vs \rightarrow s2$

$\langle \text{ML} \rangle$

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

inductive-cases *halloc-elim-cases*:

$$G \vdash (\text{Some } xc, s) \text{ -halloc } oi \rightarrow a \rightarrow s'$$

$$G \vdash (\text{Norm } s) \text{ -halloc } oi \rightarrow a \rightarrow s'$$

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*: $\llbracket G \vdash s \text{ -sxalloc} \rightarrow s';$

$$\bigwedge s. \llbracket s' = \text{Norm } s \rrbracket \implies P;$$

$$\bigwedge j s. \llbracket s' = (\text{Some } (\text{Jump } j), s) \rrbracket \implies P;$$

$$\bigwedge e s. \llbracket s' = (\text{Some } (\text{Error } e), s) \rrbracket \implies P;$$

$$\bigwedge a s. \llbracket s' = (\text{Some } (\text{Xcpt } (\text{Loc } a)), s) \rrbracket \implies P$$

$$\rrbracket \implies P$$

$\langle \text{proof} \rangle$

declare *not-None-eq* [*simp del*]

declare *split-paired-All* [*simp del*] *split-paired-Ex* [*simp del*]

$\langle \text{ML} \rangle$

inductive-cases *eval-cases*: $G \vdash s -t \succ \rightarrow (v, s')$

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

$G \vdash (\text{Some } xc, s) -t$	$\succ \rightarrow (v, s')$
$G \vdash \text{Norm } s -\text{In1r } \text{Skip}$	$\succ \rightarrow (x, s')$
$G \vdash \text{Norm } s -\text{In1r } (\text{Jmp } j)$	$\succ \rightarrow (x, s')$
$G \vdash \text{Norm } s -\text{In1r } (l \cdot c)$	$\succ \rightarrow (x, s')$
$G \vdash \text{Norm } s -\text{In3 } ([\])$	$\succ \rightarrow (v, s')$
$G \vdash \text{Norm } s -\text{In3 } (e \# es)$	$\succ \rightarrow (v, s')$
$G \vdash \text{Norm } s -\text{In1l } (\text{Lit } w)$	$\succ \rightarrow (v, s')$
$G \vdash \text{Norm } s -\text{In1l } (\text{UnOp } \text{unop } e)$	$\succ \rightarrow (v, s')$
$G \vdash \text{Norm } s -\text{In1l } (\text{BinOp } \text{binop } e1 \ e2)$	$\succ \rightarrow (v, s')$
$G \vdash \text{Norm } s -\text{In2 } (\text{LVar } vn)$	$\succ \rightarrow (v, s')$
$G \vdash \text{Norm } s -\text{In1l } (\text{Cast } T \ e)$	$\succ \rightarrow (v, s')$
$G \vdash \text{Norm } s -\text{In1l } (e \ \text{InstOf } T)$	$\succ \rightarrow (v, s')$
$G \vdash \text{Norm } s -\text{In1l } (\text{Super})$	$\succ \rightarrow (v, s')$
$G \vdash \text{Norm } s -\text{In1l } (\text{Acc } va)$	$\succ \rightarrow (v, s')$
$G \vdash \text{Norm } s -\text{In1r } (\text{Expr } e)$	$\succ \rightarrow (x, s')$
$G \vdash \text{Norm } s -\text{In1r } (c1 ;; c2)$	$\succ \rightarrow (x, s')$
$G \vdash \text{Norm } s -\text{In1l } (\text{Methd } C \ \text{sig})$	$\succ \rightarrow (x, s')$
$G \vdash \text{Norm } s -\text{In1l } (\text{Body } D \ c)$	$\succ \rightarrow (x, s')$
$G \vdash \text{Norm } s -\text{In1l } (e0 \ ? \ e1 : e2)$	$\succ \rightarrow (v, s')$
$G \vdash \text{Norm } s -\text{In1r } (\text{If } (e) \ c1 \ \text{Else } c2)$	$\succ \rightarrow (x, s')$
$G \vdash \text{Norm } s -\text{In1r } (l \cdot \text{While } (e) \ c)$	$\succ \rightarrow (x, s')$
$G \vdash \text{Norm } s -\text{In1r } (c1 \ \text{Finally } c2)$	$\succ \rightarrow (x, s')$
$G \vdash \text{Norm } s -\text{In1r } (\text{Throw } e)$	$\succ \rightarrow (x, s')$
$G \vdash \text{Norm } s -\text{In1l } (\text{NewC } C)$	$\succ \rightarrow (v, s')$
$G \vdash \text{Norm } s -\text{In1l } (\text{New } T[e])$	$\succ \rightarrow (v, s')$
$G \vdash \text{Norm } s -\text{In1l } (\text{Ass } va \ e)$	$\succ \rightarrow (v, s')$
$G \vdash \text{Norm } s -\text{In1r } (\text{Try } c1 \ \text{Catch } (tn \ vn) \ c2)$	$\succ \rightarrow (x, s')$
$G \vdash \text{Norm } s -\text{In2 } (\{\text{accC}, \text{statDeclC}, \text{stat}\} e..fn)$	$\succ \rightarrow (v, s')$
$G \vdash \text{Norm } s -\text{In2 } (e1.[e2])$	$\succ \rightarrow (v, s')$
$G \vdash \text{Norm } s -\text{In1l } (\{\text{accC}, \text{statT}, \text{mode}\} e.mn(\{pT\}p))$	$\succ \rightarrow (v, s')$
$G \vdash \text{Norm } s -\text{In1r } (\text{Init } C)$	$\succ \rightarrow (x, s')$

declare *not-None-eq* [*simp*]

declare *split-paired-All* [*simp*] *split-paired-Ex* [*simp*]

$\langle \text{ML} \rangle$

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 *case* *t* *of*

In1 *ec* \implies (*case* *ec* *of*
Inl *e* \implies ($\exists v. w = \text{In1 } v$)
| *Inr* *c* $\implies w = \diamond$)
| *In2* *e* \implies ($\exists v. w = \text{In2 } v$)
| *In3* *e* \implies ($\exists v. w = \text{In3 } v$))

$\langle \text{proof} \rangle$

The following simplification procedures set up the proper injections of terms and their corresponding values in the evaluation relation: E.g. an expression (injection *In1l* into terms) always evaluates to ordinary values (injection *In1* into generalised values *vals*).

lemma *eval-expr-eq*: $G \vdash s -\text{In1l } t \succ \rightarrow (w, s') = (\exists v. w = \text{In1 } v \wedge G \vdash s -t \succ \rightarrow v \rightarrow s')$

$\langle \text{proof} \rangle$

lemma *eval-var-eq*: $G \vdash s -In2 t \succ \rightarrow (w, s') = (\exists vf. w=In2 vf \wedge G \vdash s -t \dot{=} \succ vf \rightarrow s')$
 ⟨proof⟩

lemma *eval-exprs-eq*: $G \vdash s -In3 t \succ \rightarrow (w, s') = (\exists vs. w=In3 vs \wedge G \vdash s -t \dot{=} \succ vs \rightarrow s')$
 ⟨proof⟩

lemma *eval-stmt-eq*: $G \vdash s -In1r t \succ \rightarrow (w, s') = (w=\diamond \wedge G \vdash s -t \rightarrow s')$
 ⟨proof⟩

⟨ML⟩

declare *halloc.Abrupt* [intro!] *eval.Abrupt* [intro!] *AbruptIs* [intro!]

Callee, InsInitE, InsInitV, FinA are only used in smallstep semantics, not in the bigstep semantics.
 So their is no valid evaluation of these terms

lemma *eval-Callee*: $G \vdash Norm s -Callee l e -\succ v \rightarrow s' = False$
 ⟨proof⟩

lemma *eval-InsInitE*: $G \vdash Norm s -InsInitE c e -\succ v \rightarrow s' = False$
 ⟨proof⟩

lemma *eval-InsInitV*: $G \vdash Norm s -InsInitV c w -\succ v \rightarrow s' = False$
 ⟨proof⟩

lemma *eval-FinA*: $G \vdash Norm s -FinA a c \rightarrow s' = False$
 ⟨proof⟩

lemma *eval-no-abrupt-lemma*:
 $\bigwedge s s'. G \vdash s -t \succ \rightarrow (w, s') \implies normal s' \longrightarrow normal s$
 ⟨proof⟩

lemma *eval-no-abrupt*:
 $G \vdash (x, s) -t \succ \rightarrow (w, Norm s') =$
 $(x = None \wedge G \vdash Norm s -t \succ \rightarrow (w, Norm s'))$
 ⟨proof⟩

⟨ML⟩

lemma *eval-abrupt-lemma*:
 $G \vdash s -t \succ \rightarrow (v, s') \implies abrupt s = Some xc \longrightarrow s' = s \wedge v = undefined3 t$
 ⟨proof⟩

lemma *eval-abrupt*:
 $G \vdash (Some xc, s) -t \succ \rightarrow (w, s') =$
 $(s' = (Some xc, s) \wedge w = undefined3 t \wedge$
 $G \vdash (Some xc, s) -t \succ \rightarrow (undefined3 t, (Some xc, s)))$
 ⟨proof⟩

$\langle ML \rangle$

lemma *LitI*: $G \vdash s \text{ --Lit } v \text{ --}\gamma \text{ (if normal } s \text{ then } v \text{ else undefined)} \rightarrow s$
 $\langle \text{proof} \rangle$

lemma *SkipI* [*intro!*]: $G \vdash s \text{ --Skip} \rightarrow s$
 $\langle \text{proof} \rangle$

lemma *ExprI*: $G \vdash s \text{ --}e \text{ --}\gamma v \rightarrow s' \implies G \vdash s \text{ --Expr } e \rightarrow s'$
 $\langle \text{proof} \rangle$

lemma *CompI*: $\llbracket G \vdash s \text{ --}c1 \rightarrow s1; G \vdash s1 \text{ --}c2 \rightarrow s2 \rrbracket \implies G \vdash s \text{ --}c1;; c2 \rightarrow s2$
 $\langle \text{proof} \rangle$

lemma *CondI*:
 $\bigwedge s1. \llbracket G \vdash s \text{ --}e \text{ --}\gamma b \rightarrow s1; G \vdash s1 \text{ --(if the-Bool } b \text{ then } e1 \text{ else } e2) \text{ --}\gamma v \rightarrow s2 \rrbracket \implies$
 $G \vdash s \text{ --}e \text{ ? } e1 : e2 \text{ --}\gamma \text{ (if normal } s1 \text{ then } v \text{ else undefined)} \rightarrow s2$
 $\langle \text{proof} \rangle$

lemma *IfI*: $\llbracket G \vdash s \text{ --}e \text{ --}\gamma v \rightarrow s1; G \vdash s1 \text{ --(if the-Bool } v \text{ then } c1 \text{ else } c2) \rightarrow s2 \rrbracket$
 $\implies G \vdash s \text{ --If}(e) \text{ } c1 \text{ Else } c2 \rightarrow s2$
 $\langle \text{proof} \rangle$

lemma *MethdI*: $G \vdash s \text{ --body } G \text{ } C \text{ sig} \text{ --}\gamma v \rightarrow s'$
 $\implies G \vdash s \text{ --Methd } C \text{ sig} \text{ --}\gamma v \rightarrow s'$
 $\langle \text{proof} \rangle$

lemma *eval-Call*:
 $\llbracket G \vdash \text{Norm } s0 \text{ --}e \text{ --}\gamma a' \rightarrow s1; G \vdash s1 \text{ --ps} \dot{\text{--}} \gamma pvs \rightarrow s2;$
 $D = \text{invocation-declclass } G \text{ mode (store } s2) \text{ } a' \text{ statT } (\text{name=mn,parTs=pTs});$
 $s3 = \text{init-lvars } G \text{ } D (\text{name=mn,parTs=pTs}) \text{ mode } a' \text{ pvs } s2;$
 $s3' = \text{check-method-access } G \text{ accC statT mode } (\text{name=mn,parTs=pTs}) \text{ } a' \text{ } s3;$
 $G \vdash s3' \text{ --Methd } D (\text{name=mn,parTs=pTs}) \text{ --}\gamma v \rightarrow s4;$
 $s4' = \text{restore-lvars } s2 \text{ } s4 \rrbracket \implies$
 $G \vdash \text{Norm } s0 \text{ --}\{ \text{accC, statT, mode} \} e \cdot \text{mn}(\{ pTs \} ps) \text{ --}\gamma v \rightarrow s4'$
 $\langle \text{proof} \rangle$

lemma *eval-Init*:
 $\llbracket \text{if initd } C \text{ (globs } s0) \text{ then } s3 = \text{Norm } s0$
 $\text{ else } G \vdash \text{Norm (init-class-obj } G \text{ } C \text{ } s0)$
 $\text{ --(if } C = \text{Object then Skip else Init (super (the (class } G \text{ } C)))} \rightarrow s1 \wedge$
 $G \vdash \text{set-lvars Map.empty } s1 \text{ --(init (the (class } G \text{ } C)))} \rightarrow s2 \wedge$
 $s3 = \text{restore-lvars } s1 \text{ } s2 \rrbracket \implies$
 $G \vdash \text{Norm } s0 \text{ --Init } C \rightarrow s3$
 $\langle \text{proof} \rangle$

lemma *init-done*: $\text{initd } C \text{ } s \implies G \vdash s \text{ --Init } C \rightarrow s$
 $\langle \text{proof} \rangle$

lemma *eval-StatRef*:

$G \vdash s \text{ --StatRef } rt \text{ --}\succ (if \text{ abrupt } s = None \text{ then } Null \text{ else } undefined) \rightarrow s$
 ⟨proof⟩

lemma *SkipD* [*dest!*]: $G \vdash s \text{ --Skip} \rightarrow s' \implies s' = s$
 ⟨proof⟩

lemma *Skip-eq* [*simp*]: $G \vdash s \text{ --Skip} \rightarrow s' = (s = s')$
 ⟨proof⟩

lemma *init-retains-locals* [*rule-format (no-asm)*]: $G \vdash s \text{ --}t \succ \rightarrow (w, s') \implies$
 $(\forall C. t = In1r (Init C) \longrightarrow locals (store s) = locals (store s'))$
 ⟨proof⟩

lemma *halloc-xcpt* [*dest!*]:

$\bigwedge s'. G \vdash (Some \text{ xc}, s) \text{ --halloc } oi \succ a \rightarrow s' \implies s' = (Some \text{ xc}, s)$
 ⟨proof⟩

lemma *eval-Methd*:

$G \vdash s \text{ --In1l}(\text{body } G \ C \ sig) \succ \rightarrow (w, s')$
 $\implies G \vdash s \text{ --In1l}(\text{Methd } C \ sig) \succ \rightarrow (w, s')$
 ⟨proof⟩

lemma *eval-Body*: $\llbracket G \vdash Norm \ s0 \text{ --Init } D \rightarrow s1; G \vdash s1 \text{ --}c \rightarrow s2;$
 $res = the (locals (store s2) Result);$
 $s3 = (if (\exists l. abrupt \ s2 = Some (Jump (Break l)) \vee$
 $abrupt \ s2 = Some (Jump (Cont l)))$
 $then \ abupd (\lambda x. Some (Error CrossMethodJump)) \ s2$
 $else \ s2);$
 $s4 = abupd (absorb Ret) \ s3 \rrbracket \implies$
 $G \vdash Norm \ s0 \text{ --Body } D \ c \text{ --}\succ res \rightarrow s4$
 ⟨proof⟩

lemma *eval-binop-arg2-indep*:

$\neg \text{ need-second-arg } binop \ v1 \implies \text{eval-binop } binop \ v1 \ x = \text{eval-binop } binop \ v1 \ y$
 ⟨proof⟩

lemma *eval-BinOp-arg2-indepI*:

assumes *eval-e1*: $G \vdash Norm \ s0 \text{ --}e1 \text{ --}\succ v1 \rightarrow s1$ **and**
no-need: $\neg \text{ need-second-arg } binop \ v1$
shows $G \vdash Norm \ s0 \text{ --BinOp } binop \ e1 \ e2 \text{ --}\succ (\text{eval-binop } binop \ v1 \ v2) \rightarrow s1$
 (**is** *?EvalBinOp* *v2*)
 ⟨proof⟩

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)]:

$G \vdash s \text{-t} \succ \rightarrow (w, s') \implies (\forall w' s''. G \vdash s \text{-t} \succ \rightarrow (w', s'') \longrightarrow w' = w \wedge s'' = s')$
 ⟨proof⟩

lemma *single-valued-eval*:

single-valued $\{(s, t), (v, s')\}. G \vdash s \text{-t} \succ \rightarrow (v, s')$
 ⟨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*: $\wedge fd. wf-fdecl\ G\ P\ fd = is-acc-type\ G\ P\ (type\ (snd\ fd))$
 <*proof*>

declare *wf-fdecl-def2* [*iff*]

type and expression names

datatype *tnam'* = *HasFoo'* | *Base'* | *Ext'* | *Main'*
datatype *vnam'* = *arr'* | *vee'* | *z'* | *e'*
datatype *label'* = *lab1'*

axiomatization

tnam' :: *tnam'* \Rightarrow *tnam* **and**
vnam' :: *vnam'* \Rightarrow *vname* **and**
label' :: *label'* \Rightarrow *label*

where

inj-tnam' [*simp*]: $\wedge x\ y. (tnam'\ x = tnam'\ y) = (x = y)$ **and**
inj-vnam' [*simp*]: $\wedge x\ y. (vnam'\ x = vnam'\ y) = (x = y)$ **and**
inj-label' [*simp*]: $\wedge x\ y. (label'\ x = label'\ y) = (x = y)$ **and**

surj-tnam': $\wedge n. \exists m. n = tnam'\ m$ **and**
surj-vnam': $\wedge n. \exists m. n = vnam'\ m$ **and**
surj-label': $\wedge n. \exists m. n = label'\ m$

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 = (|name=foo,parTs=[Class Base] |)

```

definition

```
foo-mhead :: mhead
where foo-mhead = (|access=Public,static=False,pars=[z],resT=Class Base|)
```

definition

```
Base-foo :: mdecl
where Base-foo = (foo-sig, (|access=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 :: qname => qname => var
where arr-viewed-from acc C = {acc C,Base,True}StatRef (Class T 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 \mapsto HasFooInt)*
 ⟨*proof*⟩

lemma *table-classes-Object* [*simp*]:

table-of Classes Object = *Some* (λ*access=Public*,*cfields=[]*
 ,*methods=Object-mdecls*
 ,*init=Skip*,*super=undefined*,*superIfs=[]*)

⟨*proof*⟩

lemma *table-classes-SXcpt* [*simp*]:

table-of Classes (SXcpt xn)
 = *Some* (λ*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 == (λ*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!]: $(Object, C) \in (subcls1\ tprg)^+ \implies R$
 ⟨proof⟩

lemma *not-Throwable-subcls-SXcpt* [elim!]:
 $(SXcpt\ Throwable, SXcpt\ xn) \in (subcls1\ tprg)^+ \implies R$
 ⟨proof⟩

lemma *not-SXcpt-n-subcls-SXcpt-n* [elim!]:
 $(SXcpt\ xn, SXcpt\ xn) \in (subcls1\ tprg)^+ \implies R$
 ⟨proof⟩

lemma *not-Base-subcls-Ext* [elim!]: $(Base, Ext) \in (subcls1\ tprg)^+ \implies R$
 ⟨proof⟩

lemma *not-TName-n-subcls-TName-n* [rule-format (no-asm), elim!]:
 $((\{pid=java-lang, tid=TName\ tn\}), (\{pid=java-lang, tid=TName\ tn\}))$
 $\in (subcls1\ tprg)^+ \implies R$
 ⟨proof⟩

lemma *ws-idecl-HasFoo*: *ws-idecl tprg HasFoo* []
 ⟨proof⟩

lemma *ws-cdecl-Object*: *ws-cdecl tprg Object any*
 ⟨proof⟩

lemma *ws-cdecl-Throwable*: *ws-cdecl tprg (SXcpt Throwable) Object*
 ⟨proof⟩

lemma *ws-cdecl-SXcpt*: *ws-cdecl tprg (SXcpt xn) (SXcpt Throwable)*
 ⟨proof⟩

lemma *ws-cdecl-Base*: *ws-cdecl tprg Base Object*
 ⟨proof⟩

lemma *ws-cdecl-Ext*: *ws-cdecl tprg Ext Base*
 ⟨proof⟩

lemma *ws-cdecl-Main*: *ws-cdecl tprg Main Object*
 ⟨proof⟩

lemmas *ws-cdecls = ws-cdecl-SXcpt ws-cdecl-Object ws-cdecl-Throwable*
ws-cdecl-Base ws-cdecl-Ext ws-cdecl-Main

declare *not-Object-subcls-any* [rule del]
not-Throwable-subcls-SXcpt [rule del]

not-SXcpt-n-subcls-SXcpt-n [rule del]
not-Base-subcls-Ext [rule del] *not-TName-n-subcls-TName-n* [rule del]

lemma *ws-idecl-all*:

$G = \text{tprg} \implies (\forall (I, i) \in \text{set } \text{Ifaces}. \text{ws-idecl } G \ I \ (\text{isuperIfs } i))$
 ⟨proof⟩

lemma *ws-cdecl-all*: $G = \text{tprg} \implies (\forall (C, c) \in \text{set } \text{Classes}. \text{ws-cdecl } G \ C \ (\text{super } c))$
 ⟨proof⟩

lemma *ws-tprg*: *ws-prog tprg*

⟨proof⟩

misc program properties (independent of well-structuredness)

lemma *single-iface* [simp]: *is-iface tprg I = (I = HasFoo)*

⟨proof⟩

lemma *empty-subint1* [simp]: *subint1 tprg = {}*

⟨proof⟩

lemma *unique-ifaces*: *unique Ifaces*

⟨proof⟩

lemma *unique-classes*: *unique Classes*

⟨proof⟩

lemma *SXcpt-subcls-Throwable* [simp]: *tprg ⊢ SXcpt xn ≤_C SXcpt Throwable*

⟨proof⟩

lemma *Ext-subclseq-Base* [simp]: *tprg ⊢ Ext ≤_C Base*

⟨proof⟩

lemma *Ext-subcls-Base* [simp]: *tprg ⊢ Ext <_C Base*

⟨proof⟩

fields and method lookup

lemma *fields-tprg-Object* [simp]: *DeclConcepts.fields tprg Object = []*

⟨proof⟩

lemma *fields-tprg-Throwable* [simp]:

DeclConcepts.fields tprg (SXcpt Throwable) = []

⟨proof⟩

lemma *fields-tprg-SXcpt* [simp]: *DeclConcepts.fields tprg (SXcpt xn) = []*

⟨proof⟩

lemmas $fields-rec' = fields-rec$ [OF - ws-tprg]

lemma $fields-Base$ [simp]:

$DeclConcepts.fields\ tprg\ Base$

$= [((arr, Base), (\access=Public, static=True, type=PrimT\ Boolean.[])),$
 $((vee, Base), (\access=Public, static=False, type=Iface\ HasFoo\ []))]$

$\langle proof \rangle$

lemma $fields-Ext$ [simp]:

$DeclConcepts.fields\ tprg\ Ext$

$= [((vee, Ext), (\access=Public, static=False, type= PrimT\ Integer))]]$
 $@\ DeclConcepts.fields\ tprg\ Base$

$\langle proof \rangle$

lemmas $imethds-rec' = imethds-rec$ [OF - ws-tprg]

lemmas $methd-rec' = methd-rec$ [OF - ws-tprg]

lemma $imethds-HasFoo$ [simp]:

$imethds\ tprg\ HasFoo = set-option \circ Map.empty(foo-sig \rightarrow (HasFoo, foo-mhead))$

$\langle proof \rangle$

lemma $methd-tprg-Object$ [simp]: $methd\ tprg\ Object = Map.empty$

$\langle proof \rangle$

lemma $methd-Base$ [simp]:

$methd\ tprg\ Base = table-of [(\lambda(s,m). (s, Base, m))\ Base-foo]$

$\langle proof \rangle$

lemma $memberid-Base-foo-simp$ [simp]:

$memberid\ (mdecl\ Base-foo) = mid\ foo-sig$

$\langle proof \rangle$

lemma $memberid-Ext-foo-simp$ [simp]:

$memberid\ (mdecl\ Ext-foo) = mid\ foo-sig$

$\langle proof \rangle$

lemma $Base-declares-foo$:

$tprg \vdash mdecl\ Base-foo\ declared-in\ Base$

$\langle proof \rangle$

lemma $foo-sig-not-undeclared-in-Base$:

$\neg tprg \vdash mid\ foo-sig\ undeclared-in\ Base$

$\langle proof \rangle$

lemma $Ext-declares-foo$:

$tprg \vdash mdecl\ Ext-foo\ declared-in\ Ext$

$\langle proof \rangle$

lemma *foo-sig-not-undeclared-in-Ext*:
 $\neg \text{tprg} \vdash \text{mid } \text{foo-sig } \text{undeclared-in } \text{Ext}$
 ⟨proof⟩

lemma *Base-foo-not-inherited-in-Ext*:
 $\neg \text{tprg} \vdash \text{Ext } \text{inherits } (\text{Base}, \text{mdecl } \text{Base-foo})$
 ⟨proof⟩

lemma *Ext-method-inheritance*:
 $\text{filter-tab } (\lambda \text{sig } m. \text{tprg} \vdash \text{Ext } \text{inherits } \text{method } \text{sig } m)$
 $(\text{Map.empty}(\text{fst } ((\lambda(s, m). (s, \text{Base}, m)) \text{Base-foo}) \mapsto$
 $\text{snd } ((\lambda(s, m). (s, \text{Base}, m)) \text{Base-foo})))$
 $= \text{Map.empty}$
 ⟨proof⟩

lemma *methd-Ext* [simp]: $\text{methd } \text{tprg } \text{Ext} =$
 $\text{table-of } [(\lambda(s, m). (s, \text{Ext}, m)) \text{Ext-foo}]$
 ⟨proof⟩

accessibility

lemma *classesDefined*:
 $\llbracket \text{class } \text{tprg } C = \text{Some } c; C \neq \text{Object} \rrbracket \implies \exists \text{sc. class } \text{tprg } (\text{super } c) = \text{Some } \text{sc}$
 ⟨proof⟩

lemma *superclassesBase* [simp]: $\text{superclasses } \text{tprg } \text{Base} = \{\text{Object}\}$
 ⟨proof⟩

lemma *superclassesExt* [simp]: $\text{superclasses } \text{tprg } \text{Ext} = \{\text{Base}, \text{Object}\}$
 ⟨proof⟩

lemma *superclassesMain* [simp]: $\text{superclasses } \text{tprg } \text{Main} = \{\text{Object}\}$
 ⟨proof⟩

lemma *HasFoo-accessible* [simp]: $\text{tprg} \vdash (\text{Iface } \text{HasFoo}) \text{ accessible-in } P$
 ⟨proof⟩

lemma *HasFoo-is-acc-iface* [simp]: $\text{is-acc-iface } \text{tprg } P \text{ HasFoo}$
 ⟨proof⟩

lemma *HasFoo-is-acc-type* [simp]: $\text{is-acc-type } \text{tprg } P (\text{Iface } \text{HasFoo})$
 ⟨proof⟩

lemma *Base-accessible* [simp]: $\text{tprg} \vdash (\text{Class } \text{Base}) \text{ accessible-in } P$
 ⟨proof⟩

lemma *Base-is-acc-class* [simp]: $\text{is-acc-class } \text{tprg } P \text{ Base}$

$\langle proof \rangle$

lemma *Base-is-acc-type*[simp]: *is-acc-type* tprg *P* (*Class Base*)

$\langle proof \rangle$

lemma *Ext-accessible*[simp]: tprg \vdash (*Class Ext*) *accessible-in P*

$\langle proof \rangle$

lemma *Ext-is-acc-class*[simp]: *is-acc-class* tprg *P Ext*

$\langle proof \rangle$

lemma *Ext-is-acc-type*[simp]: *is-acc-type* tprg *P* (*Class Ext*)

$\langle proof \rangle$

lemma *accmethd-tprg-Object* [simp]: *accmethd* tprg *S Object* = *Map.empty*

$\langle proof \rangle$

lemma *snd-special-simp*: *snd* (($\lambda(s, m). (s, a, m)$) *x*) = (*a, snd x*)

$\langle proof \rangle$

lemma *fst-special-simp*: *fst* (($\lambda(s, m). (s, a, m)$) *x*) = *fst x*

$\langle proof \rangle$

lemma *foo-sig-undeclared-in-Object*:

tprg \vdash mid *foo-sig undeclared-in Object*

$\langle proof \rangle$

lemma *unique-sig-Base-foo*:

tprg \vdash mdecl (*sig, snd Base-foo*) *declared-in Base* \implies *sig=foo-sig*

$\langle proof \rangle$

lemma *Base-foo-no-override*:

tprg, sig \vdash (*Base, (snd Base-foo)*) *overrides old* \implies *P*

$\langle proof \rangle$

lemma *Base-foo-no-stat-override*:

tprg, sig \vdash (*Base, (snd Base-foo)*) *overrides_S old* \implies *P*

$\langle proof \rangle$

lemma *Base-foo-no-hide*:

tprg, sig \vdash (*Base, (snd Base-foo)*) *hides old* \implies *P*

$\langle proof \rangle$

lemma *Ext-foo-no-hide*:

tprg, sig \vdash (*Ext, (snd Ext-foo)*) *hides old* \implies *P*

$\langle \text{proof} \rangle$

lemma *unique-sig-Ext-foo*:

$\text{tprg} \vdash \text{mdecl } (\text{sig}, \text{snd Ext-foo}) \text{ declared-in Ext} \implies \text{sig} = \text{foo-sig}$
 $\langle \text{proof} \rangle$

lemma *Ext-foo-override*:

$\text{tprg}, \text{sig} \vdash (\text{Ext}, (\text{snd Ext-foo})) \text{ overrides old}$
 $\implies \text{old} = (\text{Base}, (\text{snd Base-foo}))$
 $\langle \text{proof} \rangle$

lemma *Ext-foo-stat-override*:

$\text{tprg}, \text{sig} \vdash (\text{Ext}, (\text{snd Ext-foo})) \text{ overrides}_S \text{ old}$
 $\implies \text{old} = (\text{Base}, (\text{snd Base-foo}))$
 $\langle \text{proof} \rangle$

lemma *Base-foo-member-of-Base*:

$\text{tprg} \vdash (\text{Base}, \text{mdecl Base-foo}) \text{ member-of Base}$
 $\langle \text{proof} \rangle$

lemma *Base-foo-member-in-Base*:

$\text{tprg} \vdash (\text{Base}, \text{mdecl Base-foo}) \text{ member-in Base}$
 $\langle \text{proof} \rangle$

lemma *Ext-foo-member-of-Ext*:

$\text{tprg} \vdash (\text{Ext}, \text{mdecl Ext-foo}) \text{ member-of Ext}$
 $\langle \text{proof} \rangle$

lemma *Ext-foo-member-in-Ext*:

$\text{tprg} \vdash (\text{Ext}, \text{mdecl Ext-foo}) \text{ member-in Ext}$
 $\langle \text{proof} \rangle$

lemma *Base-foo-permits-acc*:

$\text{tprg} \vdash (\text{Base}, \text{mdecl Base-foo}) \text{ in Base permits-acc-from } S$
 $\langle \text{proof} \rangle$

lemma *Base-foo-accessible* [simp]:

$\text{tprg} \vdash (\text{Base}, \text{mdecl Base-foo}) \text{ of Base accessible-from } S$
 $\langle \text{proof} \rangle$

lemma *Base-foo-dyn-accessible* [simp]:

$\text{tprg} \vdash (\text{Base}, \text{mdecl Base-foo}) \text{ in Base dyn-accessible-from } S$
 $\langle \text{proof} \rangle$

lemma *accmethd-Base* [simp]:

$\text{accmethd tprg } S \text{ Base} = \text{methd tprg Base}$
 $\langle \text{proof} \rangle$

lemma *Ext-foo-permits-acc*:

$tprg \vdash (Ext, mdecl\ Ext-foo)$ in *Ext permits-acc-from S*
 ⟨proof⟩

lemma *Ext-foo-accessible [simp]*:

$tprg \vdash (Ext, mdecl\ Ext-foo)$ of *Ext accessible-from S*
 ⟨proof⟩

lemma *Ext-foo-dyn-accessible [simp]*:

$tprg \vdash (Ext, mdecl\ Ext-foo)$ in *Ext dyn-accessible-from S*
 ⟨proof⟩

lemma *Ext-foo-overrides-Base-foo*:

$tprg \vdash (Ext, Ext-foo)$ overrides $(Base, Base-foo)$
 ⟨proof⟩

lemma *accmethd-Ext [simp]*:

$accmethd\ tprg\ S\ Ext = methd\ tprg\ Ext$
 ⟨proof⟩

lemma *cls-Ext: class tprg Ext = Some ExtCl*

⟨proof⟩

lemma *dynmethd-Ext-foo*:

$dynmethd\ tprg\ Base\ Ext\ (\!name = foo, parTs = [Class\ Base])$
 $= Some\ (Ext, snd\ Ext-foo)$
 ⟨proof⟩

lemma *Base-fields-accessible [simp]*:

$accfield\ tprg\ S\ Base$
 $= table-of((map\ (\lambda((n,d),f).(n,(d,f))))\ (DeclConcepts.fields\ tprg\ Base))$
 ⟨proof⟩

lemma *arr-member-of-Base*:

$tprg \vdash (Base, fdecl\ (arr,$
 $\quad (\!access = Public, static = True, type = PrimT\ Boolean.[])))$
member-of Base
 ⟨proof⟩

lemma *arr-member-in-Base*:

$tprg \vdash (Base, fdecl\ (arr,$
 $\quad (\!access = Public, static = True, type = PrimT\ Boolean.[])))$
member-in Base
 ⟨proof⟩

lemma *arr-member-of-Ext*:

$tprg \vdash (Base, fdecl\ (arr,$
 $\quad (\!access = Public, static = True, type = PrimT\ Boolean.[])))$

member-of Ext
 ⟨proof⟩

lemma *arr-member-in-Ext*:
 tprg⊢(Base, fdecl (arr,
 (⟦access = Public, static = True, type = PrimT Boolean.[]⟧)))
 member-in Ext
 ⟨proof⟩

lemma *Ext-fields-accessible[simp]*:
 accfield tprg S Ext
 = table-of((map (λ((n,d),f).(n,(d,f)))) (DeclConcepts.fields tprg Ext))
 ⟨proof⟩

lemma *arr-Base-dyn-accessible [simp]*:
 tprg⊢(Base, fdecl (arr, (⟦access=Public,static=True ,type=PrimT Boolean.[]⟧)))
 in Base dyn-accessible-from S
 ⟨proof⟩

lemma *arr-Ext-dyn-accessible[simp]*:
 tprg⊢(Base, fdecl (arr, (⟦access=Public,static=True ,type=PrimT Boolean.[]⟧)))
 in Ext dyn-accessible-from S
 ⟨proof⟩

lemma *array-of-PrimT-acc [simp]*:
 is-acc-type tprg java-lang (PrimT t.[])
 ⟨proof⟩

lemma *PrimT-acc [simp]*:
 is-acc-type tprg java-lang (PrimT t)
 ⟨proof⟩

lemma *Object-acc [simp]*:
 is-acc-class tprg java-lang Object
 ⟨proof⟩

well-formedness

lemma *wf-HasFoo*: wf-idecl tprg (HasFoo, HasFooInt)
 ⟨proof⟩

declare *member-is-static-simp [simp]*
declare *wt.Skip [rule del] wt.Init [rule del]*
 ⟨ML⟩

lemmas *wtIs = wt-Call wt-Super wt-FVar wt-StatRef wt-intros*
lemmas *daIs = assigned.select-convs da-Skip da-NewC da-Lit da-Super da.intros*

lemmas *Base-foo-defs = Base-foo-def foo-sig-def foo-mhead-def*
lemmas *Ext-foo-defs = Ext-foo-def foo-sig-def*

lemma *wf-Base-foo*: *wf-mdecl tprg Base Base-foo*
 ⟨*proof*⟩

lemma *wf-Ext-foo*: *wf-mdecl tprg Ext Ext-foo*
 ⟨*proof*⟩

declare *mhead-resTy-simp* [*simp add*]

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*:
 $appl-methds tprg S (ClassT Base) (\name=foo, parTs=[NT]) =$
 $\{((ClassT Base, (\access=Public,static=False,pars=[z],resT=Class Base))$
 $, [Class Base])\}$
 ⟨*proof*⟩

lemma *max-spec-Base-foo*: $max-spec tprg S (ClassT Base) (\name=foo,parTs=[NT]) =$
 $\{((ClassT Base, (\access=Public,static=False,pars=[z],resT=Class Base))$
 $, [Class Base])\}$
 ⟨*proof*⟩

well-typedness

schematic-goal *wt-test*: $(\prg=tprg,cls=Main,lcl=Map.empty(VName e \mapsto Class Base)) \vdash test ?pTs::\sqrt$

<proof>

definite assignment

schematic-goal *da-test*: (\downarrow prg=*tprg*,cls=*Main*,lcl=*Map.empty*(*VName* $e \mapsto$ *Class Base*))
 $\vdash \{ \} \gg \langle \text{test } ?pTs \rangle \gg (\downarrow nrm = \{ VName\ e \}, brk = \lambda\ l.\ UNIV)$

<proof>

execution

lemma *alloc-one*: $\bigwedge a\ obj.\ \llbracket \text{the } (new\text{-Addr } h) = a; \text{atleast-free } h\ (Suc\ n) \rrbracket \implies$
 $new\text{-Addr } h = \text{Some } a \wedge \text{atleast-free } (h(a \mapsto obj))\ n$

<proof>

declare *fvar-def2* [*simp*] *avar-def2* [*simp*] *init-lvars-def2* [*simp*]
declare *init-obj-def* [*simp*] *var-tys-def* [*simp*] *fields-table-def* [*simp*]
declare *BaseCl-def* [*simp*] *ExtCl-def* [*simp*] *Ext-foo-def* [*simp*]
Base-foo-defs [*simp*]

<ML>

lemmas *eval-Is* = *eval-Init eval-StatRef AbruptIs eval-intros*

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 == (\downarrow tag=*Arr* (*PrimT Boolean*) 2
,values= *Map.empty*(*Inr 0* \mapsto *Bool False*)(*Inr 1* \mapsto *Bool False*))

abbreviation

obj-b == (\downarrow tag=*CInst Ext*
,values=(*Map.empty*(*Inl* (*vec*, *Base*) \mapsto *Null*)
(*Inl* (*vec*, *Ext*) \mapsto *Intg 0*)))

abbreviation

obj-c == (\downarrow tag=*CInst* (*SXCpt NullPointer*),values=*CONST Map.empty*)

abbreviation *arr-N* == *Map.empty*(*Inl* (*arr*, *Base*) \mapsto *Null*)

abbreviation *arr-a* == *Map.empty*(*Inl* (*arr*, *Base*) \mapsto *Addr a*)

abbreviation

globs1 == *Map.empty*(*Inr Ext* \mapsto (\downarrow tag=*undefined*, values=*Map.empty*))
(*Inr Base* \mapsto (\downarrow tag=*undefined*, values=*arr-N*))
(*Inr Object* \mapsto (\downarrow tag=*undefined*, values=*Map.empty*))

abbreviation

globs2 == *Map.empty*(*Inr Ext* \mapsto (\downarrow tag=*undefined*, values=*Map.empty*))
(*Inr Object* \mapsto (\downarrow tag=*undefined*, values=*Map.empty*))
(*Inl a* \mapsto *obj-a*)
(*Inr Base* \mapsto (\downarrow tag=*undefined*, values=*arr-a*))

abbreviation *globs3* == *globs2*(*Inl b* \mapsto *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 $gext :: st \Rightarrow st \Rightarrow bool$ ($- \leq | -$ [71,71] 70) **where**
 $s \leq | s' \equiv \forall r. \forall obj \in globs\ s\ r. \exists obj' \in globs\ s'\ r. tag\ obj' = tag\ obj$

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*:

$\llbracket s \leq | s'; globs\ s\ r = Some\ obj \rrbracket$
 $\implies \exists obj'. globs\ s'\ r = Some\ obj' \wedge tag\ obj' = tag\ obj$
<proof>

lemma *rev-gext-objD*:

$\llbracket globs\ s\ r = Some\ obj; s \leq | s' \rrbracket$
 $\implies \exists obj'. globs\ s'\ r = Some\ obj' \wedge tag\ obj' = tag\ obj$
<proof>

lemma *init-class-obj-inited*:

$init\ class\ obj\ G\ C\ s1 \leq | s2 \implies inited\ C\ (globs\ s2)$
<proof>

lemma *gext-refl* [*intro!*, *simp*]: $s \leq | s$

<proof>

lemma *gext-gupd* [*simp*, *elim!*]: $\bigwedge s. globs\ s\ r = None \implies s \leq | gupd(r \mapsto x)s$

<proof>

lemma *gext-new* [*simp*, *elim!*]: $\bigwedge s. \text{globs } s \ r = \text{None} \implies s \leq | \text{init-obj } G \ \text{oi } r \ s$
 ⟨*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* :: $\text{prog} \Rightarrow \text{st} \Rightarrow \text{val} \Rightarrow \text{ty} \Rightarrow \text{bool}$ ($-, \vdash :: \preceq$ [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*: $G, s \vdash v :: \preceq \text{PrimT } \text{Boolean} \implies \exists b. v = \text{Bool } b$
 ⟨*proof*⟩

lemma *conf-litval* [*rule-format* (*no-asm*)]:

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*: $\llbracket \text{heap } s \ a = \text{Some obj}; G \vdash \text{obj-ty obj} \preceq T \rrbracket \Longrightarrow G, s \vdash \text{Addr } a :: \preceq T$
 ⟨proof⟩

lemma *defval-conf* [*rule-format* (*no-asm*), *elim*]:

is-type $G \ T \longrightarrow G, s \vdash \text{default-val } T :: \preceq T$
 ⟨proof⟩

lemma *conf-widen* [*rule-format* (*no-asm*), *elim*]:

$G \vdash T \preceq T' \Longrightarrow G, s \vdash x :: \preceq T \longrightarrow \text{ws-prog } G \longrightarrow G, s \vdash x :: \preceq T'$
 ⟨proof⟩

lemma *conf-gext* [*rule-format* (*no-asm*), *elim*]:

$G, s \vdash v :: \preceq T \longrightarrow s \leq | s' \longrightarrow G, s \uparrow v :: \preceq T$
 ⟨proof⟩

lemma *conf-list-widen* [*rule-format* (*no-asm*)]:

$\text{ws-prog } G \Longrightarrow$
 $\forall Ts \ Ts'. \text{list-all2 } (\text{conf } G \ s) \ \text{vs } Ts$
 $\longrightarrow G \vdash Ts \preceq Ts' \longrightarrow \text{list-all2 } (\text{conf } G \ s) \ \text{vs } Ts'$
 ⟨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

lconf :: *prog* \Rightarrow *st* \Rightarrow ('a, val) table \Rightarrow ('a, ty) table \Rightarrow bool (-, + -[: \preceq]- [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*: $\llbracket G, s \vdash \text{vs} :: \preceq Ts; Ts \ n = \text{Some } T \rrbracket \Longrightarrow G, s \vdash (\text{the } (\text{vs } n)) :: \preceq T$
 ⟨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*]: $G, \text{lupd}(vn \mapsto v) \text{s} \vdash l [:: \preceq] L = G, \text{s} \vdash l [:: \preceq] L$

<proof>

lemma *lconf-new*: $\llbracket L \text{ vn} = \text{None}; G, \text{s} \vdash l [:: \preceq] L \rrbracket \implies G, \text{s} \vdash l(vn \mapsto v) [:: \preceq] L$

<proof>

lemma *lconf-upd*: $\llbracket G, \text{s} \vdash l [:: \preceq] L; G, \text{s} \vdash v :: \preceq T; L \text{ vn} = \text{Some } T \rrbracket \implies$

$G, \text{s} \vdash l(vn \mapsto v) [:: \preceq] L$

<proof>

lemma *lconf-ext*: $\llbracket G, \text{s} \vdash l [:: \preceq] L; G, \text{s} \vdash v :: \preceq T \rrbracket \implies G, \text{s} \vdash l(vn \mapsto v) [:: \preceq] L(vn \mapsto T)$

<proof>

lemma *lconf-map-sum* [*simp*]:

$G, \text{s} \vdash l1 (+) l2 [:: \preceq] L1 (+) L2 = (G, \text{s} \vdash l1 [:: \preceq] L1 \wedge G, \text{s} \vdash l2 [:: \preceq] L2)$

<proof>

lemma *lconf-ext-list* [*rule-format (no-asm)*]:

$\bigwedge X. \llbracket G, \text{s} \vdash l [:: \preceq] L \rrbracket \implies$

$\forall vs \ Ts. \text{distinct } vns \longrightarrow \text{length } Ts = \text{length } vns$

$\longrightarrow \text{list-all2 } (\text{conf } G \ s) \ vs \ Ts \longrightarrow G, \text{s} \vdash l(vns[\mapsto]vs) [:: \preceq] L(vns[\mapsto]Ts)$

<proof>

lemma *lconf-deallocL*: $\llbracket G, \text{s} \vdash l [:: \preceq] L(vn \mapsto T); L \text{ vn} = \text{None} \rrbracket \implies G, \text{s} \vdash l [:: \preceq] L$

<proof>

lemma *lconf-gext* [*elim*]: $\llbracket G, \text{s} \vdash l [:: \preceq] L; s \leq |s^\eta \rrbracket \implies G, \text{s}^\eta \vdash l [:: \preceq] L$

<proof>

lemma *lconf-empty* [*simp, intro!*]: $G, \text{s} \vdash vs [:: \preceq] \text{Map.empty}$

<proof>

lemma *lconf-init-vals* [*intro!*]:

$\forall n. \forall T \in fs \ n : \text{is-type } G \ T \implies G, \text{s} \vdash \text{init-vals } fs [:: \preceq] fs$

<proof>

weak value list conformance

Only if the value is defined it has to conform to its type. This is the contribution of the definite assignment analysis to the notion of conformance. The definite assignment analysis ensures that the program only attempts to access local variables that actually have a defined value in the state. So conformance must only ensure that the defined values are of the right type, and not also that the value is defined.

definition

$wlconf :: prog \Rightarrow st \Rightarrow ('a, val) table \Rightarrow ('a, ty) table \Rightarrow bool$ $(-, \vdash -[\sim::\preceq])$ $- [71, 71, 71, 71] 70)$
where $G, s \vdash vs[\sim::\preceq] Ts = (\forall n. \forall T \in Ts n: \forall v \in vs n: G, s \vdash v::\preceq T)$

lemma $wlconfD$: $\llbracket G, s \vdash vs[\sim::\preceq] Ts; Ts n = Some T; vs n = Some v \rrbracket \Longrightarrow G, s \vdash v::\preceq T$
 $\langle proof \rangle$

lemma $wlconf-cong$ $[simp]$: $\bigwedge s. G, set-locals x s \vdash l[\sim::\preceq] L = G, s \vdash l[\sim::\preceq] L$
 $\langle proof \rangle$

lemma $wlconf-lupd$ $[simp]$: $G, lupd(vn \mapsto v) s \vdash l[\sim::\preceq] L = G, s \vdash l[\sim::\preceq] L$
 $\langle proof \rangle$

lemma $wlconf-upd$: $\llbracket G, s \vdash l[\sim::\preceq] L; G, s \vdash v::\preceq T; L vn = Some T \rrbracket \Longrightarrow$
 $G, s \vdash l(vn \mapsto v)[\sim::\preceq] L$
 $\langle proof \rangle$

lemma $wlconf-ext$: $\llbracket G, s \vdash l[\sim::\preceq] L; G, s \vdash v::\preceq T \rrbracket \Longrightarrow G, s \vdash l(vn \mapsto v)[\sim::\preceq] L(vn \mapsto T)$
 $\langle proof \rangle$

lemma $wlconf-map-sum$ $[simp]$:
 $G, s \vdash l1 (+) l2[\sim::\preceq] L1 (+) L2 = (G, s \vdash l1[\sim::\preceq] L1 \wedge G, s \vdash l2[\sim::\preceq] L2)$
 $\langle proof \rangle$

lemma $wlconf-ext-list$ $[rule-format (no-asm)]$:
 $\bigwedge X. \llbracket G, s \vdash l[\sim::\preceq] L \rrbracket \Longrightarrow$
 $\forall vs Ts. distinct\ vns \longrightarrow length\ Ts = length\ vns$
 $\longrightarrow list-all2 (conf\ G\ s)\ vs\ Ts \longrightarrow G, s \vdash l(vns[\mapsto] vs)[\sim::\preceq] L(vns[\mapsto] Ts)$
 $\langle proof \rangle$

lemma $wlconf-deallocL$: $\llbracket G, s \vdash l[\sim::\preceq] L(vn \mapsto T); L vn = None \rrbracket \Longrightarrow G, s \vdash l[\sim::\preceq] L$
 $\langle proof \rangle$

lemma $wlconf-geat$ $[elim]$: $\llbracket G, s \vdash l[\sim::\preceq] L; s \leq |s'| \rrbracket \Longrightarrow G, s' \vdash l[\sim::\preceq] L$
 $\langle proof \rangle$

lemma $wlconf-empty$ $[simp, intro!]$: $G, s \vdash vs[\sim::\preceq] Map.empty$
 $\langle proof \rangle$

lemma $wlconf-empty-vals$: $G, s \vdash Map.empty[\sim::\preceq] ts$
 $\langle proof \rangle$

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*:

$G, s \vdash l[\preceq]L \implies G, s \vdash l[\sim::\preceq]L$
 ⟨proof⟩

object conformance

definition

$oconf :: prog \Rightarrow st \Rightarrow obj \Rightarrow oref \Rightarrow bool \ (-, + :: \preceq \sqrt{-} \ [71, 71, 71, 71] \ 70)$ **where**
 $(G, s \vdash obj :: \preceq \sqrt{r}) = (G, s \vdash \text{values} \ obj[\preceq] \text{var-tys} \ G \ (\text{tag} \ obj) \ r \wedge$
 (case r of
 $\text{Heap } a \Rightarrow is\text{-type} \ G \ (\text{obj-ty} \ obj)$
 $| \text{Stat } C \Rightarrow \text{True}))$

lemma *oconf-is-type*: $G, s \vdash obj :: \preceq \sqrt{\text{Heap } a} \implies is\text{-type} \ G \ (\text{obj-ty} \ obj)$
 ⟨proof⟩

lemma *oconf-lconf*: $G, s \vdash obj :: \preceq \sqrt{r} \implies G, s \vdash \text{values} \ obj[\preceq] \text{var-tys} \ G \ (\text{tag} \ obj) \ r$
 ⟨proof⟩

lemma *oconf-cong [simp]*: $G, \text{set-locals} \ l \ s \vdash obj :: \preceq \sqrt{r} = G, s \vdash obj :: \preceq \sqrt{r}$
 ⟨proof⟩

lemma *oconf-init-obj-lemma*:

$\llbracket \wedge C \ c. \text{class} \ G \ C = \text{Some} \ c \implies \text{unique} \ (\text{DeclConcepts.fields} \ G \ C);$
 $\wedge C \ c \ f \ \text{fld}. \llbracket \text{class} \ G \ C = \text{Some} \ c;$
 $\text{table-of} \ (\text{DeclConcepts.fields} \ G \ C) \ f = \text{Some} \ \text{fld} \rrbracket$
 $\implies is\text{-type} \ G \ (\text{type} \ \text{fld});$
 (case r of
 $\text{Heap } a \Rightarrow is\text{-type} \ G \ (\text{obj-ty} \ obj)$
 $| \text{Stat } C \Rightarrow is\text{-class} \ G \ C)$
 $\rrbracket \implies G, s \vdash obj \ (\llbracket \text{values} := \text{init-vals} \ (\text{var-tys} \ G \ (\text{tag} \ obj) \ r) \rrbracket) :: \preceq \sqrt{r}$
 ⟨proof⟩

state conformance

definition

$conforms :: state \Rightarrow env' \Rightarrow bool \ (- :: \preceq - \ [71, 71] \ 70)$ **where**
 $xs :: \preceq E =$
 (let $(G, L) = E; s = \text{snd} \ xs; l = \text{locals} \ s$ in
 $(\forall r. \forall obj \in \text{globs} \ s \ r: G, s \vdash obj :: \preceq \sqrt{r}) \wedge G, s \vdash l \ [\sim::\preceq]L \wedge$
 $(\forall a. \text{fst} \ xs = \text{Some} \ (\text{Xcpt} \ (\text{Loc} \ a)) \longrightarrow G, s \vdash \text{Addr} \ a :: \preceq \text{Class} \ (\text{SXcpt} \ \text{Throwable})) \wedge$
 $(\text{fst} \ xs = \text{Some} \ (\text{Jump} \ \text{Ret}) \longrightarrow l \ \text{Result} \neq \text{None}))$

conforms

lemma *conforms-globsD*:

$\llbracket (x, s) :: \preceq (G, L); \text{globs} \ s \ r = \text{Some} \ obj \rrbracket \implies G, s \vdash obj :: \preceq \sqrt{r}$
 ⟨proof⟩

lemma conforms-localD: $(x, s)::\preceq(G, L) \implies G, s \vdash \text{locals } s[\sim::\preceq]L$
 ⟨proof⟩

lemma conforms-XcptLocD: $\llbracket (x, s)::\preceq(G, L); x = \text{Some } (Xcpt (Loc a)) \rrbracket \implies$
 $G, s \vdash \text{Addr } a::\preceq \text{Class } (SXcpt \text{ Throwable})$
 ⟨proof⟩

lemma conforms-RetD: $\llbracket (x, s)::\preceq(G, L); x = \text{Some } (Jump \text{ Ret}) \rrbracket \implies$
 $(\text{locals } s) \text{ Result} \neq \text{None}$
 ⟨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]:
 $j = \text{Ret} \longrightarrow \text{locals } s \text{ Result} \neq \text{None}$
 $\implies ((\text{Some } (Jump j), s)::\preceq(G, L)) = (\text{Norm } s::\preceq(G, L))$
 ⟨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]:
 $((\text{Some } (Error e), s)::\preceq(G, L)) = (\text{Norm } s::\preceq(G, L))$
 ⟨proof⟩

lemma conforms-raise-if [iff]:
 $((\text{raise-if } c \text{ xn } x, s)::\preceq(G, L)) = ((x, s)::\preceq(G, L))$
 ⟨proof⟩

lemma conforms-error-if [iff]:
 $((\text{error-if } c \text{ err } x, s)::\preceq(G, L)) = ((x, s)::\preceq(G, L))$
 ⟨proof⟩

lemma conforms-NormI: $(x, s)::\preceq(G, L) \implies \text{Norm } s::\preceq(G, L)$
 ⟨proof⟩

lemma conforms-absorb [rule-format]:
 $(a, b)::\preceq(G, L) \longrightarrow (\text{absorb } j \ a, b)::\preceq(G, L)$
 ⟨proof⟩

lemma conformsI: $\llbracket \forall r. \forall \text{obj} \in \text{globs } s \ r: G, s \vdash \text{obj}::\preceq \sqrt{r};$
 $G, s \vdash \text{locals } s[\sim::\preceq]L;$
 $\forall a. x = \text{Some } (Xcpt (Loc a)) \longrightarrow G, s \vdash \text{Addr } a::\preceq \text{Class } (SXcpt \text{ Throwable});$

$x = \text{Some } (\text{Jump Ret}) \longrightarrow \text{locals } s \text{ Result } \neq \text{None}] \Longrightarrow$
 $(x, s)::\preceq(G, L)$
 ⟨proof⟩

lemma conforms-xconf: $\llbracket (x, s)::\preceq(G, L);$
 $\forall a. x' = \text{Some } (\text{Xcpt } (\text{Loc } a)) \longrightarrow G, s \vdash \text{Addr } a::\preceq \text{Class } (\text{SXcpt } \text{Throwable});$
 $x' = \text{Some } (\text{Jump Ret}) \longrightarrow \text{locals } s \text{ Result } \neq \text{None}] \Longrightarrow$
 $(x', s)::\preceq(G, L)$
 ⟨proof⟩

lemma conforms-lupd:
 $\llbracket (x, s)::\preceq(G, L); L \text{ vn} = \text{Some } T; G, s \vdash v::\preceq T] \Longrightarrow (x, \text{lupd}(vn \mapsto v)s)::\preceq(G, L)$
 ⟨proof⟩

lemmas conforms-allocL-aux = conforms-localD [THEN wlconf-ext]

lemma conforms-allocL:
 $\llbracket (x, s)::\preceq(G, L); G, s \vdash v::\preceq T] \Longrightarrow (x, \text{lupd}(vn \mapsto v)s)::\preceq(G, L(vn \mapsto T))$
 ⟨proof⟩

lemmas conforms-deallocL-aux = conforms-localD [THEN wlconf-deallocL]

lemma conforms-deallocL: $\bigwedge s. \llbracket s::\preceq(G, L(vn \mapsto T)); L \text{ vn} = \text{None}] \Longrightarrow s::\preceq(G, L)$
 ⟨proof⟩

lemma conforms-geat: $\llbracket (x, s)::\preceq(G, L); s \leq |s';$
 $\forall r. \forall \text{obj} \in \text{globs } s' \ r: G, s \vdash \text{obj}::\preceq \sqrt{r};$
 $\text{locals } s' = \text{locals } s] \Longrightarrow (x, s')::\preceq(G, L)$
 ⟨proof⟩

lemma conforms-xgeat:
 $\llbracket (x, s)::\preceq(G, L); (x', s')::\preceq(G, L); s' \leq |s; \text{dom } (\text{locals } s') \subseteq \text{dom } (\text{locals } s)]$
 $\Longrightarrow (x', s)::\preceq(G, L)$
 ⟨proof⟩

lemma conforms-gupd: $\bigwedge \text{obj}. \llbracket (x, s)::\preceq(G, L); G, s \vdash \text{obj}::\preceq \sqrt{r}; s \leq |gupd(r \mapsto \text{obj})s]$
 $\Longrightarrow (x, \text{gupd}(r \mapsto \text{obj})s)::\preceq(G, L)$
 ⟨proof⟩

lemma conforms-upd-gobj: $\llbracket (x, s)::\preceq(G, L); \text{globs } s \ r = \text{Some } \text{obj};$
 $\text{var-tys } G \ (\text{tag } \text{obj}) \ r \ n = \text{Some } T; G, s \vdash v::\preceq T] \Longrightarrow (x, \text{upd-gobj } r \ n \ v \ s)::\preceq(G, L)$
 ⟨proof⟩

lemma conforms-set-locals:
 $\llbracket (x, s)::\preceq(G, L); G, s \vdash l[\sim::\preceq]L; x = \text{Some } (\text{Jump Ret}) \longrightarrow l \text{ Result } \neq \text{None}]$
 $\Longrightarrow (x, \text{set-locals } l \ s)::\preceq(G, L)$
 ⟨proof⟩

lemma *conforms-locals*:

$$\begin{aligned} & \llbracket (a,b)::\preceq(G, L); L x = \text{Some } T; \text{locals } b \ x \neq \text{None} \rrbracket \\ & \implies G, b \vdash \text{the } (\text{locals } b \ x)::\preceq T \\ & \langle \text{proof} \rangle \end{aligned}$$

lemma *conforms-return*:

$$\begin{aligned} & \wedge s'. \llbracket (x,s)::\preceq(G, L); (x',s')::\preceq(G, L'); s \leq |s'; x' \neq \text{Some } (\text{Jump Ret}) \rrbracket \implies \\ & \quad (x', \text{set-locals } (\text{locals } s) \ s')::\preceq(G, L) \\ & \langle \text{proof} \rangle \end{aligned}$$

end

Chapter 18

DefiniteAssignmentCorrect

1 Correctness of Definite Assignment

theory *DefiniteAssignmentCorrect* **imports** *WellForm Eval* **begin**

declare [[*simproc del: wt-expr wt-var wt-exprs wt-stmt*]]

lemma *sxalloc-no-jump*:

assumes *sxalloc*: $G \vdash s0 \text{ --sxalloc--} \rightarrow s1$ **and**
no-jmp: $\text{abrupt } s0 \neq \text{Some } (Jump\ j)$

shows $\text{abrupt } s1 \neq \text{Some } (Jump\ j)$

<proof>

lemma *sxalloc-no-jump'*:

assumes *sxalloc*: $G \vdash s0 \text{ --sxalloc--} \rightarrow s1$ **and**
jump: $\text{abrupt } s1 = \text{Some } (Jump\ j)$

shows $\text{abrupt } s0 = \text{Some } (Jump\ j)$

<proof>

lemma *halloc-no-jump*:

assumes *halloc*: $G \vdash s0 \text{ --halloc } oi \text{ --} \rightarrow a \rightarrow s1$ **and**
no-jmp: $\text{abrupt } s0 \neq \text{Some } (Jump\ j)$

shows $\text{abrupt } s1 \neq \text{Some } (Jump\ j)$

<proof>

lemma *halloc-no-jump'*:

assumes *halloc*: $G \vdash s0 \text{ --halloc } oi \text{ --} \rightarrow a \rightarrow s1$ **and**
jump: $\text{abrupt } s1 = \text{Some } (Jump\ j)$

shows $\text{abrupt } s0 = \text{Some } (Jump\ j)$

<proof>

lemma *Body-no-jump*:

assumes *eval*: $G \vdash s0 \text{ --Body } D\ c \text{ --} \rightarrow v \rightarrow s1$ **and**
jump: $\text{abrupt } s0 \neq \text{Some } (Jump\ j)$

shows $\text{abrupt } s1 \neq \text{Some } (Jump\ j)$

<proof>

lemma *Methd-no-jump*:

assumes *eval*: $G \vdash s0 \text{ --Methd } D\ sig \text{ --} \rightarrow v \rightarrow s1$ **and**

jump: $\text{abrupt } s0 \neq \text{Some } (\text{Jump } j)$
shows $\text{abrupt } s1 \neq \text{Some } (\text{Jump } j)$
 ⟨proof⟩

lemma *jumpNestingOkS-mono*:
assumes *jumpNestingOk-l'*: $\text{jumpNestingOkS } \text{jmps}' \ c$
and *subset*: $\text{jmps}' \subseteq \text{jmps}$
shows $\text{jumpNestingOkS } \text{jmps} \ c$
 ⟨proof⟩

corollary *jumpNestingOk-mono*:
assumes *jmpOk*: $\text{jumpNestingOk } \text{jmps}' \ t$
and *subset*: $\text{jmps}' \subseteq \text{jmps}$
shows $\text{jumpNestingOk } \text{jmps} \ t$
 ⟨proof⟩

lemma *assign-abrupt-propagation*:
assumes *f-ok*: $\text{abrupt } (f \ n \ s) \neq x$
and *ass*: $\text{abrupt } (\text{assign } f \ n \ s) = x$
shows $\text{abrupt } s = x$
 ⟨proof⟩

lemma *wt-init-comp-ty'*:
is-acc-type (*prg Env*) (*pid* (*cls Env*)) $T \implies \text{Env} \vdash \text{init-comp-ty } T :: \checkmark$
 ⟨proof⟩

lemma *fvar-upd-no-jump*:
assumes *upd*: $\text{upd} = \text{snd } (\text{fst } (\text{fvar } \text{statDeclC } \text{stat } \text{fn } a \ s'))$
and *noJmp*: $\text{abrupt } s \neq \text{Some } (\text{Jump } j)$
shows $\text{abrupt } (\text{upd } \text{val } s) \neq \text{Some } (\text{Jump } j)$
 ⟨proof⟩

lemma *avar-state-no-jump*:
assumes *jmp*: $\text{abrupt } (\text{snd } (\text{avar } G \ i \ a \ s)) = \text{Some } (\text{Jump } j)$
shows $\text{abrupt } s = \text{Some } (\text{Jump } j)$
 ⟨proof⟩

lemma *avar-upd-no-jump*:
assumes *upd*: $\text{upd} = \text{snd } (\text{fst } (\text{avar } G \ i \ a \ s'))$
and *noJmp*: $\text{abrupt } s \neq \text{Some } (\text{Jump } j)$
shows $\text{abrupt } (\text{upd } \text{val } s) \neq \text{Some } (\text{Jump } j)$
 ⟨proof⟩

The next theorem expresses: If jumps (breaks, continues, returns) are nested correctly, we won't find an unexpected jump in the result state of the evaluation. For example, a break can't leave its enclosing loop, an return can't leave its enclosing method. To prove this, the method call is critical. Although the wellformedness of the whole program guarantees that the jumps (breaks, continues and returns) are nested correctly in all method bodies, the call rule alone does not guarantee that I will call a method or even a class that is part of the program due to dynamic binding! To be able to ensure this we need a kind of conformance of the state, like in the typesafety proof. But then we will redo the typesafety proof here. It would be nice if we could find an easy precondition that will

guarantee that all calls will actually call classes and methods of the current program, which can be instantiated in the typesafety proof later on. To fix this problem, I have instrumented the semantic definition of a call to filter out any breaks in the state and to throw an error instead.

To get an induction hypothesis which is strong enough to perform the proof, we can't just assume *jumpNestingOk* for the empty set and conclude, that no jump at all will be in the resulting state, because the set is altered by the statements *Lab* and *While*.

The wellformedness of the program is used to ensure that for all classinitialisations and methods the nesting of jumps is wellformed, too.

theorem *jumpNestingOk-eval*:

assumes *eval*: $G \vdash s0 \text{ -t}\rightarrow (v, s1)$
and *jmpOk*: *jumpNestingOk* *jmps* *t*
and *wt*: $Env \vdash t :: T$
and *wf*: *wf-prog* *G*
and *G*: *prg* *Env* = *G*
and *no-jmp*: $\forall j. \text{abrupt } s0 = \text{Some } (Jump\ j) \rightarrow j \in \text{jmps}$
 (is ?Jmp *jmps* *s0*)
shows $(\forall j. \text{fst } s1 = \text{Some } (Jump\ j) \rightarrow j \in \text{jmps}) \wedge$
 (normal *s1* \rightarrow
 $(\forall w\ upd. v = In2\ (w, upd)$
 $\rightarrow (\forall s\ j\ val.$
 $\text{abrupt } s \neq \text{Some } (Jump\ j) \rightarrow$
 $\text{abrupt } (upd\ val\ s) \neq \text{Some } (Jump\ j)))$)
 (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* $\vdash s0 \text{ -t}\rightarrow (v, s1)$ **and**
 jmpOk: *jumpNestingOk* {} *t* **and**
 no-jmp: $\text{abrupt } s0 \neq \text{Some } (Jump\ j)$ **and**
 wt: $Env \vdash t :: T$ **and**
 wf: *wf-prog* (*prg* *Env*)

shows $\text{abrupt } s1 \neq \text{Some } (Jump\ j) \wedge$
 (normal *s1* $\rightarrow v = In2\ (w, upd)$
 $\rightarrow \text{abrupt } s \neq \text{Some } (Jump\ j')$
 $\rightarrow \text{abrupt } (upd\ val\ s) \neq \text{Some } (Jump\ j')$)

<proof>

lemmas *jumpNestingOk-eval-no-jumpE*

= *jumpNestingOk-eval-no-jump* [THEN *conjE*, *rule-format*]

corollary *eval-expression-no-jump*:

assumes *eval*: *prg* *Env* $\vdash s0 \text{ -e}\rightarrow v \rightarrow s1$ **and**
 no-jmp: $\text{abrupt } s0 \neq \text{Some } (Jump\ j)$ **and**
 wt: $Env \vdash e :: T$ **and**
 wf: *wf-prog* (*prg* *Env*)

shows $\text{abrupt } s1 \neq \text{Some } (Jump\ j)$

<proof>

corollary *eval-var-no-jump*:

assumes *eval*: *prg* *Env* $\vdash s0 \text{ -var}\Rightarrow (w, upd) \rightarrow s1$ **and**
 no-jmp: $\text{abrupt } s0 \neq \text{Some } (Jump\ j)$ **and**
 wt: $Env \vdash var :: T$ **and**
 wf: *wf-prog* (*prg* *Env*)

shows $\text{abrupt } s1 \neq \text{Some } (\text{Jump } j) \wedge$
 $(\text{normal } s1 \longrightarrow$
 $(\text{abrupt } s \neq \text{Some } (\text{Jump } j')$
 $\longrightarrow \text{abrupt } (\text{upd val } s) \neq \text{Some } (\text{Jump } j'))$
 ⟨proof⟩

lemmas $\text{eval-var-no-jumpE} = \text{eval-var-no-jump} [\text{THEN conjE, rule-format}]$

corollary $\text{eval-statement-no-jump}$:

assumes $\text{eval}: \text{prg Env} \vdash s0 -c \rightarrow s1$ **and**
 $\text{jmpOk}: \text{jumpNestingOkS } \{ \} c$ **and**
 $\text{no-jmp}: \text{abrupt } s0 \neq \text{Some } (\text{Jump } j)$ **and**
 $\text{wt}: \text{Env} \vdash c::\surd$ **and**
 $\text{wf}: \text{wf-prog } (\text{prg Env})$
shows $\text{abrupt } s1 \neq \text{Some } (\text{Jump } j)$
 ⟨proof⟩

corollary $\text{eval-expression-list-no-jump}$:

assumes $\text{eval}: \text{prg Env} \vdash s0 -es \dot{=} v \rightarrow s1$ **and**
 $\text{no-jmp}: \text{abrupt } s0 \neq \text{Some } (\text{Jump } j)$ **and**
 $\text{wt}: \text{Env} \vdash es::\dot{=} T$ **and**
 $\text{wf}: \text{wf-prog } (\text{prg Env})$
shows $\text{abrupt } s1 \neq \text{Some } (\text{Jump } j)$
 ⟨proof⟩

lemma $\text{union-subseteq-elim} [\text{elim}]: \llbracket A \cup B \subseteq C; \llbracket A \subseteq C; B \subseteq C \rrbracket \implies P \rrbracket \implies P$
 ⟨proof⟩

lemma $\text{dom-locals-halloc-mono}$:

assumes $\text{halloc}: G \vdash s0 -\text{halloc } oi \dot{=} a \rightarrow s1$
shows $\text{dom } (\text{locals } (\text{store } s0)) \subseteq \text{dom } (\text{locals } (\text{store } s1))$
 ⟨proof⟩

lemma $\text{dom-locals-sxalloc-mono}$:

assumes $\text{sxalloc}: G \vdash s0 -\text{sxalloc} \rightarrow s1$
shows $\text{dom } (\text{locals } (\text{store } s0)) \subseteq \text{dom } (\text{locals } (\text{store } s1))$
 ⟨proof⟩

lemma $\text{dom-locals-assign-mono}$:

assumes $f\text{-ok}: \text{dom } (\text{locals } (\text{store } s)) \subseteq \text{dom } (\text{locals } (\text{store } (f \ n \ s)))$
shows $\text{dom } (\text{locals } (\text{store } s)) \subseteq \text{dom } (\text{locals } (\text{store } (\text{assign } f \ n \ s)))$
 ⟨proof⟩

lemma $\text{dom-locals-lvar-mono}$:

$\text{dom } (\text{locals } (\text{store } s)) \subseteq \text{dom } (\text{locals } (\text{store } (\text{snd } (\text{lvar } vn \ s') \ \text{val } s)))$
 ⟨proof⟩

lemma $\text{dom-locals-fvar-vvar-mono}$:

$$\begin{aligned} & \text{dom (locals (store s))} \\ & \subseteq \text{dom (locals (store (snd (fst (fvar statDeclC stat fn a s')) val s)))} \\ & \langle \text{proof} \rangle \end{aligned}$$

lemma *dom-locals-fvar-mono*:

$$\begin{aligned} & \text{dom (locals (store s))} \\ & \subseteq \text{dom (locals (store (snd (fvar statDeclC stat fn a s))))} \\ & \langle \text{proof} \rangle \end{aligned}$$

lemma *dom-locals-avar-vvar-mono*:

$$\begin{aligned} & \text{dom (locals (store s))} \\ & \subseteq \text{dom (locals (store (snd (fst (avar G i a s')) val s)))} \\ & \langle \text{proof} \rangle \end{aligned}$$

lemma *dom-locals-avar-mono*:

$$\begin{aligned} & \text{dom (locals (store s))} \\ & \subseteq \text{dom (locals (store (snd (avar G i a s))))} \\ & \langle \text{proof} \rangle \end{aligned}$$

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*:

$$\begin{aligned} & \text{assumes } \text{eval: } G \vdash s0 \text{ -t> } \rightarrow (v, s1) \\ & \text{shows } \text{dom (locals (store s0))} \subseteq \text{dom (locals (store s1))} \wedge \\ & \quad (\forall vv. v = \text{In2 } vv \wedge \text{normal } s1 \\ & \quad \rightarrow (\forall s \text{ val. } \text{dom (locals (store s))} \\ & \quad \quad \subseteq \text{dom (locals (store ((snd vv) val s)))))) \\ & \langle \text{proof} \rangle \end{aligned}$$

lemma *dom-locals-eval-mono-elim*:

$$\begin{aligned} & \text{assumes } \text{eval: } G \vdash s0 \text{ -t> } \rightarrow (v, s1) \\ & \text{obtains } \text{dom (locals (store s0))} \subseteq \text{dom (locals (store s1))} \text{ and} \\ & \quad \wedge vv \text{ s val. } \llbracket v = \text{In2 } vv; \text{normal } s1 \rrbracket \\ & \quad \quad \Rightarrow \text{dom (locals (store s))} \\ & \quad \quad \subseteq \text{dom (locals (store ((snd vv) val s)))} \\ & \langle \text{proof} \rangle \end{aligned}$$

lemma *halloc-no-abrupt*:

$$\begin{aligned} & \text{assumes } \text{halloc: } G \vdash s0 \text{ -halloc } oi \text{ > } a \rightarrow s1 \text{ and} \\ & \quad \text{normal: normal } s1 \\ & \text{shows } \text{normal } s0 \\ & \langle \text{proof} \rangle \end{aligned}$$

lemma *salloc-mono-no-abrupt*:

assumes *salloc*: $G \vdash s0 - \text{salloc} \rightarrow s1$ **and**
normal: *normal s1*

shows *normal s0*

<proof>

lemma *union-subseteqI*: $\llbracket A \cup B \subseteq C; A' \subseteq A; B' \subseteq B \rrbracket \implies A' \cup B' \subseteq C$

<proof>

lemma *union-subseteqII*: $\llbracket A \cup B \subseteq C; A' \subseteq A \rrbracket \implies A' \cup B \subseteq C$

<proof>

lemma *union-subseteqIr*: $\llbracket A \cup B \subseteq C; B' \subseteq B \rrbracket \implies A \cup B' \subseteq C$

<proof>

lemma *subseteq-union-transl* [*trans*]: $\llbracket A \subseteq B; B \cup C \subseteq D \rrbracket \implies A \cup C \subseteq D$

<proof>

lemma *subseteq-union-transr* [*trans*]: $\llbracket A \subseteq B; C \cup B \subseteq D \rrbracket \implies A \cup C \subseteq D$

<proof>

lemma *union-subseteq-weaken*: $\llbracket A \cup B \subseteq C; \llbracket A \subseteq C; B \subseteq C \rrbracket \implies P \rrbracket \implies P$

<proof>

lemma *assigns-good-approx*:

assumes

eval: $G \vdash s0 - t \succ \rightarrow (v, s1)$ **and**

normal: *normal s1*

shows *assigns t* $\subseteq \text{dom}(\text{locals}(\text{store } s1))$

<proof>

corollary *assignsE-good-approx*:

assumes

eval: $\text{prg } Env \vdash s0 - e \succ v \rightarrow s1$ **and**

normal: *normal s1*

shows *assignsE e* $\subseteq \text{dom}(\text{locals}(\text{store } s1))$

<proof>

corollary *assignsV-good-approx*:

assumes

eval: $\text{prg } Env \vdash s0 - v = \succ vf \rightarrow s1$ **and**

normal: *normal s1*

shows *assignsV v* $\subseteq \text{dom}(\text{locals}(\text{store } s1))$

<proof>

corollary *assignsEs-good-approx*:

assumes

eval: $\text{prg } Env \vdash s0 - es \doteq \succ vs \rightarrow s1$ **and**

normal: *normal s1*

shows *assignsEs es* $\subseteq \text{dom}(\text{locals}(\text{store } s1))$

<proof>

lemma *constVal-eval*:

assumes *const*: $\text{constVal } e = \text{Some } c$ **and**
eval: $G \vdash \text{Norm } s0 \text{ } -e-\succ v \rightarrow s$
shows $v = c \wedge \text{normal } s$
 ⟨*proof*⟩

lemmas *constVal-eval-elim* = *constVal-eval* [THEN *conjE*]

lemma *eval-unop-type*:

typeof dt (*eval-unop unop v*) = *Some* (*PrimT* (*unop-type unop*))
 ⟨*proof*⟩

lemma *eval-binop-type*:

typeof dt (*eval-binop binop v1 v2*) = *Some* (*PrimT* (*binop-type binop*))
 ⟨*proof*⟩

lemma *constVal-Boolean*:

assumes *const*: $\text{constVal } e = \text{Some } c$ **and**
wt: $\text{Env} \vdash e :: -\text{PrimT Boolean}$
shows *typeof empty-dt c* = *Some* (*PrimT Boolean*)
 ⟨*proof*⟩

lemma *assigns-if-good-approx*:

assumes
eval: $\text{prg Env} \vdash s0 \text{ } -e-\succ b \rightarrow s1$ **and**
normal: *normal s1* **and**
bool: $\text{Env} \vdash e :: -\text{PrimT Boolean}$
shows *assigns-if* (*the-Bool b*) $e \subseteq \text{dom} (\text{locals } (\text{store } s1))$
 ⟨*proof*⟩

lemma *assigns-if-good-approx'*:

assumes *eval*: $G \vdash s0 \text{ } -e-\succ b \rightarrow s1$
and *normal*: *normal s1*
and *bool*: $(\text{prg} = G, \text{cls} = C, \text{lcl} = L) \vdash e :: -(\text{PrimT Boolean})$
shows *assigns-if* (*the-Bool b*) $e \subseteq \text{dom} (\text{locals } (\text{store } s1))$
 ⟨*proof*⟩

lemma *subset-Intl*: $A \subseteq C \implies A \cap B \subseteq C$
 ⟨*proof*⟩

lemma *subset-Intr*: $B \subseteq C \implies A \cap B \subseteq C$
 ⟨*proof*⟩

lemma *da-good-approx*:

assumes *eval*: $\text{prg Env} \vdash s0 \text{ } -t-\succ \rightarrow (v, s1)$ **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 (*normal s1* $\implies (\text{nrm } A \subseteq \text{dom} (\text{locals } (\text{store } s1)))$) \wedge

$$\begin{aligned}
& (\forall l. \text{abrupt } s1 = \text{Some } (\text{Jump } (\text{Break } l)) \wedge \text{normal } s0 \\
& \quad \longrightarrow (\text{brk } A \ l \subseteq \text{dom } (\text{locals } (\text{store } s1)))) \wedge \\
& (\text{abrupt } s1 = \text{Some } (\text{Jump } \text{Ret}) \wedge \text{normal } s0 \\
& \quad \longrightarrow \text{Result} \in \text{dom } (\text{locals } (\text{store } s1))) \\
& (\text{is } ?\text{NormalAssigned } s1 \ A \wedge ?\text{BreakAssigned } s0 \ s1 \ A \wedge ?\text{ResAssigned } s0 \ s1) \\
& \langle \text{proof} \rangle
\end{aligned}$$

lemma *da-good-approxE*:

assumes

$\text{prg } \text{Env} \vdash s0 \text{ } \text{-t}\succ \rightarrow (v, s1)$ **and** $\text{Env} \vdash t :: T$ **and**
 $\text{Env} \vdash \text{dom } (\text{locals } (\text{store } s0)) \gg t \gg A$ **and** $\text{wf-prog } (\text{prg } \text{Env})$

obtains

$\text{normal } s1 \implies \text{norm } A \subseteq \text{dom } (\text{locals } (\text{store } s1))$ **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 $\text{eval}: G \vdash s0 \text{ } \text{-t}\succ \rightarrow (v, s1)$

and $\text{wt}: (\text{prg}=G, \text{cls}=C, \text{lcl}=L) \vdash t :: T$

and $\text{da}: (\text{prg}=G, \text{cls}=C, \text{lcl}=L) \vdash \text{dom } (\text{locals } (\text{store } s0)) \gg t \gg A$

and $\text{wf}: \text{wf-prog } G$

obtains $\text{normal } s1 \implies \text{norm } A \subseteq \text{dom } (\text{locals } (\text{store } s1))$ **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 $\llbracket \text{simproc } \text{add}: \text{wt-expr } \text{wt-var } \text{wt-exprs } \text{wt-stmt} \rrbracket$

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*

<proof>

lemma *error-free-sxalloc:*

assumes *sxalloc: $G \vdash s0 \text{ --sxalloc } \rightarrow s1$ and *error-free-s0: error-free s0**

shows *error-free s1*

<proof>

lemma *error-free-check-field-access-eq:*

error-free (check-field-access $G \text{ acc}C \text{ statDecl}C \text{ fn stat } a \text{ } s$)

\implies *(check-field-access $G \text{ acc}C \text{ statDecl}C \text{ fn stat } a \text{ } s$) = s*

<proof>

lemma *error-free-check-method-access-eq:*

error-free (check-method-access $G \text{ acc}C \text{ stat}T \text{ mode sig } a' \text{ } s$)

\implies *(check-method-access $G \text{ acc}C \text{ stat}T \text{ mode sig } a' \text{ } s$) = s*

<proof>

lemma *error-free-FVar-lemma:*

error-free s

\implies *error-free (abupd (if stat then id else np a) s)*

<proof>

lemma *error-free-init-lvars [simp,intro]:*

error-free s \implies

error-free (init-lvars $G \text{ } C \text{ sig mode } a \text{ } pvs \text{ } s$)

<proof>

lemma *error-free-LVar-lemma*:
 $error\text{-}free\ s \implies error\text{-}free\ (assign\ (\lambda v. supd\ lupd(vn \mapsto v))\ w\ s)$
 ⟨proof⟩

lemma *error-free-throw* [*simp,intro*]:
 $error\text{-}free\ s \implies error\text{-}free\ (abupd\ (throw\ x)\ s)$
 ⟨proof⟩

result conformance

definition
 $assign\text{-}conforms :: st \Rightarrow (val \Rightarrow state \Rightarrow state) \Rightarrow ty \Rightarrow env' \Rightarrow bool\ (-\leq|\text{-}\leq\text{-}::\leq\text{-}\ [71,71,71,71]\ 70)$
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
 $rconf :: prog \Rightarrow lenv \Rightarrow st \Rightarrow term \Rightarrow vals \Rightarrow tys \Rightarrow bool\ (-, -, \vdash, \succ :: \leq\text{-}\ [71,71,71,71,71,71]\ 70)$
where
 $G, L, s \vdash t \succ v :: \leq T =$
 (case T of
 $Inl\ T \Rightarrow if\ (\exists\ var. t = In2\ var)$
 then $(\forall\ n. (the\text{-}In2\ t) = LVar\ n$
 $\longrightarrow (fst\ (the\text{-}In2\ v) = the\ (locals\ s\ n)) \wedge$
 $(locals\ s\ n \neq None \longrightarrow G, s \vdash fst\ (the\text{-}In2\ v) :: \leq T)) \wedge$
 $(\neg (\exists\ n. the\text{-}In2\ t = LVar\ n) \longrightarrow (G, s \vdash fst\ (the\text{-}In2\ v) :: \leq T)) \wedge$
 $(s \leq |snd\ (the\text{-}In2\ v) \leq T :: \leq (G, L))$
 else $G, s \vdash the\text{-}In1\ v :: \leq T$
 | $Inr\ Ts \Rightarrow list\text{-}all2\ (conf\ G\ s)\ (the\text{-}In3\ v)\ Ts$

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-In1* [*simp*]:
 $G, L, s \vdash In1\ ec \succ In1\ v :: \leq Inl\ T = G, s \vdash v :: \leq T$
 ⟨proof⟩

lemma *rconf-In2-no-LVar* [*simp*]:
 $\forall\ n. va \neq LVar\ n \implies$
 $G, L, s \vdash In2\ va \succ In2\ vf :: \leq Inl\ T = (G, s \vdash fst\ vf :: \leq T \wedge s \leq |snd\ vf \leq T :: \leq (G, L))$
 ⟨proof⟩

lemma *rconf-In2-LVar* [*simp*]:

$$\begin{aligned} & va = LVar\ n \implies \\ & G, L, s \vdash In2\ va \succ In2\ vf :: \preceq Inl\ T \\ & = ((fst\ vf = the\ (locals\ s\ n)) \wedge \\ & \quad (locals\ s\ n \neq None \longrightarrow G, s \vdash fst\ vf :: \preceq T) \wedge s \leq |snd\ vf \preceq T :: \preceq (G, L)) \\ & \langle proof \rangle \end{aligned}$$

lemma *rconf-In3* [*simp*]:

$$G, L, s \vdash In3\ es \succ In3\ vs :: \preceq Inr\ Ts = list-all2\ (\lambda v\ T.\ G, s \vdash v :: \preceq T)\ vs\ Ts$$

<proof>

fits and conf

lemma *conf-fits*: $G, s \vdash v :: \preceq T \implies G, s \vdash v\ fits\ T$

<proof>

lemma *fits-conf*:

$$\llbracket G, s \vdash v :: \preceq T; G \vdash T \preceq? T'; G, s \vdash v\ fits\ T'; ws-prog\ G \rrbracket \implies G, s \vdash v :: \preceq T'$$

<proof>

lemma *fits-Array*:

$$\llbracket G, s \vdash v :: \preceq T; G \vdash T'. [] \preceq T. []; G, s \vdash v\ fits\ T'; ws-prog\ G \rrbracket \implies G, s \vdash v :: \preceq T'$$

<proof>

gext

lemma *halloc-gext*: $\bigwedge s1\ s2.\ G \vdash s1 -halloc\ oi \succ a \rightarrow s2 \implies snd\ s1 \leq |snd\ s2$

<proof>

lemma *sxalloc-gext*: $\bigwedge s1\ s2.\ G \vdash s1 -sxalloc \rightarrow s2 \implies snd\ s1 \leq |snd\ s2$

<proof>

lemma *eval-gext-lemma* [*rule-format* (*no-asm*)]:

$$\begin{aligned} & G \vdash s -t \succ \rightarrow (w, s') \implies snd\ s \leq |snd\ s' \wedge (case\ w\ of \\ & \quad In1\ v \Rightarrow True \\ & \quad | In2\ vf \Rightarrow normal\ s \longrightarrow (\forall v\ x\ s.\ s \leq |snd\ (assign\ (snd\ vf)\ v\ (x, s))) \\ & \quad | In3\ vs \Rightarrow True) \\ & \langle proof \rangle \end{aligned}$$

lemma *evar-gext-f*:

$$G \vdash Norm\ s1 -e \succ vf \rightarrow s2 \implies s \leq |snd\ (assign\ (snd\ vf)\ v\ (x, s))$$

<proof>

lemmas *eval-gext* = *eval-gext-lemma* [*THEN* *conjunct1*]

lemma *eval-gext'*: $G \vdash (x1, s1) -t \succ \rightarrow (w, (x2, s2)) \implies s1 \leq |s2$

<proof>

lemma *init-yields-initd*: $G \vdash Norm\ s1 -Init\ C \rightarrow s2 \implies initd\ C\ s2$

<proof>

Lemmas

lemma *obj-ty-obj-class1*:

$\llbracket wf\text{-prog } G; is\text{-type } G (obj\text{-ty } obj) \rrbracket \implies is\text{-class } G (obj\text{-class } obj)$
 $\langle proof \rangle$

lemma *oconf-init-obj*:

$\llbracket wf\text{-prog } G;$
 $(case\ r\ of\ Heap\ a \Rightarrow is\text{-type } G (obj\text{-ty } obj) \mid Stat\ C \Rightarrow is\text{-class } G\ C)$
 $\rrbracket \implies G, s \vdash obj \ (values := init\text{-vals } (var\text{-tys } G (tag\ obj)\ r)) :: \preceq \sqrt{r}$
 $\langle proof \rangle$

lemma *conforms-newG*: $\llbracket globs\ s\ oref = None; (x, s) :: \preceq (G, L);$

$wf\text{-prog } G; case\ oref\ of\ Heap\ a \Rightarrow is\text{-type } G (obj\text{-ty } (\!tag=oi, values=vs))$
 $\mid Stat\ C \Rightarrow is\text{-class } G\ C \rrbracket \implies$
 $(x, init\text{-obj } G\ oi\ oref\ s) :: \preceq (G, L)$
 $\langle proof \rangle$

lemma *conforms-init-class-obj*:

$\llbracket (x, s) :: \preceq (G, L); wf\text{-prog } G; class\ G\ C = Some\ y; \neg\ inited\ C\ (globs\ s) \rrbracket \implies$
 $(x, init\text{-class-obj } G\ C\ s) :: \preceq (G, L)$
 $\langle proof \rangle$

lemma *fst-init-lvars[simp]*:

$fst\ (init\text{-lvars } G\ C\ sig\ (invmode\ m\ e)\ a'\ pvs\ (x, s)) =$
 $(if\ is\text{-static } m\ then\ x\ else\ (np\ a')\ x)$
 $\langle proof \rangle$

lemma *halloc-conforms*: $\bigwedge s1. \llbracket G \vdash s1 \text{-halloc } oi \succ a \rightarrow s2; wf\text{-prog } G; s1 :: \preceq (G, L);$

$is\text{-type } G (obj\text{-ty } (\!tag=oi, values=fs)) \rrbracket \implies s2 :: \preceq (G, L)$
 $\langle proof \rangle$

lemma *halloc-type-sound*:

$\bigwedge s1. \llbracket G \vdash s1 \text{-halloc } oi \succ a \rightarrow (x, s); wf\text{-prog } G; s1 :: \preceq (G, L);$
 $T = obj\text{-ty } (\!tag=oi, values=fs); is\text{-type } G\ T \rrbracket \implies$
 $(x, s) :: \preceq (G, L) \wedge (x = None \longrightarrow G, s \vdash Addr\ a :: \preceq T)$
 $\langle proof \rangle$

lemma *sxalloc-type-sound*:

$\bigwedge s1\ s2. \llbracket G \vdash s1 \text{-sxalloc} \rightarrow s2; wf\text{-prog } G \rrbracket \implies$
 $case\ fst\ s1\ of$
 $None \Rightarrow s2 = s1$
 $\mid Some\ abr \Rightarrow (case\ abr\ of$
 $Xcpt\ x \Rightarrow (\exists a. fst\ s2 = Some(Xcpt\ (Loc\ a)) \wedge$
 $(\forall L. s1 :: \preceq (G, L) \longrightarrow s2 :: \preceq (G, L)))$
 $\mid Jump\ j \Rightarrow s2 = s1$
 $\mid Error\ e \Rightarrow s2 = s1)$
 $\langle proof \rangle$

lemma *wt-init-comp-ty*:

is-acc-type G (*pid* C) $T \implies (\text{prg}=G, \text{cls}=C, \text{lcl}=L) \vdash \text{init-comp-ty } T :: \surd$
 ⟨proof⟩

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

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

definition

DynT-prop :: [*prog, inv-mode, qtname, ref-ty*] \Rightarrow *bool* ($\dashv\rightarrow \dashv\leq$ [71, 71, 71, 71] 70)

where

$G \vdash \text{mode} \rightarrow D \preceq t = (\text{mode} = \text{IntVir} \longrightarrow \text{is-class } G D \wedge$
 (if $(\exists T. t = \text{ArrayT } T)$ then $D = \text{Object}$ else $G \vdash \text{Class } D \preceq \text{RefT } t$)

lemma *DynT-propI*:

$\llbracket (x, s) :: \preceq (G, L); G, s \vdash a' :: \preceq \text{RefT } \text{statT}; \text{wf-prog } G; \text{mode} = \text{IntVir} \longrightarrow a' \neq \text{Null} \rrbracket$
 $\implies G \vdash \text{mode} \rightarrow \text{invocation-class } \text{mode } s a' \text{statT} \preceq \text{statT}$
 ⟨proof⟩

lemma *invocation-methd*:

$\llbracket \text{wf-prog } G; \text{statT} \neq \text{NullT};$
 $(\forall \text{statC}. \text{statT} = \text{ClassT } \text{statC} \longrightarrow \text{is-class } G \text{statC});$
 $(\forall I. \text{statT} = \text{IfaceT } I \longrightarrow \text{is-iface } G I \wedge \text{mode} \neq \text{SuperM});$
 $(\forall T. \text{statT} = \text{ArrayT } T \longrightarrow \text{mode} \neq \text{SuperM});$
 $G \vdash \text{mode} \rightarrow \text{invocation-class } \text{mode } s a' \text{statT} \preceq \text{statT};$
 $\text{dynlookup } G \text{statT} (\text{invocation-class } \text{mode } s a' \text{statT}) \text{sig} = \text{Some } m \rrbracket$
 $\implies \text{methd } G (\text{invocation-declclass } G \text{mode } s a' \text{statT } \text{sig}) \text{sig} = \text{Some } m$
 ⟨proof⟩

lemma *DynT-mheadsD*:

$\llbracket G \vdash \text{invmode } \text{sm } e \rightarrow \text{invC} \preceq \text{statT};$
 $\text{wf-prog } G; (\text{prg}=G, \text{cls}=C, \text{lcl}=L) \vdash e :: \neg \text{RefT } \text{statT};$
 $(\text{statDeclT}, \text{sm}) \in \text{mheads } G C \text{statT } \text{sig};$
 $\text{invC} = \text{invocation-class } (\text{invmode } \text{sm } e) s a' \text{statT};$
 $\text{declC} = \text{invocation-declclass } G (\text{invmode } \text{sm } e) s a' \text{statT } \text{sig}$
 $\rrbracket \implies$
 $\exists dm.$
 $\text{methd } G \text{declC } \text{sig} = \text{Some } dm \wedge \text{dynlookup } G \text{statT } \text{invC } \text{sig} = \text{Some } dm \wedge$
 $G \vdash \text{resTy } (\text{methd } dm) \preceq \text{resTy } \text{sm} \wedge$
 $\text{wf-mdecl } G \text{declC} (\text{sig}, \text{methd } dm) \wedge$
 $\text{declC} = \text{declclass } dm \wedge$
 $\text{is-static } dm = \text{is-static } \text{sm} \wedge$
 $\text{is-class } G \text{invC} \wedge \text{is-class } G \text{declC} \wedge G \vdash \text{invC} \preceq_C \text{declC} \wedge$
 (if $\text{invmode } \text{sm } e = \text{IntVir}$
 then $(\forall \text{statC}. \text{statT} = \text{ClassT } \text{statC} \longrightarrow G \vdash \text{invC} \preceq_C \text{statC})$
 else $(\exists \text{statC}. \text{statT} = \text{ClassT } \text{statC} \wedge G \vdash \text{statC} \preceq_C \text{declC})$
 $\vee (\forall \text{statC}. \text{statT} \neq \text{ClassT } \text{statC} \wedge \text{declC} = \text{Object})$) \wedge
 $\text{statDeclT} = \text{ClassT } (\text{declclass } dm)$
 ⟨proof⟩

corollary *DynT-mheadsE* [*consumes* γ]:

— Same as *DynT-mheadsD* but better suited for application in typesafety proof

assumes *invC-compatible*: $G \vdash \text{mode} \rightarrow \text{invC} \preceq \text{statT}$

and *wf*: *wf-prog* G

and *wt-e*: $(\text{prg}=G, \text{cls}=C, \text{lcl}=L) \vdash e :: \neg \text{RefT } \text{statT}$

and $mheads: (statDeclT, sm) \in mheads\ G\ C\ statT\ sig$
and $mode: mode = invmode\ sm\ e$
and $invC: invC = invocation-class\ mode\ s\ a'\ statT$
and $declC: declC = invocation-declclass\ G\ mode\ s\ a'\ statT\ sig$
and $dm: \bigwedge dm. \llbracket methd\ G\ declC\ sig = Some\ dm;$
 $dynlookup\ G\ statT\ invC\ sig = Some\ dm;$
 $G \vdash resTy\ (mthd\ dm) \preceq resTy\ sm;$
 $wf-mdecl\ G\ declC\ (sig, mthd\ dm);$
 $declC = declclass\ dm;$
 $is-static\ dm = is-static\ sm;$
 $is-class\ G\ invC; is-class\ G\ declC; G \vdash invC \preceq_C\ declC;$
 $(if\ invmode\ sm\ e = IntVir$
 $then\ (\forall\ statC. statT = ClassT\ statC \longrightarrow G \vdash invC \preceq_C\ statC)$
 $else\ (\ (\exists\ statC. statT = ClassT\ statC \wedge G \vdash statC \preceq_C\ declC)$
 $\vee\ (\forall\ statC. statT \neq ClassT\ statC \wedge declC = Object)) \wedge$
 $statDeclT = ClassT\ (declclass\ dm)) \rrbracket \Longrightarrow P$

shows P

$\langle proof \rangle$

lemma *DynT-conf*: $\llbracket G \vdash invocation-class\ mode\ s\ a'\ statT \preceq_C\ declC; wf-prog\ G;$
 $isrtype\ G\ (statT);$
 $G, s \vdash a' :: \preceq RefT\ statT; mode = IntVir \longrightarrow a' \neq Null;$
 $mode \neq IntVir \longrightarrow (\exists\ statC. statT = ClassT\ statC \wedge G \vdash statC \preceq_C\ declC)$
 $\vee\ (\forall\ statC. statT \neq ClassT\ statC \wedge declC = Object) \rrbracket$
 $\Longrightarrow G, s \vdash a' :: \preceq Class\ declC$

$\langle proof \rangle$

lemma *Ass-lemma*:

$\llbracket G \vdash Norm\ s0 - var = \triangleright (w, f) \rightarrow Norm\ s1; G \vdash Norm\ s1 - e \rightarrow v \rightarrow Norm\ s2;$
 $G, s2 \vdash v :: \preceq eT; s1 \leq |s2 \longrightarrow assign\ f\ v\ (Norm\ s2) :: \preceq (G, L) \rrbracket$
 $\Longrightarrow assign\ f\ v\ (Norm\ s2) :: \preceq (G, L) \wedge$
 $(normal\ (assign\ f\ v\ (Norm\ s2)) \longrightarrow G, store\ (assign\ f\ v\ (Norm\ s2)) \vdash v :: \preceq eT)$

$\langle proof \rangle$

lemma *Throw-lemma*: $\llbracket G \vdash tn \preceq_C\ SXcpt\ Throwable; wf-prog\ G; (x1, s1) :: \preceq (G, L);$
 $x1 = None \longrightarrow G, s1 \vdash a' :: \preceq Class\ tn \rrbracket \Longrightarrow (throw\ a'\ x1, s1) :: \preceq (G, L)$

$\langle proof \rangle$

lemma *Try-lemma*: $\llbracket G \vdash obj-ty\ (the\ (globs\ s1'\ (Heap\ a))) \preceq Class\ tn;$
 $(Some\ (Xcpt\ (Loc\ a)), s1') :: \preceq (G, L); wf-prog\ G \rrbracket$
 $\Longrightarrow Norm\ (lupd\ (vn \mapsto Addr\ a)\ s1') :: \preceq (G, L(vn \mapsto Class\ tn))$

$\langle proof \rangle$

lemma *Fin-lemma*:

$\llbracket G \vdash Norm\ s1 - c2 \rightarrow (x2, s2); wf-prog\ G; (Some\ a, s1) :: \preceq (G, L); (x2, s2) :: \preceq (G, L);$
 $dom\ (locals\ s1) \subseteq dom\ (locals\ s2) \rrbracket$
 $\Longrightarrow (abrupt-if\ True\ (Some\ a)\ x2, s2) :: \preceq (G, L)$

$\langle proof \rangle$

lemma *FVar-lemma1*:

$\llbracket table-of\ (DeclConcepts.fields\ G\ statC)\ (fn, statDeclC) = Some\ f ;$
 $x2 = None \longrightarrow G, s2 \vdash a :: \preceq Class\ statC; wf-prog\ G; G \vdash statC \preceq_C\ statDeclC;$

$statDeclC \neq Object;$
 $class\ G\ statDeclC = Some\ y;$ $(x2, s2)::\preceq(G, L); s1 \leq |s2;$
 $initd\ statDeclC\ (globs\ s1);$
 $(if\ static\ f\ then\ id\ else\ np\ a)\ x2 = None]$
 \implies
 $\exists\ obj.\ globs\ s2\ (if\ static\ f\ then\ Inr\ statDeclC\ else\ Inl\ (the-Addr\ a))$
 $\quad = Some\ obj \wedge$
 $var-tys\ G\ (tag\ obj)\ (if\ static\ f\ then\ Inr\ statDeclC\ else\ Inl\ (the-Addr\ a))$
 $\quad (Inl\ (fn, statDeclC)) = Some\ (type\ f)$
 $\langle proof \rangle$

lemma *FVar-lemma2: error-free state*

$\implies error-free$
 $(assign$
 $\quad (\lambda v.\ supd$
 $\quad\quad (upd-gobj$
 $\quad\quad\quad (if\ static\ field\ then\ Inr\ statDeclC$
 $\quad\quad\quad\quad else\ Inl\ (the-Addr\ a))$
 $\quad\quad\quad (Inl\ (fn,\ statDeclC))\ v))$
 $\quad w\ state)$

$\langle proof \rangle$

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*]

$\langle ML \rangle$

lemma *FVar-lemma:*

$\llbracket ((v, f), Norm\ s2') = fvar\ statDeclC\ (static\ field)\ fn\ a\ (x2, s2);$
 $G \vdash statC \preceq_C statDeclC;$
 $table-of\ (DeclConcepts.fields\ G\ statC)\ (fn,\ statDeclC) = Some\ field;$
 $wf-prog\ G;$
 $x2 = None \implies G, s2 \vdash a::\preceq Class\ statC;$
 $statDeclC \neq Object;$ $class\ G\ statDeclC = Some\ y;$
 $(x2, s2)::\preceq(G, L); s1 \leq |s2;$ $initd\ statDeclC\ (globs\ s1) \rrbracket \implies$
 $G, s2 \uparrow v::\preceq type\ field \wedge s2' \leq |f \preceq type\ field::\preceq(G, L)$

$\langle proof \rangle$

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:* $\llbracket globs\ s\ (Inl\ a) = Some\ obj; tag\ obj = Arr\ ty\ i;$

$the-Intg\ i'\ in-bounds\ i; wf-prog\ G; G \vdash ty.\llbracket \preceq Tb.\llbracket;$ $Norm\ s::\preceq(G, L)$
 $\rrbracket \implies G, s \vdash the\ ((values\ obj)\ (Inr\ (the-Intg\ i'))::\preceq Tb$

$\langle proof \rangle$

lemma *obj-split:* $\exists\ t\ vs.\ obj = (\{tag=t, values=vs\})$

$\langle proof \rangle$

lemma *AVar-lemma2: error-free state*

$\implies error-free$
 $(assign$

$(\lambda v (x, s').$
 $((\text{raise-if } (\neg G, s \vdash v \text{ fits } T) \text{ ArrStore}) x,$
 $\text{upd-gobj } (\text{Inl } a) (\text{Inr } (\text{the-Intg } i)) v s')$
 $w \text{ state})$
 $\langle \text{proof} \rangle$

lemma *AVar-lemma*: $\llbracket \text{wf-prog } G; G \vdash (x1, s1) -e2 \rightarrow i \rightarrow (x2, s2);$
 $((v, f), \text{Norm } s2') = \text{avar } G \ i \ a \ (x2, s2); x1 = \text{None} \longrightarrow G, s1 \vdash a :: \preceq \text{Ta} \cdot \llbracket ;$
 $(x2, s2) :: \preceq (G, L); s1 \leq |s2 \rrbracket \implies G, s2 \vdash v :: \preceq \text{Ta} \wedge s2' \leq |f \preceq \text{Ta} :: \preceq (G, L)$
 $\langle \text{proof} \rangle$

Call

lemma *conforms-init-lvars-lemma*: $\llbracket \text{wf-prog } G;$
 $\text{wf-mhead } G \ P \ \text{sig } \text{mh};$
 $\text{list-all2 } (\text{conf } G \ s) \ \text{pvs } \ \text{pTsa}; G \vdash \text{pTsa} [\preceq] (\text{parTs } \text{sig}) \rrbracket \implies$
 $G, s \vdash \text{Map.empty } (\text{pars } \text{mh} [\mapsto] \text{pvs})$
 $[\sim :: \preceq] \text{table-of lvars } (\text{pars } \text{mh} [\mapsto] \text{parTs } \text{sig})$
 $\langle \text{proof} \rangle$

lemma *lconf-map-lname* [*simp*]:
 $G, s \vdash (\text{case-lname } l1 \ l2) [\preceq] (\text{case-lname } L1 \ L2)$
 $=$
 $(G, s \vdash l1 [\preceq] L1 \wedge G, s \vdash (\lambda x :: \text{unit} . l2) [\preceq] (\lambda x :: \text{unit} . L2))$
 $\langle \text{proof} \rangle$

lemma *wlconf-map-lname* [*simp*]:
 $G, s \vdash (\text{case-lname } l1 \ l2) [\sim :: \preceq] (\text{case-lname } L1 \ L2)$
 $=$
 $(G, s \vdash l1 [\sim :: \preceq] L1 \wedge G, s \vdash (\lambda x :: \text{unit} . l2) [\sim :: \preceq] (\lambda x :: \text{unit} . L2))$
 $\langle \text{proof} \rangle$

lemma *lconf-map-ename* [*simp*]:
 $G, s \vdash (\text{case-ename } l1 \ l2) [\preceq] (\text{case-ename } L1 \ L2)$
 $=$
 $(G, s \vdash l1 [\preceq] L1 \wedge G, s \vdash (\lambda x :: \text{unit} . l2) [\preceq] (\lambda x :: \text{unit} . L2))$
 $\langle \text{proof} \rangle$

lemma *wlconf-map-ename* [*simp*]:
 $G, s \vdash (\text{case-ename } l1 \ l2) [\sim :: \preceq] (\text{case-ename } L1 \ L2)$
 $=$
 $(G, s \vdash l1 [\sim :: \preceq] L1 \wedge G, s \vdash (\lambda x :: \text{unit} . l2) [\sim :: \preceq] (\lambda x :: \text{unit} . L2))$
 $\langle \text{proof} \rangle$

lemma *defval-conf1* [*rule-format* (*no-asm*), *elim*]:
 $\text{is-type } G \ T \longrightarrow (\exists v \in \text{Some } (\text{default-val } T): G, s \vdash v :: \preceq T)$
 $\langle \text{proof} \rangle$

lemma *np-no-jump*: $x \neq \text{Some } (\text{Jump } j) \implies (\text{np } a') \ x \neq \text{Some } (\text{Jump } j)$

⟨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*:

[[*wf-mhead* G (*pid declC*) *sig* (*mhead* (*mthd dm*))]; *wf-prog* G ;
list-all2 (*conf* G s) *pvs pTsa*; $G \vdash pTsa[\preceq](parTs sig)$;
 $(x, s) :: \preceq(G, L)$;
methd G *declC* *sig* = *Some dm*;
isrtype G *statT*;
 $G \vdash invC \preceq_C declC$;
 $G, s \vdash a' :: \preceq RefT statT$;
invmode (*mhd sm*) $e = IntVir \longrightarrow a' \neq Null$;
invmode (*mhd sm*) $e \neq IntVir \longrightarrow$
 $(\exists statC. statT = ClassT statC \wedge G \vdash statC \preceq_C declC)$
 $\vee (\forall statC. statT \neq ClassT statC \wedge declC = Object)$;
invC = *invocation-class* (*invmode* (*mhd sm*) e) s a' *statT*;
declC = *invocation-declclass* G (*invmode* (*mhd sm*) e) s a' *statT* *sig*;
 $x \neq Some$ (*Jump Ret*)

]] \implies

init-lvars G *declC* *sig* (*invmode* (*mhd sm*) e) a'
pvs $(x, s) :: \preceq(G, \lambda k.$
 (*case* k of
 EName $e \Rightarrow$ (*case* e of
 VNam v
 \Rightarrow (*table-of* (*lcls* (*mbody* (*mthd dm*)))
 (*pars* (*mthd dm*) \mapsto *parTs* *sig*)) v
 | *Res* \Rightarrow *Some* (*resTy* (*mthd dm*)))
 | *This* \Rightarrow *if* (*is-static* (*mthd sm*))
 then None *else Some* (*Class 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*: *wf-prog* G **and**
not-Null: $\neg stat \longrightarrow a \neq Null$ **and**
conform-a: $G, (store s) \vdash a :: \preceq Class statC$ **and**
conform-s: $s :: \preceq(G, L)$ **and**
normal-s: *normal* s **and**
wt-e: ($\text{prg} = G, \text{cls} = accC, \text{lcl} = L$) $\vdash e :: -Class statC$ **and**
f: *accfield* G *accC* *statC* *fn* = *Some f* **and**
dynC: *if* *stat* *then* *dynC* = *declclass f*
 else *dynC* = *obj-class* (*lookup-obj* (*store s*) a) **and**
stat: *if* *stat* *then* (*is-static* f) *else* ($\neg is-static$ f)
shows *table-of* (*DeclConcepts.fields* G *dynC*) (*fn, declclass f*) = *Some* (*fld f*) \wedge
 $G \vdash Field$ fn f *in* *dynC* *dyn-accessible-from* *accC*

⟨proof⟩

lemma *error-free-field-access*:

assumes *accfield*: $\text{accfield } G \text{ accC statC fn} = \text{Some } (\text{statDeclC}, f)$ **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: $(v, s2') = \text{fvar statDeclC } (\text{is-static } f) \text{ fn } a \text{ } s2$ **and**
wf: *wf-prog* *G*
shows *check-field-access* *G accC statDeclC fn (is-static f) a s2' = s2'*
 ⟨*proof*⟩

lemma *call-access-ok*:

assumes *invC-prop*: $G \vdash \text{invmode statM } e \rightarrow \text{invC} \preceq \text{statT}$
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*: $\text{invC} = \text{invocation-class } (\text{invmode statM } e) \text{ } s \text{ } a \text{ } \text{statT}$
shows $\exists \text{ dynM}. \text{dynlookup } G \text{ statT invC sig} = \text{Some dynM} \wedge$
 $G \vdash \text{Method sig dynM in invC dyn-accessible-from accC}$
 ⟨*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: $\text{max-spec } G \text{ accC statT } (\text{name} = \text{mn}, \text{parTs} = \text{pTs})$
 $= \{((\text{statDeclT}, \text{statM}), \text{pTs}')\}$ **and**
conf-s2: $s2 :: \preceq(G, L)$ **and**
conf-a: $\text{normal } s1 \implies G, \text{store } s1 \vdash a :: \preceq \text{RefT statT}$ **and**
invProp: $\text{normal } s3 \implies$
 $G \vdash \text{invmode statM } e \rightarrow \text{invC} \preceq \text{statT}$ **and**
s3: $s3 = \text{init-lvars } G \text{ invDeclC } (\text{name} = \text{mn}, \text{parTs} = \text{pTs}')$
 $(\text{invmode statM } e) \text{ } a \text{ } vs \text{ } s2$ **and**
invC: $\text{invC} = \text{invocation-class } (\text{invmode statM } e) (\text{store } s2) \text{ } a \text{ } \text{statT}$ **and**
invDeclC: $\text{invDeclC} = \text{invocation-declclass } G (\text{invmode statM } e) (\text{store } s2)$
 $\text{ } a \text{ } \text{statT } (\text{name} = \text{mn}, \text{parTs} = \text{pTs}')$ **and**
wf: *wf-prog* *G*
shows *check-method-access* *G accC statT (invmode statM e) (name=mn, parTs=pTs')* *a 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*:

$\bigwedge \text{tab } qs \text{ } x \text{ } y. \text{length } ps = \text{length } qs$
 $\implies \text{tab}(ps[\mapsto]qs)(x \mapsto y) = \text{tab}(ps@[x][\mapsto]qs@[y])$
 ⟨*proof*⟩

lemma *map-upd-cong*: $\text{tab} = \text{tab}' \implies \text{tab}(x \mapsto y) = \text{tab}'(x \mapsto y)$

⟨*proof*⟩

lemma *map-upd-cong-ext*: $tab\ z = tab'\ z \implies (tab(x \mapsto y))\ z = (tab'(x \mapsto y))\ z$
 ⟨proof⟩

lemma *map-upds-cong*: $tab = tab' \implies tab(xs[\mapsto]ys) = tab'(xs[\mapsto]ys)$
 ⟨proof⟩

lemma *map-upds-cong-ext*:
 $\bigwedge\ tab\ tab'\ ys.\ tab\ z = tab'\ z \implies (tab(xs[\mapsto]ys))\ z = (tab'(xs[\mapsto]ys))\ z$
 ⟨proof⟩

lemma *map-upd-override*: $(tab(x \mapsto y))\ x = (tab'(x \mapsto y))\ x$
 ⟨proof⟩

lemma *map-upds-eq-length-suffix*: $\bigwedge\ tab\ qs.$
 $length\ ps = length\ qs \implies tab(ps@xs[\mapsto]qs) = tab(ps[\mapsto]qs)(xs[\mapsto][])$
 ⟨proof⟩

lemma *map-upds-upds-eq-length-prefix-simp*:
 $\bigwedge\ tab\ qs.\ length\ ps = length\ qs$
 $\implies tab(ps[\mapsto]qs)(xs[\mapsto]ys) = tab(ps@xs[\mapsto]qs@ys)$
 ⟨proof⟩

lemma *map-upd-cut-irrelevant*:
 $\llbracket (tab(x \mapsto y))\ vn = Some\ el;\ (tab'(x \mapsto y))\ vn = None \rrbracket$
 $\implies tab\ vn = Some\ el$
 ⟨proof⟩

lemma *map-upd-Some-expand*:
 $\llbracket tab\ vn = Some\ z \rrbracket$
 $\implies \exists\ z.\ (tab(x \mapsto y))\ vn = Some\ z$
 ⟨proof⟩

lemma *map-upds-Some-expand*:
 $\bigwedge\ tab\ ys\ z.\ \llbracket tab\ vn = Some\ z \rrbracket$
 $\implies \exists\ z.\ (tab(xs[\mapsto]ys))\ vn = Some\ z$
 ⟨proof⟩

lemma *map-upd-Some-swap*:
 $(tab(r \mapsto w)(u \mapsto v))\ vn = Some\ z \implies \exists\ z.\ (tab(u \mapsto v)(r \mapsto w))\ vn = Some\ z$
 ⟨proof⟩

lemma *map-upd-None-swap*:
 $(tab(r \mapsto w)(u \mapsto v))\ vn = None \implies (tab(u \mapsto v)(r \mapsto w))\ vn = None$
 ⟨proof⟩

lemma *map-eq-upd-eq*: $tab\ vn = tab'\ vn \implies (tab(x \mapsto y))\ vn = (tab'(x \mapsto y))\ vn$
 ⟨proof⟩

lemma *map-upd-in-expansion-map-swap*:
 $\llbracket (tab(x \mapsto y))\ vn = Some\ z; tab\ vn \neq Some\ z \rrbracket$
 $\implies (tab'(x \mapsto y))\ vn = Some\ z$
 ⟨proof⟩

lemma *map-upds-in-expansion-map-swap*:
 $\bigwedge tab\ tab'\ ys\ z. \llbracket (tab(xs[\mapsto]ys))\ vn = Some\ z; tab\ vn \neq Some\ z \rrbracket$
 $\implies (tab'(xs[\mapsto]ys))\ vn = Some\ z$
 ⟨proof⟩

lemma *map-upds-Some-swap*:
assumes $r\text{-}u$: $(tab(r \mapsto w)(u \mapsto v)(xs[\mapsto]ys))\ vn = Some\ z$
shows $\exists z. (tab(u \mapsto v)(r \mapsto w)(xs[\mapsto]ys))\ vn = Some\ z$
 ⟨proof⟩

lemma *map-upds-Some-insert*:
assumes z : $(tab(xs[\mapsto]ys))\ vn = Some\ z$
shows $\exists z. (tab(u \mapsto v)(xs[\mapsto]ys))\ vn = Some\ z$
 ⟨proof⟩

lemma *map-upds-None-cut*:
assumes *expand-None*: $(tab(xs[\mapsto]ys))\ vn = None$
shows $tab\ vn = None$
 ⟨proof⟩

lemma *map-upds-cut-irrelevant*:
 $\bigwedge tab\ tab'\ ys. \llbracket (tab(xs[\mapsto]ys))\ vn = Some\ el; (tab'(xs[\mapsto]ys))\ vn = None \rrbracket$
 $\implies tab\ vn = Some\ el$
 ⟨proof⟩

lemma *dom-vname-split*:
 $dom\ (case\ lname\ (case\ ename\ (tab(x \mapsto y)(xs[\mapsto]ys))\ a)\ b)$
 $= dom\ (case\ lname\ (case\ ename\ (tab(x \mapsto y))\ a)\ b) \cup$
 $dom\ (case\ lname\ (case\ ename\ (tab(xs[\mapsto]ys))\ a)\ b)$
(is $?List\ x\ xs\ y\ ys = ?Hd\ x\ y \cup ?Tl\ xs\ ys$
 ⟨proof⟩

lemma *dom-map-upd*: $\bigwedge tab. dom\ (tab(x \mapsto y)) = dom\ tab \cup \{x\}$
 ⟨proof⟩

lemma *dom-map-upds*: $\bigwedge tab\ ys. length\ xs = length\ ys$
 $\implies dom\ (tab(xs[\mapsto]ys)) = dom\ tab \cup set\ xs$
 ⟨proof⟩

lemma *dom-case-ename-None-simp*:

$\text{dom} (\text{case-ename } \text{vname-tab } \text{None}) = \text{VName } \text{' } (\text{dom } \text{vname-tab})$
 ⟨proof⟩

lemma *dom-case-ename-Some-simp*:

$\text{dom} (\text{case-ename } \text{vname-tab } (\text{Some } a)) = \text{VName } \text{' } (\text{dom } \text{vname-tab}) \cup \{\text{Res}\}$
 ⟨proof⟩

lemma *dom-case-lname-None-simp*:

$\text{dom} (\text{case-lname } \text{ename-tab } \text{None}) = \text{EName } \text{' } (\text{dom } \text{ename-tab})$
 ⟨proof⟩

lemma *dom-case-lname-Some-simp*:

$\text{dom} (\text{case-lname } \text{ename-tab } (\text{Some } a)) = \text{EName } \text{' } (\text{dom } \text{ename-tab}) \cup \{\text{This}\}$
 ⟨proof⟩

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*:

$f \text{' } g \text{' } A = (f \circ g) \text{' } A$
 ⟨proof⟩

lemma *dom-locals-init-lvars*:

assumes *m*: $m = (\text{methd } (\text{the } (\text{methd } G \ C \ \text{sig})))$
assumes *len*: $\text{length } (\text{pars } m) = \text{length } \text{pvs}$
shows $\text{dom} (\text{locals } (\text{store } (\text{init-lvars } G \ C \ \text{sig } (\text{invmode } m \ e) \ a \ \text{pvs } s)))$
 = *parameters m*

⟨proof⟩

lemma *da-e2-BinOp*:

assumes *da*: $(\text{prg} = G, \text{cls} = \text{accC}, \text{lcl} = L)$
 $\text{dom} (\text{locals } (\text{store } s0)) \gg \langle \text{BinOp } \text{binop } e1 \ e2 \rangle_e \gg A$
and *wt-e1*: $(\text{prg} = G, \text{cls} = \text{accC}, \text{lcl} = L) \vdash e1 :: -e1T$
and *wt-e2*: $(\text{prg} = G, \text{cls} = \text{accC}, \text{lcl} = L) \vdash e2 :: -e2T$
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*: $G \vdash s0 \ -e1 \ -\gamma \ v1 \rightarrow s1$
and *conf-v1*: $G, \text{store } s1 \vdash v1 :: \preceq e1T$
and *wf*: *wf-prog G*
shows $\exists E2. (\text{prg} = G, \text{cls} = \text{accC}, \text{lcl} = L) \vdash \text{dom} (\text{locals } (\text{store } s1))$
 $\gg (\text{if need-second-arg } \text{binop } v1 \text{ then } \langle e2 \rangle_e \text{ else } \langle \text{Skip} \rangle_s) \gg E2$

⟨proof⟩

main proof of type safety

lemma *eval-type-sound*:

assumes *eval*: $G \vdash s0 \ -t\gamma \rightarrow (v, s1)$

and $wt: (\downarrow prg = G, cls = accC, lcl = L) \vdash t :: T$
and $da: (\downarrow prg = G, cls = accC, lcl = L) \vdash \text{dom} (\text{locals} (\text{store } s0)) \gg t \gg A$
and $wf: wf\text{-prog } G$
and $conf\text{-}s0: s0 :: \preceq (G, L)$
shows $s1 :: \preceq (G, L) \wedge (\text{normal } s1 \longrightarrow G, L, \text{store } s1 \vdash t \gg v :: \preceq T) \wedge$
 $(\text{error-free } s0 = \text{error-free } s1)$
 $\langle \text{proof} \rangle$

corollary *eval-type-soundE* [consumes 5]:

assumes $eval: G \vdash s0 \text{ -}t \gg \rightarrow (v, s1)$
and $conf: s0 :: \preceq (G, L)$
and $wt: (\downarrow prg = G, cls = accC, lcl = L) \vdash t :: T$
and $da: (\downarrow prg = G, cls = accC, lcl = L) \vdash \text{dom} (\text{locals} (\text{snd } s0)) \gg t \gg A$
and $wf: wf\text{-prog } G$
and $elim: \llbracket s1 :: \preceq (G, L); \text{normal } s1 \Longrightarrow G, L, \text{snd } s1 \vdash t \gg v :: \preceq T; \text{error-free } s0 = \text{error-free } s1 \rrbracket \Longrightarrow P$
shows P
 $\langle \text{proof} \rangle$

corollary *eval-ts*:

$\llbracket G \vdash s \text{ -}e \text{ -} \gg v \rightarrow s'; wf\text{-prog } G; s :: \preceq (G, L); (\downarrow prg = G, cls = C, lcl = L) \vdash e :: \text{-}T;$
 $(\downarrow prg = G, cls = C, lcl = L) \vdash \text{dom} (\text{locals} (\text{store } s)) \gg \text{In1 } e \gg A \rrbracket$
 $\Longrightarrow s' :: \preceq (G, L) \wedge (\text{normal } s' \longrightarrow G, \text{store } s' \vdash v :: \preceq T) \wedge$
 $(\text{error-free } s = \text{error-free } s')$
 $\langle \text{proof} \rangle$

corollary *evals-ts*:

$\llbracket G \vdash s \text{ -}es \text{ -} \gg vs \rightarrow s'; wf\text{-prog } G; s :: \preceq (G, L); (\downarrow prg = G, cls = C, lcl = L) \vdash es :: \text{-}Ts;$
 $(\downarrow prg = G, cls = C, lcl = L) \vdash \text{dom} (\text{locals} (\text{store } s)) \gg \text{In3 } es \gg A \rrbracket$
 $\Longrightarrow s' :: \preceq (G, L) \wedge (\text{normal } s' \longrightarrow \text{list-all2} (\text{conf } G (\text{store } s')) \text{ vs } Ts) \wedge$
 $(\text{error-free } s = \text{error-free } s')$
 $\langle \text{proof} \rangle$

corollary *evar-ts*:

$\llbracket G \vdash s \text{ -}v \text{ -} \gg vf \rightarrow s'; wf\text{-prog } G; s :: \preceq (G, L); (\downarrow prg = G, cls = C, lcl = L) \vdash v :: T;$
 $(\downarrow prg = G, cls = C, lcl = L) \vdash \text{dom} (\text{locals} (\text{store } s)) \gg \text{In2 } v \gg A \rrbracket \Longrightarrow$
 $s' :: \preceq (G, L) \wedge (\text{normal } s' \longrightarrow G, L, (\text{store } s) \vdash \text{In2 } v \gg \text{In2 } vf :: \preceq \text{In1 } T) \wedge$
 $(\text{error-free } s = \text{error-free } s')$
 $\langle \text{proof} \rangle$

theorem *exec-ts*:

$\llbracket G \vdash s \text{ -}c \rightarrow s'; wf\text{-prog } G; s :: \preceq (G, L); (\downarrow prg = G, cls = C, lcl = L) \vdash c :: \checkmark;$
 $(\downarrow prg = G, cls = C, lcl = L) \vdash \text{dom} (\text{locals} (\text{store } s)) \gg \text{In1r } c \gg A \rrbracket$
 $\Longrightarrow s' :: \preceq (G, L) \wedge (\text{error-free } s \longrightarrow \text{error-free } s')$
 $\langle \text{proof} \rangle$

lemma *wf-eval-Fin*:

assumes $wf: wf\text{-prog } G$
and $wt\text{-}c1: (\downarrow prg = G, cls = C, lcl = L) \vdash \text{In1r } c1 :: \text{In1} (\text{PrimT } \text{Void})$
and $da\text{-}c1: (\downarrow prg = G, cls = C, lcl = L) \vdash \text{dom} (\text{locals} (\text{store} (\text{Norm } s0))) \gg \text{In1r } c1 \gg A$
and $conf\text{-}s0: \text{Norm } s0 :: \preceq (G, L)$
and $eval\text{-}c1: G \vdash \text{Norm } s0 \text{ -}c1 \rightarrow (x1, s1)$
and $eval\text{-}c2: G \vdash \text{Norm } s1 \text{ -}c2 \rightarrow s2$
and $s3: s3 = \text{abupd} (\text{abrupt-if } (x1 \neq \text{None}) x1) s2$
shows $G \vdash \text{Norm } s0 \text{ -}c1 \text{ Finally } c2 \rightarrow s3$
 $\langle \text{proof} \rangle$

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 *eval*: $G \vdash s0 \text{ -t} \rightarrow (v, s1)$
and *wt*: $(\text{prg}=G, \text{cls}=\text{acc}C, \text{lcl}=L) \vdash t :: T$
and *da*: $(\text{prg}=G, \text{cls}=\text{acc}C, \text{lcl}=L) \vdash \text{dom} (\text{locals} (\text{store } s0)) \gg t \gg A$
and *wf*: *wf-prog* G
and *abrupt*: $\bigwedge s t \text{ abr } L \text{ acc}C T A.$
 $\llbracket (\text{prg}=G, \text{cls}=\text{acc}C, \text{lcl}=L) \vdash t :: T;$
 $\llbracket (\text{prg}=G, \text{cls}=\text{acc}C, \text{lcl}=L) \vdash \text{dom} (\text{locals} (\text{store} (\text{Some } \text{abr}, s))) \gg t \gg A$
 $\rrbracket \implies P L \text{ acc}C (\text{Some } \text{abr}, s) t (\text{undefined3 } t) (\text{Some } \text{abr}, s)$
and *skip*: $\bigwedge s L \text{ acc}C. P L \text{ acc}C (\text{Norm } s) \langle \text{Skip} \rangle_s \diamond (\text{Norm } s)$
and *expr*: $\bigwedge e s0 s1 v L \text{ acc}C eT E.$
 $\llbracket (\text{prg}=G, \text{cls}=\text{acc}C, \text{lcl}=L) \vdash e :: -eT;$
 $\llbracket (\text{prg}=G, \text{cls}=\text{acc}C, \text{lcl}=L)$
 $\vdash \text{dom} (\text{locals} (\text{store} ((\text{Norm } s0)::\text{state}))) \gg \langle e \rangle_e \gg E;$
 $P L \text{ acc}C (\text{Norm } s0) \langle e \rangle_e [v]_e s1 \rrbracket$
 $\implies P L \text{ acc}C (\text{Norm } s0) \langle \text{Expr } e \rangle_s \diamond s1$
and *lab*: $\bigwedge c l s0 s1 L \text{ acc}C C.$
 $\llbracket (\text{prg}=G, \text{cls}=\text{acc}C, \text{lcl}=L) \vdash c :: \sqrt{;};$
 $\llbracket (\text{prg}=G, \text{cls}=\text{acc}C, \text{lcl}=L)$
 $\vdash \text{dom} (\text{locals} (\text{store} ((\text{Norm } s0)::\text{state}))) \gg \langle c \rangle_s \gg C;$
 $P L \text{ acc}C (\text{Norm } s0) \langle c \rangle_s \diamond s1 \rrbracket$
 $\implies P L \text{ acc}C (\text{Norm } s0) \langle l \cdot c \rangle_s \diamond (\text{abupd } (\text{absorb } l) s1)$
and *comp*: $\bigwedge c1 c2 s0 s1 s2 L \text{ acc}C C1.$
 $\llbracket G \vdash \text{Norm } s0 \text{ -c1} \rightarrow s1; G \vdash s1 \text{ -c2} \rightarrow s2;$
 $\llbracket (\text{prg}=G, \text{cls}=\text{acc}C, \text{lcl}=L) \vdash c1 :: \sqrt{;};$
 $\llbracket (\text{prg}=G, \text{cls}=\text{acc}C, \text{lcl}=L) \vdash c2 :: \sqrt{;};$
 $\llbracket (\text{prg}=G, \text{cls}=\text{acc}C, \text{lcl}=L) \vdash$
 $\text{dom} (\text{locals} (\text{store} ((\text{Norm } s0)::\text{state}))) \gg \langle c1 \rangle_s \gg C1;$
 $P L \text{ acc}C (\text{Norm } s0) \langle c1 \rangle_s \diamond s1;$
 $\bigwedge Q. \llbracket \text{normal } s1;$
 $\bigwedge C2. \llbracket (\text{prg}=G, \text{cls}=\text{acc}C, \text{lcl}=L)$
 $\vdash \text{dom} (\text{locals} (\text{store } s1)) \gg \langle c2 \rangle_s \gg C2;$
 $P L \text{ acc}C s1 \langle c2 \rangle_s \diamond s2 \rrbracket \implies Q$
 $\rrbracket \implies Q$
 $\rrbracket \implies P L \text{ acc}C (\text{Norm } s0) \langle c1;; c2 \rangle_s \diamond s2$
and *if*: $\bigwedge b c1 c2 e s0 s1 s2 L \text{ acc}C E.$
 $\llbracket G \vdash \text{Norm } s0 \text{ -e} \text{ -} \triangleright b \rightarrow s1;$
 $G \vdash s1 \text{ -(if the-Bool } b \text{ then } c1 \text{ else } c2) \rightarrow s2;$
 $\llbracket (\text{prg}=G, \text{cls}=\text{acc}C, \text{lcl}=L) \vdash e :: -\text{Prim}T \text{ Boolean};$
 $\llbracket (\text{prg}=G, \text{cls}=\text{acc}C, \text{lcl}=L) \vdash (\text{if the-Bool } b \text{ then } c1 \text{ else } c2) :: \sqrt{;};$
 $\llbracket (\text{prg}=G, \text{cls}=\text{acc}C, \text{lcl}=L) \vdash$
 $\text{dom} (\text{locals} (\text{store} ((\text{Norm } s0)::\text{state}))) \gg \langle e \rangle_e \gg E;$
 $P L \text{ acc}C (\text{Norm } s0) \langle e \rangle_e [b]_e s1;$
 $\bigwedge Q. \llbracket \text{normal } s1;$
 $\bigwedge C. \llbracket (\text{prg}=G, \text{cls}=\text{acc}C, \text{lcl}=L) \vdash (\text{dom} (\text{locals} (\text{store } s1)))$
 $\gg \langle \text{if the-Bool } b \text{ then } c1 \text{ else } c2 \rangle_s \gg C;$
 $P L \text{ acc}C s1 \langle \text{if the-Bool } b \text{ then } c1 \text{ else } c2 \rangle_s \diamond s2$

```

      ] ==> Q
    ] ==> Q
  ] ==> P L accC (Norm s0) (If(e) c1 Else c2)_s ◇ s2
shows P L accC s0 t v s1
<proof>

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
  (+- -->----> '(-, -) [61,61,80,61,0,0] 60)
and evaln :: [prog, state, var, vvar, nat, state] => bool
  (+- --=>----> - [61,61,90,61,61,61] 60)
and eval-n :: [prog, state, expr, val, nat, state] => bool
  (+- ---->----> - [61,61,80,61,61,61] 60)
and evalsn :: [prog, state, expr list, val list, nat, state] => bool
  (+- --=>----> - [61,61,61,61,61,61] 60)
and execn :: [prog, state, stmt, nat, state] => bool
  (+- ---->----> - [61,61,65, 61,61] 60)
for G :: prog
```

where

```
G⊢s -c -n→ s' ≡ G⊢s -In1r c>-n→ (◇ , s')
| G⊢s -e->v -n→ s' ≡ G⊢s -In1l e>-n→ (In1 v , s')
| G⊢s -e=>vf -n→ s' ≡ G⊢s -In2 e>-n→ (In2 vf , s')
| G⊢s -e≠>v -n→ s' ≡ G⊢s -In3 e>-n→ (In3 v , s')
```

— propagation of abrupt completion

```
| Abrupt: G⊢(Some xc,s) -t>-n→ (undefined3 t,(Some xc,s))
```

— evaluation of variables

```
| LVar: G⊢Norm s -LVar vn=>lvar vn s-n→ Norm s
```

- | *FVar*: $\llbracket G \vdash \text{Norm } s0 \text{ -Init statDeclC -}n \rightarrow s1; G \vdash s1 \text{ -}e \rightarrow a \text{ -}n \rightarrow s2;$
 $(v, s2') = \text{fvar statDeclC stat fn } a \text{ } s2;$
 $s3 = \text{check-field-access } G \text{ accC statDeclC fn stat } a \text{ } s2 \rrbracket \implies$
 $G \vdash \text{Norm } s0 \text{ -}\{accC, statDeclC, stat\}e..fn \rightarrow v \text{ -}n \rightarrow s3$
- | *AVar*: $\llbracket G \vdash \text{Norm } s0 \text{ -}e1 \rightarrow a \text{ -}n \rightarrow s1; G \vdash s1 \text{ -}e2 \rightarrow i \text{ -}n \rightarrow s2;$
 $(v, s2') = \text{avar } G \text{ } i \text{ } a \text{ } s2 \rrbracket \implies$
 $G \vdash \text{Norm } s0 \text{ -}e1.[e2] \rightarrow v \text{ -}n \rightarrow s2'$
- evaluation of expressions
- | *NewC*: $\llbracket G \vdash \text{Norm } s0 \text{ -Init } C \text{ -}n \rightarrow s1;$
 $G \vdash s1 \text{ -halloc } (C \text{Inst } C) \rightarrow a \rightarrow s2 \rrbracket \implies$
 $G \vdash \text{Norm } s0 \text{ -NewC } C \rightarrow \text{Addr } a \text{ -}n \rightarrow s2$
- | *NewA*: $\llbracket G \vdash \text{Norm } s0 \text{ -init-comp-ty } T \text{ -}n \rightarrow s1; G \vdash s1 \text{ -}e \rightarrow i' \text{ -}n \rightarrow s2;$
 $G \vdash \text{abupd } (\text{check-neg } i') \text{ } s2 \text{ -halloc } (\text{Arr } T \text{ } (\text{the-Intg } i')) \rightarrow a \rightarrow s3 \rrbracket \implies$
 $G \vdash \text{Norm } s0 \text{ -New } T[e] \rightarrow \text{Addr } a \text{ -}n \rightarrow s3$
- | *Cast*: $\llbracket G \vdash \text{Norm } s0 \text{ -}e \rightarrow v \text{ -}n \rightarrow s1;$
 $s2 = \text{abupd } (\text{raise-if } (\neg G, \text{snd } s1 \vdash v \text{ fits } T) \text{ } \text{ClassCast}) \text{ } s1 \rrbracket \implies$
 $G \vdash \text{Norm } s0 \text{ -Cast } T \text{ } e \rightarrow v \text{ -}n \rightarrow s2$
- | *Inst*: $\llbracket G \vdash \text{Norm } s0 \text{ -}e \rightarrow v \text{ -}n \rightarrow s1;$
 $b = (v \neq \text{Null} \wedge G, \text{store } s1 \vdash v \text{ fits } \text{RefT } T) \rrbracket \implies$
 $G \vdash \text{Norm } s0 \text{ -}e \text{ InstOf } T \rightarrow \text{Bool } b \text{ -}n \rightarrow s1$
- | *Lit*: $G \vdash \text{Norm } s \text{ -Lit } v \rightarrow v \text{ -}n \rightarrow \text{Norm } s$
- | *UnOp*: $\llbracket G \vdash \text{Norm } s0 \text{ -}e \rightarrow v \text{ -}n \rightarrow s1 \rrbracket$
 $\implies G \vdash \text{Norm } s0 \text{ -UnOp } \text{unop } e \rightarrow (\text{eval-unop } \text{unop } v) \text{ -}n \rightarrow s1$
- | *BinOp*: $\llbracket G \vdash \text{Norm } s0 \text{ -}e1 \rightarrow v1 \text{ -}n \rightarrow s1;$
 $G \vdash s1 \text{ -(if need-second-arg binop } v1 \text{ then (In1l } e2) \text{ else (In1r Skip))}$
 $\rightarrow \text{-}n \rightarrow (\text{In1 } v2, s2) \rrbracket$
 $\implies G \vdash \text{Norm } s0 \text{ -BinOp } \text{binop } e1 \text{ } e2 \rightarrow (\text{eval-binop } \text{binop } v1 \text{ } v2) \text{ -}n \rightarrow s2$
- | *Super*: $G \vdash \text{Norm } s \text{ -Super} \rightarrow \text{val-this } s \text{ -}n \rightarrow \text{Norm } s$
- | *Acc*: $\llbracket G \vdash \text{Norm } s0 \text{ -}va \rightarrow (v, f) \text{ -}n \rightarrow s1 \rrbracket \implies$
 $G \vdash \text{Norm } s0 \text{ -Acc } va \rightarrow v \text{ -}n \rightarrow s1$
- | *Ass*: $\llbracket G \vdash \text{Norm } s0 \text{ -}va \rightarrow (w, f) \text{ -}n \rightarrow s1;$
 $G \vdash s1 \text{ -}e \rightarrow v \text{ -}n \rightarrow s2 \rrbracket \implies$
 $G \vdash \text{Norm } s0 \text{ -}va := e \rightarrow v \text{ -}n \rightarrow \text{assign } f \text{ } v \text{ } s2$
- | *Cond*: $\llbracket G \vdash \text{Norm } s0 \text{ -}e0 \rightarrow b \text{ -}n \rightarrow s1;$
 $G \vdash s1 \text{ -(if the-Bool } b \text{ then } e1 \text{ else } e2) \rightarrow v \text{ -}n \rightarrow s2 \rrbracket \implies$
 $G \vdash \text{Norm } s0 \text{ -}e0 \text{ ? } e1 : e2 \rightarrow v \text{ -}n \rightarrow s2$
- | *Call*:
 $\llbracket G \vdash \text{Norm } s0 \text{ -}e \rightarrow a' \text{ -}n \rightarrow s1; G \vdash s1 \text{ -args} \rightarrow vs \text{ -}n \rightarrow s2;$
 $D = \text{invocation-declclass } G \text{ mode } (\text{store } s2) \text{ } a' \text{ } \text{statT } (\text{name}=\text{mn}, \text{parTs}=\text{pTs});$
 $s3 = \text{init-lvars } G \text{ } D \text{ } (\text{name}=\text{mn}, \text{parTs}=\text{pTs}) \text{ mode } a' \text{ } vs \text{ } s2;$
 $s3' = \text{check-method-access } G \text{ accC statT mode } (\text{name}=\text{mn}, \text{parTs}=\text{pTs}) \text{ } a' \text{ } s3;$

- $$\begin{array}{l} G\vdash s3' - \text{Methd } D \ (\text{name}=\text{mn}, \text{parTs}=\text{pTs}) \text{-}\succ v - n \rightarrow s4 \\ \Downarrow \\ \implies \\ G\vdash \text{Norm } s0 \text{-}\{\text{accC}, \text{statT}, \text{mode}\}e.\text{mn}(\{\text{pTs}\}\text{args}) \text{-}\succ v - n \rightarrow (\text{restore-lvars } s2 \ s4) \end{array}$$
- | *Methd*: $\llbracket G\vdash \text{Norm } s0 \text{-body } G \ D \ \text{sig} \text{-}\succ v - n \rightarrow s1 \rrbracket \implies$
 $G\vdash \text{Norm } s0 \text{-Methd } D \ \text{sig} \text{-}\succ v - \text{Suc } n \rightarrow s1$
- | *Body*: $\llbracket G\vdash \text{Norm } s0 \text{-Init } D \text{-}n \rightarrow s1; G\vdash s1 \text{-}c \text{-}n \rightarrow s2;$
 $s3 = (\text{if } (\exists l. \text{abrupt } s2 = \text{Some } (\text{Jump } (\text{Break } l))) \vee$
 $\text{abrupt } s2 = \text{Some } (\text{Jump } (\text{Cont } l)))$
 $\text{then } \text{abupd } (\lambda x. \text{Some } (\text{Error CrossMethodJump})) \ s2$
 $\text{else } s2 \rrbracket \implies$
 $G\vdash \text{Norm } s0 \text{-Body } D \ c$
 $\text{-}\succ \text{the } (\text{locals } (\text{store } s2) \ \text{Result}) \text{-}n \rightarrow \text{abupd } (\text{absorb } \text{Ret}) \ s3$
- evaluation of expression lists
- | *Nil*:
 $G\vdash \text{Norm } s0 \text{-}\llbracket \text{::}\succ \rrbracket \text{-}n \rightarrow \text{Norm } s0$
- | *Cons*: $\llbracket G\vdash \text{Norm } s0 \text{-}e \text{-}\succ v \text{-}n \rightarrow s1;$
 $G\vdash \quad s1 \text{-}es \text{::}\succ vs \text{-}n \rightarrow s2 \rrbracket \implies$
 $G\vdash \text{Norm } s0 \text{-}e \# es \text{::}\succ v \# vs \text{-}n \rightarrow s2$
- execution of statements
- | *Skip*:
 $G\vdash \text{Norm } s \text{-Skip} \text{-}n \rightarrow \text{Norm } s$
- | *Expr*: $\llbracket G\vdash \text{Norm } s0 \text{-}e \text{-}\succ v \text{-}n \rightarrow s1 \rrbracket \implies$
 $G\vdash \text{Norm } s0 \text{-Expr } e \text{-}n \rightarrow s1$
- | *Lab*: $\llbracket G\vdash \text{Norm } s0 \text{-}c \text{-}n \rightarrow s1 \rrbracket \implies$
 $G\vdash \text{Norm } s0 \text{-}l \cdot c \text{-}n \rightarrow \text{abupd } (\text{absorb } l) \ s1$
- | *Comp*: $\llbracket G\vdash \text{Norm } s0 \text{-}c1 \text{-}n \rightarrow s1;$
 $G\vdash \quad s1 \text{-}c2 \text{-}n \rightarrow s2 \rrbracket \implies$
 $G\vdash \text{Norm } s0 \text{-}c1;; c2 \text{-}n \rightarrow s2$
- | *If*: $\llbracket G\vdash \text{Norm } s0 \text{-}e \text{-}\succ b \text{-}n \rightarrow s1;$
 $G\vdash \quad s1 \text{-}(\text{if the-Bool } b \text{ then } c1 \text{ else } c2) \text{-}n \rightarrow s2 \rrbracket \implies$
 $G\vdash \text{Norm } s0 \text{-If}(e) \ c1 \ \text{Else } c2 \text{-}n \rightarrow s2$
- | *Loop*: $\llbracket G\vdash \text{Norm } s0 \text{-}e \text{-}\succ b \text{-}n \rightarrow s1;$
 $\text{if the-Bool } b$
 $\text{then } (G\vdash s1 \text{-}c \text{-}n \rightarrow s2 \wedge$
 $G\vdash (\text{abupd } (\text{absorb } (\text{Cont } l)) \ s2) \text{-}l \cdot \text{While}(e) \ c \text{-}n \rightarrow s3)$
 $\text{else } s3 = s1 \rrbracket \implies$
 $G\vdash \text{Norm } s0 \text{-}l \cdot \text{While}(e) \ c \text{-}n \rightarrow s3$
- | *Jmp*: $G\vdash \text{Norm } s \text{-Jmp } j \text{-}n \rightarrow (\text{Some } (\text{Jump } j), \ s)$
- | *Throw*: $\llbracket G\vdash \text{Norm } s0 \text{-}e \text{-}\succ a' \text{-}n \rightarrow s1 \rrbracket \implies$
 $G\vdash \text{Norm } s0 \text{-Throw } e \text{-}n \rightarrow \text{abupd } (\text{throw } a') \ s1$
- | *Try*: $\llbracket G\vdash \text{Norm } s0 \text{-}c1 \text{-}n \rightarrow s1; G\vdash s1 \text{-}x\text{alloc} \rightarrow s2;$
 $\text{if } G, s2 \vdash \text{catch } tn \text{ then } G\vdash \text{new-xcpt-var } vn \ s2 \text{-}c2 \text{-}n \rightarrow s3 \text{ else } s3 = s2 \rrbracket$
 \implies

$$G \vdash \text{Norm } s0 - \text{Try } c1 \text{ Catch}(tn \text{ } vn) \text{ } c2 - n \rightarrow s3$$

| *Fin*: $\llbracket G \vdash \text{Norm } s0 - c1 - n \rightarrow (x1, s1);$
 $G \vdash \text{Norm } s1 - c2 - n \rightarrow s2;$
 $s3 = (\text{if } (\exists \text{ err. } x1 = \text{Some } (\text{Error } \text{err}))$
 $\text{then } (x1, s1)$
 $\text{else } \text{abupd } (\text{abrupt-if } (x1 \neq \text{None}) \text{ } x1) \text{ } s2) \rrbracket \implies$
 $G \vdash \text{Norm } s0 - c1 \text{ Finally } c2 - n \rightarrow s3$

| *Init*: $\llbracket \text{the } (\text{class } G \text{ } C) = c;$
 $\text{if } \text{inited } C \text{ (globs } s0) \text{ then } s3 = \text{Norm } s0$
 $\text{else } (G \vdash \text{Norm } (\text{init-class-obj } G \text{ } C \text{ } s0)$
 $\text{---} (\text{if } C = \text{Object then Skip else Init (super } c)) - n \rightarrow s1 \wedge$
 $G \vdash \text{set-lvars Map.empty } s1 - \text{init } c - n \rightarrow s2 \wedge$
 $s3 = \text{restore-lvars } s1 \text{ } s2) \rrbracket$
 \implies
 $G \vdash \text{Norm } s0 - \text{Init } C - n \rightarrow s3$

monos

if-bool-eq-conj

declare *if-split* [*split del*] *if-split-asm* [*split del*]
option.split [*split del*] *option.split-asm* [*split del*]
not-None-eq [*simp del*]
split-paired-All [*simp del*] *split-paired-Ex* [*simp del*]
 $\langle ML \rangle$

inductive-cases *evaln-cases*: $G \vdash s - t \succ - n \rightarrow (v, s')$

inductive-cases *evaln-elim-cases*:

$G \vdash (\text{Some } xc, s) - t$	$\succ - n \rightarrow (v, s')$
$G \vdash \text{Norm } s - \text{In1r Skip}$	$\succ - n \rightarrow (x, s')$
$G \vdash \text{Norm } s - \text{In1r (Jmp } j)$	$\succ - n \rightarrow (x, s')$
$G \vdash \text{Norm } s - \text{In1r (l. } c)$	$\succ - n \rightarrow (x, s')$
$G \vdash \text{Norm } s - \text{In3 } (\llbracket \rrbracket)$	$\succ - n \rightarrow (v, s')$
$G \vdash \text{Norm } s - \text{In3 } (e \# es)$	$\succ - n \rightarrow (v, s')$
$G \vdash \text{Norm } s - \text{In1l (Lit } w)$	$\succ - n \rightarrow (v, s')$
$G \vdash \text{Norm } s - \text{In1l (UnOp unop } e)$	$\succ - n \rightarrow (v, s')$
$G \vdash \text{Norm } s - \text{In1l (BinOp binop } e1 \text{ } e2)$	$\succ - n \rightarrow (v, s')$
$G \vdash \text{Norm } s - \text{In2 (LVar } vn)$	$\succ - n \rightarrow (v, s')$
$G \vdash \text{Norm } s - \text{In1l (Cast } T \text{ } e)$	$\succ - n \rightarrow (v, s')$
$G \vdash \text{Norm } s - \text{In1l (e InstOf } T)$	$\succ - n \rightarrow (v, s')$
$G \vdash \text{Norm } s - \text{In1l (Super)}$	$\succ - n \rightarrow (v, s')$
$G \vdash \text{Norm } s - \text{In1l (Acc } va)$	$\succ - n \rightarrow (v, s')$
$G \vdash \text{Norm } s - \text{In1r (Expr } e)$	$\succ - n \rightarrow (x, s')$
$G \vdash \text{Norm } s - \text{In1r (c1;; c2)$	$\succ - n \rightarrow (x, s')$
$G \vdash \text{Norm } s - \text{In1l (Methd } C \text{ } sig)$	$\succ - n \rightarrow (x, s')$
$G \vdash \text{Norm } s - \text{In1l (Body } D \text{ } c)$	$\succ - n \rightarrow (x, s')$
$G \vdash \text{Norm } s - \text{In1l (e0 ? e1 : e2)$	$\succ - n \rightarrow (v, s')$
$G \vdash \text{Norm } s - \text{In1r (If(e) c1 Else c2)$	$\succ - n \rightarrow (x, s')$
$G \vdash \text{Norm } s - \text{In1r (l. While(e) c)$	$\succ - n \rightarrow (x, s')$
$G \vdash \text{Norm } s - \text{In1r (c1 Finally c2)$	$\succ - n \rightarrow (x, s')$
$G \vdash \text{Norm } s - \text{In1r (Throw } e)$	$\succ - n \rightarrow (x, s')$
$G \vdash \text{Norm } s - \text{In1l (NewC } C)$	$\succ - n \rightarrow (v, s')$
$G \vdash \text{Norm } s - \text{In1l (New } T[e])$	$\succ - n \rightarrow (v, s')$
$G \vdash \text{Norm } s - \text{In1l (Ass } va \text{ } e)$	$\succ - n \rightarrow (v, s')$
$G \vdash \text{Norm } s - \text{In1r (Try } c1 \text{ Catch}(tn \text{ } vn) \text{ } c2)$	$\succ - n \rightarrow (x, s')$
$G \vdash \text{Norm } s - \text{In2 } (\{accC, statDeclC, stat\} e..fn)$	$\succ - n \rightarrow (v, s')$
$G \vdash \text{Norm } s - \text{In2 } (e1.[e2])$	$\succ - n \rightarrow (v, s')$

$$\begin{array}{l} G\vdash \text{Norm } s - \text{In1l } (\{accC, statT, mode\}e \cdot mn(\{pT\}p)) \succ -n \rightarrow (v, s') \\ G\vdash \text{Norm } s - \text{In1r } (\text{Init } C) \succ -n \rightarrow (x, s') \end{array}$$

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*: $G\vdash s - t \succ -n \rightarrow (w, s') \implies \text{case } t \text{ of } \text{In1 } ec \implies$
 $(\text{case } ec \text{ of } \text{Inl } e \implies (\exists v. w = \text{In1 } v) \mid \text{Inr } c \implies w = \diamond)$
 $\mid \text{In2 } e \implies (\exists v. w = \text{In2 } v) \mid \text{In3 } e \implies (\exists v. w = \text{In3 } v)$
 ⟨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*: $G\vdash s - e \succ -n \rightarrow (v, s') \implies$
 $\text{fst } s = \text{Some } xc \implies s' = s \wedge v = \text{undefined3 } e$
 ⟨proof⟩

lemma *evaln-abrupt*:

$\wedge s'. G \vdash (\text{Some } xc, s) \text{ -e}\succ\text{-n} \rightarrow (w, s') = (s' = (\text{Some } xc, s) \wedge$
 $w = \text{undefined} \exists e \wedge G \vdash (\text{Some } xc, s) \text{ -e}\succ\text{-n} \rightarrow (\text{undefined} \exists e, (\text{Some } xc, s)))$
 ⟨proof⟩

⟨ML⟩

lemma *evaln-LitI*: $G \vdash s \text{ -Lit } v \text{ -}\succ \text{ (if normal } s \text{ then } v \text{ else undefined) -n} \rightarrow s$

⟨proof⟩

lemma *CondI*:

$\wedge s1. \llbracket G \vdash s \text{ -e}\succ\text{-b-n} \rightarrow s1; G \vdash s1 \text{ - (if the-Bool } b \text{ then } e1 \text{ else } e2) \text{ -}\succ\text{-v-n} \rightarrow s2 \rrbracket \implies$
 $G \vdash s \text{ -e } ? e1 : e2 \text{ -}\succ \text{ (if normal } s1 \text{ then } v \text{ else undefined) -n} \rightarrow s2$
 ⟨proof⟩

lemma *evaln-SkipI* [*intro!*]: $G \vdash s \text{ -Skip-n} \rightarrow s$

⟨proof⟩

lemma *evaln-ExprI*: $G \vdash s \text{ -e}\succ\text{-v-n} \rightarrow s' \implies G \vdash s \text{ -Expr } e \text{ -n} \rightarrow s'$

⟨proof⟩

lemma *evaln-CompI*: $\llbracket G \vdash s \text{ -c1-n} \rightarrow s1; G \vdash s1 \text{ -c2-n} \rightarrow s2 \rrbracket \implies G \vdash s \text{ -c1;; c2-n} \rightarrow s2$

⟨proof⟩

lemma *evaln-IfI*:

$\llbracket G \vdash s \text{ -e}\succ\text{-v-n} \rightarrow s1; G \vdash s1 \text{ - (if the-Bool } v \text{ then } c1 \text{ else } c2) \text{ -n} \rightarrow s2 \rrbracket \implies$
 $G \vdash s \text{ -If } (e) \text{ c1 Else } c2 \text{ -n} \rightarrow s2$

⟨proof⟩

lemma *evaln-SkipD* [*dest!*]: $G \vdash s \text{ -Skip-n} \rightarrow s' \implies s' = s$

⟨proof⟩

lemma *evaln-Skip-eq* [*simp*]: $G \vdash s \text{ -Skip-n} \rightarrow s' = (s = s')$

⟨proof⟩

evaln implies eval

lemma *evaln-eval*:

assumes *evaln*: $G \vdash s0 \text{ -t}\succ\text{-n} \rightarrow (v, s1)$

shows $G \vdash s0 \text{ -t}\succ \rightarrow (v, s1)$

⟨proof⟩

lemma *Suc-le-D-lemma*: $\llbracket \text{Suc } n \leq m'; (\wedge m. n \leq m \implies P (\text{Suc } m)) \rrbracket \implies P m'$

⟨proof⟩

lemma *evaln-nonstrict* [*rule-format (no-asm), elim*]:

$G \vdash s \text{ -t}\succ\text{-n} \rightarrow (w, s') \implies \forall m. n \leq m \longrightarrow G \vdash s \text{ -t}\succ\text{-m} \rightarrow (w, s')$

⟨proof⟩

lemmas *evaln-nonstrict-Suc* = *evaln-nonstrict* [OF - le-refl [THEN le-SucI]]

lemma *evaln-max2*: $\llbracket G \vdash s1 - t1 \succ - n1 \rightarrow (w1, s1'); G \vdash s2 - t2 \succ - n2 \rightarrow (w2, s2') \rrbracket \implies$
 $G \vdash s1 - t1 \succ - \max n1 n2 \rightarrow (w1, s1') \wedge G \vdash s2 - t2 \succ - \max n1 n2 \rightarrow (w2, s2')$
 <proof>

corollary *evaln-max2E* [consumes 2]:

$\llbracket G \vdash s1 - t1 \succ - n1 \rightarrow (w1, s1'); G \vdash s2 - t2 \succ - n2 \rightarrow (w2, s2') \rrbracket$
 $\llbracket G \vdash s1 - t1 \succ - \max n1 n2 \rightarrow (w1, s1'); G \vdash s2 - t2 \succ - \max n1 n2 \rightarrow (w2, s2') \rrbracket \implies P \rrbracket \implies P$
 <proof>

lemma *evaln-max3*:

$\llbracket G \vdash s1 - t1 \succ - n1 \rightarrow (w1, s1'); G \vdash s2 - t2 \succ - n2 \rightarrow (w2, s2'); G \vdash s3 - t3 \succ - n3 \rightarrow (w3, s3') \rrbracket \implies$
 $G \vdash s1 - t1 \succ - \max (\max n1 n2) n3 \rightarrow (w1, s1') \wedge$
 $G \vdash s2 - t2 \succ - \max (\max n1 n2) n3 \rightarrow (w2, s2') \wedge$
 $G \vdash s3 - t3 \succ - \max (\max n1 n2) n3 \rightarrow (w3, s3')$
 <proof>

corollary *evaln-max3E*:

$\llbracket G \vdash s1 - t1 \succ - n1 \rightarrow (w1, s1'); G \vdash s2 - t2 \succ - n2 \rightarrow (w2, s2'); G \vdash s3 - t3 \succ - n3 \rightarrow (w3, s3') \rrbracket$
 $\llbracket G \vdash s1 - t1 \succ - \max (\max n1 n2) n3 \rightarrow (w1, s1') \rrbracket$
 $G \vdash s2 - t2 \succ - \max (\max n1 n2) n3 \rightarrow (w2, s2')$
 $G \vdash s3 - t3 \succ - \max (\max n1 n2) n3 \rightarrow (w3, s3')$
 $\rrbracket \implies P$
 $\rrbracket \implies P$
 <proof>

lemma *le-max3I1*: $(n2 :: nat) \leq \max n1 (\max n2 n3)$

<proof>

lemma *le-max3I2*: $(n3 :: nat) \leq \max n1 (\max n2 n3)$

<proof>

declare [[*simproc del*: *wt-expr wt-var wt-exprs wt-stmt*]]

eval implies evaln

lemma *eval-evaln*:

assumes *eval*: $G \vdash s0 - t \succ \rightarrow (v, s1)$
shows $\exists n. G \vdash s0 - t \succ - n \rightarrow (v, s1)$
 <proof>

end

Chapter 21

Trans

theory *Trans* **imports** *Evaln* **begin**

definition

```
groundVar :: var ⇒ bool where
groundVar v ⇔ (case v of
  LVar ln ⇒ True
  | {accC,statDeclC,stat}e..fn ⇒ ∃ a. e=Lit a
  | e1.[e2] ⇒ ∃ a i. e1 = Lit a ∧ e2 = Lit i
  | InsInitV c v ⇒ 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 ⇒ bool
where groundExprs es ⇔ (∀ e ∈ set es. ∃ v. e = Lit v)
```

primrec *the-val*:: expr ⇒ val

```
where the-val (Lit v) = v
```

primrec *the-var*:: prog ⇒ state ⇒ var ⇒ (vvar × 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

$SKIP :: expr$
where $SKIP == Lit\ Unit$

inductive

$step :: [prog, term \times state, term \times state] \Rightarrow bool\ (\dashv\vdash \mapsto 1\ -[61,82,82]\ 81)$
for $G :: prog$

where

$Abrupt:$ $\llbracket \forall v. t \neq \langle Lit\ v \rangle;$
 $\forall t. t \neq \langle l \cdot Skip \rangle;$
 $\forall C\ vn\ c. t \neq \langle Try\ Skip\ Catch\ (C\ vn)\ c \rangle;$
 $\forall x\ c. t \neq \langle Skip\ Finally\ c \rangle \wedge xc \neq Xcpt\ x;$
 $\forall a\ c. t \neq \langle FinA\ a\ c \rangle \rrbracket$
 \implies
 $G \vdash (t, Some\ xc, s) \mapsto 1\ (\langle Lit\ undefined \rangle, Some\ xc, s)$

| $InsInitE:$ $\llbracket G \vdash (\langle c \rangle, Norm\ s) \mapsto 1\ (\langle c \rangle, s') \rrbracket$
 \implies
 $G \vdash (\langle InsInitE\ c\ e \rangle, Norm\ s) \mapsto 1\ (\langle InsInitE\ c'\ e \rangle, s')$

| $NewC:$ $G \vdash (\langle NewC\ C \rangle, Norm\ s) \mapsto 1\ (\langle InsInitE\ (Init\ C)\ (NewC\ C) \rangle, Norm\ s)$
| $NewCInitd:$ $\llbracket G \vdash Norm\ s \text{ --halloc } (CInst\ C) \succ a \rightarrow s' \rrbracket$
 \implies
 $G \vdash (\langle InsInitE\ Skip\ (NewC\ C) \rangle, Norm\ s) \mapsto 1\ (\langle Ref\ a \rangle, s')$

| $NewA:$
 $G \vdash (\langle New\ T[e] \rangle, Norm\ s) \mapsto 1\ (\langle InsInitE\ (init\ comp\ ty\ T)\ (New\ T[e]) \rangle, Norm\ s)$
| $InsInitNewAIdx:$
 $\llbracket G \vdash (\langle e \rangle, Norm\ s) \mapsto 1\ (\langle e \rangle, s') \rrbracket$
 \implies
 $G \vdash (\langle InsInitE\ Skip\ (New\ T[e]) \rangle, Norm\ s) \mapsto 1\ (\langle InsInitE\ Skip\ (New\ T[e']) \rangle, s')$
| $InsInitNewA:$
 $\llbracket G \vdash abupd\ (check\ neg\ i)\ (Norm\ s) \text{ --halloc } (Arr\ T\ (the\ Intg\ i)) \succ a \rightarrow s' \rrbracket$
 \implies
 $G \vdash (\langle InsInitE\ Skip\ (New\ T[Lit\ i]) \rangle, Norm\ s) \mapsto 1\ (\langle Ref\ a \rangle, s')$

| $CastE:$
 $\llbracket G \vdash (\langle e \rangle, Norm\ s) \mapsto 1\ (\langle e \rangle, s') \rrbracket$
 \implies
 $G \vdash (\langle Cast\ T\ e \rangle, None, s) \mapsto 1\ (\langle Cast\ T\ e \rangle, s')$

| $Cast:$
 $\llbracket s' = abupd\ (raise\ if\ (\neg G, s \vdash v\ fits\ T)\ ClassCast)\ (Norm\ s) \rrbracket$
 \implies

$$G\vdash(\langle\text{Cast } T \text{ (Lit } v)\rangle, \text{Norm } s) \mapsto 1 (\langle\text{Lit } v\rangle, s')$$

$$| \text{InstE: } \llbracket G\vdash(\langle e\rangle, \text{Norm } s) \mapsto 1 (\langle e'::\text{expr}\rangle, s') \rrbracket$$

$$\implies$$

$$G\vdash(\langle e \text{ InstOf } T\rangle, \text{Norm } s) \mapsto 1 (\langle e'\rangle, s')$$

$$| \text{Inst: } \llbracket b = (v \neq \text{Null} \wedge G, s \vdash v \text{ fits Ref } T) \rrbracket$$

$$\implies$$

$$G\vdash(\langle\text{Lit } v \text{ InstOf } T\rangle, \text{Norm } s) \mapsto 1 (\langle\text{Lit } (\text{Bool } b)\rangle, s')$$

$$| \text{UnOpE: } \llbracket G\vdash(\langle e\rangle, \text{Norm } s) \mapsto 1 (\langle e'\rangle, s') \rrbracket$$

$$\implies$$

$$G\vdash(\langle\text{UnOp unop } e\rangle, \text{Norm } s) \mapsto 1 (\langle\text{UnOp unop } e'\rangle, s')$$

$$| \text{UnOp: } G\vdash(\langle\text{UnOp unop (Lit } v)\rangle, \text{Norm } s) \mapsto 1 (\langle\text{Lit (eval-unop unop } v)\rangle, \text{Norm } s)$$

$$| \text{BinOpE1: } \llbracket G\vdash(\langle e1\rangle, \text{Norm } s) \mapsto 1 (\langle e1'\rangle, s') \rrbracket$$

$$\implies$$

$$G\vdash(\langle\text{BinOp binop } e1 \text{ } e2\rangle, \text{Norm } s) \mapsto 1 (\langle\text{BinOp binop } e1' \text{ } e2\rangle, s')$$

$$| \text{BinOpE2: } \llbracket \text{need-second-arg binop } v1; G\vdash(\langle e2\rangle, \text{Norm } s) \mapsto 1 (\langle e2'\rangle, s') \rrbracket$$

$$\implies$$

$$G\vdash(\langle\text{BinOp binop (Lit } v1) \text{ } e2\rangle, \text{Norm } s)$$

$$\mapsto 1 (\langle\text{BinOp binop (Lit } v1) \text{ } e2'\rangle, s')$$

$$| \text{BinOpTerm: } \llbracket \neg \text{need-second-arg binop } v1 \rrbracket$$

$$\implies$$

$$G\vdash(\langle\text{BinOp binop (Lit } v1) \text{ } e2\rangle, \text{Norm } s)$$

$$\mapsto 1 (\langle\text{Lit } v1\rangle, \text{Norm } s)$$

$$| \text{BinOp: } G\vdash(\langle\text{BinOp binop (Lit } v1) \text{ (Lit } v2)\rangle, \text{Norm } s)$$

$$\mapsto 1 (\langle\text{Lit (eval-binop binop } v1 \text{ } v2)\rangle, \text{Norm } s)$$

$$| \text{Super: } G\vdash(\langle\text{Super}\rangle, \text{Norm } s) \mapsto 1 (\langle\text{Lit (val-this } s)\rangle, \text{Norm } s)$$

$$| \text{AccVA: } \llbracket G\vdash(\langle va\rangle, \text{Norm } s) \mapsto 1 (\langle va'\rangle, s') \rrbracket$$

$$\implies$$

$$G\vdash(\langle\text{Acc } va\rangle, \text{Norm } s) \mapsto 1 (\langle\text{Acc } va'\rangle, s')$$

$$| \text{Acc: } \llbracket \text{groundVar } va; ((v, vf), s') = \text{the-var } G (\text{Norm } s) \text{ } va \rrbracket$$

$$\implies$$

$$G\vdash(\langle\text{Acc } va\rangle, \text{Norm } s) \mapsto 1 (\langle\text{Lit } v\rangle, s')$$

$$| \text{AssVA: } \llbracket G\vdash(\langle va\rangle, \text{Norm } s) \mapsto 1 (\langle va'\rangle, s') \rrbracket$$

$$\implies$$

$$G\vdash(\langle va:=e\rangle, \text{Norm } s) \mapsto 1 (\langle va' := e\rangle, s')$$

$$| \text{AssE: } \llbracket \text{groundVar } va; G\vdash(\langle e\rangle, \text{Norm } s) \mapsto 1 (\langle e'\rangle, s') \rrbracket$$

$$\implies$$

$$G\vdash(\langle va:=e\rangle, \text{Norm } s) \mapsto 1 (\langle va := e'\rangle, s')$$

$$| \text{Ass: } \llbracket \text{groundVar } va; ((w, f), s') = \text{the-var } G (\text{Norm } s) \text{ } va \rrbracket$$

$$\implies$$

$$G\vdash(\langle va := (\text{Lit } v)\rangle, \text{Norm } s) \mapsto 1 (\langle\text{Lit } v, \text{assign } f \text{ } v \text{ } s'\rangle)$$

$$| \text{CondC: } \llbracket G\vdash(\langle e0\rangle, \text{Norm } s) \mapsto 1 (\langle e0'\rangle, s') \rrbracket$$

$$\implies$$

$$G\vdash(\langle e0? \text{ } e1:e2\rangle, \text{Norm } s) \mapsto 1 (\langle e0'? \text{ } e1:e2\rangle, s')$$

$$| \text{Cond: } G\vdash(\langle\text{Lit } b? \text{ } e1:e2\rangle, \text{Norm } s) \mapsto 1 (\langle\text{if the-Bool } b \text{ then } e1 \text{ else } e2\rangle, \text{Norm } s)$$

<i>CallTarget</i> :	$\llbracket G\vdash(\langle e \rangle, \text{Norm } s) \mapsto 1 (\langle e' \rangle, s') \rrbracket$ \implies $G\vdash(\langle \{ \text{acc}C, \text{stat}T, \text{mode} \} e \cdot \text{mn}(\{ pTs \} \text{args}) \rangle, \text{Norm } s)$ $\mapsto 1 (\langle \{ \text{acc}C, \text{stat}T, \text{mode} \} e' \cdot \text{mn}(\{ pTs \} \text{args}) \rangle, s')$
<i>CallArgs</i> :	$\llbracket G\vdash(\langle \text{args} \rangle, \text{Norm } s) \mapsto 1 (\langle \text{args}' \rangle, s') \rrbracket$ \implies $G\vdash(\langle \{ \text{acc}C, \text{stat}T, \text{mode} \} \text{Lit } a \cdot \text{mn}(\{ pTs \} \text{args}) \rangle, \text{Norm } s)$ $\mapsto 1 (\langle \{ \text{acc}C, \text{stat}T, \text{mode} \} \text{Lit } a \cdot \text{mn}(\{ pTs \} \text{args}') \rangle, s')$
<i>Call</i> :	$\llbracket \text{groundExprs } \text{args}; \text{vs} = \text{map the-val } \text{args};$ $D = \text{invocation-declclass } G \text{ mode } s \text{ a stat}T (\llbracket \text{name}=\text{mn}, \text{par}Ts=\text{pTs} \rrbracket);$ $s' = \text{init-lvars } G \text{ D } (\llbracket \text{name}=\text{mn}, \text{par}Ts=\text{pTs} \rrbracket) \text{ mode } a' \text{ vs } (\text{Norm } s) \rrbracket$ \implies $G\vdash(\langle \{ \text{acc}C, \text{stat}T, \text{mode} \} \text{Lit } a \cdot \text{mn}(\{ pTs \} \text{args}) \rangle, \text{Norm } s)$ $\mapsto 1 (\langle \langle \text{Callee } (\text{locals } s) (\text{Methd } D (\llbracket \text{name}=\text{mn}, \text{par}Ts=\text{pTs} \rrbracket)) \rangle \rangle, s')$
<i>Callee</i> :	$\llbracket G\vdash(\langle e \rangle, \text{Norm } s) \mapsto 1 (\langle e' :: \text{expr} \rangle, s') \rrbracket$ \implies $G\vdash(\langle \langle \text{Callee lcls-caller } e \rangle \rangle, \text{Norm } s) \mapsto 1 (\langle e' \rangle, s')$
<i>CalleeRet</i> :	$G\vdash(\langle \langle \text{Callee lcls-caller } (\text{Lit } v) \rangle \rangle, \text{Norm } s)$ $\mapsto 1 (\langle \langle \text{Lit } v \rangle, (\text{set-lvars lcls-caller } (\text{Norm } s)) \rangle \rangle)$
<i>Methd</i> :	$G\vdash(\langle \langle \text{Methd } D \text{ sig} \rangle \rangle, \text{Norm } s) \mapsto 1 (\langle \langle \text{body } G \text{ D sig} \rangle \rangle, \text{Norm } s)$
<i>Body</i> :	$G\vdash(\langle \langle \text{Body } D \text{ c} \rangle \rangle, \text{Norm } s) \mapsto 1 (\langle \langle \text{InsInitE } (\text{Init } D) (\text{Body } D \text{ c}) \rangle \rangle, \text{Norm } s)$
<i>InsInitBody</i> :	$\llbracket G\vdash(\langle c \rangle, \text{Norm } s) \mapsto 1 (\langle c' \rangle, s') \rrbracket$ \implies $G\vdash(\langle \langle \text{InsInitE Skip } (\text{Body } D \text{ c}) \rangle \rangle, \text{Norm } s) \mapsto 1 (\langle \langle \text{InsInitE Skip } (\text{Body } D \text{ c}') \rangle \rangle, s')$
<i>InsInitBodyRet</i> :	$G\vdash(\langle \langle \text{InsInitE Skip } (\text{Body } D \text{ Skip}) \rangle \rangle, \text{Norm } s)$ $\mapsto 1 (\langle \langle \text{Lit } (\text{the } (\langle \text{locals } s \rangle \text{ Result})) \rangle, \text{abupd } (\text{absorb Ret}) (\text{Norm } s) \rangle \rangle)$
<i>FVar</i> :	$\llbracket \neg \text{inited statDeclC } (\text{globs } s) \rrbracket$ \implies $G\vdash(\langle \langle \{ \text{acc}C, \text{statDeclC}, \text{stat} \} e \cdot \text{fn} \rangle \rangle, \text{Norm } s)$ $\mapsto 1 (\langle \langle \text{InsInitV } (\text{Init statDeclC}) (\{ \text{acc}C, \text{statDeclC}, \text{stat} \} e \cdot \text{fn}) \rangle \rangle, \text{Norm } s)$
<i>InsInitFVarE</i> :	$\llbracket G\vdash(\langle e \rangle, \text{Norm } s) \mapsto 1 (\langle e' \rangle, s') \rrbracket$ \implies $G\vdash(\langle \langle \text{InsInitV Skip } (\{ \text{acc}C, \text{statDeclC}, \text{stat} \} e \cdot \text{fn}) \rangle \rangle, \text{Norm } s)$ $\mapsto 1 (\langle \langle \text{InsInitV Skip } (\{ \text{acc}C, \text{statDeclC}, \text{stat} \} e' \cdot \text{fn}) \rangle \rangle, s')$
<i>InsInitFVar</i> :	$G\vdash(\langle \langle \text{InsInitV Skip } (\{ \text{acc}C, \text{statDeclC}, \text{stat} \} \text{Lit } a \cdot \text{fn}) \rangle \rangle, \text{Norm } s)$ $\mapsto 1 (\langle \langle \{ \text{acc}C, \text{statDeclC}, \text{stat} \} \text{Lit } a \cdot \text{fn} \rangle \rangle, \text{Norm } s)$
<p>— Notice, that we do not have literal values for <i>vars</i>. The rules for accessing variables (<i>Acc</i>) and assigning to variables (<i>Ass</i>), test this with the predicate <i>groundVar</i>. After initialisation is done and the <i>FVar</i> is evaluated, we can't just throw away the <i>InsInitFVar</i> term and return a literal value, as in the cases of <i>New</i> or <i>NewC</i>. Instead we just return the evaluated <i>FVar</i> and test for initialisation in the rule <i>FVar</i>.</p>	
<i>AVarE1</i> :	$\llbracket G\vdash(\langle e1 \rangle, \text{Norm } s) \mapsto 1 (\langle e1' \rangle, s') \rrbracket$ \implies $G\vdash(\langle \langle e1 \rangle, [e2] \rangle \rangle, \text{Norm } s) \mapsto 1 (\langle \langle e1' \rangle, [e2] \rangle \rangle, s')$

$$\begin{array}{l}
| \text{AVarE2: } G\vdash(\langle e2 \rangle, \text{Norm } s) \mapsto 1 (\langle e2' \rangle, s') \\
\quad \Longrightarrow \\
\quad G\vdash(\langle \text{Lit } a.[e2] \rangle, \text{Norm } s) \mapsto 1 (\langle \text{Lit } a.[e2'] \rangle, s')
\end{array}$$

— *Nil* is fully evaluated

$$\begin{array}{l}
| \text{ConsHd: } \llbracket G\vdash(\langle e::\text{expr} \rangle, \text{Norm } s) \mapsto 1 (\langle e'::\text{expr} \rangle, s') \rrbracket \\
\quad \Longrightarrow \\
\quad G\vdash(\langle e\#es \rangle, \text{Norm } s) \mapsto 1 (\langle e'\#es \rangle, s')
\end{array}$$

$$\begin{array}{l}
| \text{ConsTl: } \llbracket G\vdash(\langle es \rangle, \text{Norm } s) \mapsto 1 (\langle es' \rangle, s') \rrbracket \\
\quad \Longrightarrow \\
\quad G\vdash(\langle (\text{Lit } v)\#es \rangle, \text{Norm } s) \mapsto 1 (\langle (\text{Lit } v)\#es' \rangle, s')
\end{array}$$

$$| \text{Skip: } G\vdash(\langle \text{Skip} \rangle, \text{Norm } s) \mapsto 1 (\langle \text{SKIP} \rangle, \text{Norm } s)$$

$$\begin{array}{l}
| \text{ExprE: } \llbracket G\vdash(\langle e \rangle, \text{Norm } s) \mapsto 1 (\langle e' \rangle, s') \rrbracket \\
\quad \Longrightarrow \\
\quad G\vdash(\langle \text{Expr } e \rangle, \text{Norm } s) \mapsto 1 (\langle \text{Expr } e' \rangle, s') \\
| \text{Expr: } G\vdash(\langle \text{Expr } (\text{Lit } v) \rangle, \text{Norm } s) \mapsto 1 (\langle \text{Skip} \rangle, \text{Norm } s)
\end{array}$$

$$\begin{array}{l}
| \text{LabC: } \llbracket G\vdash(\langle c \rangle, \text{Norm } s) \mapsto 1 (\langle c' \rangle, s') \rrbracket \\
\quad \Longrightarrow \\
\quad G\vdash(\langle l \cdot c \rangle, \text{Norm } s) \mapsto 1 (\langle l \cdot c' \rangle, s') \\
| \text{Lab: } G\vdash(\langle l \cdot \text{Skip} \rangle, s) \mapsto 1 (\langle \text{Skip} \rangle, \text{abupd } (\text{absorb } l) s)
\end{array}$$

$$\begin{array}{l}
| \text{CompC1: } \llbracket G\vdash(\langle c1 \rangle, \text{Norm } s) \mapsto 1 (\langle c1' \rangle, s') \rrbracket \\
\quad \Longrightarrow \\
\quad G\vdash(\langle c1;; c2 \rangle, \text{Norm } s) \mapsto 1 (\langle c1';; c2 \rangle, s')
\end{array}$$

$$| \text{Comp: } G\vdash(\langle \text{Skip};; c2 \rangle, \text{Norm } s) \mapsto 1 (\langle c2 \rangle, \text{Norm } s)$$

$$\begin{array}{l}
| \text{IfE: } \llbracket G\vdash(\langle e \rangle, \text{Norm } s) \mapsto 1 (\langle e' \rangle, s') \rrbracket \\
\quad \Longrightarrow \\
\quad G\vdash(\langle \text{If } (e) \text{ } s1 \text{ Else } s2 \rangle, \text{Norm } s) \mapsto 1 (\langle \text{If } (e') \text{ } s1 \text{ Else } s2 \rangle, s') \\
| \text{If: } G\vdash(\langle \text{If } (\text{Lit } v) \text{ } s1 \text{ Else } s2 \rangle, \text{Norm } s) \\
\quad \mapsto 1 (\langle \text{if the-Bool } v \text{ then } s1 \text{ else } s2 \rangle, \text{Norm } s)
\end{array}$$

$$\begin{array}{l}
| \text{Loop: } G\vdash(\langle l \cdot \text{While}(e) \text{ } c \rangle, \text{Norm } s) \\
\quad \mapsto 1 (\langle \text{If}(e) (\text{Cont } l \cdot c;; l \cdot \text{While}(e) \text{ } c) \text{ Else Skip} \rangle, \text{Norm } s)
\end{array}$$

$$| \text{Jmp: } G\vdash(\langle \text{Jmp } j \rangle, \text{Norm } s) \mapsto 1 (\langle \text{Skip} \rangle, (\text{Some } (\text{Jump } j), s))$$

$$\begin{array}{l}
| \text{ThrowE: } \llbracket G\vdash(\langle e \rangle, \text{Norm } s) \mapsto 1 (\langle e' \rangle, s') \rrbracket \\
\quad \Longrightarrow \\
\quad G\vdash(\langle \text{Throw } e \rangle, \text{Norm } s) \mapsto 1 (\langle \text{Throw } e' \rangle, s') \\
| \text{Throw: } G\vdash(\langle \text{Throw } (\text{Lit } a) \rangle, \text{Norm } s) \mapsto 1 (\langle \text{Skip} \rangle, \text{abupd } (\text{throw } a) (\text{Norm } s))
\end{array}$$

| *TryC1*: $\llbracket G \vdash (\langle c1 \rangle, \text{Norm } s) \mapsto 1 (\langle c1 \rangle, s') \rrbracket$
 \implies
 $G \vdash (\langle \text{Try } c1 \text{ Catch}(C \text{ vn}) c2 \rangle, \text{Norm } s) \mapsto 1 (\langle \text{Try } c1' \text{ Catch}(C \text{ vn}) c2 \rangle, s')$

| *Try*: $\llbracket G \vdash s \text{ --salloc--} \rightarrow s' \rrbracket$
 \implies
 $G \vdash (\langle \text{Try Skip Catch}(C \text{ vn}) c2 \rangle, s)$
 $\mapsto 1 (\text{if } G, s \vdash \text{catch } C \text{ then } (\langle c2 \rangle, \text{new-xcpt-var } \text{vn } s') \text{ else } (\langle \text{Skip} \rangle, s'))$

| *FinC1*: $\llbracket G \vdash (\langle c1 \rangle, \text{Norm } s) \mapsto 1 (\langle c1 \rangle, s') \rrbracket$
 \implies
 $G \vdash (\langle c1 \text{ Finally } c2 \rangle, \text{Norm } s) \mapsto 1 (\langle c1' \text{ Finally } c2 \rangle, s')$

| *Fin*: $G \vdash (\langle \text{Skip Finally } c2 \rangle, (a, s)) \mapsto 1 (\langle \text{FinA } a \text{ } c2 \rangle, \text{Norm } s)$

| *FinAC*: $\llbracket G \vdash (\langle c \rangle, s) \mapsto 1 (\langle c \rangle, s') \rrbracket$
 \implies
 $G \vdash (\langle \text{FinA } a \text{ } c \rangle, s) \mapsto 1 (\langle \text{FinA } a \text{ } c \rangle, s')$

| *FinA*: $G \vdash (\langle \text{FinA } a \text{ Skip} \rangle, s) \mapsto 1 (\langle \text{Skip} \rangle, \text{abupd } (\text{abrupt-if } (a \neq \text{None}) a) s)$

| *Init1*: $\llbracket \text{inited } C \text{ (globs } s) \rrbracket$
 \implies
 $G \vdash (\langle \text{Init } C \rangle, \text{Norm } s) \mapsto 1 (\langle \text{Skip} \rangle, \text{Norm } s)$

| *Init*: $\llbracket \text{the (class } G \text{ } C) = c; \neg \text{inited } C \text{ (globs } s) \rrbracket$
 \implies
 $G \vdash (\langle \text{Init } C \rangle, \text{Norm } s)$
 $\mapsto 1 (\langle (\text{if } C = \text{Object then Skip else (Init (super } c));$
 $\text{Expr (Callee (locals } s) (\text{InsInitE (init } c) \text{ SKIP}))} \rangle,$
 $\text{Norm (init-class-obj } G \text{ } C \text{ } s))$

— *InsInitE* is just used as trick to embed the statement *init c* into an expression

| *InsInitESKIP*:
 $G \vdash (\langle \text{InsInitE Skip SKIP} \rangle, \text{Norm } s) \mapsto 1 (\langle \text{SKIP} \rangle, \text{Norm } s)$

abbreviation

stepn:: $[\text{prog}, \text{term} \times \text{state}, \text{nat}, \text{term} \times \text{state}] \Rightarrow \text{bool } (\vdash \mapsto - [61, 82, 82] \ 81)$
where $G \vdash p \mapsto n \ p' \equiv (p, p') \in \{(x, y). \text{step } G \ x \ y\}^{\wedge n}$

abbreviation

steptr:: $[\text{prog}, \text{term} \times \text{state}, \text{term} \times \text{state}] \Rightarrow \text{bool } (\vdash \mapsto * - [61, 82, 82] \ 81)$
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 (-i Thomas Kleymann)
- expressions not flattened to elementary assignments (as usual for axiomatic semantics) but treated first-class =i explicit result value handling
- intermediate values not on triple, but on assertion level (with result entry)
- multiple results with semantical substitution mechanism not requiring a stack
- because of dynamic method binding, terms need to be dependent on state. this is also useful for conditional expressions and statements
- result values in triples exactly as in eval relation (also for xcpt states)
- validity: additional assumption of state conformance and well-typedness, which is required for soundness and thus rule hazard required of completeness

restrictions:

- all triples in a derivation are of the same type (due to weak polymorphism)

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* :: [*p**trn*] => *p**trn* (Val:- [951] 950)
-*Var* :: [*p**trn*] => *p**trn* (Var:- [951] 950)

$-Vals :: [pttrn] \Rightarrow pttrn \quad (Vals:- [951] 950)$

translations

$\lambda Val:v . b == (\lambda v. b) \circ CONST \textit{the-In1}$

$\lambda Var:v . b == (\lambda v. b) \circ CONST \textit{the-In2}$

$\lambda Vals:v. b == (\lambda v. b) \circ CONST \textit{the-In3}$

— relation on result values, state and auxiliary variables

type-synonym $'a \textit{ assn} = res \Rightarrow state \Rightarrow 'a \Rightarrow bool$

translations

$(type) 'a \textit{ assn} <= (type) \textit{ vals} \Rightarrow state \Rightarrow 'a \Rightarrow bool$

definition

$\textit{ assn-imp} :: 'a \textit{ assn} \Rightarrow 'a \textit{ assn} \Rightarrow bool \textit{ (infixr} \Rightarrow 25)$

where $(P \Rightarrow Q) = (\forall Y s Z. P Y s Z \longrightarrow Q Y s Z)$

lemma $\textit{ assn-imp-def2} \textit{ [iff]}: (P \Rightarrow Q) = (\forall Y s Z. P Y s Z \longrightarrow Q Y s Z)$

$\langle \textit{proof} \rangle$

assertion transformers

2 peek-and

definition

$\textit{ peek-and} :: 'a \textit{ assn} \Rightarrow (state \Rightarrow bool) \Rightarrow 'a \textit{ assn} \textit{ (infixl} \wedge. 13)$

where $(P \wedge. p) = (\lambda Y s Z. P Y s Z \wedge p s)$

lemma $\textit{ peek-and-def2} \textit{ [simp]}: \textit{ peek-and} P p Y s = (\lambda Z. (P Y s Z \wedge p s))$

$\langle \textit{proof} \rangle$

lemma $\textit{ peek-and-Not} \textit{ [simp]}: (P \wedge. (\lambda s. \neg f s)) = (P \wedge. \textit{Not} \circ f)$

$\langle \textit{proof} \rangle$

lemma $\textit{ peek-and-and} \textit{ [simp]}: \textit{ peek-and} (\textit{peek-and} P p) p = \textit{peek-and} P p$

$\langle \textit{proof} \rangle$

lemma $\textit{ peek-and-commut}: (P \wedge. p \wedge. q) = (P \wedge. q \wedge. p)$

$\langle \textit{proof} \rangle$

abbreviation

$\textit{Normal} :: 'a \textit{ assn} \Rightarrow 'a \textit{ assn}$

where $\textit{Normal} P == P \wedge. \textit{normal}$

lemma $\textit{ peek-and-Normal} \textit{ [simp]}: \textit{peek-and} (\textit{Normal} P) p = \textit{Normal} (\textit{peek-and} P p)$

$\langle \textit{proof} \rangle$

3 assn-supd

definition

$\textit{ assn-supd} :: 'a \textit{ assn} \Rightarrow (state \Rightarrow state) \Rightarrow 'a \textit{ assn} \textit{ (infixl} ;. 13)$

where $(P ;. f) = (\lambda Y s' Z. \exists s. P Y s Z \wedge s' = f s)$

lemma *assn-supd-def2* [simp]: $\text{assn-supd } P f Y s' Z = (\exists s. P Y s Z \wedge s' = f s)$
 ⟨proof⟩

4 supd-assn

definition

supd-assn :: $(\text{state} \Rightarrow \text{state}) \Rightarrow 'a \text{ assn} \Rightarrow 'a \text{ assn}$ (**infixr** .; 13)
where $(f .; P) = (\lambda Y s. P Y (f s))$

lemma *supd-assn-def2* [simp]: $(f .; P) Y s = P Y (f s)$
 ⟨proof⟩

lemma *supd-assn-supdD* [elim]: $((f .; Q) ;. f) Y s Z \Longrightarrow Q Y s Z$
 ⟨proof⟩

lemma *supd-assn-supdI* [elim]: $Q Y s Z \Longrightarrow (f .; (Q ;. f)) Y s Z$
 ⟨proof⟩

5 subst-res

definition

subst-res :: $'a \text{ assn} \Rightarrow \text{res} \Rightarrow 'a \text{ assn}$ (**←←** [60,61] 60)
where $P \leftarrow w = (\lambda Y. P w)$

lemma *subst-res-def2* [simp]: $(P \leftarrow w) Y = P w$
 ⟨proof⟩

lemma *subst-subst-res* [simp]: $P \leftarrow w \leftarrow v = P \leftarrow w$
 ⟨proof⟩

lemma *peek-and-subst-res* [simp]: $(P \wedge. p) \leftarrow w = (P \leftarrow w \wedge. p)$
 ⟨proof⟩

6 subst-Bool

definition

subst-Bool :: $'a \text{ assn} \Rightarrow \text{bool} \Rightarrow 'a \text{ assn}$ (**←=** [60,61] 60)
where $P \leftarrow = b = (\lambda Y s Z. \exists v. P (\text{Val } v) s Z \wedge (\text{normal } s \longrightarrow \text{the-Bool } v = b))$

lemma *subst-Bool-def2* [simp]:
 $(P \leftarrow = b) Y s Z = (\exists v. P (\text{Val } v) s Z \wedge (\text{normal } s \longrightarrow \text{the-Bool } v = b))$
 ⟨proof⟩

lemma *subst-Bool-the-BoolI*: $P (\text{Val } b) s Z \Longrightarrow (P \leftarrow = \text{the-Bool } b) Y s Z$
 ⟨proof⟩

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

$-peek-res :: pptrn \Rightarrow 'a\ assn \Rightarrow 'a\ assn \quad (\lambda\cdot. - [0,3] 3)$

translations

$\lambda w.. P == CONST peek-res (\lambda w. P)$

lemma $peek-res-def2$ $[simp]$: $peek-res P Y = P Y Y$

$\langle proof \rangle$

lemma $peek-res-subst-res$ $[simp]$: $peek-res P \leftarrow w = P w \leftarrow w$

$\langle proof \rangle$

lemma $peek-subst-res-allI$:

$(\bigwedge a. T a (P (f a) \leftarrow f a)) \implies \forall a. T a (peek-res P \leftarrow f a)$

$\langle proof \rangle$

8 ign-res

definition

$ign-res :: 'a\ assn \Rightarrow 'a\ assn \ (-\downarrow [1000] 1000)$

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

$-peek-st :: pptrn \Rightarrow 'a\ assn \Rightarrow 'a\ assn \quad (\lambda\cdot. - [0,3] 3)$

translations

$\lambda s.. P == CONST peek-st (\lambda s. P)$

lemma $peek-st-def2$ $[simp]$: $(\lambda s.. Pf s) Y s = Pf (store s) Y s$

$\langle proof \rangle$

lemma *peek-st-triv* [*simp*]: $(\lambda s.. P) = P$
 ⟨*proof*⟩

lemma *peek-st-st* [*simp*]: $(\lambda s.. \lambda s'.. P s s') = (\lambda s.. P s s)$
 ⟨*proof*⟩

lemma *peek-st-split* [*simp*]: $(\lambda s.. \lambda Y s'. P s Y s') = (\lambda Y s. P (\text{store } s) Y s)$
 ⟨*proof*⟩

lemma *peek-st-subst-res* [*simp*]: $(\lambda s.. P s) \leftarrow w = (\lambda s.. P s \leftarrow w)$
 ⟨*proof*⟩

lemma *peek-st-Normal* [*simp*]: $(\lambda s.. (\text{Normal } (P s))) = \text{Normal } (\lambda s.. P s)$
 ⟨*proof*⟩

10 ign-res-eq

definition

ign-res-eq :: 'a *assn* \Rightarrow *res* \Rightarrow 'a *assn* $(\downarrow = -$ [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 *assn* \Rightarrow 'a *assn* (**infixr** ..; 13)
where $(vf \ ..; P) = (\lambda Y s. \text{let } (v, s') = vf s \text{ in } P (\text{Var } v) s')$

lemma *RefVar-def2* [*simp*]: $(vf \ ..; P) Y s =$
 $P (\text{Var } (\text{fst } (vf s))) (\text{snd } (vf s))$
 ⟨*proof*⟩

12 allocation

definition

Alloc :: *prog* \Rightarrow *obj-tag* \Rightarrow 'a *assn* \Rightarrow 'a *assn*
where $\text{Alloc } G \text{ otag } P = (\lambda Y s Z. \forall s' a. G \vdash s \text{ -halloc } \text{otag} \triangleright a \rightarrow s' \longrightarrow P (\text{Val } (\text{Addr } a)) s' Z)$

definition

$SXAlloc :: prog \Rightarrow 'a\ assn \Rightarrow 'a\ assn$
where $SXAlloc\ G\ P = (\lambda Y\ s\ Z. \forall s'. G \vdash s \text{ -- } salloc \rightarrow s' \longrightarrow P\ Y\ s'\ Z)$

lemma $Alloc\text{-}def2$ [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 $SXAlloc\text{-}def2$ [simp]:
 $SXAlloc\ G\ P\ Y\ s\ Z = (\forall s'. G \vdash s \text{ -- } salloc \rightarrow s' \longrightarrow P\ Y\ s'\ Z)$
 <proof>

validity**definition**

$type\text{-}ok :: prog \Rightarrow term \Rightarrow state \Rightarrow bool$ **where**
 $type\text{-}ok\ G\ t\ s =$
 $(\exists L\ T\ C\ A. (normal\ s \longrightarrow (\downarrow prg=G,cls=C,lcl=L) \vdash t :: T \wedge$
 $(\downarrow prg=G,cls=C,lcl=L) \vdash dom\ (locals\ (store\ s)) \gg t \gg A)$
 $\wedge s :: \preceq(G,L))$

datatype $'a\ triple = triple\ ('a\ assn)\ term\ ('a\ assn)$
 $(\{(1-)\} / \text{--} \succ / \{(1-)\})$ [3,65,3] 75

type-synonym $'a\ triples = 'a\ triple\ set$

abbreviation

$var\text{-}triple :: ['a\ assn, var, 'a\ assn] \Rightarrow 'a\ triple$
 $(\{(1-)\} / \text{--} \succ / \{(1-)\})$ [3,80,3] 75
where $\{P\}\ e \text{ --} \succ \{Q\} == \{P\}\ In2\ e \succ \{Q\}$

abbreviation

$expr\text{-}triple :: ['a\ assn, expr, 'a\ assn] \Rightarrow 'a\ triple$
 $(\{(1-)\} / \text{--} \succ / \{(1-)\})$ [3,80,3] 75
where $\{P\}\ e \text{ --} \succ \{Q\} == \{P\}\ In1l\ e \succ \{Q\}$

abbreviation

$exprs\text{-}triple :: ['a\ assn, expr\ list, 'a\ assn] \Rightarrow 'a\ triple$
 $(\{(1-)\} / \text{--} \# \succ / \{(1-)\})$ [3,65,3] 75
where $\{P\}\ e \text{ --} \# \succ \{Q\} == \{P\}\ In3\ e \succ \{Q\}$

abbreviation

$stmt\text{-}triple :: ['a\ assn, stmt, 'a\ assn] \Rightarrow 'a\ triple$
 $(\{(1-)\} / \text{--} \cdot / \{(1-)\})$ [3,65,3] 75
where $\{P\}\ .c. \{Q\} == \{P\}\ In1r\ c \succ \{Q\}$

notation (ASCII)

$triple\ (\{(1-)\} / \text{--} \succ / \{(1-)\})$ [3,65,3] 75 **and**
 $var\text{-}triple\ (\{(1-)\} / \text{--} \succ / \{(1-)\})$ [3,80,3] 75 **and**
 $expr\text{-}triple\ (\{(1-)\} / \text{--} \succ / \{(1-)\})$ [3,80,3] 75 **and**
 $exprs\text{-}triple\ (\{(1-)\} / \text{--} \# \succ / \{(1-)\})$ [3,65,3] 75

lemma $inj\text{-}triple$: $inj\ (\lambda(P,t,Q). \{P\}\ t \succ \{Q\})$
 <proof>

lemma *triple-inj-eq*: $(\{P\} t \succ \{Q\} = \{P'\} t' \succ \{Q'\}) = (P=P' \wedge t=t' \wedge Q=Q')$
 ⟨proof⟩

definition *mtriples* :: $('c \Rightarrow 'a \text{ sig} \Rightarrow 'a \text{ assn}) \Rightarrow ('c \Rightarrow 'a \text{ sig} \Rightarrow \text{expr}) \Rightarrow$
 $('c \Rightarrow 'a \text{ sig} \Rightarrow 'a \text{ assn}) \Rightarrow ('c \times 'a \text{ sig}) \text{ set} \Rightarrow 'a \text{ triples } (\{(1-)\} / \text{---} / \{(1-)\} | -)[3,65,3,65] 75)$

where

$\{\{P\} \text{ tf} \text{---} \{Q\} | \text{ms}\} = (\lambda(C, \text{sig}). \{\text{Normal}(P \ C \ \text{sig})\} \text{ tf } C \ \text{sig} \text{---} \{Q \ C \ \text{sig}\}) \text{'ms}$

definition

triple-valid :: $\text{prog} \Rightarrow \text{nat} \Rightarrow 'a \text{ triple} \Rightarrow \text{bool} \text{ (---:- [61,0, 58] 57)}$

where

$G \models n:t =$
 $(\text{case } t \text{ of } \{P\} t \succ \{Q\} \Rightarrow$
 $\forall Y \ s \ Z. P \ Y \ s \ Z \longrightarrow \text{type-ok } G \ t \ s \longrightarrow$
 $(\forall Y' \ s'. G \vdash s \text{---} t \text{---} n \longrightarrow (Y', s') \longrightarrow Q \ Y' \ s' \ Z))$

abbreviation

triples-valid:: $\text{prog} \Rightarrow \text{nat} \Rightarrow 'a \text{ triples} \Rightarrow \text{bool} \text{ (---:- [61,0, 58] 57)}$

where $G \models n:ts == \text{Ball } ts \text{ (triple-valid } G \ n)$

notation (ASCII)

triples-valid (---:- [61,0, 58] 57)

definition

ax-valids :: $\text{prog} \Rightarrow 'b \text{ triples} \Rightarrow 'a \text{ triples} \Rightarrow \text{bool} \text{ (-,--- [61,58,58] 57)}$

where $(G, A) \models ts = (\forall n. G \models n:A \longrightarrow G \models n:ts)$

abbreviation

ax-valid :: $\text{prog} \Rightarrow 'b \text{ triples} \Rightarrow 'a \text{ triple} \Rightarrow \text{bool} \text{ (-,--- [61,58,58] 57)}$

where $G, A \models t == G, A \models \{t\}$

notation (ASCII)

ax-valid (-,--- [61,58,58] 57)

lemma *triple-valid-def2*: $G \models n:\{P\} t \succ \{Q\} =$

$(\forall Y \ s \ Z. P \ Y \ s \ Z$
 $\longrightarrow (\exists L. (\text{normal } s \longrightarrow (\exists C \ T \ A. (\text{prg}=G, \text{cls}=C, \text{lcl}=L) \vdash t::T \wedge$
 $(\text{prg}=G, \text{cls}=C, \text{lcl}=L) \vdash \text{dom } (\text{locals } (\text{store } s)) \gg t \gg A)) \wedge$
 $s::\preceq(G, L))$
 $\longrightarrow (\forall Y' \ s'. G \vdash s \text{---} t \text{---} n \longrightarrow (Y', s') \longrightarrow Q \ Y' \ s' \ Z))$

⟨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⟩

inductive

ax-derivs :: $\text{prog} \Rightarrow 'a \text{ triples} \Rightarrow 'a \text{ triples} \Rightarrow \text{bool} \text{ (-,--- [61,58,58] 57)}$

and *ax-deriv* :: $\text{prog} \Rightarrow 'a \text{ triples} \Rightarrow 'a \text{ triple} \Rightarrow \text{bool} \text{ (-,+ [61,58,58] 57)}$

for $G :: \text{prog}$

where

$G, A \vdash t \equiv G, A \models \{t\}$

- | *empty*: $G, A \vdash \{\}$
| *insert*: $\llbracket G, A \vdash t; G, A \vdash ts \rrbracket \implies G, A \vdash \text{insert } t \ ts$
- | *asm*: $ts \subseteq A \implies G, A \vdash ts$
- | *weaken*: $\llbracket G, A \vdash ts'; ts \subseteq ts' \rrbracket \implies G, A \vdash ts$
- | *conseq*: $\forall Y \ s \ Z \ . \ P \ Y \ s \ Z \ \longrightarrow (\exists P' \ Q'. G, A \vdash \{P'\} \ t \succ \{Q'\} \wedge (\forall Y' \ s' \ Z'. P' \ Y' \ s' \ Z' \longrightarrow Q' \ Y' \ s' \ Z')) \implies G, A \vdash \{P\} \ t \succ \{Q\}$
- | *hazard*: $G, A \vdash \{P \ \wedge. \text{Not} \circ \text{type-ok } G \ t\} \ t \succ \{Q\}$
- | *Abrupt*: $G, A \vdash \{P \leftarrow (\text{undefined3 } t) \ \wedge. \text{Not} \circ \text{normal}\} \ t \succ \{P\}$
- variables
- | *LVar*: $G, A \vdash \{\text{Normal } (\lambda s.. P \leftarrow \text{Var } (\text{lvar } vn \ s))\} \ \text{LVar } vn \ \succ \{P\}$
- | *FVar*: $\llbracket G, A \vdash \{\text{Normal } P\} \ . \text{Init } C. \{Q\}; G, A \vdash \{Q\} \ e \succ \{\lambda \text{Val}:a.. \text{fvar } C \ \text{stat} \ \text{fn } a \ ..; R\} \rrbracket \implies G, A \vdash \{\text{Normal } P\} \ \{\text{acc } C, C, \text{stat}\} e.. \text{fn} \succ \{R\}$
- | *AVar*: $\llbracket G, A \vdash \{\text{Normal } P\} \ e1 \succ \{Q\}; \forall a. G, A \vdash \{Q \leftarrow \text{Val } a\} \ e2 \succ \{\lambda \text{Val}:i.. \text{avar } G \ i \ a \ ..; R\} \rrbracket \implies G, A \vdash \{\text{Normal } P\} \ e1.[e2] \succ \{R\}$
- expressions
- | *NewC*: $\llbracket G, A \vdash \{\text{Normal } P\} \ . \text{Init } C. \{\text{Alloc } G \ (C \text{Inst } C) \ Q\} \rrbracket \implies G, A \vdash \{\text{Normal } P\} \ \text{NewC } C \succ \{Q\}$
- | *NewA*: $\llbracket G, A \vdash \{\text{Normal } P\} \ . \text{init-comp-ty } T. \{Q\}; G, A \vdash \{Q\} \ e \succ \{\lambda \text{Val}:i.. \text{abupd } (\text{check-neg } i) \ .; \text{Alloc } G \ (\text{Arr } T \ (\text{the-Intg } i)) \ R\} \rrbracket \implies G, A \vdash \{\text{Normal } P\} \ \text{New } T[e] \succ \{R\}$
- | *Cast*: $\llbracket G, A \vdash \{\text{Normal } P\} \ e \succ \{\lambda \text{Val}:v.. \lambda s.. \text{abupd } (\text{raise-if } (\neg G, s \vdash v \ \text{fits } T) \ \text{ClassCast}) \ .; Q \leftarrow \text{Val } v\} \rrbracket \implies G, A \vdash \{\text{Normal } P\} \ \text{Cast } T \ e \succ \{Q\}$
- | *Inst*: $\llbracket G, A \vdash \{\text{Normal } P\} \ e \succ \{\lambda \text{Val}:v.. \lambda s.. Q \leftarrow \text{Val } (\text{Bool } (v \neq \text{Null} \ \wedge \ G, s \vdash v \ \text{fits } \text{RefT } T))\} \rrbracket \implies G, A \vdash \{\text{Normal } P\} \ e \ \text{InstOf } T \succ \{Q\}$
- | *Lit*: $G, A \vdash \{\text{Normal } (P \leftarrow \text{Val } v)\} \ \text{Lit } v \succ \{P\}$
- | *UnOp*: $\llbracket G, A \vdash \{\text{Normal } P\} \ e \succ \{\lambda \text{Val}:v.. Q \leftarrow \text{Val } (\text{eval-unop } \text{unop } v)\} \rrbracket \implies G, A \vdash \{\text{Normal } P\} \ \text{UnOp } \text{unop} \ e \succ \{Q\}$
- | *BinOp*: $\llbracket G, A \vdash \{\text{Normal } P\} \ e1 \succ \{Q\}; \forall v1. G, A \vdash \{Q \leftarrow \text{Val } v1\} \ (\text{if need-second-arg binop } v1 \ \text{then } (\text{In1l } e2) \ \text{else } (\text{In1r } \text{Skip})) \succ \{\lambda \text{Val}:v2.. R \leftarrow \text{Val } (\text{eval-binop } \text{binop } v1 \ v2)\} \rrbracket \implies G, A \vdash \{\text{Normal } P\} \ \text{BinOp } \text{binop} \ e1 \ e2 \succ \{R\}$

- | *Super*: $G, A \vdash \{ \text{Normal } (\lambda s.. P \leftarrow \text{Val } (\text{val-this } s)) \} \text{ Super} \multimap \{ P \}$
- | *Acc*: $\llbracket G, A \vdash \{ \text{Normal } P \} \text{ va} \multimap \{ \lambda \text{Var}:(v,f).. Q \leftarrow \text{Val } v \} \rrbracket \implies$
 $G, A \vdash \{ \text{Normal } P \} \text{ Acc va} \multimap \{ Q \}$
- | *Ass*: $\llbracket G, A \vdash \{ \text{Normal } P \} \text{ va} \multimap \{ Q \};$
 $\forall \text{vf}. G, A \vdash \{ Q \leftarrow \text{Var } \text{vf} \} \text{ e} \multimap \{ \lambda \text{Val}:v.. \text{assign } (\text{snd } \text{vf}) \text{ v } .; R \} \rrbracket \implies$
 $G, A \vdash \{ \text{Normal } P \} \text{ va} := \text{e} \multimap \{ R \}$
- | *Cond*: $\llbracket G, A \vdash \{ \text{Normal } P \} \text{ e0} \multimap \{ P' \};$
 $\forall b. G, A \vdash \{ P' \leftarrow = b \} (\text{if } b \text{ then } \text{e1} \text{ else } \text{e2}) \multimap \{ Q \} \rrbracket \implies$
 $G, A \vdash \{ \text{Normal } P \} \text{ e0 } ? \text{e1} : \text{e2} \multimap \{ Q \}$
- | *Call*:
 $\llbracket G, A \vdash \{ \text{Normal } P \} \text{ e} \multimap \{ Q \}; \forall a. G, A \vdash \{ Q \leftarrow \text{Val } a \} \text{ args} \multimap \{ R \ a \};$
 $\forall a \text{ vs } \text{invC } \text{declC } l. G, A \vdash \{ (R \ a \leftarrow \text{Vals } \text{vs } \wedge.$
 $(\lambda s. \text{declC} = \text{invocation-declclass } G \text{ mode } (\text{store } s) \ a \ \text{statT } (\text{name} = \text{mn}, \text{parTs} = \text{pTs})) \wedge$
 $\text{invC} = \text{invocation-class } \text{mode } (\text{store } s) \ a \ \text{statT } \wedge$
 $l = \text{locals } (\text{store } s)) \};$
 $\text{init-lvars } G \ \text{declC } (\text{name} = \text{mn}, \text{parTs} = \text{pTs}) \ \text{mode } a \ \text{vs} \wedge.$
 $(\lambda s. \text{normal } s \longrightarrow G \vdash \text{mode} \rightarrow \text{invC} \preceq \text{statT}) \rrbracket \implies$
 $\text{Methd } \text{declC } (\text{name} = \text{mn}, \text{parTs} = \text{pTs}) \multimap \{ \text{set-lvars } l .; S \} \rrbracket \implies$
 $G, A \vdash \{ \text{Normal } P \} \{ \text{accC}, \text{statT}, \text{mode} \} \text{e} \cdot \text{mn}(\{ \text{pTs} \} \text{args}) \multimap \{ S \}$
- | *Methd*: $\llbracket G, A \cup \{ \{ P \} \text{ Methd} \multimap \{ Q \} \mid \text{ms} \} \vdash \{ \{ P \} \text{ body } G \multimap \{ Q \} \mid \text{ms} \} \rrbracket \implies$
 $G, A \vdash \{ \{ P \} \text{ Methd} \multimap \{ Q \} \mid \text{ms} \}$
- | *Body*: $\llbracket G, A \vdash \{ \text{Normal } P \} \text{ .Init } D. \{ Q \};$
 $G, A \vdash \{ Q \} \text{ .c. } \{ \lambda s.. \text{abupd } (\text{absorb } \text{Ret}) .; R \leftarrow (\text{In1 } (\text{the } (\text{locals } s \ \text{Result}))) \rrbracket \implies$
 $G, A \vdash \{ \text{Normal } P \} \text{ Body } D \text{ c} \multimap \{ R \}$
- expression lists
- | *Nil*: $G, A \vdash \{ \text{Normal } (P \leftarrow \text{Vals } []) \} [] \multimap \{ P \}$
- | *Cons*: $\llbracket G, A \vdash \{ \text{Normal } P \} \text{ e} \multimap \{ Q \};$
 $\forall v. G, A \vdash \{ Q \leftarrow \text{Val } v \} \text{ es} \multimap \{ \lambda \text{Vals}:\text{vs}.. R \leftarrow \text{Vals } (v \# \text{vs}) \} \rrbracket \implies$
 $G, A \vdash \{ \text{Normal } P \} \text{ e} \# \text{es} \multimap \{ R \}$
- statements
- | *Skip*: $G, A \vdash \{ \text{Normal } (P \leftarrow \diamond) \} \text{ .Skip. } \{ P \}$
- | *Expr*: $\llbracket G, A \vdash \{ \text{Normal } P \} \text{ e} \multimap \{ Q \leftarrow \diamond \} \rrbracket \implies$
 $G, A \vdash \{ \text{Normal } P \} \text{ .Expr } e. \{ Q \}$
- | *Lab*: $\llbracket G, A \vdash \{ \text{Normal } P \} \text{ .c. } \{ \text{abupd } (\text{absorb } l) .; Q \} \rrbracket \implies$
 $G, A \vdash \{ \text{Normal } P \} \text{ .l. c. } \{ Q \}$
- | *Comp*: $\llbracket G, A \vdash \{ \text{Normal } P \} \text{ .c1. } \{ Q \};$
 $G, A \vdash \{ Q \} \text{ .c2. } \{ R \} \rrbracket \implies$
 $G, A \vdash \{ \text{Normal } P \} \text{ .c1}; \text{c2. } \{ R \}$
- | *If*: $\llbracket G, A \vdash \{ \text{Normal } P \} \text{ e} \multimap \{ P' \};$
 $\forall b. G, A \vdash \{ P' \leftarrow = b \} (\text{if } b \text{ then } \text{c1} \text{ else } \text{c2}). \{ Q \} \rrbracket \implies$
 $G, A \vdash \{ \text{Normal } P \} \text{ .If}(e) \ \text{c1} \ \text{Else} \ \text{c2. } \{ Q \}$

| *Loop*: $\llbracket G, A \vdash \{P\} e \multimap \{P'\};$
 $G, A \vdash \{Normal (P' \leftarrow True)\} .c. \{abupd (absorb (Cont l)) .; P\} \rrbracket \Longrightarrow$
 $G, A \vdash \{P\} .l. While(e) c. \{(P' \leftarrow False) \downarrow = \diamond\}$

| *Jump*: $G, A \vdash \{Normal (abupd (\lambda a. (Some (Jump j)))) .; P \leftarrow \diamond\} .Jump j. \{P\}$

| *Throw*: $\llbracket G, A \vdash \{Normal P\} e \multimap \{\lambda Val:a.: abupd (throw a) .; Q \leftarrow \diamond\} \rrbracket \Longrightarrow$
 $G, A \vdash \{Normal P\} .Throw e. \{Q\}$

| *Try*: $\llbracket G, A \vdash \{Normal P\} .c1. \{SXAlloc G Q\};$
 $G, A \vdash \{Q \wedge (\lambda s. G, s \vdash catch C) .; new-xcpt-var vn\} .c2. \{R\};$
 $(Q \wedge (\lambda s. \neg G, s \vdash catch C)) \Rightarrow R \rrbracket \Longrightarrow$
 $G, A \vdash \{Normal P\} .Try c1 Catch(C vn) c2. \{R\}$

| *Finally*: $\llbracket G, A \vdash \{Normal P\} .c1. \{Q\};$
 $\forall x. G, A \vdash \{Q \wedge (\lambda s. x = fst s) .; abupd (\lambda x. None)\}$
 $.c2. \{abupd (abrupt-if (x \neq None) x) .; R\} \rrbracket \Longrightarrow$
 $G, A \vdash \{Normal P\} .c1 Finally c2. \{R\}$

| *Done*: $G, A \vdash \{Normal (P \leftarrow \diamond \wedge initd C)\} .Init C. \{P\}$

| *Init*: $\llbracket the (class G C) = c;$
 $G, A \vdash \{Normal ((P \wedge Not \circ initd C) .; supd (init-class-obj G C))\}$
 $.(if C = Object then Skip else Init (super c)). \{Q\};$
 $\forall l. G, A \vdash \{Q \wedge (\lambda s. l = locals (store s)) .; set-lvars Map.empty\}$
 $.init c. \{set-lvars l .; R\} \rrbracket \Longrightarrow$
 $G, A \vdash \{Normal (P \wedge Not \circ initd C)\} .Init C. \{R\}$

— Some dummy rules for the intermediate terms *Callee*, *InsInitE*, *InsInitV*, *FinA* only used by the smallstep semantics.

| *InsInitV*: $G, A \vdash \{Normal P\} InsInitV c v \multimap \{Q\}$
| *InsInitE*: $G, A \vdash \{Normal P\} InsInitE c e \multimap \{Q\}$
| *Callee*: $G, A \vdash \{Normal P\} Callee l e \multimap \{Q\}$
| *FinA*: $G, A \vdash \{Normal P\} .FinA a c. \{Q\}$

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*: $\llbracket G, A' \Vdash ts; G, A \Vdash A' \rrbracket \Longrightarrow G, A \Vdash ts$
<proof>

lemma *ax-thin* [rule-format (no-asm)]:

$G, (A' :: 'a \text{ triple set}) \Vdash (ts :: 'a \text{ triple set}) \Longrightarrow \forall A. A' \subseteq A \longrightarrow G, A \Vdash ts$
<proof>

lemma *ax-thin-insert*: $G, (A :: 'a \text{ triple set}) \Vdash (t :: 'a \text{ triple}) \Longrightarrow G, insert x A \Vdash t$
<proof>

lemma *subset-mtriples-iff*:

$ts \subseteq \{\{P\} mb \multimap \{Q\} \mid ms\} = (\exists ms'. ms' \subseteq ms \wedge ts = \{\{P\} mb \multimap \{Q\} \mid ms'\})$
<proof>

lemma *weaken*:

$$G, (A :: 'a \text{ triple set}) \vdash (ts' :: 'a \text{ triple set}) \implies \forall ts. ts \subseteq ts' \longrightarrow G, A \vdash ts$$

<proof>

rules derived from conseq

In the following rules we often have to give some type annotations like: $G, A \vdash \{P\} t \succ \{Q\}$. Given only the term above without annotations, Isabelle would infer a more general type were we could have different types of auxiliary variables in the assumption set (A) and in the triple itself (P and Q). But *ax-derivs.Methd* enforces the same type in the inductive definition of the derivation. So we have to restrict the types to be able to apply the rules.

lemma *conseq12*: $\llbracket G, (A :: 'a \text{ triple set}) \vdash \{P' :: 'a \text{ assn}\} t \succ \{Q'\};$
 $\forall Y s Z. P Y s Z \longrightarrow (\forall Y' s'. (\forall Y Z'. P' Y s Z' \longrightarrow Q' Y' s' Z') \longrightarrow$
 $Q Y' s' Z) \rrbracket$
 $\implies G, A \vdash \{P :: 'a \text{ assn}\} t \succ \{Q\}$
<proof>

lemma *conseq12'*: $\llbracket G, (A :: 'a \text{ triple set}) \vdash \{P' :: 'a \text{ assn}\} t \succ \{Q'\}; \forall s Y' s'.$
 $(\forall Y Z. P' Y s Z \longrightarrow Q' Y' s' Z) \longrightarrow$
 $(\forall Y Z. P Y s Z \longrightarrow Q Y' s' Z) \rrbracket$
 $\implies G, A \vdash \{P :: 'a \text{ assn}\} t \succ \{Q\}$
<proof>

lemma *conseq12-from-conseq12'*: $\llbracket G, (A :: 'a \text{ triple set}) \vdash \{P' :: 'a \text{ assn}\} t \succ \{Q'\};$
 $\forall Y s Z. P Y s Z \longrightarrow (\forall Y' s'. (\forall Y Z'. P' Y s Z' \longrightarrow Q' Y' s' Z') \longrightarrow$
 $Q Y' s' Z) \rrbracket$
 $\implies G, A \vdash \{P :: 'a \text{ assn}\} t \succ \{Q\}$
<proof>

lemma *conseq1*: $\llbracket G, (A :: 'a \text{ triple set}) \vdash \{P' :: 'a \text{ assn}\} t \succ \{Q\}; P \Rightarrow P' \rrbracket$
 $\implies G, A \vdash \{P :: 'a \text{ assn}\} t \succ \{Q\}$
<proof>

lemma *conseq2*: $\llbracket G, (A :: 'a \text{ triple set}) \vdash \{P :: 'a \text{ assn}\} t \succ \{Q'\}; Q' \Rightarrow Q \rrbracket$
 $\implies G, A \vdash \{P :: 'a \text{ assn}\} t \succ \{Q\}$
<proof>

lemma *ax-escape*:

$$\llbracket \forall Y s Z. P Y s Z$$

$$\longrightarrow G, (A :: 'a \text{ triple set}) \vdash \{\lambda Y' s' (Z' :: 'a). (Y', s') = (Y, s)\}$$

$$t \succ$$

$$\{\lambda Y s Z'. Q Y s Z\}$$

$$\rrbracket \implies G, A \vdash \{P :: 'a \text{ assn}\} t \succ \{Q :: 'a \text{ assn}\}$$

<proof>

lemma *ax-constant*: $\llbracket C \implies G, (A :: 'a \text{ triple set}) \vdash \{P :: 'a \text{ assn}\} t \succ \{Q\} \rrbracket$
 $\implies G, A \vdash \{\lambda Y s Z. C \wedge P Y s Z\} t \succ \{Q\}$
<proof>

lemma *ax-impossible* [*intro*]:

$$G, (A::'a \text{ triple set}) \vdash \{\lambda Y s Z. \text{False}\} t \succ \{Q::'a \text{ assn}\}$$

<proof>

lemma *ax-nochange-lemma*: $\llbracket P \ Y \ s; \text{All } ((=) \ w) \rrbracket \implies P \ w \ s$

<proof>

lemma *ax-nochange*:

$$G, (A::(\text{res} \times \text{state}) \text{ triple set}) \vdash \{\lambda Y s Z. (Y, s) = Z\} t \succ \{\lambda Y s Z. (Y, s) = Z\}$$

$$\implies G, A \vdash \{P::(\text{res} \times \text{state}) \text{ assn}\} t \succ \{P\}$$

<proof>

lemma *ax-trivial*: $G, (A::'a \text{ triple set}) \vdash \{P::'a \text{ assn}\} t \succ \{\lambda Y s Z. \text{True}\}$

<proof>

lemma *ax-disj*:

$$\llbracket G, (A::'a \text{ triple set}) \vdash \{P1::'a \text{ assn}\} t \succ \{Q1\}; G, A \vdash \{P2::'a \text{ assn}\} t \succ \{Q2\} \rrbracket$$

$$\implies G, A \vdash \{\lambda Y s Z. P1 \ Y \ s \ Z \vee P2 \ Y \ s \ Z\} t \succ \{\lambda Y s Z. Q1 \ Y \ s \ Z \vee Q2 \ Y \ s \ Z\}$$

<proof>

lemma *ax-supd-shuffle*:

$$(\exists Q. G, (A::'a \text{ triple set}) \vdash \{P::'a \text{ assn}\} .c1. \{Q\} \wedge G, A \vdash \{Q ; f\} .c2. \{R\}) =$$

$$(\exists Q'. G, A \vdash \{P\} .c1. \{f ; Q'\} \wedge G, A \vdash \{Q'\} .c2. \{R\})$$

<proof>

lemma *ax-cases*:

$$\llbracket G, (A::'a \text{ triple set}) \vdash \{P \wedge. \quad C\} t \succ \{Q::'a \text{ assn}\};$$

$$G, A \vdash \{P \wedge. \text{Not} \circ C\} t \succ \{Q\} \rrbracket \implies G, A \vdash \{P\} t \succ \{Q\}$$

<proof>

lemma *ax-adapt*: $G, (A::'a \text{ triple set}) \vdash \{P::'a \text{ assn}\} t \succ \{Q\}$

$$\implies G, A \vdash \{\text{adapt-pre } P \ Q \ Q'\} t \succ \{Q'\}$$

<proof>

lemma *adapt-pre-adapt*: $G, (A::'a \text{ triple set}) \models \{P::'a \text{ assn}\} t \succ \{Q\}$

$$\longrightarrow G, A \models \{\text{adapt-pre } P \ Q \ Q'\} t \succ \{Q'\}$$

<proof>

lemma *adapt-pre-weakest*:

$$\forall G \ (A::'a \text{ triple set}) \ t. \ G, A \models \{P\} t \succ \{Q\} \longrightarrow G, A \models \{P'\} t \succ \{Q\} \implies$$

$$P' \Rightarrow \text{adapt-pre } P \ Q \ (Q'::'a \text{ assn})$$

<proof>

lemma peek-and-forget1-Normal:

$$G, (A::'a \text{ triple set}) \vdash \{ \text{Normal } P \} t \succ \{ Q::'a \text{ assn} \}$$

$$\implies G, A \vdash \{ \text{Normal } (P \wedge. p) \} t \succ \{ Q \}$$

<proof>

lemma peek-and-forget1:

$$G, (A::'a \text{ triple set}) \vdash \{ P::'a \text{ assn} \} t \succ \{ Q \}$$

$$\implies G, A \vdash \{ P \wedge. p \} t \succ \{ Q \}$$

<proof>

lemmas ax-NormalD = peek-and-forget1 [of - - - - normal]

lemma peek-and-forget2:

$$G, (A::'a \text{ triple set}) \vdash \{ P::'a \text{ assn} \} t \succ \{ Q \wedge. p \}$$

$$\implies G, A \vdash \{ P \} t \succ \{ Q \}$$

<proof>

lemma ax-subst-Val-allI:

$$\forall v. G, (A::'a \text{ triple set}) \vdash \{ (P' \quad v) \leftarrow \text{Val } v \} t \succ \{ (Q \ v)::'a \text{ assn} \}$$

$$\implies \forall v. G, A \vdash \{ (\lambda w.. P' (\text{the-In1 } w)) \leftarrow \text{Val } v \} t \succ \{ Q \ v \}$$

<proof>

lemma ax-subst-Var-allI:

$$\forall v. G, (A::'a \text{ triple set}) \vdash \{ (P' \quad v) \leftarrow \text{Var } v \} t \succ \{ (Q \ v)::'a \text{ assn} \}$$

$$\implies \forall v. G, A \vdash \{ (\lambda w.. P' (\text{the-In2 } w)) \leftarrow \text{Var } v \} t \succ \{ Q \ v \}$$

<proof>

lemma ax-subst-Vals-allI:

$$(\forall v. G, (A::'a \text{ triple set}) \vdash \{ (P' \quad v) \leftarrow \text{Vals } v \} t \succ \{ (Q \ v)::'a \text{ assn} \})$$

$$\implies \forall v. G, A \vdash \{ (\lambda w.. P' (\text{the-In3 } w)) \leftarrow \text{Vals } v \} t \succ \{ Q \ v \}$$

<proof>

alternative axioms

lemma ax-Lit2:

$$G, (A::'a \text{ triple set}) \vdash \{ \text{Normal } P::'a \text{ assn} \} \text{Lit } v \succ \{ \text{Normal } (P \downarrow = \text{Val } v) \}$$

<proof>

lemma ax-Lit2-test-complete:

$$G, (A::'a \text{ triple set}) \vdash \{ \text{Normal } (P \leftarrow \text{Val } v)::'a \text{ assn} \} \text{Lit } v \succ \{ P \}$$

<proof>

lemma ax-LVar2: $G, (A::'a \text{ triple set}) \vdash \{ \text{Normal } P::'a \text{ assn} \} \text{LVar } vn \succ \{ \text{Normal } (\lambda s.. P \downarrow = \text{Var } (\text{lvar } vn \ s)) \}$

<proof>

lemma ax-Super2: $G, (A::'a \text{ triple set}) \vdash$

$$\{ \text{Normal } P::'a \text{ assn} \} \text{Super} \succ \{ \text{Normal } (\lambda s.. P \downarrow = \text{Val } (\text{val-this } s)) \}$$

<proof>

lemma *ax-Nil2*:

$G, (A::'a \text{ triple set}) \vdash \{ \text{Normal } P::'a \text{ assn} \} \square \doteq \doteq \{ \text{Normal } (P \downarrow = \text{Vals } \square) \}$
 ⟨proof⟩

misc derived structural rules

lemma *ax-finite-mtriples-lemma*: $\llbracket F \subseteq ms; \text{finite } ms; \forall (C, sig) \in ms.$

$G, (A::'a \text{ triple set}) \vdash \{ \text{Normal } (P \ C \ sig)::'a \text{ assn} \} \text{ mb } C \ sig \dashv \dashv \{ Q \ C \ sig \} \rrbracket \implies$
 $G, A \vdash \{ \{ P \} \text{ mb } \dashv \dashv \{ Q \} \mid F \}$

⟨proof⟩

lemmas *ax-finite-mtriples* = *ax-finite-mtriples-lemma* [*OF subset-refl*]

lemma *ax-derivs-insertD*:

$G, (A::'a \text{ triple set}) \vdash \text{insert } (t::'a \text{ triple}) \ ts \implies G, A \vdash t \wedge G, A \vdash ts$
 ⟨proof⟩

lemma *ax-methods-spec*:

$\llbracket G, (A::'a \text{ triple set}) \vdash \text{case-prod } f \ ' \ ms; (C, sig) \in ms \rrbracket \implies G, A \vdash ((f \ C \ sig)::'a \text{ triple})$
 ⟨proof⟩

lemma *ax-finite-pointwise-lemma* [*rule-format*]: $\llbracket F \subseteq ms; \text{finite } ms \rrbracket \implies$

$(\forall (C, sig) \in F. G, (A::'a \text{ triple set}) \vdash (f \ C \ sig)::'a \text{ triple})) \longrightarrow (\forall (C, sig) \in ms. G, A \vdash (g \ C \ sig)::'a \text{ triple})) \longrightarrow$
 $G, A \vdash \text{case-prod } f \ ' \ F \longrightarrow G, A \vdash \text{case-prod } g \ ' \ F$

⟨proof⟩

lemmas *ax-finite-pointwise* = *ax-finite-pointwise-lemma* [*OF subset-refl*]

lemma *ax-no-hazard*:

$G, (A::'a \text{ triple set}) \vdash \{ P \ \wedge. \ \text{type-ok } G \ t \} \ t \dashv \dashv \{ Q::'a \text{ assn} \} \implies G, A \vdash \{ P \} \ t \dashv \dashv \{ Q \}$
 ⟨proof⟩

lemma *ax-free-wt*:

$(\exists T \ L \ C. (\text{prg} = G, \text{cls} = C, \text{lcl} = L) \vdash t::T)$
 $\longrightarrow G, (A::'a \text{ triple set}) \vdash \{ \text{Normal } P \} \ t \dashv \dashv \{ Q::'a \text{ assn} \} \implies$
 $G, A \vdash \{ \text{Normal } P \} \ t \dashv \dashv \{ Q \}$

⟨proof⟩

⟨ML⟩

declare *ax-Abrupts* [*intro!*]

lemmas *ax-Normal-cases* = *ax-cases* [*of - - - normal*]

lemma *ax-Skip* [*intro!*]: $G, (A::'a \text{ triple set}) \vdash \{ P \leftarrow \diamond \} \ . \text{Skip}. \{ P::'a \text{ assn} \}$

⟨proof⟩

lemmas *ax-SkipI* = *ax-Skip* [*THEN conseq1*]

derived rules for methd call

lemma *ax-Call-known-DynT*:

$\llbracket G \vdash \text{IntVir} \rightarrow C \preceq \text{statT};$
 $\forall a \ \text{vs } l. G, A \vdash \{ (R \ a \leftarrow \text{Vals } \text{vs} \ \wedge. (\lambda s. l = \text{locals } (store \ s))) ;$
 $\text{init-lvars } G \ C \ (\text{name} = mn, \text{parTs} = pTs) \ \text{IntVir } a \ \text{vs} \} \rrbracket$

$$\text{Methd } C \ (\!| \text{name=mn,parTs=pTs} \!) \text{-} \succ \ \{ \text{set-lvars } l \ ; \ S \};$$

$$\forall a. G, A \vdash \{ Q \leftarrow \text{Val } a \} \text{ args} \doteq \succ$$

$$\{ R \ a \ \wedge. \ (\lambda s. C = \text{obj-class } (\text{the } (\text{heap } (\text{store } s) \ (\text{the-Addr } a)))) \ \wedge$$

$$C = \text{invocation-declclass}$$

$$G \ \text{IntVir } (\text{store } s) \ a \ \text{statT } (\!| \text{name=mn,parTs=pTs} \!) \ \};$$

$$G, (A :: 'a \ \text{triple set}) \vdash \{ \text{Normal } P \} \ e \text{-} \succ \ \{ Q :: 'a \ \text{assn} \}$$

$$\implies G, A \vdash \{ \text{Normal } P \} \ \{ \text{accC, statT, IntVir} \} e \cdot \text{mn}(\{ pTs \} \text{args}) \text{-} \succ \ \{ S \}$$

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

lemma ax-Call-Static:

$$\llbracket \forall a \ \text{vs } l. G, A \vdash \{ R \ a \leftarrow \text{Vals } \text{vs} \ \wedge. \ (\lambda s. l = \text{locals } (\text{store } s)) \ ; \ ;$$

$$\text{init-lvars } G \ C \ (\!| \text{name=mn,parTs=pTs} \!) \ \text{Static any-Addr vs} \}$$

$$\text{Methd } C \ (\!| \text{name=mn,parTs=pTs} \!) \text{-} \succ \ \{ \text{set-lvars } l \ ; \ S \};$$

$$G, A \vdash \{ \text{Normal } P \} \ e \text{-} \succ \ \{ Q \};$$

$$\forall a. G, (A :: 'a \ \text{triple set}) \vdash \{ Q \leftarrow \text{Val } a \} \text{ args} \doteq \succ \ \{ (R :: \text{val} \Rightarrow 'a \ \text{assn}) \ a$$

$$\wedge. \ (\lambda s. C = \text{invocation-declclass}$$

$$G \ \text{Static } (\text{store } s) \ a \ \text{statT } (\!| \text{name=mn,parTs=pTs} \!)) \}$$

$$\rrbracket \implies G, A \vdash \{ \text{Normal } P \} \ \{ \text{accC, statT, Static} \} e \cdot \text{mn}(\{ pTs \} \text{args}) \text{-} \succ \ \{ S \}$$

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

lemma ax-Methd1:

$$\llbracket G, A \cup \{ \{ P \} \ \text{Methd} \text{-} \succ \ \{ Q \} \mid ms \} \vdash \{ \{ P \} \ \text{body } G \text{-} \succ \ \{ Q \} \mid ms \}; \ (C, \text{sig}) \in ms \rrbracket \implies$$

$$G, A \vdash \{ \text{Normal } (P \ C \ \text{sig}) \} \ \text{Methd } C \ \text{sig} \text{-} \succ \ \{ Q \ C \ \text{sig} \}$$

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

lemma ax-MethdN:

$$G, \text{insert}(\{ \text{Normal } P \} \ \text{Methd } C \ \text{sig} \text{-} \succ \ \{ Q \}) \ A \vdash$$

$$\{ \text{Normal } P \} \ \text{body } G \ C \ \text{sig} \text{-} \succ \ \{ Q \} \implies$$

$$G, A \vdash \{ \text{Normal } P \} \ \text{Methd } C \ \text{sig} \text{-} \succ \ \{ Q \}$$

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

lemma ax-StatRef:

$$G, (A :: 'a \ \text{triple set}) \vdash \{ \text{Normal } (P \leftarrow \text{Val } \text{Null}) \} \ \text{StatRef } \text{rt} \text{-} \succ \ \{ P :: 'a \ \text{assn} \}$$

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

rules derived from Init and Done

lemma ax-InitS: $\llbracket \text{the } (\text{class } G \ C) = c; C \neq \text{Object};$

$$\forall l. G, A \vdash \{ Q \ \wedge. \ (\lambda s. l = \text{locals } (\text{store } s)) \ ; \ ; \ \text{set-lvars } \text{Map.empty} \}$$

$$\ . \text{init } c. \ \{ \text{set-lvars } l \ ; \ R \};$$

$$G, A \vdash \{ \text{Normal } ((P \ \wedge. \ \text{Not} \circ \text{initd } C) \ ; \ ; \ \text{supd } (\text{init-class-obj } G \ C)) \}$$

$$\ . \text{Init } (\text{super } c). \ \{ Q \} \rrbracket \implies$$

$$G, (A :: 'a \ \text{triple set}) \vdash \{ \text{Normal } (P \ \wedge. \ \text{Not} \circ \text{initd } C) \} \ . \text{Init } C. \ \{ R :: 'a \ \text{assn} \}$$

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

lemma ax-Init-Skip-lemma:

$$\forall l. G, (A :: 'a \ \text{triple set}) \vdash \{ P \leftarrow \diamond \ \wedge. \ (\lambda s. l = \text{locals } (\text{store } s)) \ ; \ ; \ \text{set-lvars } l' \}$$

$$\ . \text{Skip}. \ \{ (\text{set-lvars } l \ ; \ ; \ P) :: 'a \ \text{assn} \}$$

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

lemma ax-triv-InitS: $\llbracket \text{the } (\text{class } G \ C) = c; \text{init } c = \text{Skip}; C \neq \text{Object};$

$$\begin{aligned}
& P \leftarrow \diamond \Rightarrow (\text{supd } (\text{init-class-obj } G \ C) \ .; P); \\
& G, A \vdash \{ \text{Normal } (P \wedge \text{initd } C) \} \ . \text{Init } (\text{super } c) \ . \{ (P \wedge \text{initd } C) \leftarrow \diamond \} \} \Longrightarrow \\
& G, (A :: 'a \text{ triple set}) \vdash \{ \text{Normal } P \leftarrow \diamond \} \ . \text{Init } C \ . \{ (P \wedge \text{initd } C) :: 'a \text{ assn} \} \\
\langle \text{proof} \rangle
\end{aligned}$$

lemma *ax-Init-Object*: $\text{wf-prog } G \Longrightarrow G, (A :: 'a \text{ triple set}) \vdash$
 $\{ \text{Normal } ((\text{supd } (\text{init-class-obj } G \ \text{Object}) \ .; P \leftarrow \diamond) \wedge \text{Not } \circ \text{initd } \text{Object}) \}$
 $\ . \text{Init } \text{Object} \ . \{ (P \wedge \text{initd } \text{Object}) :: 'a \text{ assn} \}$
 $\langle \text{proof} \rangle$

lemma *ax-triv-Init-Object*: $\llbracket \text{wf-prog } G;$
 $(P :: 'a \text{ assn}) \Rightarrow (\text{supd } (\text{init-class-obj } G \ \text{Object}) \ .; P) \rrbracket \Longrightarrow$
 $G, (A :: 'a \text{ triple set}) \vdash \{ \text{Normal } P \leftarrow \diamond \} \ . \text{Init } \text{Object} \ . \{ P \wedge \text{initd } \text{Object} \}$
 $\langle \text{proof} \rangle$

introduction rules for Alloc and SXAlloc

lemma *ax-SXAlloc-Normal*:
 $G, (A :: 'a \text{ triple set}) \vdash \{ P :: 'a \text{ assn} \} \ .c. \{ \text{Normal } Q \}$
 $\Longrightarrow G, A \vdash \{ P \} \ .c. \{ \text{SXAlloc } G \ Q \}$
 $\langle \text{proof} \rangle$

lemma *ax-Alloc*:
 $G, (A :: 'a \text{ triple set}) \vdash \{ P :: 'a \text{ assn} \} \ t \succ$
 $\{ \text{Normal } (\lambda Y \ (x, s) \ Z. (\forall a. \text{new-Addr } (\text{heap } s) = \text{Some } a \longrightarrow$
 $Q \ (\text{Val } (\text{Addr } a)) \ (\text{Norm}(\text{init-obj } G \ (\text{CInst } C) \ (\text{Heap } a) \ s)) \ Z)) \wedge$
 $\text{heap-free } (\text{Suc } (\text{Suc } 0))) \}$
 $\Longrightarrow G, A \vdash \{ P \} \ t \succ \{ \text{Alloc } G \ (\text{CInst } C) \ Q \}$
 $\langle \text{proof} \rangle$

lemma *ax-Alloc-Arr*:
 $G, (A :: 'a \text{ triple set}) \vdash \{ P :: 'a \text{ assn} \} \ t \succ$
 $\{ \lambda \text{Val} : i. \text{Normal } (\lambda Y \ (x, s) \ Z. \neg \text{the-Intg } i < 0 \wedge$
 $(\forall a. \text{new-Addr } (\text{heap } s) = \text{Some } a \longrightarrow$
 $Q \ (\text{Val } (\text{Addr } a)) \ (\text{Norm } (\text{init-obj } G \ (\text{Arr } T \ (\text{the-Intg } i)) \ (\text{Heap } a) \ s)) \ Z)) \wedge$
 $\text{heap-free } (\text{Suc } (\text{Suc } 0))) \}$
 \Longrightarrow
 $G, A \vdash \{ P \} \ t \succ \{ \lambda \text{Val} : i. \text{abupd } (\text{check-neg } i) \ .; \text{Alloc } G \ (\text{Arr } T \ (\text{the-Intg } i)) \ Q \}$
 $\langle \text{proof} \rangle$

lemma *ax-SXAlloc-catch-SXcpt*:
 $\llbracket G, (A :: 'a \text{ triple set}) \vdash \{ P :: 'a \text{ assn} \} \ t \succ$
 $\{ (\lambda Y \ (x, s) \ Z. x = \text{Some } (\text{Xcpt } (\text{Std } xn)) \wedge$
 $(\forall a. \text{new-Addr } (\text{heap } s) = \text{Some } a \longrightarrow$
 $Q \ Y \ (\text{Some } (\text{Xcpt } (\text{Loc } a)), \text{init-obj } G \ (\text{CInst } (\text{SXcpt } xn)) \ (\text{Heap } a) \ s) \ Z))$
 $\wedge \text{heap-free } (\text{Suc } (\text{Suc } 0))) \rrbracket$
 \Longrightarrow
 $G, A \vdash \{ P \} \ t \succ \{ \text{SXAlloc } G \ (\lambda Y \ s \ Z. Q \ Y \ s \ Z \wedge G, s \vdash \text{catch } \text{SXcpt } xn) \}$
 $\langle \text{proof} \rangle$

end

Chapter 23

AxSound

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* (\models :-:[61,0, 58] 57)
where
 $G \models n :: t =$
(*case t of* {*P*} *t* \succ {*Q*} \Rightarrow
 $\forall Y s Z. P Y s Z \longrightarrow (\forall L. s :: \preceq(G, L)$
 $\longrightarrow (\forall T C A. (normal\ s \longrightarrow ((\text{prg}=G, \text{cls}=C, \text{lcl}=L) \vdash t :: T \wedge$
 $(\text{prg}=G, \text{cls}=C, \text{lcl}=L) \vdash \text{dom} (\text{locals} (\text{store } s)) \gg t \gg A)) \longrightarrow$
 $(\forall Y' s'. G \vdash s -t \succ -n \rightarrow (Y', s') \longrightarrow Q Y' s' Z \wedge s' :: \preceq(G, L))))))$

This definition differs from the ordinary *triple-valid-def* manly in the conclusion: We also ensures conformance of the result state. So we don't have to apply the type soundness lemma all the time during induction. This definition is only introduced for the soundness proof of the axiomatic semantics, in the end we will conclude to the ordinary definition.

definition

ax-valids2 :: *prog* \Rightarrow 'a *triples* \Rightarrow 'a *triples* \Rightarrow *bool* (\models :-:[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*: $G \models n :: \{P\} t \succ \{Q\} =$

$(\forall Y s Z. P Y s Z \longrightarrow (\forall Y' s'. G \vdash s -t \succ -n \rightarrow (Y', s') \longrightarrow$
 $(\forall L. s :: \preceq(G, L) \longrightarrow (\forall T C A. (normal\ s \longrightarrow ((\text{prg}=G, \text{cls}=C, \text{lcl}=L) \vdash t :: T \wedge$
 $(\text{prg}=G, \text{cls}=C, \text{lcl}=L) \vdash \text{dom} (\text{locals} (\text{store } s)) \gg t \gg A)) \longrightarrow$
 $Q Y' s' Z \wedge s' :: \preceq(G, L))))))$
<proof>

lemma *triple-valid2-eq* [*rule-format* (*no-asm*)]:

wf-prog *G* \implies *triple-valid2* *G* = *triple-valid* *G*
<proof>

lemma *ax-valids2-eq*: *wf-prog* *G* \implies $G, A \models :: ts = G, A \models ts$

<proof>

lemma *triple-valid2-Suc* [*rule-format* (*no-asm*)]: $G \models Suc\ n :: t \longrightarrow G \models n :: t$

<proof>

lemma *Method-triple-valid2-0*: $G \models 0 :: \{ \text{Normal } P \} \text{ Method } C \text{ sig} \multimap \{ Q \}$

<proof>

lemma *Method-triple-valid2-SucI*:

$\llbracket G \models n :: \{ \text{Normal } P \} \text{ body } G \text{ C sig} \multimap \{ Q \} \rrbracket$

$\implies G \models \text{Suc } n :: \{ \text{Normal } P \} \text{ Method } C \text{ sig} \multimap \{ Q \}$

<proof>

lemma *triples-valid2-Suc*:

$\text{Ball } ts \text{ (triple-valid2 } G \text{ (Suc } n)) \implies \text{Ball } ts \text{ (triple-valid2 } G \text{ } n)$

<proof>

lemma $G \models n :: \text{insert } t \text{ } A = (G \models n :: t \wedge G \models n :: A)$

<proof>

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 $G, A \models \{ \{ P \} \text{ Method} \multimap \{ Q \} \mid ms \}$

<proof>

lemma *valids2-inductI*: $\forall s \ t \ n \ Y' \ s'. G \vdash s \multimap \multimap n \rightarrow (Y', s') \longrightarrow t = c \longrightarrow$

$\text{Ball } A \text{ (triple-valid2 } G \text{ } n) \longrightarrow (\forall Y \ Z. P \ Y \ s \ Z \longrightarrow$

$(\forall L. s :: \preceq (G, L) \longrightarrow$

$(\forall T \ C \ A. (\text{normal } s \longrightarrow (\llbracket \text{prg} = G, \text{cls} = C, \text{lcl} = L \rrbracket \vdash t :: T) \wedge$

$(\llbracket \text{prg} = G, \text{cls} = C, \text{lcl} = L \rrbracket \vdash \text{dom } (\text{locals } (\text{store } s)) \gg t \gg A) \longrightarrow$

$Q \ Y' \ s' \ Z \wedge s' :: \preceq (G, L))) \implies$

$G, A \models \{ \{ P \} \text{ c} \multimap \{ Q \} \}$

<proof>

lemma *da-good-approx-evalnE* [*consumes 4*]:

assumes *evaln*: $G \vdash s0 \multimap \multimap n \rightarrow (v, s1)$

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

<proof>

lemma *validI*:

assumes $I: \bigwedge n \ s0 \ L \ \text{acc} \ C \ T \ C \ v \ s1 \ Y \ Z.$

$\llbracket \forall t \in A. G \models n :: t; s0 :: \preceq (G, L);$

$\text{normal } s0 \implies (\llbracket \text{prg} = G, \text{cls} = \text{acc} \ C, \text{lcl} = L \rrbracket \vdash t :: T;$

$$\text{normal } s0 \implies (\text{prg}=G, \text{cls}=\text{acc}C, \text{lcl}=L) \vdash \text{dom } (\text{locals } (\text{store } s0)) \gg t \gg C;$$

$$G \vdash s0 -t \succ -n \rightarrow (v, s1); P \ Y \ s0 \ Z \implies Q \ v \ s1 \ Z \ \wedge \ s1 :: \preceq (G, L)$$

shows $G, A \models :: \{ \{P\} \ t \succ \{Q\} \}$
 ⟨proof⟩

declare $[[\text{simproc } \text{add}: \text{wt-expr } \text{wt-var } \text{wt-exprs } \text{wt-stmt}]]$

lemma *valid-stmtI*:

assumes $I: \bigwedge n \ s0 \ L \ \text{acc}C \ C \ s1 \ Y \ Z.$

$\forall t \in A. G \models n :: t; s0 :: \preceq (G, L);$

$\text{normal } s0 \implies (\text{prg}=G, \text{cls}=\text{acc}C, \text{lcl}=L) \vdash c :: \surd;$

$\text{normal } s0 \implies (\text{prg}=G, \text{cls}=\text{acc}C, \text{lcl}=L) \vdash \text{dom } (\text{locals } (\text{store } s0)) \gg \langle c \rangle_s \gg C;$

$G \vdash s0 -c -n \rightarrow s1; P \ Y \ s0 \ Z \implies Q \ \diamond \ s1 \ Z \ \wedge \ s1 :: \preceq (G, L)$

shows $G, A \models :: \{ \{P\} \ \langle c \rangle_s \succ \{Q\} \}$
 ⟨proof⟩

lemma *valid-stmt-NormalI*:

assumes $I: \bigwedge n \ s0 \ L \ \text{acc}C \ C \ s1 \ Y \ Z.$

$\forall t \in A. G \models n :: t; s0 :: \preceq (G, L); \text{normal } s0; (\text{prg}=G, \text{cls}=\text{acc}C, \text{lcl}=L) \vdash c :: \surd;$

$(\text{prg}=G, \text{cls}=\text{acc}C, \text{lcl}=L) \vdash \text{dom } (\text{locals } (\text{store } s0)) \gg \langle c \rangle_s \gg C;$

$G \vdash s0 -c -n \rightarrow s1; (\text{Normal } P) \ Y \ s0 \ Z \implies Q \ \diamond \ s1 \ Z \ \wedge \ s1 :: \preceq (G, L)$

shows $G, A \models :: \{ \{ \text{Normal } P \} \ \langle c \rangle_s \succ \{Q\} \}$
 ⟨proof⟩

lemma *valid-var-NormalI*:

assumes $I: \bigwedge n \ s0 \ L \ \text{acc}C \ T \ C \ \text{vf} \ s1 \ Y \ Z.$

$\forall t \in A. G \models n :: t; s0 :: \preceq (G, L); \text{normal } s0;$

$(\text{prg}=G, \text{cls}=\text{acc}C, \text{lcl}=L) \vdash t :: T;$

$(\text{prg}=G, \text{cls}=\text{acc}C, \text{lcl}=L) \vdash \text{dom } (\text{locals } (\text{store } s0)) \gg \langle t \rangle_v \gg C;$

$G \vdash s0 -t \succ \text{vf} -n \rightarrow s1; (\text{Normal } P) \ Y \ s0 \ Z \implies$

$\implies Q \ (\text{In2 } \text{vf}) \ s1 \ Z \ \wedge \ s1 :: \preceq (G, L)$

shows $G, A \models :: \{ \{ \text{Normal } P \} \ \langle t \rangle_v \succ \{Q\} \}$
 ⟨proof⟩

lemma *valid-expr-NormalI*:

assumes $I: \bigwedge n \ s0 \ L \ \text{acc}C \ T \ C \ v \ s1 \ Y \ Z.$

$\forall t \in A. G \models n :: t; s0 :: \preceq (G, L); \text{normal } s0;$

$(\text{prg}=G, \text{cls}=\text{acc}C, \text{lcl}=L) \vdash t :: -T;$

$(\text{prg}=G, \text{cls}=\text{acc}C, \text{lcl}=L) \vdash \text{dom } (\text{locals } (\text{store } s0)) \gg \langle t \rangle_e \gg C;$

$G \vdash s0 -t \succ v -n \rightarrow s1; (\text{Normal } P) \ Y \ s0 \ Z \implies$

$\implies Q \ (\text{In1 } v) \ s1 \ Z \ \wedge \ s1 :: \preceq (G, L)$

shows $G, A \models :: \{ \{ \text{Normal } P \} \ \langle t \rangle_e \succ \{Q\} \}$
 ⟨proof⟩

lemma *valid-expr-list-NormalI*:

assumes $I: \bigwedge n \ s0 \ L \ \text{acc}C \ T \ C \ \text{vs} \ s1 \ Y \ Z.$

$\forall t \in A. G \models n :: t; s0 :: \preceq (G, L); \text{normal } s0;$

$(\text{prg}=G, \text{cls}=\text{acc}C, \text{lcl}=L) \vdash t :: \doteq T;$

$(\text{prg}=G, \text{cls}=\text{acc}C, \text{lcl}=L) \vdash \text{dom } (\text{locals } (\text{store } s0)) \gg \langle t \rangle_l \gg C;$

$G \vdash s0 -t \doteq \text{vs} -n \rightarrow s1; (\text{Normal } P) \ Y \ s0 \ Z \implies$

$\implies Q \ (\text{In3 } \text{vs}) \ s1 \ Z \ \wedge \ s1 :: \preceq (G, L)$

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*: $normal \ s0 \ \Longrightarrow \ (\text{prg} = G, \text{cls} = \text{acc}C, \text{lcl} = L) \vdash t :: T$
and *da*: $normal \ s0 \ \Longrightarrow \ (\text{prg} = G, \text{cls} = \text{acc}C, \text{lcl} = L) \vdash \text{dom} \ (\text{locals} \ (\text{store} \ s0)) \gg t \gg C$
and *elim*: $\llbracket Q \ v \ s1 \ Z; s1 :: \preceq(G, L) \rrbracket \Longrightarrow \text{concl}$
shows *concl*
 $\langle \text{proof} \rangle$

lemma *all-empty*: $(\forall x. P) = P$
 $\langle \text{proof} \rangle$

corollary *evaln-type-sound*:
assumes *evaln*: $G \vdash s0 \ -t \succ -n \rightarrow (v, s1)$ **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 $s1 :: \preceq(G, L) \wedge (normal \ s1 \ \longrightarrow \ G, L, \text{store} \ s1 \vdash t \succ v :: \preceq T) \wedge$
 $(error\text{-free} \ s0 = error\text{-free} \ s1)$
 $\langle \text{proof} \rangle$

corollary *dom-locals-evaln-mono-elim* [consumes 1]:
assumes
evaln: $G \vdash s0 \ -t \succ -n \rightarrow (v, s1)$ **and**
hyps: $\llbracket \text{dom} \ (\text{locals} \ (\text{store} \ s0)) \subseteq \text{dom} \ (\text{locals} \ (\text{store} \ s1));$
 $\wedge \ v \ s \ \text{val}. \llbracket v = \text{In2} \ v \ v; normal \ s1 \rrbracket$
 $\Longrightarrow \text{dom} \ (\text{locals} \ (\text{store} \ s))$
 $\subseteq \text{dom} \ (\text{locals} \ (\text{store} \ ((\text{snd} \ v \ v) \ \text{val} \ s))) \rrbracket \Longrightarrow P$
shows P
 $\langle \text{proof} \rangle$

lemma *evaln-no-abrupt*:
 $\wedge s \ s'. \llbracket G \vdash s \ -t \succ -n \rightarrow (w, s') \rrbracket; normal \ s' \rrbracket \Longrightarrow normal \ s$
 $\langle \text{proof} \rangle$

declare *inj-term-simps* [simp]

lemma *ax-sound2*:
assumes $wf: wf\text{-prog} \ G$
and *deriv*: $G, A \vdash ts$
shows $G, A \models :: ts$
 $\langle \text{proof} \rangle$
declare *inj-term-simps* [simp del]

theorem *ax-sound*:
 $wf\text{-prog} \ G \Longrightarrow G, (A :: 'a \ \text{triple} \ \text{set}) \vdash (ts :: 'a \ \text{triple} \ \text{set}) \Longrightarrow G, A \models ts$
 $\langle \text{proof} \rangle$

lemma *sound-valid2-lemma*:

$\llbracket \forall v n. \text{Ball } A \text{ (triple-valid2 } G \ n) \longrightarrow P \ v \ n; \text{Ball } A \text{ (triple-valid2 } G \ n) \rrbracket$
 $\implies P \ v \ n$

<proof>

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 (-j, 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]:
 $nyinitcls\ G\ ((init\ lvars\ G\ C\ sig\ mode\ a'\ pvs)\ s) = nyinitcls\ G\ s$
 ⟨proof⟩

lemma *nyinitcls-emptyD*: $\llbracket nyinitcls\ G\ s = \{\};\ is\ class\ G\ C \rrbracket \implies initd\ C\ s$
 ⟨proof⟩

lemma *card-Suc-lemma*:
 $\llbracket card\ (insert\ a\ A) \leq Suc\ n;\ a \notin A;\ finite\ A \rrbracket \implies card\ A \leq n$
 ⟨proof⟩

lemma *nyinitcls-le-SucD*:
 $\llbracket card\ (nyinitcls\ G\ (x,s)) \leq Suc\ n;\ \neg\ initd\ C\ (globs\ s);\ class\ G\ C = Some\ y \rrbracket \implies$
 $card\ (nyinitcls\ G\ (x,init\ class\ obj\ G\ C\ s)) \leq n$
 ⟨proof⟩

lemma *initd-gext'*: $\llbracket s \leq |s';\ initd\ C\ (globs\ s) \rrbracket \implies initd\ C\ (globs\ s')$
 ⟨proof⟩

lemma *nyinitcls-gext*: $snd\ s \leq |snd\ s' \implies nyinitcls\ G\ s' \subseteq nyinitcls\ G\ s$
 ⟨proof⟩

lemma *card-nyinitcls-gext*:
 $\llbracket snd\ s \leq |snd\ s';\ card\ (nyinitcls\ G\ s) \leq n \rrbracket \implies card\ (nyinitcls\ G\ s') \leq n$
 ⟨proof⟩

init-le

definition

init-le :: *prog* \Rightarrow *nat* \Rightarrow *state* \Rightarrow *bool* (+*init-le*- [51,51] 50)
 where $G \vdash init \leq n = (\lambda s. card\ (nyinitcls\ G\ s) \leq n)$

lemma *init-le-def2* [simp]: $(G \vdash init \leq n)\ s = (card\ (nyinitcls\ G\ s) \leq n)$
 ⟨proof⟩

lemma

All-init-leD:
 $\forall n::nat. G, (A::'a\ triple\ set) \vdash \{P \wedge. G \vdash init \leq n\} t > \{Q::'a\ assn\}$
 $\implies G, A \vdash \{P\} t > \{Q\}$
 ⟨proof⟩

Most General Triples and Formulas

definition

remember-init-state :: *state assn* (\doteq)

where $\dot{=} \equiv \lambda Y s Z. s = Z$

lemma *remember-init-state-def2* [simp]: $\dot{=} Y = (=)$

<proof>

definition

MGF :: [state assn, term, prog] \Rightarrow state triple ($\{-\} \dashv \rightarrow \{-\}$) [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 ($\{=:n\} \dashv \rightarrow \{-\}$) [3,65,3]62)

where $\{=:n\} t \succ \{G \rightarrow\} = \{\dot{=} \wedge. G \vdash \text{init} \leq n\} t \succ \{G \rightarrow\}$

lemma *MGF-valid*: $wf\text{-prog } G \Longrightarrow G, \{\dot{=}\} \vdash \{\dot{=}\} t \succ \{G \rightarrow\}$

<proof>

lemma *MGF-res-eq-lemma* [simp]:

$(\forall Y' Y s. Y = Y' \wedge P s \longrightarrow Q s) = (\forall s. P s \longrightarrow Q s)$

<proof>

lemma *MGFn-def2*:

$G, A \vdash \{=:n\} t \succ \{G \rightarrow\} = G, A \vdash \{\dot{=} \wedge. G \vdash \text{init} \leq n\}$

$t \succ \{\lambda Y s' s. G \vdash s - t \succ \rightarrow (Y, s')\}$

<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*:

$G, (A :: \text{state triple set}) \vdash \{=:n\} t \succ \{G \rightarrow\} \Longrightarrow$

$G, A \vdash \{(\lambda Y' s' s. s' = s \wedge P s) \wedge. G \vdash \text{init} \leq n\}$

$t \succ \{(\lambda Y' s' s. G \vdash s - t \succ \rightarrow (Y', s') \wedge P s) \wedge. G \vdash \text{init} \leq n\}$

<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*: $G, A \vdash \{\text{Normal } \dot{=}\} t \succ \{G \rightarrow\} \Longrightarrow$

$G, (A :: \text{state triple set}) \vdash \{\dot{=} :: \text{state assn}\} t \succ \{G \rightarrow\}$

<proof>

lemma *MGFNormalD*:

$G, (A :: \text{state triple set}) \vdash \{\dot{=}\} t \succ \{G \rightarrow\} \Longrightarrow G, A \vdash \{\text{Normal } \dot{=}\} t \succ \{G \rightarrow\}$

<proof>

Additionally to *MGFNormalI*, we also expand the definition of the most general formula here

lemma *MGFn-NormalI*:

$G, (A :: \text{state triple set}) \vdash \{\text{Normal}((\lambda Y' s' s. s' = s \wedge \text{normal } s) \wedge. G \vdash \text{init} \leq n)\} t \succ$

$$\{\lambda Y s' s. G \vdash s - t \succ \rightarrow (Y, s')\} \implies G, A \vdash \{=:n\} t \succ \{G \rightarrow\}$$

<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*:

$$\begin{aligned} & (\exists T L C. (\text{prg}=G, \text{cls}=C, \text{lcl}=L) \vdash t :: T) \\ & \longrightarrow G, (A :: \text{state triple set}) \vdash \{=:n\} t \succ \{G \rightarrow\} \\ & \implies G, A \vdash \{=:n\} t \succ \{G \rightarrow\} \end{aligned}$$

<proof>

To derive the most general formula, we can restrict ourselves to welltyped terms and assume that the state in the precondition conforms to the environment. All type violations can be uniformly handled by the hazard rule.

lemma *MGFn-free-wt-NormalConformI*:

$$\begin{aligned} & (\forall T L C. (\text{prg}=G, \text{cls}=C, \text{lcl}=L) \vdash t :: T) \\ & \longrightarrow G, (A :: \text{state triple set}) \\ & \quad \vdash \{ \text{Normal}((\lambda Y' s' s. s'=s \wedge \text{normal } s) \wedge. G \vdash \text{init} \leq n) \wedge. (\lambda s. s :: \preceq(G, L)) \} \\ & \quad t \succ \\ & \quad \{ \lambda Y s' s. G \vdash s - t \succ \rightarrow (Y, s') \} \\ & \implies G, A \vdash \{=:n\} t \succ \{G \rightarrow\} \end{aligned}$$

<proof>

To derive the most general formula, we can restrict ourselves to welltyped terms and assume that the state in the precondition conforms to the environment and that the term is definitely assigned with respect to this state. All type violations can be uniformly handled by the hazard rule.

lemma *MGFn-free-wt-da-NormalConformI*:

$$\begin{aligned} & (\forall T L C B. (\text{prg}=G, \text{cls}=C, \text{lcl}=L) \vdash t :: T) \\ & \longrightarrow G, (A :: \text{state triple set}) \\ & \quad \vdash \{ \text{Normal}((\lambda Y' s' s. s'=s \wedge \text{normal } s) \wedge. G \vdash \text{init} \leq n) \wedge. (\lambda s. s :: \preceq(G, L)) \\ & \quad \wedge. (\lambda s. (\text{prg}=G, \text{cls}=C, \text{lcl}=L) \vdash \text{dom } (\text{locals } (\text{store } s)) \gg t \gg B) \} \\ & \quad t \succ \\ & \quad \{ \lambda Y s' s. G \vdash s - t \succ \rightarrow (Y, s') \} \\ & \implies G, A \vdash \{=:n\} t \succ \{G \rightarrow\} \end{aligned}$$

<proof>

main lemmas

lemma *MGFn-Init*:

assumes *mgf-hyp*: $\forall m. \text{Suc } m \leq n \longrightarrow (\forall t. G, A \vdash \{=:m\} t \succ \{G \rightarrow\})$
shows $G, (A :: \text{state triple set}) \vdash \{=:n\} \langle \text{Init } C \rangle_s \succ \{G \rightarrow\}$

<proof>

lemmas *MGFn-InitD* = *MGFn-Init* [THEN *MGFnD*, THEN *ax-NormalD*]

lemma *MGFn-Call*:

assumes *mgf-methods*:
 $\forall C \text{ sig}. G, (A :: \text{state triple set}) \vdash \{=:n\} \langle (\text{Methd } C \text{ sig}) \rangle_e \succ \{G \rightarrow\}$
and *mgf-e*: $G, A \vdash \{=:n\} \langle e \rangle_e \succ \{G \rightarrow\}$
and *mgf-ps*: $G, A \vdash \{=:n\} \langle ps \rangle_l \succ \{G \rightarrow\}$
and *wf*: *wf-prog* *G*
shows $G, A \vdash \{=:n\} \langle \{ \text{acc } C, \text{stat } T, \text{mode} \} e \cdot \text{mn}(\{ pTs' \} ps) \rangle_e \succ \{G \rightarrow\}$

<proof>

lemma *eval-expression-no-jump'*:

assumes *eval*: $G \vdash s0 - e \succ v \rightarrow s1$
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 $wf: wf\text{-prog } G$
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

$\text{unroll} :: \text{prog} \Rightarrow \text{label} \Rightarrow \text{expr} \Rightarrow \text{stmt} \Rightarrow (\text{state} \times \text{state}) \text{ set}$ **where**
 $\text{unroll } G \ l \ e \ c = \{(s, t). \exists v \ s1 \ s2.$
 $\quad G \vdash s \ -e \rightarrow v \rightarrow s1 \wedge \text{the-Bool } v \wedge \text{normal } s1 \wedge$
 $\quad G \vdash s1 \ -c \rightarrow s2 \wedge t = (\text{abupd } (\text{absorb } (\text{Cont } l)) \ s2)\}$

lemma unroll-while:

assumes $\text{unroll}: (s, t) \in (\text{unroll } G \ l \ e \ c)^*$
and $\text{eval-e}: G \vdash t \ -e \rightarrow v \rightarrow s'$
and $\text{normal-termination}: \text{normal } s' \longrightarrow \neg \text{the-Bool } v$
and $wt: (\text{prg}=G, \text{cls}=C, \text{lcl}=L) \vdash e :: -T$
and $wf: wf\text{-prog } G$
shows $G \vdash s \ -l \cdot \text{While}(e) \ c \rightarrow s'$
 $\langle \text{proof} \rangle$

lemma MGFn-Loop:

assumes $\text{mfg-e}: G, (A :: \text{state triple set}) \vdash \{=:n\} \langle e \rangle_{e \succ} \{G \rightarrow\}$
and $\text{mfg-c}: G, A \vdash \{=:n\} \langle c \rangle_{s \succ} \{G \rightarrow\}$
and $wf: wf\text{-prog } G$
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 $\text{mfg-init}: G, A \vdash \{=:n\} \langle \text{Init statDeclC} \rangle_{s \succ} \{G \rightarrow\}$
and $\text{mfg-e}: G, A \vdash \{=:n\} \langle e \rangle_{e \succ} \{G \rightarrow\}$
and $wf: wf\text{-prog } G$
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 $wf: wf\text{-prog } G$
and $\text{mfg-c1}: G, A \vdash \{=:n\} \langle c1 \rangle_{s \succ} \{G \rightarrow\}$
and $\text{mfg-c2}: G, A \vdash \{=:n\} \langle c2 \rangle_{s \succ} \{G \rightarrow\}$
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 $\text{eval-init}: G \vdash \text{Norm } s0 \ -\text{Init } D \rightarrow s1$
and $\text{eval-c}: G \vdash s1 \ -c \rightarrow s2$
and $\text{jmpOk}: \text{jumpNestingOkS } \{\text{Ret}\} \ c$
and $wt\text{-c}: (\text{prg}=G, \text{cls}=C, \text{lcl}=L) \vdash c :: \checkmark$

and $clsD: class\ G\ D = Some\ d$
and $wf: wf\text{-}prog\ G$
shows $\forall l. abrupt\ s2 \neq Some\ (Jump\ (Break\ l)) \wedge$
 $abrupt\ s2 \neq Some\ (Jump\ (Cont\ l))$
 <proof>

lemma *MGFn-Body*:

assumes $wf: wf\text{-}prog\ G$
and $mgf\text{-}init: G, A \vdash \{=:n\} \langle Init\ D \rangle_s \succ \{G \rightarrow\}$
and $mgf\text{-}c: G, A \vdash \{=:n\} \langle c \rangle_s \succ \{G \rightarrow\}$
shows $G, (A::state\ triple\ set) \vdash \{=:n\} \langle Body\ D\ c \rangle_e \succ \{G \rightarrow\}$
 <proof>

lemma *MGFn-lemma*:

assumes $mgf\text{-}methds:$
 $\bigwedge n. \forall C\ sig. G, (A::state\ triple\ set) \vdash \{=:n\} \langle Methd\ C\ sig \rangle_e \succ \{G \rightarrow\}$
and $wf: wf\text{-}prog\ G$
shows $\bigwedge t. G, A \vdash \{=:n\} t \succ \{G \rightarrow\}$
 <proof>

lemma *MGF-asm*:

$\llbracket \forall C\ sig. is\text{-}methd\ G\ C\ sig \longrightarrow G, A \vdash \{\doteq\} In1l\ (Methd\ C\ sig) \succ \{G \rightarrow\}; wf\text{-}prog\ G \rrbracket$
 $\implies G, (A::state\ triple\ set) \vdash \{\doteq\} t \succ \{G \rightarrow\}$
 <proof>

nested version

lemma *nesting-lemma' [rule-format (no-asm)]*:

assumes $ax\text{-}derivs\text{-}asm: \bigwedge A\ ts. ts \subseteq A \implies P\ A\ ts$
and $MGF\text{-}nested\text{-}Methd: \bigwedge A\ pn. \forall b \in bdy\ pn. P\ (insert\ (mgf\text{-}call\ pn)\ A)\ \{mgf\ b\}$
 $\implies P\ A\ \{mgf\text{-}call\ pn\}$
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 $\forall A. A \subseteq uA \longrightarrow n \leq card\ uA \longrightarrow card\ A = card\ uA - n$
 $\longrightarrow (\forall t. P\ A\ \{mgf\ t\})$
 <proof>

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 $P\ \{\} \ \{mgf\ t\}$
 <proof>

lemma *MGF-nested-Methd*: \llbracket

$G, insert\ (\{Normal\ \doteq\} \langle Methd\ C\ sig \rangle_e \succ \{G \rightarrow\})\ A$
 $\vdash \{Normal\ \doteq\} \langle body\ G\ C\ sig \rangle_e \succ \{G \rightarrow\}$
 $\rrbracket \implies G, A \vdash \{Normal\ \doteq\} \langle Methd\ C\ sig \rangle_e \succ \{G \rightarrow\}$
 <proof>

lemma *MGF-deriv*: $wf\text{-prog } G \implies G, (\{\} :: \text{state triple set}) \vdash \{\dot{=}\} t \succ \{G \rightarrow\}$
 ⟨proof⟩

simultaneous version

lemma *MGF-simult-Methd-lemma*: $\text{finite } ms \implies$
 $G, A \cup (\lambda(C, sig). \{Normal \dot{=}\} \langle Methd \ C \ sig \rangle_e \succ \{G \rightarrow\}) \text{ ' } ms$
 $\vdash (\lambda(C, sig). \{Normal \dot{=}\} \langle body \ G \ C \ sig \rangle_e \succ \{G \rightarrow\}) \text{ ' } ms \implies$
 $G, A \vdash (\lambda(C, sig). \{Normal \dot{=}\} \langle Methd \ C \ sig \rangle_e \succ \{G \rightarrow\}) \text{ ' } ms$
 ⟨proof⟩

lemma *MGF-simult-Methd*: $wf\text{-prog } G \implies$
 $G, (\{\} :: \text{state triple set}) \vdash (\lambda(C, sig). \{Normal \dot{=}\} \langle Methd \ C \ sig \rangle_e \succ \{G \rightarrow\})$
 ‘ *Collect (case-prod (is-methd G))*
 ⟨proof⟩

corollaries

lemma *eval-to-evaln*: $\llbracket G \vdash s \text{ --} t \succ \rightarrow (Y', s'); \text{type-ok } G \ t \ s; \text{wf-prog } G \rrbracket$
 $\implies \exists n. G \vdash s \text{ --} t \succ \text{--} n \rightarrow (Y', s')$
 ⟨proof⟩

lemma *MGF-complete*:
assumes *valid*: $G, \{\} \models \{P\} t \succ \{Q\}$
and *mgf*: $G, (\{\} :: \text{state triple set}) \vdash \{\dot{=}\} t \succ \{G \rightarrow\}$
and *wf*: $wf\text{-prog } G$
shows $G, (\{\} :: \text{state triple set}) \vdash \{P :: \text{state assn}\} t \succ \{Q\}$
 ⟨proof⟩

theorem *ax-complete*:
assumes *wf*: $wf\text{-prog } G$
and *valid*: $G, \{\} \models \{P :: \text{state assn}\} t \succ \{Q\}$
shows $G, (\{\} :: \text{state triple set}) \vdash \{P\} t \succ \{Q\}$
 ⟨proof⟩

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:*

$Q = (\lambda Y (x,s) Z. x \neq \text{None} \longrightarrow \text{the-Bool} (\text{the} (\text{locals } s \ i))) \implies$
 $G, (\{::'a \ \text{triple set}\}) \vdash \{\text{Normal} (\lambda Y s Z::'a. \ \text{True})\}$
.lab1 • *While*(*Lit* (*Bool True*)) (*If*(*Acc* (*LVar i*)) (*Throw* (*Acc* (*LVar xcpt*))) *Else*
 (*Expr* (*Ass* (*LVar i*) (*Acc* (*LVar j*))))). {*Q*}
 ⟨*proof*⟩

end