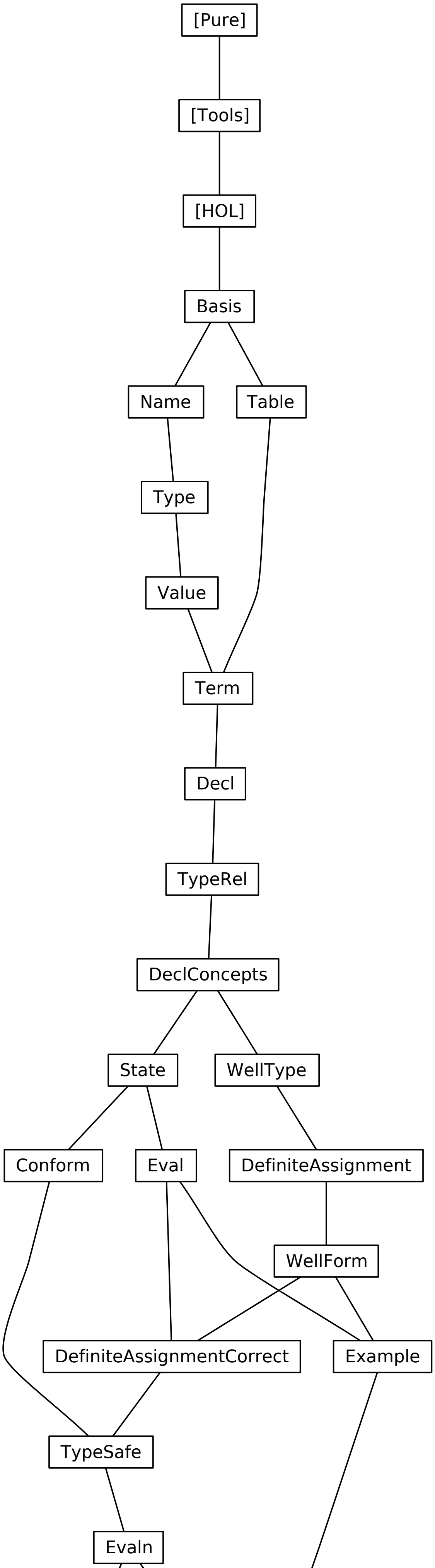
Java Source and Bytecode Formalizations in Isabelle: Bali

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

March 13, 2025

Contents


[Pure]
[Tools]
[HOL]
Basis
Name
Table
Type
Value
Term
Decl
TypeRel
DeclConcepts
State
WellType
Conform
Eval
DefiniteAssignment
WellForm
DefiniteAssignmentCorrect
Example
TypeSafe
Evaln

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*: $\{p.\ P\ (case\text{-}prod\ f\ p)\} = \{(a,b).\ P\ (f\ a\ b)\}$
⟨*proof*⟩

lemma *subset-insertD*: $A \subseteq insert\ x\ B \Longrightarrow A \subseteq B \wedge x \notin A \vee (\exists B'.\ A = insert\ x\ B' \wedge B' \subseteq B)$
⟨*proof*⟩

abbreviation *nat3* :: *nat* (‹*3*›) **where** $3 \equiv Suc\ 2$
abbreviation *nat4* :: *nat* (‹*4*›) **where** $4 \equiv Suc\ 3$

lemma *irrefl-tranclI′*: $r^{-1} \cap r^{+} = \{\} \Longrightarrow \forall x.\ (x,\ x) \notin r^{+}$
⟨*proof*⟩

lemma *trancl-rtrancl-trancl*: $[\![(x,\ y) \in r^{+};\ (y,\ z) \in r^{*}]\!] \Longrightarrow (x,\ z) \in r^{+}$
⟨*proof*⟩

lemma *rtrancl-into-trancl3*: $[\![(a,\ b) \in r^{*};\ a \neq b]\!] \Longrightarrow (a,\ b) \in r^{+}$
⟨*proof*⟩

lemma *rtrancl-into-rtrancl2*: $[\![(a,\ b) \in\ r;\ (b,\ c) \in r^{*}]\!] \Longrightarrow (a,\ c) \in r^{*}$
⟨*proof*⟩

lemma *triangle-lemma*:
assumes *unique*: $\bigwedge a\ b\ c.\ [\![(a,b)\in r;\ (a,c)\in r]\!] \Longrightarrow b = c$
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 (*Refl*) $a = b$
    | (*Trancl*) $(a,b) \in r^+$
  ⟨*proof*⟩

lemma *Ball-weaken*: $[\![Ball\ s\ P;\ \bigwedge x.\ P\ x \longrightarrow Q\ x]\!] \Longrightarrow Ball\ s\ Q$
  ⟨*proof*⟩

lemma *finite-SetCompr2*:
  finite $\{f\ y\ x\ |x\ y.\ P\ y\}$ **if** *finite* (*Collect P*)
    $\forall y.\ P\ y \longrightarrow finite\ (range\ (f\ y))$
⟨*proof*⟩

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

pairs

lemma *surjective-pairing5*:
  $p = (fst\ p,\ fst\ (snd\ p),\ fst\ (snd\ (snd\ p)),\ fst\ (snd\ (snd\ (snd\ p))),$
    $snd\ (snd\ (snd\ (snd\ p))))$
  ⟨*proof*⟩

lemma *fst-splitE* [*elim*!]:
  assumes $fst\ s' = x'$
  obtains $x\ s$ **where** $s' = (x,s)$ **and** $x = x'$
  ⟨*proof*⟩

lemma *fst-in-set-lemma*: $(x,\ y) \in set\ l \Longrightarrow x \in fst\ `\ set\ l$
  ⟨*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*) *sum3* = *In1* '*a* | *In2* '*b* | *In3* '*c*

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

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

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

abbreviation *In1l* :: '*al* ⇒ ('*al* + '*ar*, '*b*, '*c*) *sum3*
where *In1l e* ≡ *In1* (*Inl e*)

abbreviation *In1r* :: '*ar* ⇒ ('*al* + '*ar*, '*b*, '*c*) *sum3*
where *In1r c* ≡ *In1* (*Inr c*)

abbreviation *the-In1l* :: ('*al* + '*ar*, '*b*, '*c*) *sum3* ⇒ '*al*
where *the-In1l* ≡ *the-Inl* ∘ *the-In1*

abbreviation *the-In1r* :: ('*al* + '*ar*, '*b*, '*c*) *sum3* ⇒ '*ar*
where *the-In1r* ≡ *the-Inr* ∘ *the-In1*

⟨*ML*⟩

quantifiers for option type

syntax
-Oall :: [*pttrn*, '*a option*, *bool*] ⇒ *bool* (‹(*3*! -:-:/ -)› [*0*,*0*,*10*] *10*)
-Oex :: [*pttrn*, '*a option*, *bool*] ⇒ *bool* (‹(*3*? -:-:/ -)› [*0*,*0*,*10*] *10*)

syntax (*symbols*)
-Oall :: [*pttrn*, '*a option*, *bool*] ⇒ *bool* (‹(*3*∀ -∈-:/ -)› [*0*,*0*,*10*] *10*)
-Oex :: [*pttrn*, '*a option*, *bool*] ⇒ *bool* (‹(*3*∃ -∈-:/ -)› [*0*,*0*,*10*] *10*)

syntax-consts
-Oall ⇌ *Ball* **and**
-Oex ⇌ *Bex*

translations
∀ *x*∈*A*: *P* ⇌ ∀ *x*∈*CONST set-option A*. *P*
∃ *x*∈*A*: *P* ⇌ ∃ *x*∈*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* :: $('a \times 'b)$ *list* $\Rightarrow$ *bool*
where *unique* = *distinct* $\circ$ *map fst*

lemma *uniqueD*: *unique l* $\Longrightarrow (x, y) \in$ *set l* $\Longrightarrow (x', y') \in$ *set l* $\Longrightarrow x = x' \Longrightarrow y = y'$
⟨*proof*⟩

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

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

lemma *unique-ConsD*: *unique* $(x\#xs) \Longrightarrow$ *unique xs*
⟨*proof*⟩

lemma *unique-single* [*simp*]: $\bigwedge p.$ *unique* $[p]$
⟨*proof*⟩

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

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

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

list patterns

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

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

syntax
-lpttrn :: [*pttrn*, *pttrn*] $\Rightarrow$ *pttrn* (‹-#/-› [*901*,*900*] *900*)
syntax-consts
-lpttrn $\rightleftharpoons$ *lsplit*
translations
$\lambda y \# x \# xs.\ b \rightleftharpoons$ *CONST lsplit* $(\lambda y\ x \# xs.\ b)$
$\lambda x \# xs.\ b \rightleftharpoons$ *CONST lsplit* $(\lambda x\ xs.\ b)$

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

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

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
$= 'a \rightharpoonup 'b$
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* <= $(type)$ $'a \rightharpoonup 'b$

lemma *map-add-find-left*[*simp*]: $n\ k = None \Longrightarrow (m ++ n)\ k = m\ k$
⟨*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 =
 (λ*k*.
  (*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 = (λ*k*. (*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* (λ*x y*. *True*) *t* = *t*
⟨*proof*⟩

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

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

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

lemma *finite-range-filter-tab*:
finite (*range t*) $\Longrightarrow$ *finite* (*range* (*filter-tab c t*))
⟨*proof*⟩

lemma *filter-tab-SomeD*[*dest*!]:
filter-tab c t k = *Some x* $\Longrightarrow$ (*t k* = *Some x*) ∧ *c k x*
⟨*proof*⟩

lemma *filter-tab-SomeI*: ⟦*t k* = *Some x*;*C k x*⟧ $\Longrightarrow$*filter-tab C t k* = *Some x*
⟨*proof*⟩

lemma *filter-tab-all-True*:
∀ *k y*. *t k* = *Some y* ⟶ *p k y* $\Longrightarrow$*filter-tab p t* = *t*
⟨*proof*⟩

lemma *filter-tab-all-True-Some*:
⟦∀ *k y*. *t k* = *Some y* ⟶ *p k y*; *t k* = *Some v*⟧ $\Longrightarrow$ *filter-tab p t k* = *Some v*
⟨*proof*⟩

lemma *filter-tab-all-False*:
∀ *k y*. *t k* = *Some y* ⟶ ¬ *p k y* $\Longrightarrow$*filter-tab p t* = *Map.empty*
⟨*proof*⟩

lemma *filter-tab-None*: *t k* = *None* $\Longrightarrow$ *filter-tab p t k* = *None*
⟨*proof*⟩

lemma *filter-tab-dom-subset*: *dom* (*filter-tab C t*) ⊆ *dom t*
⟨*proof*⟩

lemma *filter-tab-eq*: ⟦*a*=*b*⟧ $\Longrightarrow$ *filter-tab C a* = *filter-tab C b*
⟨*proof*⟩

lemma *finite-dom-filter-tab*:
finite (*dom t*) $\Longrightarrow$ *finite* (*dom* (*filter-tab C t*))
⟨*proof*⟩

lemma *filter-tab-weaken*:
⟦∀ *a* ∈ *t k*: ∃ *b* ∈ *s k*: *P a b*;
⋀ *k x y*. ⟦*t k* = *Some x*;*s k* = *Some y*⟧ $\Longrightarrow$ *cond k x* ⟶ *cond k y*
⟧ $\Longrightarrow$ ∀ *a* ∈ *filter-tab cond t k*: ∃ *b* ∈ *filter-tab cond s k*: *P a b*
⟨*proof*⟩

lemma *cond-override-filter*:
⟦⋀ *k old new*. ⟦*s k* = *Some new*; *t k* = *Some old*⟧
$\Longrightarrow$ (¬ *overC new old* ⟶ ¬ *filterC k new*) ∧
(*overC new old* ⟶ *filterC k old* ⟶ *filterC k new*)
⟧ $\Longrightarrow$
cond-override overC (*filter-tab filterC t*) (*filter-tab filterC s*)
= *filter-tab filterC* (*cond-override overC t s*)
⟨*proof*⟩

Misc

lemma *Ball-set-table*: (∀ (*x*,*y*)∈ *set l*. *P x y*) $\Longrightarrow$ ∀ *x*. ∀ *y*∈ *map-of l x*: *P x y*
⟨*proof*⟩

lemma *Ball-set-tableD*:

$[\![(\forall\ (x,y) \in set\ l.\ P\ x\ y);\ x \in set\text{-}option\ (table\text{-}of\ l\ xa)]\!] \Longrightarrow P\ xa\ x$
⟨*proof*⟩

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

lemma *table-of-Some-in-set*:
$table\text{-}of\ l\ k = Some\ x \Longrightarrow (k,x) \in set\ l$
⟨*proof*⟩

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

lemma *inj-Pair-const2*: $inj\ (\lambda k.\ (k,\ C))$
⟨*proof*⟩

lemma *table-of-mapconst-SomeI*:
$[\![table\text{-}of\ t\ k = Some\ y';\ snd\ y=y';\ fst\ y=c]\!] \Longrightarrow$
$table\text{-}of\ (map\ (\lambda(k,x).\ (k,c,x))\ t)\ k = Some\ y$
⟨*proof*⟩

lemma *table-of-mapconst-NoneI*:
$[\![table\text{-}of\ t\ k = None]\!] \Longrightarrow$
$table\text{-}of\ (map\ (\lambda(k,x).\ (k,c,x))\ t)\ k = None$
⟨*proof*⟩

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

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

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

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

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

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

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

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

lemma *table-append-Some-iff*: *table-of* (*xs@ys*) *k* = *Some z* =
(*table-of xs k* = *Some z* ∨ (*table-of xs k* = *None* ∧ *table-of ys k* = *Some z*))
⟨*proof*⟩

lemma *table-of-filter-unique-SomeD* [*rule-format* (*no-asm*)]:
table-of (*filter P xs*) *k* = *Some z* ⟹ *unique xs* ⟶ *table-of xs k* = *Some z*
⟨*proof*⟩

definition *Un-tables* :: (′*a*, ′*b*) *tables set* ⇒ (′*a*, ′*b*) *tables*
where *Un-tables ts* = (λ*k*. ⋃*t*∈*ts*. *t k*)

definition *overrides-t* :: (′*a*, ′*b*) *tables* ⇒ (′*a*, ′*b*) *tables* ⇒ (′*a*, ′*b*) *tables*
(**infixl** ‹⊕⊕› *100*)
where *s* ⊕⊕ *t* = (λ*k*. *if t k* = {} *then s k else t k*)

definition
hidings-entails :: (′*a*, ′*b*) *tables* ⇒ (′*a*, ′*c*) *tables* ⇒ (′*b* ⇒ ′*c* ⇒ *bool*) ⇒ *bool*
(‹- *hidings* - *entails* -› *20*)
where (*t hidings s entails R*) = (∀ *k*. ∀ *x*∈*t k*. ∀ *y*∈*s k*. *R x y*)

definition
— variant for unique table:
hiding-entails :: (′*a*, ′*b*) *table* ⇒ (′*a*, ′*c*) *table* ⇒ (′*b* ⇒ ′*c* ⇒ *bool*) ⇒ *bool*
(‹- *hiding* - *entails* -› *20*)
where (*t hiding s entails R*) = (∀ *k*. ∀ *x*∈*t k*: ∀ *y*∈*s k*: *R x y*)

definition
— variant for a unique table and conditional overriding:
cond-hiding-entails :: (′*a*, ′*b*) *table* ⇒ (′*a*, ′*c*) *table*
⇒ (′*b* ⇒ ′*c* ⇒ *bool*) ⇒ (′*b* ⇒ ′*c* ⇒ *bool*) ⇒ *bool*
(‹- *hiding* - *under* - *entails* -› *20*)
where (*t hiding s under C entails R*) = (∀ *k*. ∀ *x*∈*t k*: ∀ *y*∈*s k*: *C x y* ⟶ *R x y*)

Untables

lemma *Un-tablesI* [*intro*]: *t* ∈ *ts* ⟹ *x* ∈ *t k* ⟹ *x* ∈ *Un-tables ts k*
⟨*proof*⟩

lemma *Un-tablesD* [*dest*!]: *x* ∈ *Un-tables ts k* ⟹ ∃ *t*. *t* ∈ *ts* ∧ *x* ∈ *t k*
⟨*proof*⟩

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

overrides

lemma *empty-overrides-t* [*simp*]: (λ*k*. {}) ⊕⊕ *m* = *m*
⟨*proof*⟩

lemma *overrides-empty-t* [*simp*]: *m* ⊕⊕ (λ*k*. {}) = *m*
⟨*proof*⟩

lemma *overrides-t-Some-iff*:
(*x* ∈ (*s* ⊕⊕ *t*) *k*) = (*x* ∈ *t k* ∨ *t k* = {} ∧ *x* ∈ *s k*)
⟨*proof*⟩

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

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

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

hiding entails

lemma *hiding-entailsD*:
t *hiding* s *entails* $R \Longrightarrow t\ k = Some\ x \Longrightarrow s\ k = Some\ y \Longrightarrow R\ x\ y$
⟨*proof*⟩

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*:
$[\![t$ *hiding* s *under* C *entails* $R;\ t\ k = Some\ x;\ s\ k = Some\ y;\ C\ x\ y]\!] \Longrightarrow R\ x\ y$
⟨*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*: $[\![t$ *hidings* s *entails* $R;\ x \in t\ k;\ y \in s\ k]\!] \Longrightarrow R\ x\ y$
⟨*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* :: $('a \rightharpoonup 'b) => nat => bool$
where
atleast-free $m\ 0 = True$
| *atleast-free-Suc*: *atleast-free* $m\ (Suc\ n) = (\exists\, a.\ m\ a = None \wedge (\forall\, b.$ *atleast-free* $(m(a \mapsto b))\ n))$

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

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

declare *fun-upd-apply* [*simp del*]
lemma *atleast-free-SucD-lemma* [*rule-format* (*no-asm*)]:
$\forall\, m\ a.\ m\ a = None \longrightarrow (\forall\, c.$ *atleast-free* $(m(a \mapsto c))\ n) \longrightarrow$
$(\forall\, b\ d.\ a \neq b \longrightarrow$ *atleast-free* $(m(b \mapsto d))\ n)$
⟨*proof*⟩

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

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

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

end

Chapter 4

Name

1 Java names

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

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* ⇒ *lname*
  where *VName n* == *EName* (*VNam n*)

abbreviation *Result* :: *lname*
  where *Result* == *EName Res*

datatype *xname* — names of standard exceptions
  = *Throwable*
  | *NullPointer* | *OutOfMemory* | *ClassCast*
  | *NegArrSize* | *IndOutBound* | *ArrStore*

lemma *xn-cases*:
 xn = *Throwable* ∨ *xn* = *NullPointer* ∨
  xn = *OutOfMemory* ∨ *xn* = *ClassCast* ∨
  xn = *NegArrSize* ∨ *xn* = *IndOutBound* ∨ *xn* = *ArrStore*
⟨*proof*⟩

datatype *tname* — type names for standard classes and other type names
  = *Object'*
  | *SXcpt' xname*
  | *TName tnam*

record *qtname* = — qualified tname cf. 6.5.3, 6.5.4
  pid :: *pname*
  tid :: *tname*

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
  qtname-qtname-def: *qtname* (*q*::*'a qtname-scheme*) = *q*

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

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

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

definition *SXcpt* :: *xname* ⇒ *qtname*
  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* ⇒ *bool*
  where *the-Bool* (*Bool b*) = *b*

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

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

type-synonym *dyn-ty* = *loc* ⇒ *ty option*

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

primrec *default-val* :: *ty* ⇒ *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 qtname* — interface type
  | *ClassT qtname* — class type
  | *ArrayT ty* — array type

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

abbreviation *NT* == *RefT NullT*
abbreviation *Iface I* == *RefT* (*IfaceT I*)
abbreviation *Class C* == *RefT* (*ClassT C*)
abbreviation *Array* :: *ty* ⇒ *ty* (‹-.[]› [*90*] *90*)
  where *T*.[] == *RefT* (*ArrayT T*)

definition
  the-Class :: *ty* ⇒ *qtname*
  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 efford

simplifications:

- expression statement allowed for any expression
- This is modeled as a special non-assignable local variable
- Super is modeled as a general expression with the same value as This
- access to field x in current class via This.x
- NewA creates only one-dimensional arrays; initialization of further subarrays may be simulated with nested NewAs
- The 'Lit' constructor is allowed to contain a reference value. But this is assumed to be prohibited in the input language, which is enforced by the type-checking rules.
- a call of a static method via a type name may be simulated by a dummy variable
- no nested blocks with inner local variables
- no synchronized statements
- no secondary forms of if, while (e.g. no for) (may be easily simulated)
- no switch (may be simulated with if)

- the *try-catch-finally* statement is divided into the *try-catch* statement and a finally statement, which may be considered as try..finally with empty catch
- the *try-catch* statement has exactly one catch clause; multiple ones can be simulated with instanceof
- the compiler is supposed to add the annotations - during type-checking. This transformation is left out as its result is checked by the type rules anyway

type-synonym *locals* = (*lname*, *val*) *table* — local variables

datatype *jump*
= *Break label* — break
| *Cont label* — continue
| *Ret* — return from method

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

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

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

type-synonym
abopt = *abrupt option*

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

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

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

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

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

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

— function codes for binary operations

datatype *binop* = *Mul* — `*` multiplication
| *Div* — `/` division
| *Mod* — `%` remainder
| *Plus* — `+` addition
| *Minus* — `-` subtraction
| *LShift* — `<<` left shift
| *RShift* — `>>` signed right shift
| *RShiftU* — `>>>` unsigned right shift
| *Less* — `<` less than
| *Le* — `<=` less than or equal
| *Greater* — `>` greater than
| *Ge* — `>=` greater than or equal
| *Eq* — `==` equal
| *Neq* — `!=` not equal
| *BitAnd* — `&` bitwise AND
| *And* — `&` boolean AND
| *BitXor* — `^` bitwise Xor
| *Xor* — `^` boolean Xor
| *BitOr* — `|` bitwise Or
| *Or* — `|` boolean Or
| *CondAnd* — `&&` conditional And
| *CondOr* — `||` conditional Or

The boolean operators `&` and `|` strictly evaluate both of their arguments. The conditional operators `&&` and `||` only evaluate the second argument if the value of the whole expression isn't allready 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 qtname qtname 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 qtname* — class instance creation
| *NewA ty expr* (‹*New* -[-]›[*99*,*10*]*85*)
— array creation
| *Cast ty expr* — type cast
| *Inst expr ref-ty* (‹- *InstOf* -›[*85*,*99*] *85*)
— instanceof
| *Lit val* — literal value, references not allowed
| *UnOp unop expr* — unary operation
| *BinOp binop expr expr* — binary operation

| *Super* — special Super keyword
| *Acc var* — variable access

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

and *stmt*
= *Skip* — empty statement
| *Expr expr* — expression statement
| *Lab jump stmt* (‹-· -› [*99*,*66*]*66*)
— labeled statement; handles break
| *Comp stmt stmt* (‹-;; -› [*66*,*65*]*65*)
| *If′ expr stmt stmt* (‹*If*′(-′) - *Else* -› [*80*,*79*,*79*]*70*)
| *Loop label expr stmt* (‹-· *While*′(-′) -› [*99*,*80*,*79*]*70*)
| *Jmp jump* — break, continue, return
| *Throw expr*
| *TryC stmt qtname vname stmt* (‹*Try* - *Catch*′(- -′) -› [*79*,*99*,*80*,*79*]*70*)
— *Try c1 Catch*(*C vn*) *c2*
— *c1*: block were exception may be thrown
— *C*: execption class to catch
— *vn*: local name for exception used in *c2*
— *c2*: block to execute when exception is catched
| *Fin stmt stmt* (‹- *Finally* -› [*79*,*79*]*70*)
| *FinA abopt stmt* — Save abruption of first statement
— technical term for smallstep sem.)
| *Init qtname* — 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* (*VNam v*)))

abbreviation
LAss :: *vname* ⇒ *expr* ⇒*stmt* (‹-:==-› [*90*,*85*] *85*)
where *v*:==*e* == *Expr* (*Ass* (*LVar* (*EName* (*VNam 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-rews* [*simp*]

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

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

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

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

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

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

class *inj-term* =
fixes *inj-term*:: ′*a* ⇒ *term* (‹⟨-⟩› *1000*)

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

instantiation *stmt* :: *inj-term*

begin

definition
stmt-inj-term-def: ⟨*c*::*stmt*⟩ = *In1r c*

instance ⟨*proof*⟩

end

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

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

instantiation *expr* :: *inj-term*
begin

definition
expr-inj-term-def: ⟨*e*::*expr*⟩ = *In1l e*

instance ⟨*proof*⟩

end

lemma *expr-inj-term-simp*: ⟨*e*::*expr*⟩ = *In1l e*
⟨*proof*⟩

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

instantiation *var* :: *inj-term*
begin

definition
var-inj-term-def: ⟨*v*::*var*⟩ = *In2 v*

instance ⟨*proof*⟩

end

lemma *var-inj-term-simp*: ⟨*v*::*var*⟩ = *In2 v*
⟨*proof*⟩

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

class *expr-of* =
fixes *expr-of* :: *'a* ⇒ *expr*

instantiation *expr* :: *expr-of*
begin

definition
expr-of = (λ(*e*::*expr*). *e*)

instance ⟨*proof*⟩

end

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

definition
⟨*es*::′*a list*⟩ = *In3* (*map expr-of es*)

instance ⟨*proof*⟩

end

lemma *expr-list-inj-term-def*:
⟨*es*::*expr list*⟩ ≡ *In3 es*
⟨*proof*⟩

lemma *expr-list-inj-term-simp*: ⟨*es*::*expr list*⟩ = *In3 es*
⟨*proof*⟩

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

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

lemma *stmt-expr-inj-term* [*iff*]: ⟨*t*::*stmt*⟩ ≠ ⟨*w*::*expr*⟩
⟨*proof*⟩
lemma *expr-stmt-inj-term* [*iff*]: ⟨*t*::*expr*⟩ ≠ ⟨*w*::*stmt*⟩
⟨*proof*⟩
lemma *stmt-var-inj-term* [*iff*]: ⟨*t*::*stmt*⟩ ≠ ⟨*w*::*var*⟩
⟨*proof*⟩
lemma *var-stmt-inj-term* [*iff*]: ⟨*t*::*var*⟩ ≠ ⟨*w*::*stmt*⟩
⟨*proof*⟩
lemma *stmt-elist-inj-term* [*iff*]: ⟨*t*::*stmt*⟩ ≠ ⟨*w*::*expr list*⟩
⟨*proof*⟩
lemma *elist-stmt-inj-term* [*iff*]: ⟨*t*::*expr list*⟩ ≠ ⟨*w*::*stmt*⟩
⟨*proof*⟩
lemma *expr-var-inj-term* [*iff*]: ⟨*t*::*expr*⟩ ≠ ⟨*w*::*var*⟩
⟨*proof*⟩
lemma *var-expr-inj-term* [*iff*]: ⟨*t*::*var*⟩ ≠ ⟨*w*::*expr*⟩
⟨*proof*⟩
lemma *expr-elist-inj-term* [*iff*]: ⟨*t*::*expr*⟩ ≠ ⟨*w*::*expr list*⟩
⟨*proof*⟩
lemma *elist-expr-inj-term* [*iff*]: ⟨*t*::*expr list*⟩ ≠ ⟨*w*::*expr*⟩
⟨*proof*⟩
lemma *var-elist-inj-term* [*iff*]: ⟨*t*::*var*⟩ ≠ ⟨*w*::*expr list*⟩
⟨*proof*⟩
lemma *elist-var-inj-term* [*iff*]: ⟨*t*::*expr list*⟩ ≠ ⟨*w*::*var*⟩
⟨*proof*⟩

lemma *term-cases*:
$[\![\bigwedge v.\ P\ \langle v\rangle_v;\ \bigwedge e.\ P\ \langle e\rangle_e;\bigwedge c.\ P\ \langle c\rangle_s;\bigwedge l.\ P\ \langle l\rangle_l]\!]$
$\Longrightarrow P\ t$
⟨*proof*⟩

Evaluation of unary operations

primrec *eval-unop* :: *unop* ⇒ *val* ⇒ *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* (¬ *the-Bool v*)

Evaluation of binary operations

primrec *eval-binop* :: *binop* ⇒ *val* ⇒ *val* ⇒ *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*^(*nat* (*the-Intg v2*))))
| *eval-binop RShift v1 v2* = *Intg* ((*the-Intg v1*) *div* (*2*^(*nat* (*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*) ≤ (*the-Intg v2*))
| *eval-binop Greater v1 v2* = *Bool* ((*the-Intg v2*) < (*the-Intg v1*))
| *eval-binop Ge v1 v2* = *Bool* ((*the-Intg v2*) ≤ (*the-Intg v1*))

| *eval-binop Eq v1 v2* = *Bool* (*v1*=*v2*)
| *eval-binop Neq v1 v2* = *Bool* (*v1*≠*v2*)
| *eval-binop BitAnd v1 v2* = *Intg 42* — FIXME: Not yet implemented
| *eval-binop And v1 v2* = *Bool* ((*the-Bool v1*) ∧ (*the-Bool v2*))
| *eval-binop BitXor v1 v2* = *Intg 42* — FIXME: Not yet implemented
| *eval-binop Xor v1 v2* = *Bool* ((*the-Bool v1*) ≠ (*the-Bool v2*))
| *eval-binop BitOr v1 v2* = *Intg 42* — FIXME: Not yet implemented
| *eval-binop Or v1 v2* = *Bool* ((*the-Bool v1*) ∨ (*the-Bool v2*))
| *eval-binop CondAnd v1 v2* = *Bool* ((*the-Bool v1*) ∧ (*the-Bool v2*))
| *eval-binop CondOr v1 v2* = *Bool* ((*the-Bool v1*) ∨ (*the-Bool v2*))

definition
need-second-arg :: *binop* ⇒ *val* ⇒ *bool* **where**
need-second-arg binop v1 = (¬ ((*binop*=*CondAnd* ∧ ¬ *the-Bool v1*) ∨
(*binop*=*CondOr* ∧ *the-Bool v1*)))

CondAnd and *CondOr* only evalulate 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*) = (¬ *b*)
⟨*proof*⟩

lemma *need-second-arg-strict*[*simp*]:
⟦*binop*≠*CondAnd*; *binop*≠*CondOr*⟧ ⟹ *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
⟨*proof*⟩

end

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

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

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

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

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

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

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

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

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

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

lemma *acc-modi-Package-le-cases*:
assumes *Package* $\leq m$
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 uniformaly about field and method declarations we introduce the notion of a member declaration (e.g. useful to define accessiblity)

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

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

class *has-memberid* =
fixes *memberid* :: ′*a* ⇒ *memberid*

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

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

instance ⟨*proof*⟩

end

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

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

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

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

⟨*proof*⟩

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

definition
pair-memberid-def:
memberid p = *memberid* (*snd p*)

instance ⟨*proof*⟩

end

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

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

definition
is-field :: *qtname* × *memberdecl* ⇒ *bool*
where *is-field m* = (∃ *declC f*. *m*=(*declC*,*fdecl f*))

lemma *is-fieldD*: *is-field m* ⟹ ∃ *declC f*. *m*=(*declC*,*fdecl f*)
⟨*proof*⟩

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

definition
is-method :: *qtname* × *memberdecl* ⇒ *bool*
where *is-method membr* = (∃ *declC m*. *membr*=(*declC*,*mdecl m*))

lemma *is-methodD*: *is-method membr* ⟹ ∃ *declC m*. *membr*=(*declC*,*mdecl m*)
⟨*proof*⟩

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

7 Interface

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

record *iface* = *ibody* + — interface
isuperIfs:: *qtname list* — superinterface list
type-synonym
idecl — interface declaration, cf. 9.1
= *qtname* × *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::*qtname list*|)
(*type*) *iface* <= (*type*) (|*access*::*acc-modi*,*imethods*::(*sig* × *mhead*) *list*,
isuperIfs::*qtname list*,...::'*a*|)
(*type*) *idecl* <= (*type*) *qtname* × *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 :: *qtname* — superclass
superIfs:: *qtname list* — implemented interfaces
type-synonym
cdecl — class declaration, cf. 8.1
= *qtname* × *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::*qtname*,*superIfs*::*qtname list*|)
(*type*) *class* <= (*type*) (|*access*::*acc-modi*,*cfields*::*fdecl list*,
methods::*mdecl list*,*init*::*stmt*,
super::*qtname*,*superIfs*::*qtname list*,. . .::'*a*|)
(*type*) *cdecl* <= (*type*) *qtname* × *class*

definition
cbody :: *class* ⇒ *cbody*
where *cbody c* = (|*access*=*access c*, *cfields*=*cfields c*,*methods*=*methods c*,*init*=*init c*|)

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

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

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

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

standard classes

consts
Object-mdecls :: *mdecl list* — methods of Object
SXcpt-mdecls :: *mdecl list* — methods of SXcpts

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

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

lemma *ObjectC-neq-SXcptC* [*simp*]: *ObjectC* ≠ *SXcptC xn*
⟨*proof*⟩

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

definition
standard-classes :: *cdecl list* **where**
standard-classes = [*ObjectC*, *SXcptC Throwable*,
SXcptC NullPointer, *SXcptC OutOfMemory*, *SXcptC ClassCast*,
SXcptC NegArrSize , *SXcptC IndOutBound*, *SXcptC ArrStore*]

programs

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

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

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

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

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

abbreviation
is-class :: *prog* ⇒ *qtname* ⇒ *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* ⇒ (*qtname* × *qtname*) *set* — direct subinterface
where *subint1 G* = {(*I*,*J*). ∃ *i*∈*iface G I*: *J*∈*set* (*isuperIfs i*)}

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

abbreviation
subcls1-syntax :: *prog* => [*qtname*, *qtname*] => *bool* (‹-⊢-≺$_C$1-› [71,71,71] 70)
where *G*⊢*C* ≺$_C$1 *D* == (*C*,*D*) ∈ *subcls1 G*

abbreviation
subclseq-syntax :: *prog* => [*qtname*, *qtname*] => *bool* (‹-⊢-≼$_C$ -› [71,71,71] 70)
where *G*⊢*C* ≼$_C$ *D* == (*C*,*D*) ∈(*subcls1 G*)*

abbreviation
subcls-syntax :: *prog* => [*qtname*, *qtname*] => *bool* (‹-⊢-≺$_C$ -› [71,71,71] 70)
where *G*⊢*C* ≺$_C$ *D* == (*C*,*D*) ∈(*subcls1 G*)$^+$

notation (*ASCII*)
subcls1-syntax (‹-|--<:C1-› [71,71,71] 70) **and**
subclseq-syntax (‹-|--<=:C -›[71,71,71] 70) **and**
subcls-syntax (‹-|--<:C -›[71,71,71] 70)

lemma *subint1I*: ⟦*iface G I* = *Some i*; *J* ∈ *set* (*isuperIfs i*)⟧
⟹ (*I*,*J*) ∈ *subint1 G*
⟨*proof*⟩

lemma *subcls1I*:⟦*class G C* = *Some c*; *C* ≠ *Object*⟧ ⟹ (*C*,(*super c*)) ∈ *subcls1 G*
⟨*proof*⟩

lemma *subint1D*: (*I*,*J*)∈*subint1 G*⟹ ∃ *i*∈*iface G I*: *J*∈*set* (*isuperIfs i*)
⟨*proof*⟩

lemma *subcls1D*:
(*C*,*D*)∈*subcls1 G* ⟹ *C*≠*Object* ∧ (∃ *c*. *class G C* = *Some c* ∧ (*super c* = *D*))
⟨*proof*⟩

lemma *subint1-def2*:
subint1 G = (*SIGMA I*: {*I*. *is-iface G I*}. *set* (*isuperIfs* (*the* (*iface G I*))))
⟨*proof*⟩

lemma *subcls1-def2*:
subcls1 G =
(*SIGMA C*: {*C*. *is-class G C*}. {*D*. *C*≠*Object* ∧ *super* (*the*(*class G C*))=*D*})

⟨*proof*⟩

lemma *subcls-is-class*:
$[\![G\vdash C \prec_C D]\!] \Longrightarrow \exists\ c.\ class\ G\ C = Some\ c$
⟨*proof*⟩

lemma *no-subcls1-Object*: $G\vdash Object\prec_C 1\ D \Longrightarrow P$
⟨*proof*⟩

lemma *no-subcls-Object*: $G\vdash Object\prec_C D \Longrightarrow P$
⟨*proof*⟩

well-structured programs

definition
ws-idecl :: *prog* ⇒ *qtname* ⇒ *qtname list* ⇒ *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-cdecl :: *prog* ⇒ *qtname* ⇒ *qtname* ⇒ *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-prog :: *prog* ⇒ *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-progI*:
$[\![\forall (I,i)\in set\ (ifaces\ G).\ \forall J\in set\ (isuperIfs\ i).\ is\text{-}iface\ G\ J\ \wedge$
$(J,I) \notin (subint1\ G)^+;$
$\forall (C,c)\in set\ (classes\ G).\ C\neq Object \longrightarrow is\text{-}class\ G\ (super\ c) \wedge$
$((super\ c),C) \notin (subcls1\ G)^+$
$]\!] \Longrightarrow ws\text{-}prog\ G$
⟨*proof*⟩

lemma *ws-prog-ideclD*:
$[\![iface\ G\ I = Some\ i;\ J\in set\ (isuperIfs\ i);\ ws\text{-}prog\ G]\!] \Longrightarrow$
$is\text{-}iface\ G\ J \wedge (J,I)\notin(subint1\ G)^+$
⟨*proof*⟩

lemma *ws-prog-cdeclD*:
$[\![class\ G\ C = Some\ c;\ C\neq Object;\ ws\text{-}prog\ G]\!] \Longrightarrow$
$is\text{-}class\ G\ (super\ c) \wedge (super\ c,C)\notin(subcls1\ G)^+$
⟨*proof*⟩

well-foundedness

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

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

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

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

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

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

lemmas *subint1-irrefl-lemma2* = *subint1-irrefl-lemma1* [*THEN irrefl-tranclI′*]
lemmas *subcls1-irrefl-lemma2* = *subcls1-irrefl-lemma1* [*THEN irrefl-tranclI′*]

lemma *subint1-irrefl*: $[\![(x,\ y) \in subint1\ G;\ ws\text{-}prog\ G]\!] \Longrightarrow x \neq y$
⟨*proof*⟩

lemma *subcls1-irrefl*: $[\![(x,\ y) \in subcls1\ G;\ ws\text{-}prog\ G]\!] \Longrightarrow x \neq y$
⟨*proof*⟩

lemmas *subint1-acyclic* = *subint1-irrefl-lemma2* [*THEN acyclicI*]
lemmas *subcls1-acyclic* = *subcls1-irrefl-lemma2* [*THEN acyclicI*]

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

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

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

lemma *subcls1-induct* [*consumes 1*]:
  $[\![ws\text{-}prog\ G;\ \bigwedge x.\ \forall y.\ (x,\ y) \in subcls1\ G \longrightarrow P\ y \Longrightarrow P\ x]\!] \Longrightarrow P\ a$
⟨*proof*⟩

lemma *ws-subint1-induct*:
 $[\![is\text{-}iface\ G\ I;\ ws\text{-}prog\ G;\ \bigwedge I\ i.\ [\![iface\ G\ I = Some\ i\ \wedge$
   $(\forall J \in set\ (isuperIfs\ i).\ (I,J){\in}subint1\ G \wedge P\ J \wedge is\text{-}iface\ G\ J)]\!] \Longrightarrow P\ I$
  $]\!] \Longrightarrow P\ I$
⟨*proof*⟩

lemma *ws-subcls1-induct*: $[\![is\text{-}class\ G\ C;\ ws\text{-}prog\ G;$
  $\bigwedge C\ c.\ [\![class\ G\ C = Some\ c;$
 $(C \neq Object \longrightarrow (C,(super\ c)){\in}subcls1\ G\ \wedge$
                  $P\ (super\ c) \wedge is\text{-}class\ G\ (super\ c))]\!] \Longrightarrow P\ C$
 $]\!] \Longrightarrow P\ C$
⟨*proof*⟩

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

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

⟦*is-class G C*; *ws-prog G*;
  ⋀ *co*. *class G Object* = *Some co* ⟹ *P Object*;
  ⋀ *C c*. ⟦*class G C* = *Some c*; *C* ≠ *Object*; *P* (*super c*)⟧ ⟹ *P C*
⟧ ⟹ *P C*
⟨*proof*⟩

lemma *ws-class-induct''* [*consumes 2*, *case-names Object Subcls*]:
⟦*class G C* = *Some c*; *ws-prog G*;
  ⋀ *co*. *class G Object* = *Some co* ⟹ *P Object co*;
  ⋀ *C c sc*. ⟦*class G C* = *Some c*; *class G* (*super c*) = *Some sc*;
      C ≠ *Object*; *P* (*super c*) *sc*⟧ ⟹ *P C c*
⟧ ⟹ *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: ⋀*I i*. ⟦*iface G I* = *Some i*;
      ∀ *J* ∈ *set* (*isuperIfs i*).
        (*I*,*J*)∈*subint1 G* ∧ *P J* ∧ *is-iface G J*⟧ ⟹ *P I*
  shows *P I*
⟨*proof*⟩

general recursion operators for the interface and class hiearchies

function *iface-rec* :: *prog* ⇒ *qtname* ⇒ (*qtname* ⇒ *iface* ⇒ *'a set* ⇒ *'a*) ⇒ *'a*
where
[*simp del*]: *iface-rec G I f* =
  (*case iface G I of*
    None ⇒ *undefined*
   | *Some i* ⇒ *if ws-prog G*
      then f I i
        ((λ*J*. *iface-rec G J f*)‘*set* (*isuperIfs i*))
      else undefined)
⟨*proof*⟩
termination
⟨*proof*⟩

lemma *iface-rec*:
⟦*iface G I* = *Some i*; *ws-prog G*⟧ ⟹
 iface-rec G I f = *f I i* ((λ*J*. *iface-rec G J f*)‘*set* (*isuperIfs i*))
⟨*proof*⟩

function
  class-rec :: *prog* ⇒ *qtname* ⇒ *'a* ⇒ (*qtname* ⇒ *class* ⇒ *'a* ⇒ *'a*) ⇒ *'a*
where
[*simp del*]: *class-rec G C t f* =
  (*case class G C of*
    None ⇒ *undefined*
   | *Some c* ⇒ *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*: $[\![$*class G C* = *Some c*; *ws-prog G*$]\!] \Longrightarrow$
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* ⇒ *qtname* ⇒ (*sig*,*qtname* × *mhead*) *tables* **where**
— methods of an interface, with overriding and inheritance, cf. 9.2
imethds G I = *iface-rec G I*
(λ*I i ts*. (*Un-tables ts*) ⊕⊕
(*set-option* ∘ *table-of* (*map* (λ(*s*,*m*). (*s*,*I*,*m*)) (*imethods 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* ⇒ (*qtname* × *qtname*) *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* => [*qtname*, *qtname*] => *bool* (‹-⊢-≺I1-› [*71*,*71*,*71*] *70*)
where *G*⊢*I* ≺*I1 J* == (*I*,*J*) ∈ *subint1 G*

abbreviation
subint-syntax :: *prog* => [*qtname*, *qtname*] => *bool* (‹-⊢-≼I -› [*71*,*71*,*71*] *70*)
where *G*⊢*I* ≼*I J* == (*I*,*J*) ∈(*subint1 G*)* — cf. 9.1.3

abbreviation
implmt1-syntax :: *prog* => [*qtname*, *qtname*] => *bool* (‹-⊢-↝1-› [*71*,*71*,*71*] *70*)
where *G*⊢*C* ↝*1 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 (‹-|--~>1-› [*71*,*71*,*71*] *70*)

subclass and subinterface relations

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

lemma *subcls-direct1*:
$[\![class\ G\ C = Some\ c;\ C \neq Object; D=super\ c]\!] \Longrightarrow G \vdash C \preceq_C D$
⟨*proof*⟩

lemma *subcls1I1*:
$[\![class\ G\ C = Some\ c;\ C \neq Object; D=super\ c]\!] \Longrightarrow G \vdash C \prec_C 1\ D$
⟨*proof*⟩

lemma *subcls-direct2*:
$[\![class\ G\ C = Some\ c;\ C \neq Object; D=super\ c]\!] \Longrightarrow G \vdash C \prec_C D$
⟨*proof*⟩

lemma *subclseq-trans*: $[\![G \vdash A \preceq_C B;\ G \vdash B \preceq_C C]\!] \Longrightarrow G \vdash A \preceq_C C$
⟨*proof*⟩

lemma *subcls-trans*: $[\![G \vdash A \prec_C B;\ G \vdash B \prec_C C]\!] \Longrightarrow G \vdash A \prec_C C$
⟨*proof*⟩

lemma *SXcpt-subcls-Throwable-lemma*:
$[\![class\ G\ (SXcpt\ xn) = Some\ xc;$
$super\ xc = (if\ xn = Throwable\ then\ Object\ else\ \ SXcpt\ Throwable)]\!]$
$\Longrightarrow G \vdash SXcpt\ xn \preceq_C SXcpt\ Throwable$
⟨*proof*⟩

lemma *subcls-ObjectI*: $[\![is\text{-}class\ G\ C;\ ws\text{-}prog\ G]\!] \Longrightarrow G \vdash C \preceq_C Object$
⟨*proof*⟩

lemma *subclseq-ObjectD* [*dest*!]: $G \vdash Object \preceq_C C \Longrightarrow C = Object$
⟨*proof*⟩

lemma *subcls-ObjectD* [*dest*!]: $G \vdash Object \prec_C C \Longrightarrow False$
⟨*proof*⟩

lemma *subcls-ObjectI1* [*intro*!]:
$[\![C \neq Object; is\text{-}class\ G\ C; ws\text{-}prog\ G]\!] \Longrightarrow G \vdash C \prec_C Object$
⟨*proof*⟩

lemma *subcls-is-class*: $(C,D) \in (subcls1\ G)^+ \Longrightarrow is\text{-}class\ G\ C$
⟨*proof*⟩

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

lemma *single-inheritance*:
$[\![G \vdash A \prec_C 1\ B;\ G \vdash A \prec_C 1\ C]\!] \Longrightarrow B = C$
⟨*proof*⟩

lemma *subcls-compareable*:
$[\![G \vdash A \preceq_C X;\ G \vdash A \preceq_C Y$
$]\!] \Longrightarrow G \vdash X \preceq_C Y \vee G \vdash Y \preceq_C X$
⟨*proof*⟩

lemma *subcls1-irrefl*: $[\![G \vdash C \prec_C 1\ D;\ ws\text{-}prog\ G\ ]\!]$
$\Longrightarrow C \neq D$

⟨*proof*⟩

lemma *no-subcls-Object*: $G\vdash C \prec_C D \Longrightarrow C \neq Object$
⟨*proof*⟩

lemma *subcls-acyclic*: $[\![G\vdash C \prec_C D; \text{ws-prog } G]\!] \Longrightarrow \neg\ G\vdash D \prec_C C$
⟨*proof*⟩

lemma *subclseq-cases*:
assumes $G\vdash C \preceq_C D$
obtains $(Eq)\ C = D \mid (Subcls)\ G\vdash C \prec_C D$
⟨*proof*⟩

lemma *subclseq-acyclic*:
$[\![G\vdash C \preceq_C D; G\vdash D \preceq_C C; \text{ws-prog } G]\!] \Longrightarrow C=D$
⟨*proof*⟩

lemma *subcls-irrefl*: $[\![G\vdash C \prec_C D; \text{ws-prog } G]\!]$
$\Longrightarrow C \neq D$
⟨*proof*⟩

lemma *invert-subclseq*:
$[\![G\vdash C \preceq_C D; \text{ws-prog } G]\!]$
$\Longrightarrow \neg\ G\vdash D \prec_C C$
⟨*proof*⟩

lemma *invert-subcls*:
$[\![G\vdash C \prec_C D; \text{ws-prog } G]\!]$
$\Longrightarrow \neg\ G\vdash D \preceq_C C$
⟨*proof*⟩

lemma *subcls-superD*:
$[\![G\vdash C \prec_C D; \text{class } G\ C = \text{Some } c]\!] \Longrightarrow G\vdash(\text{super } c) \preceq_C D$
⟨*proof*⟩

lemma *subclseq-superD*:
$[\![G\vdash C \preceq_C D; C\neq D; \text{class } G\ C = \text{Some } c]\!] \Longrightarrow G\vdash(\text{super } c) \preceq_C D$
⟨*proof*⟩

implementation relation

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

inductive — implementation, cf. 8.1.4
implmt :: *prog* ⇒ *qtname* ⇒ *qtname* ⇒ *bool* (‹-⊢-↝-› [71,71,71] 70)
for *G* :: *prog*
where
direct: $G\vdash C\leadsto 1J \Longrightarrow G\vdash C\leadsto J$
| *subint*: $G\vdash C\leadsto 1I \Longrightarrow G\vdash I\preceq I\ J \Longrightarrow G\vdash C\leadsto J$
| *subcls1*: $G\vdash C\prec_C 1D \Longrightarrow G\vdash D\leadsto J \Longrightarrow G\vdash C\leadsto J$

lemma *implmtD*: $G\vdash C\leadsto J \Longrightarrow (\exists I.\ G\vdash C\leadsto 1I \wedge G\vdash I\preceq I\ J) \vee (\exists D.\ G\vdash C\prec_C 1D \wedge G\vdash D\leadsto J)$
⟨*proof*⟩

lemma *implmt-ObjectE* [*elim*!]: $G\vdash Object\leadsto I \Longrightarrow R$
⟨*proof*⟩

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

lemma *implmt-subint2*: $[\![G \vdash A \rightsquigarrow J;\ G \vdash J \preceq I\ K]\!] \Longrightarrow G \vdash A \rightsquigarrow K$
⟨*proof*⟩

lemma *implmt-is-class*: $G \vdash C \rightsquigarrow I \Longrightarrow$ *is-class G C*
⟨*proof*⟩

widening relation

inductive
— widening, viz. method invocation conversion, cf. 5.3 i.e. kind of syntactic subtyping
widen :: *prog* ⇒ *ty* ⇒ *ty* ⇒ *bool* (‹-⊢-≼-› [*71*,*71*,*71*] *70*)
for *G* :: *prog*
where
refl: $G \vdash T \preceq T$ — identity conversion, cf. 5.1.1
| *subint*: $G \vdash I \preceq I\ J \Longrightarrow G \vdash$ *Iface I* ≼ *Iface J* — wid.ref.conv.,cf. 5.1.4
| *int-obj*: $G \vdash$ *Iface I* ≼ *Class Object*
| *subcls*: $G \vdash C \preceq_C D \Longrightarrow G \vdash$ *Class C* ≼ *Class D*
| *implmt*: $G \vdash C \rightsquigarrow I \Longrightarrow G \vdash$ *Class C* ≼ *Iface I*
| *null*: $G \vdash NT \preceq$ *RefT R*
| *arr-obj*: $G \vdash T.[] \preceq$ *Class Object*
| *array*: $G \vdash$ *RefT S* ≼ *RefT T* $\Longrightarrow G \vdash$ *RefT S*.[] ≼ *RefT T*.[]

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

lemma *widen-PrimT*: $G \vdash$ *PrimT x* ≼ *T* ⟹ (∃ *y*. *T* = *PrimT y*)
⟨*proof*⟩

lemma *widen-PrimT2*: $G \vdash S \preceq$ *PrimT x* ⟹ ∃ *y*. *S* = *PrimT y*
⟨*proof*⟩

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

lemma *widen-PrimT-strong*: $G \vdash$ *PrimT x* ≼ *T* ⟹ *T* = *PrimT x*
⟨*proof*⟩

lemma *widen-PrimT2-strong*: $G \vdash S \preceq$ *PrimT x* ⟹ *S* = *PrimT x*
⟨*proof*⟩

Specialized versions for booleans also would work for real Java

lemma *widen-Boolean*: $G \vdash$ *PrimT Boolean* ≼ *T* ⟹ *T* = *PrimT Boolean*
⟨*proof*⟩

lemma *widen-Boolean2*: $G \vdash S \preceq$ *PrimT Boolean* ⟹ *S* = *PrimT Boolean*
⟨*proof*⟩

lemma *widen-RefT*: $G \vdash$ *RefT R* ≼ *T* ⟹ ∃ *t*. *T*=*RefT t*
⟨*proof*⟩

lemma *widen-RefT2*: $G \vdash S \preceq$ *RefT R* ⟹ ∃ *t*. *S*=*RefT t*
⟨*proof*⟩

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

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

lemma *widen-Iface-Iface*: $G \vdash Iface\ I \preceq Iface\ J \Longrightarrow G \vdash I \preceq I\ J$
⟨*proof*⟩

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

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

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

lemma *widen-Class-Class*: $G \vdash Class\ C \preceq Class\ cm \Longrightarrow G \vdash C \preceq_C cm$
⟨*proof*⟩

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

lemma *widen-Class-Iface*: $G \vdash Class\ C \preceq Iface\ I \Longrightarrow G \vdash C \leadsto I$
⟨*proof*⟩

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

lemma *widen-Array*: $G \vdash S.[] \preceq T \Longrightarrow T = Class\ Object \vee (\exists T'.\ T = T'.[] \wedge G \vdash S \preceq T')$
⟨*proof*⟩

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

lemma *widen-ArrayPrimT*: $G \vdash PrimT\ t.[] \preceq T \Longrightarrow T = Class\ Object \vee T = PrimT\ t.[]$
⟨*proof*⟩

lemma *widen-ArrayRefT*:
$G \vdash RefT\ t.[] \preceq T \Longrightarrow T = Class\ Object \vee (\exists s.\ T = RefT\ s.[] \wedge G \vdash RefT\ t \preceq RefT\ s)$
⟨*proof*⟩

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

lemma *widen-Array-Array*: $G \vdash T.[] \preceq T'.[] \Longrightarrow G \vdash T \preceq T'$
⟨*proof*⟩

lemma *widen-Array-Class*: $G \vdash S.[] \preceq Class\ C \Longrightarrow C = Object$
⟨*proof*⟩

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

lemma *widen-Object*:

assumes *isrtype G T* **and** *ws-prog G*
shows $G\vdash RefT\ T \preceq Class\ Object$
⟨*proof*⟩

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

lemma *ws-widen-trans*: $[\![G\vdash S\preceq U;\ G\vdash U\preceq T;\ ws\text{-}prog\ G]\!] \Longrightarrow G\vdash S\preceq T$
⟨*proof*⟩

lemma *widen-antisym-lemma* [*rule-format* (*no-asm*)]: $[\![G\vdash S\preceq T;$
$\forall I\ J.\ G\vdash I\preceq I\ J \land G\vdash J\preceq I\ I \longrightarrow I = J;$
$\forall C\ D.\ G\vdash C\preceq_C\ D \land G\vdash D\preceq_C\ C \longrightarrow C = D;$
$\forall I\ .\ G\vdash Object\leadsto I \longrightarrow False]\!] \Longrightarrow G\vdash T\preceq S \longrightarrow S = T$
⟨*proof*⟩

lemmas *subint-antisym* =
subint1-acyclic [*THEN acyclic-impl-antisym-rtrancl*]
lemmas *subcls-antisym* =
subcls1-acyclic [*THEN acyclic-impl-antisym-rtrancl*]

lemma *widen-antisym*: $[\![G\vdash S\preceq T;\ G\vdash T\preceq S;\ ws\text{-}prog\ G]\!] \Longrightarrow S=T$
⟨*proof*⟩

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

definition
widens :: *prog* ⇒ [*ty list*, *ty list*] ⇒ *bool* (‹-⊢-[⪯]-› [*71*,*71*,*71*] *70*)
where $G\vdash Ts[\preceq]Ts' = list\text{-}all2\ (\lambda T\ T'.\ G\vdash T\preceq T')\ Ts\ Ts'$

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

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

narrowing relation

inductive — narrowing reference conversion, cf. 5.1.5
narrow :: *prog* ⇒ *ty* ⇒ *ty* ⇒ *bool* (‹-⊢-≻-› [*71*,*71*,*71*] *70*)
for *G* :: *prog*
where
subcls: $G\vdash C\preceq_C\ D \Longrightarrow G\vdash\ Class\ D\succ Class\ C$
| *implmt*: $\neg G\vdash C\leadsto I \Longrightarrow G\vdash\ Class\ C\succ Iface\ I$
| *obj-arr*: $G\vdash Class\ Object\succ T.[\,]$
| *int-cls*: $G\vdash\ Iface\ I\succ Class\ C$
| *subint*: *imethds G I hidings imethds G J entails*
$(\lambda(md,\ mh\ )\ (md',mh').\ G\vdash mrt\ mh\preceq mrt\ mh') \Longrightarrow$
$\neg G\vdash I\preceq I\ J \Longrightarrow G\vdash\ Iface\ I\succ Iface\ J$
| *array*: $G\vdash RefT\ S\succ RefT\ T \Longrightarrow G\vdash\ RefT\ S.[\,]\succ RefT\ T.[\,]$

lemma *narrow-RefT*: $G\vdash RefT\ R\succ T \Longrightarrow \exists t.\ T=RefT\ t$
⟨*proof*⟩

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

lemma *narrow-PrimT*: *G*⊢*PrimT pt*≻*T* ⟹ ∃ *t*. *T*=*PrimT t*
⟨*proof*⟩

lemma *narrow-PrimT2*: *G*⊢*S*≻*PrimT pt* ⟹
∃ *t*. *S*=*PrimT t* ∧ *G*⊢*PrimT t*⪯*PrimT pt*
⟨*proof*⟩

These narrowing lemmata hold in Bali but are to 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*⊢*PrimT pt*≻*T* ⟹ *T*=*PrimT pt*
⟨*proof*⟩

lemma *narrow-PrimT2-strong*: *G*⊢*S*≻*PrimT pt* ⟹ *S*=*PrimT pt*
⟨*proof*⟩

Specialized versions for booleans also would work for real Java

lemma *narrow-Boolean*: *G*⊢*PrimT Boolean*≻*T* ⟹ *T*=*PrimT Boolean*
⟨*proof*⟩

lemma *narrow-Boolean2*: *G*⊢*S*≻*PrimT Boolean* ⟹ *S*=*PrimT Boolean*
⟨*proof*⟩

casting relation

inductive — casting conversion, cf. 5.5
cast :: *prog* ⇒ *ty* ⇒ *ty* ⇒ *bool* (‹-⊢-⪯? -› [*71*,*71*,*71*] *70*)
for *G* :: *prog*
where
widen: *G*⊢*S*⪯*T* ⟹ *G*⊢*S*⪯? *T*
| *narrow*: *G*⊢*S*≻*T* ⟹ *G*⊢*S*⪯? *T*

lemma *cast-RefT*: *G*⊢*RefT R*⪯? *T* ⟹ ∃ *t*. *T*=*RefT t*
⟨*proof*⟩

lemma *cast-RefT2*: *G*⊢*S*⪯? *RefT R* ⟹ ∃ *t*. *S*=*RefT t*
⟨*proof*⟩

lemma *cast-PrimT*: *G*⊢*PrimT pt*⪯? *T* ⟹ ∃ *t*. *T*=*PrimT t*
⟨*proof*⟩

lemma *cast-PrimT2*: *G*⊢*S*⪯? *PrimT pt* ⟹ ∃ *t*. *S*=*PrimT t* ∧ *G*⊢*PrimT t*⪯*PrimT pt*
⟨*proof*⟩

lemma *cast-Boolean*:
assumes *bool-cast*: *G*⊢*PrimT Boolean*⪯? *T*
shows *T*=*PrimT Boolean*
⟨*proof*⟩

lemma *cast-Boolean2*:
assumes *bool-cast*: *G*⊢*S*⪯? *PrimT Boolean*
shows *S* = *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* ⇒ *qtname* ⇒ *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* ⇒ *qtname* ⇒ *bool*
  where *is-acc-class G P C* = (*is-class G C* ∧ *G*⊢(*Class C*) *accessible-in P*)

definition
  is-acc-iface :: *prog* ⇒ *pname* ⇒ *qtname* ⇒ *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* ⇒ *pname* ⇒ *ref-ty* ⇒ *bool*
where *is-acc-reftype G P T* = (*isrtype G T* ∧ *G*⊢*T accessible-in′ P*)

lemma *is-acc-classD*:
is-acc-class G P C ⟹ *is-class G C* ∧ *G*⊢(*Class C*) *accessible-in P*
⟨*proof*⟩

lemma *is-acc-class-is-class*: *is-acc-class G P C* ⟹ *is-class G C*
⟨*proof*⟩

lemma *is-acc-ifaceD*:
is-acc-iface G P I ⟹ *is-iface G I* ∧ *G*⊢(*Iface I*) *accessible-in P*
⟨*proof*⟩

lemma *is-acc-typeD*:
is-acc-type G P T ⟹ *is-type G T* ∧ *G*⊢*T accessible-in P*
⟨*proof*⟩

lemma *is-acc-reftypeD*:
is-acc-reftype G P T ⟹ *isrtype G T* ∧ *G*⊢*T accessible-in′ P*
⟨*proof*⟩

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* ⇒ *acc-modi*

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

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

instance ⟨*proof*⟩

end

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

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

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

instance ⟨*proof*⟩

end

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

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

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

instance ⟨*proof*⟩

end

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

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

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

instance ⟨*proof*⟩

end

lemma *memberdecl-fdecl-acc-modi-simp*[*simp*]:
accmodi (*fdecl m*) = *accmodi m*
⟨*proof*⟩

lemma *memberdecl-mdecl-acc-modi-simp*[*simp*]:
accmodi (*mdecl m*) = *accmodi m*
⟨*proof*⟩

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

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

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

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

instance ⟨*proof*⟩

end

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

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

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

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

instance ⟨*proof*⟩

end

lemma *pair-declclass-simp*[*simp*]: *declclass* (*c*,*x*) = *declclass c*
⟨*proof*⟩

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

class *has-static* =
fixes *is-static* :: ′*a* ⇒ *bool*

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

instance ⟨*proof*⟩

end

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

instance ⟨*proof*⟩

end

axiomatization where
static-field-type-is-static-def: *is-static* (*m*::(′*a member-scheme*)) ≡ *static m*

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

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

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

instance ⟨*proof*⟩

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

definition
mbr :: *qtname* × *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* × *qtname*)× *field*)

— some mnemotic selectors for (*vname* × *qtname*)

definition
fname:: *vname* × *'a* ⇒ *vname*
where *fname* = *fst*
— also used for fdecl

definition
declclassf:: (*vname* × *qtname*) ⇒ *qtname*
where *declclassf* = *snd*

lemma *decliface-simp*[*simp*]: *decliface* (*I*,*m*) = *I*
⟨*proof*⟩

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

definition
fldname :: *vname* × *qtname* ⇒ *vname*
where *fldname* = *fst*

definition
fldclass :: *vname* × *qtname* ⇒ *qtname*
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 :: *qtname* × *mdecl* ⇒ *qtname* × *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* ⇒ (*qtname* × *methd*) ⇒ (*qtname* × *memberdecl*)
where *method sig m* = (*declclass m*, *mdecl* (*sig*, *mthd m*))

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

instance ⟨*proof*⟩

end

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

instance ⟨*proof*⟩

end

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

instance ⟨*proof*⟩

end

axiomatization where
mhead-ext-type-resTy-def: *resTy* (*m*::(′*b mhead-scheme*)) ≡ *resT m*

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

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

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

definition
pair-resTy-def: *resTy p* = *resTy* (*snd p*)

instance ⟨*proof*⟩

end

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

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

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

inheritable-in

G⊢*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* ⇒ (*qtname* × *memberdecl*) ⇒ *pname* ⇒ *bool* (‹- ⊢ - *inheritable*′*-in* -› [*61*,*61*,*61*] *60*)
where
G⊢*membr inheritable-in pack* =
(*case* (*accmodi membr*) *of*
Private ⇒ *False*
| *Package* ⇒ (*pid* (*declclass membr*)) = *pack*
| *Protected* ⇒ *True*
| *Public* ⇒ *True*)

abbreviation
Method-inheritable-in-syntax::
prog ⇒ (*qtname* × *mdecl*) ⇒ *pname* ⇒ *bool*
(‹- ⊢*Method* - *inheritable*′*-in* - › [*61*,*61*,*61*] *60*)
where *G*⊢*Method m inheritable-in p* == *G*⊢*methdMembr m inheritable-in p*

abbreviation

Methd-inheritable-in::
 prog ⇒ *sig* ⇒ (*qtname* × *methd*) ⇒ *pname* ⇒ *bool*
 (‹- ⊢*Methd* - - *inheritable′-in* - › [*61*,*61*,*61*,*61*] *60*)
 where *G*⊢*Methd s m inheritable-in p* == *G*⊢(*method s m*) *inheritable-in p*

declared-in/undeclared-in

definition
 cdeclaredmethd :: *prog* ⇒ *qtname* ⇒ (*sig*,*methd*) *table* **where**
 cdeclaredmethd G C =
 (*case class G C of*
 None ⇒ λ *sig*. *None*
 | *Some c* ⇒ *table-of* (*methods c*))

definition
 cdeclaredfield :: *prog* ⇒ *qtname* ⇒ (*vname*,*field*) *table* **where**
 cdeclaredfield G C =
 (*case class G C of*
 None ⇒ λ *sig*. *None*
 | *Some c* ⇒ *table-of* (*cfields c*))

definition
 declared-in :: *prog* ⇒ *memberdecl* ⇒ *qtname* ⇒ *bool* (‹-⊢ - *declared′-in* -› [*61*,*61*,*61*] *60*)
where
 G⊢*m declared-in C* = (*case m of*
 fdecl (*fn*,*f*) ⇒ *cdeclaredfield G C fn* = *Some f*
 | *mdecl* (*sig*,*m*) ⇒ *cdeclaredmethd G C sig* = *Some m*)

abbreviation
method-declared-in:: *prog* ⇒ (*qtname* × *mdecl*) ⇒ *qtname* ⇒ *bool*
 (‹-⊢*Method* - *declared′-in* -› [*61*,*61*,*61*] *60*)
 where *G*⊢*Method m declared-in C* == *G*⊢*mdecl* (*mthd m*) *declared-in C*

abbreviation
methd-declared-in:: *prog* ⇒ *sig* ⇒(*qtname* × *methd*) ⇒ *qtname* ⇒ *bool*
 (‹-⊢*Methd* - - *declared′-in* -› [*61*,*61*,*61*,*61*] *60*)
 where *G*⊢*Methd s m declared-in C* == *G*⊢*mdecl* (*s*,*mthd m*) *declared-in C*

lemma *declared-in-classD*:
 G⊢*m declared-in C* ⟹ *is-class G C*
⟨*proof*⟩

definition
 undeclared-in :: *prog* ⇒ *memberid* ⇒ *qtname* ⇒ *bool* (‹-⊢ - *undeclared′-in* -› [*61*,*61*,*61*] *60*)
where
 G⊢*m undeclared-in C* = (*case m of*
 fid fn ⇒ *cdeclaredfield G C fn* = *None*
 | *mid sig* ⇒ *cdeclaredmethd G C sig* = *None*)

members

inductive
 members :: *prog* ⇒ (*qtname* × *memberdecl*) ⇒ *qtname* ⇒ *bool*
 (‹- ⊢ - *member′-of* -› [*61*,*61*,*61*] *60*)
 for *G* :: *prog*
where

 Immediate: ⟦*G*⊢*mbr m declared-in C*;*declclass m* = *C*⟧ ⟹ *G*⊢*m member-of C*
| *Inherited*: ⟦*G*⊢*m inheritable-in* (*pid C*); *G*⊢*memberid m undeclared-in C*;

$G \vdash C \prec_C 1\ S$; $G \vdash (Class\ S)$ *accessible-in* $(pid\ C)$;$G \vdash m$ *member-of* S
$\rrbracket \Longrightarrow G \vdash m$ *member-of* C

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

abbreviation
method-member-of:: $prog \Rightarrow (qtname \times mdecl) \Rightarrow qtname \Rightarrow bool$
(‹- ⊢*Method* - *member'-of* -› [61,61,61] 60)
where $G \vdash$*Method m member-of C* == $G \vdash$(*methdMembr m*) *member-of C*

abbreviation
methd-member-of:: $prog \Rightarrow sig \Rightarrow (qtname \times methd) \Rightarrow qtname \Rightarrow bool$
(‹- ⊢*Methd* - - *member'-of* -› [61,61,61,61] 60)
where $G \vdash$*Methd s m member-of C* == $G \vdash$(*method s m*) *member-of C*

abbreviation
fieldm-member-of:: $prog \Rightarrow vname \Rightarrow (qtname \times field) \Rightarrow qtname \Rightarrow bool$
(‹- ⊢*Field* - - *member'-of* -› [61,61,61] 60)
where $G \vdash$*Field n f member-of C* == $G \vdash$*fieldm n f member-of C*

definition
inherits :: $prog \Rightarrow qtname \Rightarrow (qtname \times memberdecl) \Rightarrow bool$ (‹- ⊢ - *inherits* -› [61,61,61] 60)
where
$G \vdash C$ *inherits* m =
($G \vdash m$ *inheritable-in* $(pid\ C) \wedge G \vdash$*memberid m undeclared-in* $C \wedge$
$(\exists S.\ G \vdash C \prec_C 1\ S \wedge G \vdash (Class\ S)$ *accessible-in* $(pid\ C) \wedge G \vdash m$ *member-of* S))

lemma *inherits-member*: $G \vdash C$ *inherits* $m \Longrightarrow G \vdash m$ *member-of* C
⟨*proof*⟩

definition
member-in :: $prog \Rightarrow (qtname \times memberdecl) \Rightarrow qtname \Rightarrow bool$ (‹- ⊢ - *member'-in* -› [61,61,61] 60)
where $G \vdash m$ *member-in* C = $(\exists\ provC.\ G \vdash\ C \preceq_C provC \wedge G \vdash m$ *member-of* $provC)$

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

abbreviation
method-member-in:: $prog \Rightarrow (qtname \times mdecl) \Rightarrow qtname \Rightarrow bool$
(‹- ⊢*Method* - *member'-in* -› [61,61,61] 60)
where $G \vdash$*Method m member-in C* == $G \vdash$(*methdMembr m*) *member-in C*

abbreviation
methd-member-in:: $prog \Rightarrow sig \Rightarrow (qtname \times methd) \Rightarrow qtname \Rightarrow bool$
(‹- ⊢*Methd* - - *member'-in* -› [61,61,61,61] 60)
where $G \vdash$*Methd s m member-in C* == $G \vdash$(*method s m*) *member-in C*

lemma *member-inD*: $G \vdash m$ *member-in* C
$\Longrightarrow \exists\ provC.\ G \vdash\ C \preceq_C provC \wedge G \vdash m$ *member-of* $provC$
⟨*proof*⟩

lemma *member-inI*: $\llbracket G \vdash m$ *member-of* $provC$;$G \vdash\ C \preceq_C provC \rrbracket \Longrightarrow G \vdash m$ *member-in* C
⟨*proof*⟩

lemma *member-of-to-member-in*: $G \vdash m$ *member-of* $C \Longrightarrow G \vdash m$ *member-in* C
⟨*proof*⟩

overriding

Unfortunately the static notion of overriding (used during the typecheck of the compiler) and the dynamic notion of overriding (used during execution in the JVM) are not exactly the same.

Static overriding (used during the typecheck of the compiler)

inductive
  stat-overridesR :: *prog* ⇒ (*qtname* × *mdecl*) ⇒ (*qtname* × *mdecl*) ⇒ *bool*
   (‹- ⊢ - *overrides*$_S$ -› [61,61,61] 60)
  for *G* :: *prog*
where

  Direct: ⟦¬ *is-static new*; *msig new* = *msig old*;
    G⊢*Method new declared-in* (*declclass new*);
    G⊢*Method old declared-in* (*declclass old*);
    G⊢*Method old inheritable-in pid* (*declclass new*);
    G⊢(*declclass new*) $\prec_C$1 *superNew*;
    G ⊢*Method old member-of superNew*
    ⟧ ⟹ G⊢*new overrides*$_S$ *old*

| *Indirect*: ⟦G⊢*new overrides*$_S$ *intr*; G⊢*intr overrides*$_S$ *old*⟧
    ⟹ G⊢*new overrides*$_S$ *old*

Dynamic overriding (used during the typecheck of the compiler)

inductive
  overridesR :: *prog* ⇒ (*qtname* × *mdecl*) ⇒ (*qtname* × *mdecl*) ⇒ *bool*
   (‹- ⊢ - *overrides* -› [61,61,61] 60)
  for *G* :: *prog*
where

  Direct: ⟦¬ *is-static new*; ¬ *is-static old*; *accmodi new* ≠ *Private*;
    msig new = *msig old*;
    G⊢(*declclass new*) $\prec_C$ (*declclass old*);
    G⊢*Method new declared-in* (*declclass new*);
    G⊢*Method old declared-in* (*declclass old*);
    G⊢*Method old inheritable-in pid* (*declclass new*);
    G⊢*resTy new* ⪯ *resTy old*
    ⟧ ⟹ G⊢*new overrides old*

| *Indirect*: ⟦G⊢*new overrides intr*; G⊢*intr overrides old*⟧
    ⟹ G⊢*new overrides old*

abbreviation (*input*)
sig-stat-overrides::
 prog ⇒ *sig* ⇒ (*qtname* × *methd*) ⇒ (*qtname* × *methd*) ⇒ *bool*
    (‹-,-⊢ - *overrides*$_S$ -› [61,61,61,61] 60)
 where G,s⊢*new overrides*$_S$ *old* == G⊢(*qmdecl s new*) *overrides*$_S$ (*qmdecl s old*)

abbreviation (*input*)
sig-overrides:: *prog* ⇒ *sig* ⇒ (*qtname* × *methd*) ⇒ (*qtname* × *methd*) ⇒ *bool*
    (‹-,-⊢ - *overrides* -› [61,61,61,61] 60)
 where G,s⊢*new overrides old* == G⊢(*qmdecl s new*) *overrides* (*qmdecl s old*)

Hiding

definition
hides :: *prog* ⇒ (*qtname* × *mdecl*) ⇒ (*qtname* × *mdecl*) ⇒ *bool* (‹-⊢ - *hides* -› [*61*,*61*,*61*] *60*)
where
G⊢*new hides old* =
(*is-static new* ∧ *msig new* = *msig old* ∧
G⊢(*declclass new*) $\prec_C$ (*declclass old*) ∧
G⊢*Method new declared-in* (*declclass new*) ∧
G⊢*Method old declared-in* (*declclass old*) ∧
G⊢*Method old inheritable-in pid* (*declclass new*))

abbreviation
sig-hides:: *prog* ⇒ *sig* ⇒ (*qtname* × *methd*) ⇒ (*qtname* × *methd*) ⇒ *bool*
(‹-,-⊢ - *hides* -› [*61*,*61*,*61*,*61*] *60*)
where *G*,*s*⊢*new hides old* == *G*⊢(*qmdecl s new*) *hides* (*qmdecl s old*)

lemma *hidesI*:
⟦*is-static new*; *msig new* = *msig old*;
G⊢(*declclass new*) $\prec_C$ (*declclass old*);
G⊢*Method new declared-in* (*declclass new*);
G⊢*Method old declared-in* (*declclass old*);
G⊢*Method old inheritable-in pid* (*declclass new*)
⟧ ⟹ *G*⊢*new hides old*
⟨*proof*⟩

lemma *hidesD*:
⟦*G*⊢*new hides old*⟧ ⟹
declclass new ≠ *Object* ∧ *is-static new* ∧ *msig new* = *msig old* ∧
G⊢(*declclass new*) $\prec_C$ (*declclass old*) ∧
G⊢*Method new declared-in* (*declclass new*) ∧
G⊢*Method old declared-in* (*declclass old*)
⟨*proof*⟩

lemma *overrides-commonD*:
⟦*G*⊢*new overrides old*⟧ ⟹
declclass new ≠ *Object* ∧ ¬ *is-static new* ∧ ¬ *is-static old* ∧
accmodi new ≠ *Private* ∧
msig new = *msig old* ∧
G⊢(*declclass new*) $\prec_C$ (*declclass old*) ∧
G⊢*Method new declared-in* (*declclass new*) ∧
G⊢*Method old declared-in* (*declclass old*)
⟨*proof*⟩

lemma *ws-overrides-commonD*:
⟦*G*⊢*new overrides old*;*ws-prog G*⟧ ⟹
declclass new ≠ *Object* ∧ ¬ *is-static new* ∧ ¬ *is-static old* ∧
accmodi new ≠ *Private* ∧ *G*⊢*resTy new* ⪯ *resTy old* ∧
msig new = *msig old* ∧
G⊢(*declclass new*) $\prec_C$ (*declclass old*) ∧
G⊢*Method new declared-in* (*declclass new*) ∧
G⊢*Method old declared-in* (*declclass old*)
⟨*proof*⟩

lemma *overrides-eq-sigD*:
⟦*G*⊢*new overrides old*⟧ ⟹ *msig old*=*msig new*
⟨*proof*⟩

lemma *hides-eq-sigD*:

$[\![G\vdash new\ hides\ old]\!] \Longrightarrow msig\ old{=}msig\ new$
⟨*proof*⟩

permits access

definition
permits-acc :: *prog* ⇒ (*qtname* × *memberdecl*) ⇒ *qtname* ⇒ *qtname* ⇒ *bool* (‹- ⊢ - *in* - *permits'-acc'-from* -› [*61*,*61*,*61*,*61*] *60*)
where
G⊢*membr in cls permits-acc-from accclass* =
(*case* (*accmodi membr*) *of*
Private ⇒ (*declclass membr* = *accclass*)
| *Package* ⇒ (*pid* (*declclass membr*) = *pid accclass*)
| *Protected* ⇒ (*pid* (*declclass membr*) = *pid accclass*)
∨
(*G*⊢*accclass* $\prec_C$ *declclass membr*
∧ (*G*⊢*cls* $\preceq_C$ *accclass* ∨ *is-static membr*))
| *Public* ⇒ *True*)

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

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

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

inductive
accessible-fromR :: *prog* ⇒ *qtname* ⇒ (*qtname* × *memberdecl*) ⇒ *qtname* ⇒ *bool*
and *accessible-from* :: *prog* ⇒ (*qtname* × *memberdecl*) ⇒ *qtname* ⇒ *qtname* ⇒ *bool*
(‹- ⊢ - *of* - *accessible'-from* -› [*61*,*61*,*61*,*61*] *60*)
and *method-accessible-from* :: *prog* ⇒ (*qtname* × *mdecl*) ⇒ *qtname* ⇒ *qtname* ⇒ *bool*
(‹- ⊢*Method* - *of* - *accessible'-from* -› [*61*,*61*,*61*,*61*] *60*)
for *G* :: *prog* **and** *accclass* :: *qtname*
where
G⊢*membr of cls accessible-from accclass* ≡ *accessible-fromR G accclass membr cls*

| *G*⊢*Method m of cls accessible-from accclass* ≡ *accessible-fromR G accclass* (*methdMembr m*) *cls*

| *Immediate*: !!*membr class*.
⟦*G*⊢*membr member-of class*;
G⊢(*Class class*) *accessible-in* (*pid accclass*);
G⊢*membr in class permits-acc-from accclass*
⟧ ⟹ *G*⊢*membr of class accessible-from accclass*

| *Overriding*: !!*membr class C new old supr*.
⟦*G*⊢*membr member-of class*;
G⊢(*Class class*) *accessible-in* (*pid accclass*);
membr=(*C*,*mdecl new*);
G⊢(*C*,*new*) $overrides_S$ *old*;
G⊢*class* $\prec_C$ *supr*;
G⊢*Method old of supr accessible-from accclass*

]] ⟹ G⊢membr of class accessible-from accclass

abbreviation
methd-accessible-from::
prog ⇒ *sig* ⇒ (*qtname* × *methd*) ⇒ *qtname* ⇒ *qtname* ⇒ *bool*
(‹- ⊢*Methd* - - *of* - *accessible′-from* -› [*61*,*61*,*61*,*61*,*61*] *60*)
where
G⊢*Methd s m of cls accessible-from accclass* ==
G⊢(*method s m*) *of cls accessible-from accclass*

abbreviation
field-accessible-from::
prog ⇒ *vname* ⇒ (*qtname* × *field*) ⇒ *qtname* ⇒ *qtname* ⇒ *bool*
(‹- ⊢*Field* - - *of* - *accessible′-from* -› [*61*,*61*,*61*,*61*,*61*] *60*)
where
G⊢*Field fn f of C accessible-from accclass* ==
G⊢(*fieldm fn f*) *of C accessible-from accclass*

inductive
dyn-accessible-fromR :: *prog* ⇒ *qtname* ⇒ (*qtname* × *memberdecl*) ⇒ *qtname* ⇒ *bool*
and *dyn-accessible-from′* :: *prog* ⇒ (*qtname* × *memberdecl*) ⇒ *qtname* ⇒ *qtname* ⇒ *bool*
(‹- ⊢ - *in* - *dyn′-accessible′-from* -› [*61*,*61*,*61*,*61*] *60*)
and *method-dyn-accessible-from* :: *prog* ⇒ (*qtname* × *mdecl*) ⇒ *qtname* ⇒ *qtname* ⇒ *bool*
(‹- ⊢*Method* - *in* - *dyn′-accessible′-from* -› [*61*,*61*,*61*,*61*] *60*)
for *G* :: *prog* **and** *accclass* :: *qtname*
where
G⊢*membr in C dyn-accessible-from accC* ≡ *dyn-accessible-fromR G accC membr C*

| *G*⊢*Method m in C dyn-accessible-from accC* ≡ *dyn-accessible-fromR G accC* (*methdMembr m*) *C*

| *Immediate*: !!*class*. [[*G*⊢*membr member-in class*;
G⊢*membr in class permits-acc-from accclass*
]] ⟹ *G*⊢*membr in class dyn-accessible-from accclass*

| *Overriding*: !!*class*. [[*G*⊢*membr member-in class*;
membr=(*C*,*mdecl new*);
G⊢(*C*,*new*) *overrides old*;
G⊢*class* $\prec_C$ *supr*;
G⊢*Method old in supr dyn-accessible-from accclass*
]] ⟹ *G*⊢*membr in class dyn-accessible-from accclass*

abbreviation
methd-dyn-accessible-from::
prog ⇒ *sig* ⇒ (*qtname* × *methd*) ⇒ *qtname* ⇒ *qtname* ⇒ *bool*
(‹- ⊢*Methd* - - *in* - *dyn′-accessible′-from* -› [*61*,*61*,*61*,*61*,*61*] *60*)
where
G⊢*Methd s m in C dyn-accessible-from accC* ==
G⊢(*method s m*) *in C dyn-accessible-from accC*

abbreviation
field-dyn-accessible-from::
prog ⇒ *vname* ⇒ (*qtname* × *field*) ⇒ *qtname* ⇒ *qtname* ⇒ *bool*
(‹- ⊢*Field* - - *in* - *dyn′-accessible′-from* -› [*61*,*61*,*61*,*61*,*61*] *60*)
where
G⊢*Field fn f in dynC dyn-accessible-from accC* ==
G⊢(*fieldm fn f*) *in dynC dyn-accessible-from accC*

lemma *accessible-from-commonD*: *G*⊢*m of C accessible-from S*

$\Longrightarrow$ *G*⊢*m member-of C* ∧ *G*⊢(*Class C*) *accessible-in* (*pid S*)
⟨*proof*⟩

lemma *unique-declaration*:
⟦*G*⊢*m declared-in C*; *G*⊢*n declared-in C*; *memberid m* = *memberid n* ⟧
$\Longrightarrow$ *m* = *n*
⟨*proof*⟩

lemma *declared-not-undeclared*:
G⊢*m declared-in C* $\Longrightarrow$ ¬ *G*⊢ *memberid m undeclared-in C*
⟨*proof*⟩

lemma *undeclared-not-declared*:
G⊢ *memberid m undeclared-in C* $\Longrightarrow$ ¬ *G*⊢ *m declared-in C*
⟨*proof*⟩

lemma *not-undeclared-declared*:
¬ *G*⊢ *membr-id undeclared-in C* $\Longrightarrow$ (∃ *m*. *G*⊢*m declared-in C* ∧
membr-id = *memberid m*)
⟨*proof*⟩

lemma *unique-declared-in*:
⟦*G*⊢*m declared-in C*; *G*⊢*n declared-in C*; *memberid m* = *memberid n*⟧
$\Longrightarrow$ *m* = *n*
⟨*proof*⟩

lemma *unique-member-of*:
assumes *n*: *G*⊢*n member-of C* **and**
m: *G*⊢*m member-of C* **and**
eqid: *memberid n* = *memberid m*
shows *n*=*m*
⟨*proof*⟩

lemma *member-of-is-classD*: *G*⊢*m member-of C* $\Longrightarrow$ *is-class G C*
⟨*proof*⟩

lemma *member-of-declC*:
G⊢*m member-of C*
$\Longrightarrow$ *G*⊢*mbr m declared-in* (*declclass m*)
⟨*proof*⟩

lemma *member-of-member-of-declC*:
G⊢*m member-of C*
$\Longrightarrow$ *G*⊢*m member-of* (*declclass m*)
⟨*proof*⟩

lemma *member-of-class-relation*:
G⊢*m member-of C* $\Longrightarrow$ *G*⊢*C* $\preceq_C$ *declclass m*
⟨*proof*⟩

lemma *member-in-class-relation*:
G⊢*m member-in C* $\Longrightarrow$ *G*⊢*C* $\preceq_C$ *declclass m*
⟨*proof*⟩

lemma *stat-override-declclasses-relation*:
⟦*G*⊢(*declclass new*) $\prec_C$*1 superNew*; *G* ⊢*Method old member-of superNew* ⟧
$\Longrightarrow$ *G*⊢(*declclass new*) $\prec_C$ (*declclass old*)
⟨*proof*⟩

lemma *stat-overrides-commonD*:
⟦*G*⊢*new overrides*$_S$ *old*⟧ ⟹
declclass new ≠ *Object* ∧ ¬ *is-static new* ∧ *msig new* = *msig old* ∧
G⊢(*declclass new*) $\prec_C$ (*declclass old*) ∧
G⊢*Method new declared-in* (*declclass new*) ∧
G⊢*Method old declared-in* (*declclass old*)
⟨*proof*⟩

lemma *member-of-Package*:
assumes *G*⊢*m member-of C*
and *accmodi m* = *Package*
shows *pid* (*declclass m*) = *pid C*
⟨*proof*⟩

lemma *member-in-declC*: *G*⊢*m member-in C*⟹ *G*⊢*m member-in* (*declclass m*)
⟨*proof*⟩

lemma *dyn-accessible-from-commonD*: *G*⊢*m in C dyn-accessible-from S*
⟹ *G*⊢*m member-in C*
⟨*proof*⟩

lemma *no-Private-stat-override*:
⟦*G*⊢*new overrides*$_S$ *old*⟧ ⟹ *accmodi old* ≠ *Private*
⟨*proof*⟩

lemma *no-Private-override*: ⟦*G*⊢*new overrides old*⟧ ⟹ *accmodi old* ≠ *Private*
⟨*proof*⟩

lemma *permits-acc-inheritance*:
⟦*G*⊢*m in statC permits-acc-from accC*; *G*⊢*dynC* $\preceq_C$ *statC*
⟧ ⟹ *G*⊢*m in dynC permits-acc-from accC*
⟨*proof*⟩

lemma *permits-acc-static-declC*:
⟦*G*⊢*m in C permits-acc-from accC*; *G*⊢*m member-in C*; *is-static m*
⟧ ⟹ *G*⊢*m in* (*declclass m*) *permits-acc-from accC*
⟨*proof*⟩

lemma *dyn-accessible-from-static-declC*:
assumes *acc-C*: *G*⊢*m in C dyn-accessible-from accC* **and**
static: *is-static m*
shows *G*⊢*m in* (*declclass m*) *dyn-accessible-from accC*
⟨*proof*⟩

lemma *field-accessible-fromD*:
⟦*G*⊢*membr of C accessible-from accC*;*is-field membr*⟧
⟹ *G*⊢*membr member-of C* ∧
G⊢(*Class C*) *accessible-in* (*pid accC*) ∧
G⊢*membr in C permits-acc-from accC*
⟨*proof*⟩

lemma *field-accessible-from-permits-acc-inheritance*:
⟦*G*⊢*membr of statC accessible-from accC*; *is-field membr*; *G* ⊢ *dynC* $\preceq_C$ *statC*⟧
⟹ *G*⊢*membr in dynC permits-acc-from accC*
⟨*proof*⟩

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

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

lemma *member-of-subclseq-declC*:
$G\vdash m\ member\text{-}of\ C \Longrightarrow G\vdash C \preceq_C declclass\ m$
⟨*proof*⟩

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

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

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

corollary *member-of-stat-overrides-subcls*:
$\llbracket G\vdash Methd\ sig\ old\ member\text{-}of\ C;\ G\vdash Methd\ sig\ new\ member\text{-}of\ D; G\vdash D \preceq_C C;$
$G,sig\vdash new\ overrides_S\ old;\ ws\text{-}prog\ G \rrbracket$
$\Longrightarrow G\vdash D \prec_C C$
⟨*proof*⟩

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

lemma *accessible-inheritance*:
assumes *stat-acc*: $G\vdash m\ of\ statC\ accessible\text{-}from\ accC$ **and**
subclseq: $G\vdash dynC \preceq_C statC$ **and**

member-dynC: *G*⊢*m member-of dynC* **and**
dynC-acc: *G*⊢(*Class dynC*) *accessible-in* (*pid accC*)
shows *G*⊢*m of dynC accessible-from accC*
⟨*proof*⟩

fields and methods

type-synonym
fspec = *vname* × *qtname*

translations
(*type*) *fspec* <= (*type*) *vname* × *qtname*

definition
imethds :: *prog* ⇒ *qtname* ⇒ (*sig*,*qtname* × *mhead*) *tables* **where**
imethds G I =
iface-rec G I (λ*I i ts*. (*Un-tables ts*) ⊕⊕
(*set-option* ∘ *table-of* (*map* (λ(*s*,*m*). (*s*,*I*,*m*)) (*imethods i*))))

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

definition
accimethds :: *prog* ⇒ *pname* ⇒ *qtname* ⇒ (*sig*,*qtname* × *mhead*) *tables* **where**
accimethds G pack I =
(*if G*⊢*Iface I accessible-in pack*
then imethds G I
else (λ *k*. {}))

only returns imethds if the interface is accessible

definition
methd :: *prog* ⇒ *qtname* ⇒ (*sig*,*qtname* × *methd*) *table* **where**
methd G C =
class-rec G C Map.empty
(λ*C c subcls-mthds*.
filter-tab (λ*sig m*. *G*⊢*C inherits method sig m*)
subcls-mthds
++
table-of (*map* (λ(*s*,*m*). (*s*,*C*,*m*)) (*methods c*)))

methd G C: methods of a class C (statically visible from C), with inheritance and hiding cf. 8.4.6; Overriding is captured by *dynmethd*. Every new method with the same signature coalesces the method of a superclass.

definition
accmethd :: *prog* ⇒ *qtname* ⇒ *qtname* ⇒ (*sig*,*qtname* × *methd*) *table* **where**
accmethd G S C =
filter-tab (λ*sig m*. *G*⊢*method sig m of C accessible-from S*) (*methd G C*)

accmethd G S C: only those methods of *methd G C*, accessible from S

Note the class component in the accessibility filter. The class where method *m* is declared (*declC*) isn't necessarily accessible from the current scope *S*. The method can be made accessible through inheritance, too. So we must test accessibility of method *m* of class *C* (not *declclass m*)

definition
dynmethd :: *prog* ⇒ *qtname* ⇒ *qtname* ⇒ (*sig*,*qtname* × *methd*) *table* **where**
dynmethd G statC dynC =
(λ*sig*.
(*if G*⊢*dynC* $\preceq_C$ *statC*
then (*case methd G statC sig of*
None ⇒ *None*

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

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

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

definition

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

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

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

definition

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

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

definition

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

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

definition

```
  accfield :: prog ⇒ qtname ⇒ qtname ⇒ (vname, qtname  ×  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*⊢ (*declC*,*fdecl* (*n*,*f*)) *of C accessible-from S*)
field-tab)

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

note the class component in the accessibility filter (see also *methd*). The class declaring field *f* (*declC*) isn't necessarily accessible from scope *S*. The field can be made visible through inheritance, too. So we must test accessibility of field *f* of class *C* (not *declclass f*)

definition
is-methd :: *prog* ⇒ *qtname* ⇒ *sig* ⇒ *bool*
where *is-methd G* = (λ*C sig*. *is-class G C* ∧ *methd G C sig* ≠ *None*)

definition
efname :: ((*vname* × *qtname*) × *field*) ⇒ (*vname* × *qtname*)
where *efname* = *fst*

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

4 imethds

lemma *imethds-rec*: ⟦*iface G I* = *Some i*; *ws-prog G*⟧ ⟹
imethds G I = *Un-tables* ((λ*J*. *imethds G J*)'*set* (*isuperIfs i*)) ⊕⊕
(*set-option* ∘ *table-of* (*map* (λ(*s*,*mh*). (*s*,*I*,*mh*)) (*imethods i*)))
⟨*proof*⟩

lemma *imethds-norec*:
⟦*iface G md* = *Some i*; *ws-prog G*; *table-of* (*imethods i*) *sig* = *Some mh*⟧ ⟹
(*md*, *mh*) ∈ *imethds G md sig*
⟨*proof*⟩

lemma *imethds-declI*: ⟦*m* ∈ *imethds G I sig*; *ws-prog G*; *is-iface G I*⟧ ⟹
(∃ *i*. *iface G* (*decliface m*) = *Some i* ∧
table-of (*imethods i*) *sig* = *Some* (*mthd m*)) ∧
(*I*,*decliface m*) ∈ (*subint1 G*)* ∧ *m* ∈ *imethds G* (*decliface m*) *sig*
⟨*proof*⟩

lemma *imethds-cases*:
assumes *im*: *im* ∈ *imethds G I sig*
and *ifI*: *iface G I* = *Some i*
and *ws*: *ws-prog G*
obtains (*NewMethod*) *table-of* (*map* (λ(*s*, *mh*). (*s*, *I*, *mh*)) (*imethods i*)) *sig* = *Some im*
| (*InheritedMethod*) *J* **where** *J* ∈ *set* (*isuperIfs i*) **and** *im* ∈ *imethds G J sig*
⟨*proof*⟩

5 accimethd

lemma *accimethds-simp* [*simp*]:
G⊢*Iface I accessible-in pack* ⟹ *accimethds G pack I* = *imethds G I*
⟨*proof*⟩

lemma *accimethdsD*:
im ∈ *accimethds G pack I sig*
⟹ *im* ∈ *imethds G I sig* ∧ *G*⊢*Iface I accessible-in pack*
⟨*proof*⟩

lemma *accimethdsI*:
$[\![im \in imethds\ G\ I\ sig; G\vdash Iface\ I\ accessible\text{-}in\ pack]\!]$
$\Longrightarrow im \in accimethds\ G\ pack\ I\ sig$
⟨*proof*⟩

6 methd

lemma *methd-rec*: $[\![class\ G\ C = Some\ c;\ ws\text{-}prog\ G]\!] \Longrightarrow$
$methd\ G\ C$
$= (if\ C = Object$
$then\ Map.empty$
$else\ filter\text{-}tab\ (\lambda sig\ m.\ G\vdash C\ inherits\ method\ sig\ m)$
$(methd\ G\ (super\ c)))$
$++\ table\text{-}of\ (map\ (\lambda(s,m).\ (s,C,m))\ (methods\ c))$
⟨*proof*⟩

lemma *methd-norec*:
$[\![class\ G\ declC = Some\ c;\ ws\text{-}prog\ G; table\text{-}of\ (methods\ c)\ sig = Some\ m]\!]$
$\Longrightarrow methd\ G\ declC\ sig = Some\ (declC,\ m)$
⟨*proof*⟩

lemma *methd-declC*:
$[\![methd\ G\ C\ sig = Some\ m;\ ws\text{-}prog\ G; is\text{-}class\ G\ C]\!] \Longrightarrow$
$(\exists\ d.\ class\ G\ (declclass\ m) = Some\ d \wedge table\text{-}of\ (methods\ d)\ sig = Some\ (mthd\ m)) \wedge$
$G\vdash C \preceq_C (declclass\ m) \wedge methd\ G\ (declclass\ m)\ sig = Some\ m$
⟨*proof*⟩

lemma *methd-inheritedD*:
$[\![class\ G\ C = Some\ c;\ ws\text{-}prog\ G; methd\ G\ C\ sig = Some\ m]\!]$
$\Longrightarrow\ (declclass\ m \neq C \longrightarrow G \vdash C\ inherits\ method\ sig\ m)$
⟨*proof*⟩

lemma *methd-diff-cls*:
$[\![ws\text{-}prog\ G;\ is\text{-}class\ G\ C;\ is\text{-}class\ G\ D;$
$methd\ G\ C\ sig = m;\ methd\ G\ D\ sig = n;\ m \neq n$
$]\!] \Longrightarrow C \neq D$
⟨*proof*⟩

lemma *method-declared-inI*:
$[\![table\text{-}of\ (methods\ c)\ sig = Some\ m;\ class\ G\ C = Some\ c]\!]$
$\Longrightarrow G\vdash mdecl\ (sig,m)\ declared\text{-}in\ C$
⟨*proof*⟩

lemma *methd-declared-in-declclass*:
$[\![methd\ G\ C\ sig = Some\ m;\ ws\text{-}prog\ G; is\text{-}class\ G\ C]\!]$
$\Longrightarrow G\vdash Methd\ sig\ m\ declared\text{-}in\ (declclass\ m)$
⟨*proof*⟩

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

lemma *finite-methd*: $ws\text{-}prog\ G \Longrightarrow finite\ \{methd\ G\ C\ sig\ | sig\ C.\ is\text{-}class\ G\ C\}$

⟨*proof*⟩

lemma *finite-dom-methd*:
$[\![$*ws-prog G*; *is-class G C*$]\!] \Longrightarrow$ *finite* (*dom* (*methd G C*))
⟨*proof*⟩

7 accmethd

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

lemma *accmethd-SomeI*:
$[\![$*methd G C sig* = *Some m*; *G*$\vdash$*method sig m of C accessible-from S*$]\!]$
$\Longrightarrow$ *accmethd G S C sig* = *Some m*
⟨*proof*⟩

lemma *accmethd-declC*:
$[\![$*accmethd G S C sig* = *Some m*; *ws-prog G*; *is-class G C*$]\!] \Longrightarrow$
($\exists$ *d*. *class G* (*declclass m*)=*Some d* $\wedge$
table-of (*methods d*) *sig*=*Some* (*mthd m*)) $\wedge$
G$\vdash$*C* $\preceq_C$ (*declclass m*) $\wedge$ *methd G* (*declclass m*) *sig* = *Some m* $\wedge$
G$\vdash$*method sig m of C accessible-from S*
⟨*proof*⟩

lemma *finite-dom-accmethd*:
$[\![$*ws-prog G*; *is-class G C*$]\!] \Longrightarrow$ *finite* (*dom* (*accmethd G S C*))
⟨*proof*⟩

8 dynmethd

lemma *dynmethd-rec*:
$[\![$*class G dynC* = *Some c*; *ws-prog G*$]\!] \Longrightarrow$
dynmethd G statC dynC sig
= (*if G*$\vdash$*dynC* $\preceq_C$ *statC*
then (*case methd G statC sig of*
None $\Rightarrow$ *None*
| *Some statM*
$\Rightarrow$ (*case methd G dynC sig of*
None $\Rightarrow$ *dynmethd G statC* (*super c*) *sig*
| *Some dynM* $\Rightarrow$
(*if G*,*sig*$\vdash$ *dynM overrides statM* $\vee$ *dynM* = *statM*
then Some dynM
else (*dynmethd G statC* (*super c*) *sig*)
)))
else None)
(**is** - $\Longrightarrow$ - $\Longrightarrow$ *?Dynmethd-def dynC sig* = *?Dynmethd-rec dynC c sig*)
⟨*proof*⟩

lemma *dynmethd-C-C*:$[\![$*is-class G C*; *ws-prog G*$]\!]$
$\Longrightarrow$ *dynmethd G C C sig* = *methd G C sig*
⟨*proof*⟩

lemma *dynmethdSomeD*:
$[\![$*dynmethd G statC dynC sig* = *Some dynM*; *is-class G dynC*; *ws-prog G*$]\!]$
$\Longrightarrow$ *G*$\vdash$*dynC* $\preceq_C$ *statC* $\wedge$ ($\exists$ *statM*. *methd G statC sig* = *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 ≠ statM*
and *G,sig⊢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≠statM*
shows *dynC ≠ 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 ≠ dynM*
and *G,sig⊢ dynM overrides statM*
| (*Recursion*) *dynC ≠ Object* **and** *dynmethd G statC* (*super c*) *sig = Some dynM*
⟨*proof*⟩

lemma *dynmethd-declC*:
⟦*dynmethd G statC dynC sig = Some m*;
is-class G statC;*ws-prog G*
⟧ ⟹
(∃ *d. class G* (*declclass m*)=*Some d* ∧ *table-of* (*methods d*) *sig*=*Some* (*mthd m*)) ∧
G⊢dynC $\preceq_C$ (*declclass m*) ∧ *methd G* (*declclass m*) *sig = Some m*
⟨*proof*⟩

lemma *methd-Some-dynmethd-Some*:
assumes *statM*: *methd G statC sig = Some statM* **and**
subclseq: *G⊢dynC* $\preceq_C$ *statC* **and**
is-cls-statC: *is-class G statC* **and**
ws: *ws-prog G*
shows ∃ *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⊢dynC* $\preceq_C$ *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 ≠ statM* **and** *G,sig⊢dynM overrides statM*
⟨*proof*⟩

lemma *ws-dynmethd*:
assumes *statM*: *methd G statC sig = Some statM* **and**
subclseq: $G \vdash dynC \preceq_C statC$ **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 $\wedge$ $G \vdash resTy\ dynM \preceq resTy\ statM$
⟨*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** *dynimethd 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*: ⟦*class G C = Some c*; *ws-prog G*⟧ $\Longrightarrow$
fields G C = map (λ(*fn,ft*). ((*fn,C*),*ft*)) (*cfields c*) @
(*if C = Object then* [] *else fields G* (*super c*))
⟨*proof*⟩

lemma *fields-norec*:
⟦*class G fd = Some c*; *ws-prog G*; *table-of* (*cfields c*) *fn = Some f*⟧
$\Longrightarrow$ *table-of* (*fields G fd*) (*fn,fd*) = *Some f*
⟨*proof*⟩

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

lemma *fields-declC*:
⟦*table-of* (*fields G C*) *efn = Some f*; *ws-prog G*; *is-class G C*⟧ $\Longrightarrow$
($\exists$ *d. class G* (*declclassf efn*) = *Some d* $\wedge$
table-of (*cfields d*) (*fname efn*)=*Some f*) $\wedge$
$G \vdash C \preceq_C$ (*declclassf efn*) $\wedge$ *table-of* (*fields G* (*declclassf efn*)) *efn = Some f*
⟨*proof*⟩

lemma *fields-emptyI*: $\bigwedge y$. ⟦*ws-prog G*; *class G C = Some c*;*cfields c* = [];
C $\neq$ *Object* $\longrightarrow$ *class G* (*super c*) = *Some y* $\wedge$ *fields G* (*super c*) = []⟧ $\Longrightarrow$
fields G C = []
⟨*proof*⟩

lemma *fields-mono-lemma*:
⟦$x \in$ *set* (*fields G C*); $G \vdash D \preceq_C C$; *ws-prog G*⟧
$\Longrightarrow x \in$ *set* (*fields G D*)
⟨*proof*⟩

lemma *ws-unique-fields-lemma*:
⟦(*efn*,*fd*) ∈ *set* (*fields G* (*super c*)); *fc* ∈ *set* (*cfields c*); *ws-prog G*;
fname efn = *fname fc*; *declclassf efn* = *C*;
class G C = *Some c*; *C* ≠ *Object*; *class G* (*super c*) = *Some d*⟧ ⟹ *R*
⟨*proof*⟩

lemma *ws-unique-fields*: ⟦*is-class G C*; *ws-prog G*;
⋀*C c*. ⟦*class G C* = *Some c*⟧ ⟹ *unique* (*cfields c*) ⟧ ⟹
unique (*fields G C*)
⟨*proof*⟩

11 accfield

lemma *accfield-fields*:
accfield G S C fn = *Some f*
⟹ *table-of* (*fields G C*) (*fn*, *declclass f*) = *Some* (*fld f*)
⟨*proof*⟩

lemma *accfield-declC-is-class*:
⟦*is-class G C*; *accfield G S C en* = *Some* (*fd*, *f*); *ws-prog G*⟧ ⟹
is-class G fd
⟨*proof*⟩

lemma *accfield-accessibleD*:
accfield G S C fn = *Some f* ⟹ *G*⊢*Field fn f of C accessible-from S*
⟨*proof*⟩

12 is methd

lemma *is-methdI*:
⟦*class G C* = *Some y*; *methd G C sig* = *Some b*⟧ ⟹ *is-methd G C sig*
⟨*proof*⟩

lemma *is-methdD*:
is-methd G C sig ⟹ *class G C* ≠ *None* ∧ *methd G C sig* ≠ *None*
⟨*proof*⟩

lemma *finite-is-methd*:
ws-prog G ⟹ *finite* (*Collect* (*case-prod* (*is-methd G*)))
⟨*proof*⟩

calculation of the superclasses of a class

definition
superclasses :: *prog* ⇒ *qtname* ⇒ *qtname set* **where**
superclasses G C = *class-rec G C* {}
(λ *C c superclss*. (*if C*=*Object*
then {}
else insert (*super c*) *superclss*))

lemma *superclasses-rec*: ⟦*class G C* = *Some c*; *ws-prog G*⟧ ⟹
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*: $\bigwedge C\ c.\ [\![class\ G\ C = Some\ c;\ C \neq Object]\!]$
$\Longrightarrow \exists\, sc.\ class\ G\ (super\ c) = Some\ sc$
and *x*: *x*∈*superclasses G D*
shows *x*∈*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*: $\bigwedge C\ c.\ [\![class\ G\ C = Some\ c;\ C \neq Object]\!]$
$\Longrightarrow \exists\, sc.\ class\ G\ (super\ c) = Some\ sc$
shows *D*∈*superclasses G C* ⟨*proof*⟩

end

Chapter 11

WellType

1 Well-typedness of Java programs

theory *WellType*
imports *DeclConcepts*
begin

improvements over Java Specification 1.0:

- methods of Object can be called upon references of interface or array type

simplifications:

- the type rules include all static checks on statements and expressions, e.g. definedness of names (of parameters, locals, fields, methods)

design issues:

- unified type judgment for statements, variables, expressions, expression lists
- statements are typed like expressions with dummy type Void
- the typing rules take an extra argument that is capable of determining the dynamic type of objects. Therefore, they can be used for both checking static types and determining runtime types in transition semantics.

type-synonym *lenv*
  = (*lname*, *ty*) *table* — local variables, including This and Result

record *env* =
  prg:: *prog* — program
  cls:: *qtname* — current package and class name
  lcl:: *lenv* — local environment

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

abbreviation
pkg :: *env* ⇒ *pname* — select the current package from an environment
where *pkg e* == *pid* (*cls e*)

Static overloading: maximally specific methods

type-synonym
emhead = *ref-ty* × *mhead*

— Some mnemotic selectors for emhead
definition
declrefT :: *emhead* ⇒ *ref-ty*
where *declrefT* = *fst*

definition
mhd :: *emhead* ⇒ *mhead*
where *mhd* ≡ *snd*

lemma *declrefT-simp*[*simp*]:*declrefT* (*r*,*m*) = *r*
⟨*proof*⟩

lemma *mhd-simp*[*simp*]:*mhd* (*r*,*m*) = *m*
⟨*proof*⟩

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

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

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

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

definition
cmheads :: *prog* ⇒ *qtname* ⇒ *qtname* ⇒ *sig* ⇒ *emhead set*
where *cmheads G S C* = (*λsig*. (*λ*(*Cls*,*mthd*). (*ClassT Cls*,(*mhead mthd*))) ‘ *set-option* (*accmethd G S C sig*))

definition
Objectmheads :: *prog* ⇒ *qtname* ⇒ *sig* ⇒ *emhead set* **where**
Objectmheads G S =
(*λsig*. (*λ*(*Cls*,*mthd*). (*ClassT Cls*,(*mhead mthd*)))
‘ *set-option* (*filter-tab* (*λsig m*. *accmodi m* ≠ *Private*) (*accmethd G S Object*) *sig*))

definition
accObjectmheads :: *prog* ⇒ *qtname* ⇒ *ref-ty* ⇒ *sig* ⇒ *emhead set*
where
accObjectmheads G S T =
(*if G*⊢*RefT T accessible-in* (*pid S*)
then Objectmheads G S
else (*λsig*. {}))

primrec *mheads* :: *prog* ⇒ *qtname* ⇒ *ref-ty* ⇒ *sig* ⇒ *emhead set*
where
mheads G S NullT = (*λsig*. {})
| *mheads G S* (*IfaceT I*) = (*λsig*. (*λ*(*I*,*h*).(*IfaceT I*,*h*))
‘ *accimethds G* (*pid S*) *I sig* ∪
accObjectmheads G S (*IfaceT I*) *sig*)
| *mheads G S* (*ClassT C*) = *cmheads G S C*
| *mheads G S* (*ArrayT T*) = *accObjectmheads G S* (*ArrayT T*)

definition
— applicable methods, cf. 15.11.2.1
$appl\text{-}methds :: prog \Rightarrow qtname \Rightarrow ref\text{-}ty \Rightarrow sig \Rightarrow (emhead \times ty\ list)\ set$ **where**
$appl\text{-}methds\ G\ S\ rt = (\lambda\ sig.$
$\{(mh,pTs') \mid mh\ pTs'.\ mh \in mheads\ G\ S\ rt\ (|name=name\ sig,parTs=pTs'|) \wedge$
$G\vdash(parTs\ sig)[\preceq]pTs'\})$

definition
— more specific methods, cf. 15.11.2.2
$more\text{-}spec :: prog \Rightarrow emhead \times ty\ list \Rightarrow emhead \times ty\ list \Rightarrow bool$ **where**
$more\text{-}spec\ G = (\lambda(mh,pTs).\ \lambda(mh',pTs').\ G\vdash pTs[\preceq]pTs')$

definition
— maximally specific methods, cf. 15.11.2.2
$max\text{-}spec :: prog \Rightarrow qtname \Rightarrow ref\text{-}ty \Rightarrow sig \Rightarrow (emhead \times ty\ list)\ set$ **where**
$max\text{-}spec\ G\ S\ rt\ sig = \{m.\ m \in appl\text{-}methds\ G\ S\ rt\ sig \wedge$
$(\forall m' \in appl\text{-}methds\ G\ S\ rt\ sig.\ more\text{-}spec\ G\ m'\ m \longrightarrow m'=m)\}$

lemma *max-spec2appl-meths*:
$x \in max\text{-}spec\ G\ S\ T\ sig \Longrightarrow x \in appl\text{-}methds\ G\ S\ T\ sig$
⟨*proof*⟩

lemma *appl-methsD*: $(mh,pTs') \in appl\text{-}methds\ G\ S\ T\ (|name=mn,parTs=pTs|) \Longrightarrow$
$mh \in mheads\ G\ S\ T\ (|name=mn,parTs=pTs'|) \wedge G\vdash pTs[\preceq]pTs'$
⟨*proof*⟩

lemma *max-spec2mheads*:
$max\text{-}spec\ G\ S\ rt\ (|name=mn,parTs=pTs|) = insert\ (mh,\ pTs')\ A$
$\Longrightarrow mh \in mheads\ G\ S\ rt\ (|name=mn,parTs=pTs'|) \wedge G\vdash pTs[\preceq]pTs'$
⟨*proof*⟩

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

definition
$invmode :: ('a::type)member\text{-}scheme \Rightarrow expr \Rightarrow inv\text{-}mode$ **where**
$invmode\ m\ e = (if\ is\text{-}static\ m$
$then\ Static$
$else\ if\ e=Super\ then\ SuperM\ else\ IntVir)$

lemma *invmode-nonstatic* [*simp*]:
$invmode\ (|access=a,static=False,\ldots=x|)\ (Acc\ (LVar\ e)) = IntVir$
⟨*proof*⟩

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

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

lemma *Null-staticD*:
$a'=Null \longrightarrow (is\text{-}static\ m) \Longrightarrow invmode\ m\ e = IntVir \longrightarrow a' \neq Null$

⟨*proof*⟩

Typing for unary operations

primrec *unop-type* :: *unop* ⇒ *prim-ty*
where
unop-type UPlus = *Integer*
| *unop-type UMinus* = *Integer*
| *unop-type UBitNot* = *Integer*
| *unop-type UNot* = *Boolean*

primrec *wt-unop* :: *unop* ⇒ *ty* ⇒ *bool*
where
wt-unop UPlus t = (*t* = *PrimT Integer*)
| *wt-unop UMinus t* = (*t* = *PrimT Integer*)
| *wt-unop UBitNot t* = (*t* = *PrimT Integer*)
| *wt-unop UNot t* = (*t* = *PrimT Boolean*)

Typing for binary operations

primrec *binop-type* :: *binop* ⇒ *prim-ty*
where
binop-type Mul = *Integer*
| *binop-type Div* = *Integer*
| *binop-type Mod* = *Integer*
| *binop-type Plus* = *Integer*
| *binop-type Minus* = *Integer*
| *binop-type LShift* = *Integer*
| *binop-type RShift* = *Integer*
| *binop-type RShiftU* = *Integer*
| *binop-type Less* = *Boolean*
| *binop-type Le* = *Boolean*
| *binop-type Greater* = *Boolean*
| *binop-type Ge* = *Boolean*
| *binop-type Eq* = *Boolean*
| *binop-type Neq* = *Boolean*
| *binop-type BitAnd* = *Integer*
| *binop-type And* = *Boolean*
| *binop-type BitXor* = *Integer*
| *binop-type Xor* = *Boolean*
| *binop-type BitOr* = *Integer*
| *binop-type Or* = *Boolean*
| *binop-type CondAnd* = *Boolean*
| *binop-type CondOr* = *Boolean*

primrec *wt-binop* :: *prog* ⇒ *binop* ⇒ *ty* ⇒ *ty* ⇒ *bool*
where
wt-binop G Mul t1 t2 = ((*t1* = *PrimT Integer*) ∧ (*t2* = *PrimT Integer*))
| *wt-binop G Div t1 t2* = ((*t1* = *PrimT Integer*) ∧ (*t2* = *PrimT Integer*))
| *wt-binop G Mod t1 t2* = ((*t1* = *PrimT Integer*) ∧ (*t2* = *PrimT Integer*))
| *wt-binop G Plus t1 t2* = ((*t1* = *PrimT Integer*) ∧ (*t2* = *PrimT Integer*))
| *wt-binop G Minus t1 t2* = ((*t1* = *PrimT Integer*) ∧ (*t2* = *PrimT Integer*))
| *wt-binop G LShift t1 t2* = ((*t1* = *PrimT Integer*) ∧ (*t2* = *PrimT Integer*))
| *wt-binop G RShift t1 t2* = ((*t1* = *PrimT Integer*) ∧ (*t2* = *PrimT Integer*))
| *wt-binop G RShiftU t1 t2* = ((*t1* = *PrimT Integer*) ∧ (*t2* = *PrimT Integer*))
| *wt-binop G Less t1 t2* = ((*t1* = *PrimT Integer*) ∧ (*t2* = *PrimT Integer*))
| *wt-binop G Le t1 t2* = ((*t1* = *PrimT Integer*) ∧ (*t2* = *PrimT Integer*))
| *wt-binop G Greater t1 t2* = ((*t1* = *PrimT Integer*) ∧ (*t2* = *PrimT Integer*))
| *wt-binop G Ge t1 t2* = ((*t1* = *PrimT Integer*) ∧ (*t2* = *PrimT Integer*))

$|$ *wt-binop G Eq* $t1\ t2 = (G \vdash t1 \preceq t2 \lor G \vdash t2 \preceq t1)$
$|$ *wt-binop G Neq* $t1\ t2 = (G \vdash t1 \preceq t2 \lor G \vdash t2 \preceq t1)$
$|$ *wt-binop G BitAnd* $t1\ t2 = ((t1 = PrimT\ Integer) \land (t2 = PrimT\ Integer))$
$|$ *wt-binop G And* $t1\ t2 = ((t1 = PrimT\ Boolean) \land (t2 = PrimT\ Boolean))$
$|$ *wt-binop G BitXor* $t1\ t2 = ((t1 = PrimT\ Integer) \land (t2 = PrimT\ Integer))$
$|$ *wt-binop G Xor* $t1\ t2 = ((t1 = PrimT\ Boolean) \land (t2 = PrimT\ Boolean))$
$|$ *wt-binop G BitOr* $t1\ t2 = ((t1 = PrimT\ Integer) \land (t2 = PrimT\ Integer))$
$|$ *wt-binop G Or* $t1\ t2 = ((t1 = PrimT\ Boolean) \land (t2 = PrimT\ Boolean))$
$|$ *wt-binop G CondAnd* $t1\ t2 = ((t1 = PrimT\ Boolean) \land (t2 = PrimT\ Boolean))$
$|$ *wt-binop G CondOr* $t1\ t2 = ((t1 = PrimT\ Boolean) \land (t2 = PrimT\ Boolean))$

Typing for terms

type-synonym *tys* = *ty* + *ty list*
translations
(*type*) *tys* <= (*type*) *ty* + *ty list*

inductive *wt* :: *env* ⇒ *dyn-ty* ⇒ [*term*,*tys*] ⇒ *bool* (‹-,-⊨-::-› [51,51,51,51] 50)
and *wt-stmt* :: *env* ⇒ *dyn-ty* ⇒ *stmt* ⇒ *bool* (‹-,-⊨-::√› [51,51,51] 50)
and *ty-expr* :: *env* ⇒ *dyn-ty* ⇒ [*expr* ,*ty*] ⇒ *bool* (‹-,-⊨-::−-› [51,51,51,51] 50)
and *ty-var* :: *env* ⇒ *dyn-ty* ⇒ [*var* ,*ty*] ⇒ *bool* (‹-,-⊨-::=-› [51,51,51,51] 50)
and *ty-exprs* :: *env* ⇒ *dyn-ty* ⇒ [*expr list*, *ty list*] ⇒ *bool*
(‹-,-⊨-::≐-› [51,51,51,51] 50)
where

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

— well-typed statements

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

$|$ *Expr*: $[\![E,dt \models e::-T]\!] \Longrightarrow$
$E,dt \models Expr\ e::\surd$
— cf. 14.6
$|$ *Lab*: $E,dt \models c::\surd \Longrightarrow$
$E,dt \models l\bullet\ c::\surd$

$|$ *Comp*: $[\![E,dt \models c1::\surd;$
$E,dt \models c2::\surd]\!] \Longrightarrow$
$E,dt \models c1;;\ c2::\surd$

— cf. 14.8
$|$ *If*: $[\![E,dt \models e::-PrimT\ Boolean;$
$E,dt \models c1::\surd;$
$E,dt \models c2::\surd]\!] \Longrightarrow$
$E,dt \models If(e)\ c1\ Else\ c2::\surd$

— cf. 14.10
$|$ *Loop*: $[\![E,dt \models e::-PrimT\ Boolean;$
$E,dt \models c::\surd]\!] \Longrightarrow$
$E,dt \models l\bullet\ While(e)\ c::\surd$
— cf. 14.13, 14.15, 14.16
$|$ *Jmp*: $E,dt \models Jmp\ jump::\surd$

— cf. 14.16

| *Throw*: $[\![E,dt \models e{::}-Class\ tn;$
$prg\ E \vdash tn \preceq_C\ SXcpt\ Throwable]\!] \Longrightarrow$
$E,dt \models Throw\ e{::}\surd$
— cf. 14.18
| *Try*: $[\![E,dt \models c1{::}\surd;\ prg\ E \vdash tn \preceq_C\ SXcpt\ Throwable;$
$lcl\ E\ (VName\ vn) = None;\ E\ (\!|lcl := (lcl\ E)(VName\ vn \mapsto Class\ tn)|\!),dt \models c2{::}\surd]\!]$
$\Longrightarrow$
$E,dt \models Try\ c1\ Catch(tn\ vn)\ c2{::}\surd$

— cf. 14.18
| *Fin*: $[\![E,dt \models c1{::}\surd;\ E,dt \models c2{::}\surd]\!] \Longrightarrow$
$E,dt \models c1\ Finally\ c2{::}\surd$

| *Init*: $[\![is\text{-}class\ (prg\ E)\ C]\!] \Longrightarrow$
$E,dt \models Init\ C{::}\surd$
— *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*: $[\![is\text{-}acc\text{-}class\ (prg\ E)\ (pkg\ E)\ C]\!] \Longrightarrow$
$E,dt \models NewC\ C{::}-Class\ C$
— cf. 15.9
| *NewA*: $[\![is\text{-}acc\text{-}type\ (prg\ E)\ (pkg\ E)\ T;$
$E,dt \models i{::}-PrimT\ Integer]\!] \Longrightarrow$
$E,dt \models New\ T[i]{::}-T.[]$

— cf. 15.15
| *Cast*: $[\![E,dt \models e{::}-T;\ is\text{-}acc\text{-}type\ (prg\ E)\ (pkg\ E)\ T';$
$prg\ E \vdash T \preceq?\ T']\!] \Longrightarrow$
$E,dt \models Cast\ T'\ e{::}-T'$

— cf. 15.19.2
| *Inst*: $[\![E,dt \models e{::}-RefT\ T;\ is\text{-}acc\text{-}type\ (prg\ E)\ (pkg\ E)\ (RefT\ T');$
$prg\ E \vdash RefT\ T \preceq?\ RefT\ T']\!] \Longrightarrow$
$E,dt \models e\ InstOf\ T'{::}-PrimT\ Boolean$

— cf. 15.7.1
| *Lit*: $[\![typeof\ dt\ x = Some\ T]\!] \Longrightarrow$
$E,dt \models Lit\ x{::}-T$

| *UnOp*: $[\![E,dt \models e{::}-Te;\ wt\text{-}unop\ unop\ Te;\ T = PrimT\ (unop\text{-}type\ unop)]\!]$
$\Longrightarrow$
$E,dt \models UnOp\ unop\ e{::}-T$

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

— cf. 15.10.2, 15.11.1
| *Super*: $[\![lcl\ E\ This = Some\ (Class\ C);\ C \neq Object;$
$class\ (prg\ E)\ C = Some\ c]\!] \Longrightarrow$
$E,dt \models Super{::}-Class\ (super\ c)$

— cf. 15.13.1, 15.10.1, 15.12
| *Acc*: $[\![E,dt \models va{::}=T]\!] \Longrightarrow$
$E,dt \models Acc\ va{::}-T$

— cf. 15.25, 15.25.1
| *Ass*: $\llbracket E,dt \models va::=T;\ va \neq LVar\ This;$
$E,dt \models v\ ::-T';$
$prg\ E \vdash T' \preceq T \rrbracket \Longrightarrow$
$E,dt \models va:=v::-T'$

— cf. 15.24
| *Cond*: $\llbracket E,dt \models e0::-PrimT\ Boolean;$
$E,dt \models e1::-T1;\ E,dt \models e2::-T2;$
$prg\ E \vdash T1 \preceq T2 \wedge T = T2\ \vee\ prg\ E \vdash T2 \preceq T1 \wedge T = T1 \rrbracket \Longrightarrow$
$E,dt \models e0\ ?\ e1\ :\ e2::-T$

— cf. 15.11.1, 15.11.2, 15.11.3
| *Call*: $\llbracket E,dt \models e::-RefT\ statT;$
$E,dt \models ps::\doteq pTs;$
$max\text{-}spec\ (prg\ E)\ (cls\ E)\ statT\ (\!|name=mn,parTs=pTs|\!)$
$= \{((statDeclT,m),pTs')\}$
$\rrbracket \Longrightarrow$
$E,dt \models \{cls\ E,statT,invmode\ m\ e\}e\cdot mn(\{pTs'\}ps)::-(resTy\ m)$

| *Methd*: $\llbracket is\text{-}class\ (prg\ E)\ C;$
$methd\ (prg\ E)\ C\ sig = Some\ m;$
$E,dt \models Body\ (declclass\ m)\ (stmt\ (mbody\ (mthd\ m)))::-T \rrbracket \Longrightarrow$
$E,dt \models Methd\ C\ sig::-T$
— 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*: $\llbracket is\text{-}class\ (prg\ E)\ D;$
$E,dt \models blk::\surd;$
$(lcl\ E)\ Result = Some\ T;$
$is\text{-}type\ (prg\ E)\ T \rrbracket \Longrightarrow$
$E,dt \models Body\ D\ blk::-T$
— 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*: $\llbracket lcl\ E\ vn = Some\ T;\ is\text{-}acc\text{-}type\ (prg\ E)\ (pkg\ E)\ T \rrbracket \Longrightarrow$
$E,dt \models LVar\ vn::=T$
— cf. 15.10.1
| *FVar*: $\llbracket E,dt \models e::-Class\ C;$
$accfield\ (prg\ E)\ (cls\ E)\ C\ fn = Some\ (statDeclC,f) \rrbracket \Longrightarrow$
$E,dt \models \{cls\ E,statDeclC,is\text{-}static\ f\}e..fn::=(type\ f)$
— cf. 15.12
| *AVar*: $\llbracket E,dt \models e::-T.[];$
$E,dt \models i::-PrimT\ Integer \rrbracket \Longrightarrow$
$E,dt \models e.[i]::=T$

— well-typed expression lists

— cf. 15.11.???
| *Nil*: $E,dt \models []::\doteq[]$

— cf. 15.11.???
| *Cons*: ⟦*E*,*dt*⊨*e* ::−*T*;
E,*dt*⊨*es*::≐*Ts*⟧ ⟹
E,*dt*⊨*e*#*es*::≐*T*#*Ts*

abbreviation
wt-syntax :: *env* ⇒ [*term*,*tys*] ⇒ *bool* (‹-⊢-::-› [*51*,*51*,*51*] *50*)
where *E*⊢*t*::*T* == *E*,*empty-dt*⊨*t*:: *T*

abbreviation
wt-stmt-syntax :: *env* ⇒ *stmt* ⇒ *bool* (‹-⊢-::√› [*51*,*51*] *50*)
where *E*⊢*s*::√ == *E*⊢*In1r s* :: *Inl* (*PrimT Void*)

abbreviation
ty-expr-syntax :: *env* ⇒ [*expr*, *ty*] ⇒ *bool* (‹-⊢-::−-› [*51*,*51*,*51*] *50*)
where *E*⊢*e*::−*T* == *E*⊢*In1l e* :: *Inl T*

abbreviation
ty-var-syntax :: *env* ⇒ [*var*, *ty*] ⇒ *bool* (‹-⊢-::=-› [*51*,*51*,*51*] *50*)
where *E*⊢*e*::=*T* == *E*⊢*In2 e* :: *Inl T*

abbreviation
ty-exprs-syntax :: *env* ⇒ [*expr list*, *ty list*] ⇒ *bool* (‹-⊢-::≐-› [*51*,*51*,*51*] *50*)
where *E*⊢*e*::≐*T* == *E*⊢*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*]
⟨*ML*⟩

inductive-cases *wt-elim-cases* [*cases set*]:
E,*dt*⊨*In2* (*LVar vn*) ::*T*
E,*dt*⊨*In2* ({*accC*,*statDeclC*,*s*}*e*..*fn*)::*T*
E,*dt*⊨*In2* (*e*.[*i*]) ::*T*
E,*dt*⊨*In1l* (*NewC C*) ::*T*
E,*dt*⊨*In1l* (*New T′*[*i*]) ::*T*
E,*dt*⊨*In1l* (*Cast T′ e*) ::*T*
E,*dt*⊨*In1l* (*e InstOf T′*) ::*T*
E,*dt*⊨*In1l* (*Lit x*) ::*T*
E,*dt*⊨*In1l* (*UnOp unop e*) ::*T*
E,*dt*⊨*In1l* (*BinOp binop e1 e2*) ::*T*
E,*dt*⊨*In1l* (*Super*) ::*T*
E,*dt*⊨*In1l* (*Acc va*) ::*T*
E,*dt*⊨*In1l* (*Ass va v*) ::*T*
E,*dt*⊨*In1l* (*e0 ? e1 : e2*) ::*T*
E,*dt*⊨*In1l* ({*accC*,*statT*,*mode*}*e*·*mn*({*pT′*}*p*))::*T*
E,*dt*⊨*In1l* (*Methd C sig*) ::*T*
E,*dt*⊨*In1l* (*Body D blk*) ::*T*
E,*dt*⊨*In3* ([]) ::*Ts*
E,*dt*⊨*In3* (*e*#*es*) ::*Ts*

E,dt⊨In1r Skip ::x
E,dt⊨In1r (Expr e) ::x
E,dt⊨In1r (c1;; c2) ::x
E,dt⊨In1r (l• c) ::x
E,dt⊨In1r (If(e) c1 Else c2) ::x
E,dt⊨In1r (l• While(e) c) ::x
E,dt⊨In1r (Jmp jump) ::x
E,dt⊨In1r (Throw e) ::x
E,dt⊨In1r (Try c1 Catch(tn vn) c2)::x
E,dt⊨In1r (c1 Finally c2) ::x
E,dt⊨In1r (Init C) ::x
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 ⟹ G⊢(Class C) accessible-in P
⟨*proof*⟩

lemma *is-acc-iface-is-iface*: *is-acc-iface G P I ⟹ is-iface G I*
⟨*proof*⟩

lemma *is-acc-iface-Iface-is-accessible*:
is-acc-iface G P I ⟹ G⊢(Iface I) accessible-in P
⟨*proof*⟩

lemma *is-acc-type-is-type*: *is-acc-type G P T ⟹ is-type G T*
⟨*proof*⟩

lemma *is-acc-iface-is-accessible*:
is-acc-type G P T ⟹ G⊢T accessible-in P
⟨*proof*⟩

lemma *wt-Methd-is-methd*:
E⊢In1l (Methd C sig)::T ⟹ 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*:
⟦E,dt⊨e::−RefT statT; E,dt⊨ps::≐pTs;
max-spec (prg E) (cls E) statT (⦇name=mn,parTs=pTs⦈)
= {((statDeclC,m),pTs′)};rT=(resTy m);accC=cls E;
mode = invmode m e⟧ ⟹ E,dt⊨{accC,statT,mode}e·mn({pTs′}ps)::−rT
⟨*proof*⟩

lemma *invocationTypeExpr-noClassD*:
⟦ E⊢e::−RefT statT⟧
⟹ (∀ statC. statT ≠ ClassT statC) ⟶ invmode m e ≠ SuperM
⟨*proof*⟩

lemma *wt-Super*:
⟦lcl E This = Some (Class C); C ≠ Object; class (prg E) C = Some c; D=super c⟧
⟹ E,dt⊨Super::−Class D
⟨*proof*⟩

lemma *wt-FVar*:

$[\![E,dt \models e{::}-Class\ C;\ accfield\ (prg\ E)\ (cls\ E)\ C\ fn = Some\ (statDeclC,f);$
$sf{=}is\text{-}static\ f;\ fT{=}(type\ f);\ accC{=}cls\ E]\!]$
$\Longrightarrow E,dt \models \{accC,statDeclC,sf\}e..fn{::}{=}fT$
⟨proof⟩

lemma *wt-init* [*iff*]: $E,dt \models Init\ C{::}\surd = is\text{-}class\ (prg\ E)\ C$
⟨proof⟩

declare *wt.Skip* [*iff*]

lemma *wt-StatRef*:
$is\text{-}acc\text{-}type\ (prg\ E)\ (pkg\ E)\ (RefT\ rt) \Longrightarrow E \vdash StatRef\ rt{::}-RefT\ rt$
⟨proof⟩

lemma *wt-Inj-elim*:
$\bigwedge E.\ E,dt \models t{::}U \Longrightarrow case\ t\ of$
$\quad In1\ ec \Rightarrow (case\ ec\ of$
$\quad\quad Inl\ e \Rightarrow \exists T.\ U{=}Inl\ T$
$\quad\quad |\ Inr\ s \Rightarrow U{=}Inl\ (PrimT\ Void))$
$\quad |\ In2\ e \Rightarrow (\exists T.\ U{=}Inl\ T)$
$\quad |\ In3\ e \Rightarrow (\exists T.\ U{=}Inr\ T)$
⟨proof⟩

lemma *wt-expr-eq*: $E,dt \models In1l\ t{::}U = (\exists T.\ U{=}Inl\ T \land E,dt \models t{::}-T)$
⟨proof⟩

lemma *wt-var-eq*: $E,dt \models In2\ t{::}U = (\exists T.\ U{=}Inl\ T \land E,dt \models t{::}{=}T)$
⟨proof⟩

lemma *wt-exprs-eq*: $E,dt \models In3\ t{::}U = (\exists Ts.\ U{=}Inr\ Ts \land E,dt \models t{::}\doteq Ts)$
⟨proof⟩

lemma *wt-stmt-eq*: $E,dt \models In1r\ t{::}U = (U{=}Inl(PrimT\ Void) \land E,dt \models t{::}\surd)$
⟨proof⟩

⟨ML⟩

lemma *wt-elim-BinOp*:
$[\![E,dt \models In1l\ (BinOp\ binop\ e1\ e2){::}T;$
$\bigwedge T1\ T2\ T3.$
$[\![E,dt \models e1{::}-T1;\ E,dt \models e2{::}-T2;\ wt\text{-}binop\ (prg\ E)\ binop\ T1\ T2;$
$E,dt \models (if\ b\ then\ In1l\ e2\ else\ In1r\ Skip){::}T3;$
$T = Inl\ (PrimT\ (binop\text{-}type\ binop))]\!]$
$\Longrightarrow P]\!]$
$\Longrightarrow P$
⟨proof⟩

lemma *Inj-eq-lemma* [*simp*]:
$(\forall T.\ (\exists T'.\ T = Inj\ T' \land P\ T') \longrightarrow Q\ T) = (\forall T'.\ P\ T' \longrightarrow Q\ (Inj\ T'))$
⟨proof⟩

lemma *single-valued-tys-lemma* [*rule-format* (*no-asm*)]:
$\forall S\ T.\ G \vdash S \preceq T \longrightarrow G \vdash T \preceq S \longrightarrow S = T \Longrightarrow E,dt \models t{::}T \Longrightarrow$
$G = prg\ E \longrightarrow (\forall T'.\ E,dt \models t{::}T' \longrightarrow T = T')$
⟨proof⟩

lemma *single-valued-tys*:
ws-prog (*prg E*) $\Longrightarrow$ *single-valued* $\{(t,T).\ E,dt \models t{::}T\}$
⟨*proof*⟩

lemma *typeof-empty-is-type*: *typeof* ($\lambda a.$ *None*) v = *Some* T $\Longrightarrow$ *is-type* G T
⟨*proof*⟩

lemma *typeof-is-type*: ($\forall a.\ v \neq$ *Addr* a) $\Longrightarrow$ $\exists T.$ *typeof* dt v = *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 abruption (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· s) = jumpNestingOkS ({j} ∪ jmps) s*
| *jumpNestingOkS jmps (c1;;c2) = (jumpNestingOkS jmps c1 ∧*
jumpNestingOkS jmps c2)
| *jumpNestingOkS jmps (If(e) c1 Else c2) = (jumpNestingOkS jmps c1 ∧*
jumpNestingOkS jmps c2)
| *jumpNestingOkS jmps (l· While(e) c) = jumpNestingOkS ({Cont l} ∪ jmps) c*
— The label of the while loop only handles continue jumps. Breaks are only handled by *Lab*
| *jumpNestingOkS jmps (Jmp j) = (j ∈ jmps)*
| *jumpNestingOkS jmps (Throw e) = True*
| *jumpNestingOkS jmps (Try c1 Catch(C vn) c2) = (jumpNestingOkS jmps c1 ∧*
jumpNestingOkS jmps c2)
| *jumpNestingOkS jmps (c1 Finally c2) = (jumpNestingOkS jmps c1 ∧*
jumpNestingOkS jmps c2)
| *jumpNestingOkS jmps (Init C) = True*
— wellformedness of the program must enshure that for all initializers jumpNestingOkS holds
— Dummy analysis for intermediate smallstep term *FinA*
| *jumpNestingOkS jmps (FinA a c) = False*

definition *jumpNestingOk* :: *jump set* ⇒ *term* ⇒ *bool* **where**
jumpNestingOk jmps t = (case t of
In1 se ⇒ (case se of
Inl e ⇒ True
| *Inr s ⇒ jumpNestingOkS jmps s)*
| *In2 v ⇒ True*
| *In3 es ⇒ True)*

lemma *jumpNestingOk-expr-simp* [*simp*]: *jumpNestingOk jmps (In1l e) = True*
⟨*proof*⟩

lemma *jumpNestingOk-expr-simp1* [*simp*]: *jumpNestingOk jmps* ⟨e::*expr*⟩ = *True*
⟨*proof*⟩

lemma *jumpNestingOk-stmt-simp* [*simp*]:
jumpNestingOk jmps (In1r s) = jumpNestingOkS jmps s
⟨*proof*⟩

lemma *jumpNestingOk-stmt-simp1* [*simp*]:
jumpNestingOk jmps ⟨s::*stmt*⟩ = *jumpNestingOkS jmps s*
⟨*proof*⟩

lemma *jumpNestingOk-var-simp* [*simp*]: *jumpNestingOk jmps (In2 v) = True*
⟨*proof*⟩

lemma *jumpNestingOk-var-simp1* [*simp*]: *jumpNestingOk jmps* ⟨v::*var*⟩ = *True*
⟨*proof*⟩

lemma *jumpNestingOk-expr-list-simp* [*simp*]: *jumpNestingOk jmps (In3 es) = True*
⟨*proof*⟩

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

Calculation of assigned variables for boolean expressions

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 *Methd*, *Body*, *InsInitE* and *Callee*
| *assignsE* (*Methd 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* (⟨*e*⟩) = *assignsE e*
⟨*proof*⟩

lemma *assigns-stmt-simp* [*simp*]: *assigns* (*In1r s*) = {}
⟨*proof*⟩

lemma *assigns-stmt-simp1* [*simp*]: *assigns* (⟨*s*::*stmt*⟩) = {}

⟨*proof*⟩

lemma *assigns-var-simp* [*simp*]: *assigns* (*In2 v*) = *assignsV v*
⟨*proof*⟩

lemma *assigns-var-simp1* [*simp*]: *assigns* (⟨*v*⟩) = *assignsV v*
⟨*proof*⟩

lemma *assigns-expr-list-simp* [*simp*]: *assigns* (*In3 es*) = *assignsEs es*
⟨*proof*⟩

lemma *assigns-expr-list-simp1* [*simp*]: *assigns* (⟨*es*⟩) = *assignsEs es*
⟨*proof*⟩

3 Analysis of constant expressions

```
primrec constVal :: expr ⇒ val option
where
  constVal (NewC c)      = None
| constVal (NewA t e)    = None
| constVal (Cast t e)    = None
| constVal (Inst e r)    = None
| constVal (Lit val)     = Some val
| constVal (UnOp unop e) = (case (constVal e) of
                              None   ⇒ None
                            | Some v ⇒ Some (eval-unop unop v))
| constVal (BinOp binop e1 e2) = (case (constVal e1) of
                                     None    ⇒ None
                                   | Some v1 ⇒ (case (constVal e2) of
                                                   None    ⇒ None
                                                 | Some v2 ⇒ Some (eval-binop
                                                                binop v1 v2)))
| constVal (Super)       = None
| constVal (Acc v)       = None
| constVal (Ass v e)     = None
| constVal (Cond b e1 e2) = (case (constVal b) of
                               None   ⇒ None
                             | Some bv⇒ (case the-Bool bv of
                                           True ⇒ (case (constVal e2) of
                                                     None   ⇒ None
                                                   | Some v ⇒ constVal e1)
                                         | False⇒ (case (constVal e1) of
                                                     None   ⇒ None
                                                   | Some v ⇒ constVal e2)))
```
— Note that *constVal* (*Cond b e1 e2*) is stricter as it could be. It requires that all tree expressions are constant even if we can decide which branch to choose, provided the constant value of *b*
```
| constVal (Call accC statT mode objRef mn pTs args) = None
| constVal (Methd C sig)   = None
| constVal (Body C s)      = None
| constVal (InsInitE s e)  = None
| constVal (Callee l e)    = None
```

lemma *constVal-Some-induct* [*consumes 1*, *case-names Lit UnOp BinOp CondL CondR*]:
  assumes *const*: *constVal e* = *Some v* **and**
    hyp-Lit: ⋀ *v*. *P* (*Lit v*) **and**
    hyp-UnOp: ⋀ *unop e'*. *P e'* ⟹ *P* (*UnOp unop e'*) **and**
    hyp-BinOp: ⋀ *binop e1 e2*. ⟦*P e1*; *P e2*⟧ ⟹ *P* (*BinOp binop e1 e2*) **and**
    hyp-CondL: ⋀ *b bv e1 e2*. ⟦*constVal b* = *Some bv*; *the-Bool bv*; *P b*; *P e1*⟧
      ⟹ *P* (*b? e1* : *e2*) **and**

```
    hyp-CondR: ⋀ b bv e1 e2. ⟦constVal b = Some bv; ¬the-Bool bv; P b; P e2⟧
                    ⟹ P (b? e1 : e2)
  shows P e
⟨proof⟩

lemma assignsE-const-simp: constVal e = Some v ⟹ 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 :: bool ⇒ expr ⇒ 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))

| assigns-if b (Super)      = UNIV — can never evaluate to Boolean
| assigns-if b (Acc v)      = (assignsV v)
| assigns-if b (v := e)     = (assignsE (Ass v e))
| assigns-if b (c? e1 : e2) = (assignsE c) ∪
                               (case (constVal c) of
                                  None    ⇒ (assigns-if b e1) ∩
                                             (assigns-if b e2)
                                | Some bv ⇒ (case the-Bool bv of
                                               True  ⇒ assigns-if b e1
                                             | False ⇒ assigns-if b e2))
| assigns-if b ({accC,statT,mode}objRef·mn({pTs}args))
          = assignsE ({accC,statT,mode}objRef·mn({pTs}args))
— Only dummy analysis for intermediate expressions Methd, Body, InsInitE and Callee
| assigns-if b (Methd C sig)   = {}
| assigns-if b (Body  C s)     = {}
```

| *assigns-if b* (*InsInitE s e*) = {}
| *assigns-if b* (*Callee l e*) = {}

lemma *assigns-if-const-b-simp*:
assumes *boolConst*: *constVal e* = *Some* (*Bool b*) (**is** *?Const b e*)
shows *assigns-if b e* = {} (**is** *?Ass b e*)
⟨*proof*⟩

lemma *assigns-if-const-not-b-simp*:
assumes *boolConst*: *constVal e* = *Some* (*Bool b*) (**is** *?Const b e*)
shows *assigns-if* (¬*b*) *e* = *UNIV* (**is** *?Ass b e*)
⟨*proof*⟩

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

definition
union-ts :: (*'a,'b*) *tables* ⇒ (*'a,'b*) *tables* ⇒ (*'a,'b*) *tables* (‹- ⇒∪ -› [*67*,*67*] *65*)
where *A* ⇒∪ *B* = (λ *k*. *A k* ∪ *B k*)

definition
intersect-ts :: (*'a,'b*) *tables* ⇒ (*'a,'b*) *tables* ⇒ (*'a,'b*) *tables* (‹- ⇒∩ -› [*72*,*72*] *71*)
where *A* ⇒∩ *B* = (λ*k*. *A k* ∩ *B k*)

definition
all-union-ts :: (*'a,'b*) *tables* ⇒ *'b set* ⇒ (*'a,'b*) *tables* (**infixl** ‹⇒∪∀› *40*)
where (*A* ⇒∪∀ *B*) = (λ *k*. *A k* ∪ *B*)

Binary union of tables

lemma *union-ts-iff* [*simp*]: (*c* ∈ (*A* ⇒∪ *B*) *k*) = (*c* ∈ *A k* ∨ *c* ∈ *B k*)
⟨*proof*⟩

lemma *union-tsI1* [*elim?*]: *c* ∈ *A k* ⟹ *c* ∈ (*A* ⇒∪ *B*) *k*
⟨*proof*⟩

lemma *union-tsI2* [*elim?*]: *c* ∈ *B k* ⟹ *c* ∈ (*A* ⇒∪ *B*) *k*
⟨*proof*⟩

lemma *union-tsCI* [*intro*!]: (*c* ∉ *B k* ⟹ *c* ∈ *A k*) ⟹ *c* ∈ (*A* ⇒∪ *B*) *k*
⟨*proof*⟩

lemma *union-tsE* [*elim*!]:
⟦*c* ∈ (*A* ⇒∪ *B*) *k*; (*c* ∈ *A k* ⟹ *P*); (*c* ∈ *B k* ⟹ *P*)⟧ ⟹ *P*
⟨*proof*⟩

Binary intersection of tables

lemma *intersect-ts-iff* [*simp*]: *c* ∈ (*A* ⇒∩ *B*) *k* = (*c* ∈ *A k* ∧ *c* ∈ *B k*)
⟨*proof*⟩

lemma *intersect-tsI* [*intro*!]: ⟦*c* ∈ *A k*; *c* ∈ *B k*⟧ ⟹ *c* ∈ (*A* ⇒∩ *B*) *k*
⟨*proof*⟩

lemma *intersect-tsD1*: *c* ∈ (*A* ⇒∩ *B*) *k* ⟹ *c* ∈ *A k*
⟨*proof*⟩

lemma *intersect-tsD2*: *c* ∈ (*A* ⇒∩ *B*) *k* ⟹ *c* ∈ *B k*
⟨*proof*⟩

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

All-Union of tables and set

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

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

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

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

lemma *all-union-tsE* [*elim*!]:
 $[\![c \in (A \Rightarrow\cup_\forall\ B)\ k;\ (c \in A\ k \Longrightarrow P);\ (c \in B \Longrightarrow P)]\!] \Longrightarrow P$
  ⟨*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* ⇒ (*'a*,*'b*) *tables* ⇒ (*'a*,*'b*) *tables*
  where *rmlab k A* = (λ*x*. *if x*=*k then UNIV else A x*)

definition
  range-inter-ts :: (*'a*,*'b*) *tables* ⇒ *'b set* (‹⇒⋂ -› *80*)
  where $\Rightarrow\bigcap A = \{x\ |x.\ \forall\ k.\ x \in A\ k\}$

In *E*⊢ *B* »*t*» *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* ⇒ *lname set* ⇒ *term* ⇒ *assigned* ⇒ *bool* (‹-⊢ - »-» -› [*65*,*65*,*65*,*65*] *71*)
where
  Skip: *Env*⊢ *B* »⟨*Skip*⟩» ⦇*nrm*=*B*,*brk*=λ *l*. *UNIV*⦈

| *Expr*: *Env*⊢ *B* »⟨*e*⟩» *A*
    ⟹
    Env⊢ *B* »⟨*Expr e*⟩» *A*
| *Lab*: ⟦*Env*⊢ *B* »⟨*c*⟩» *C*; *nrm A* = *nrm C* ∩ (*brk C*) *l*; *brk A* = *rmlab l* (*brk C*)⟧
    ⟹
    Env⊢ *B* »⟨*Break l*• *c*⟩» *A*

| *Comp*: ⟦*Env*⊢ *B* »⟨*c1*⟩» *C1*; *Env*⊢ *nrm C1* »⟨*c2*⟩» *C2*;

nrm A = nrm C2; *brk A* = (*brk C1*) ⇒∩ (*brk C2*)⟧
⟹
Env⊢ *B* »⟨*c1*;; *c2*⟩» *A*

| *If*: ⟦*Env*⊢ *B* »⟨*e*⟩» *E*;
Env⊢ (*B* ∪ *assigns-if True e*) »⟨*c1*⟩» *C1*;
Env⊢ (*B* ∪ *assigns-if False e*) »⟨*c2*⟩» *C2*;
nrm A = nrm C1 ∩ *nrm C2*;
brk A = brk C1 ⇒∩ *brk C2* ⟧
⟹
Env⊢ *B* »⟨*If*(*e*) *c1 Else c2*⟩» *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*⊢ *B* »⟨*e*⟩» *E* is just used to ensure the definite assignment in expression *e*. Notice the implicit analysis of a constant boolean expression *e* in this rule. For example, if *e* is constantly *True* then *assigns-if False e* = *UNIV* and therefor *nrm C2* = *UNIV*. So finally *nrm A* = *nrm C1*. For the break maps this trick workd too, because the trivial break map will map all labels to *UNIV*. In the example, if no break occurs in *c2* the break maps will trivially map to *UNIV* and if a break occurs it will map to *UNIV* too, because *assigns-if False e* = *UNIV*. So in the intersection of the break maps the path *c2* will have no contribution.

| *Loop*: ⟦*Env*⊢ *B* »⟨*e*⟩» *E*;
Env⊢ (*B* ∪ *assigns-if True e*) »⟨*c*⟩» *C*;
nrm A = nrm C ∩ (*B* ∪ *assigns-if False e*);
brk A = brk C⟧
⟹
Env⊢ *B* »⟨*l*· *While*(*e*) *c*⟩» *A*

— The *Loop* rule resembles some of the ideas of the *If* rule. For the *nrm A* the set *B* ∪ *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*: ⟦*jump=Ret* ⟶ *Result* ∈ *B*;
nrm A = *UNIV*;
brk A = (*case jump of*
Break l ⇒ λ *k*. *if k=l then B else UNIV*
| *Cont l* ⇒ λ *k*. *UNIV*
| *Ret* ⇒ λ *k*. *UNIV*)⟧
⟹
Env⊢ *B* »⟨*Jmp jump*⟩» *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 enshure that the result value is assigned.

| *Throw*: ⟦*Env*⊢ *B* »⟨*e*⟩» *E*; *nrm A* = *UNIV*; *brk A* = (λ *l*. *UNIV*)⟧
⟹ *Env*⊢ *B* »⟨*Throw e*⟩» *A*

| *Try*: ⟦*Env*⊢ *B* »⟨*c1*⟩» *C1*;
Env⦇*lcl* := (*lcl Env*)(*VName vn*↦*Class C*)⦈⊢ (*B* ∪ {*VName vn*}) »⟨*c2*⟩» *C2*;
nrm A = nrm C1 ∩ *nrm C2*;
brk A = brk C1 ⇒∩ *brk C2*⟧
⟹ *Env*⊢ *B* »⟨*Try c1 Catch*(*C vn*) *c2*⟩» *A*

| *Fin*: ⟦*Env*⊢ *B* »⟨*c1*⟩» *C1*;
Env⊢ *B* »⟨*c2*⟩» *C2*;
nrm A = nrm C1 ∪ *nrm C2*;
brk A = ((*brk C1*) ⇒∪∀ (*nrm C2*)) ⇒∩ (*brk C2*)⟧

$\Longrightarrow$
Env⊢ *B* »⟨*c1 Finally c2*⟩» *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* ($\Rightarrow\cup_\forall$). If *c2* exits with a break this break will appear in the overall result state. We don't know if *c1* completed normally or abruptly (maybe with an exception not only a break) so *c1* has no contribution to the break map following this path.

— Evaluation of expressions and the break sets of definite assignment: Thinking of a Java expression we assume that we can never have a break statement inside of a expression. So for all expressions the break sets could be set to the trivial one: $\lambda l.$ *UNIV*. But we can't trivially proof, that evaluating an expression will never result in a break, allthough Java expressions allready syntactically don't allow nested stetements in them. The reason are the nested class initialzation statements which are inserted by the evaluation rules. So to proof the absence of a break we need to ensure, that the initialization statements will never end up in a break. In a wellfromed 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*: *Env*⊢ *B* »⟨*Init C*⟩» (|*nrm*=*B*,*brk*=λ *l. UNIV*|)

— Wellformedness of a program will ensure, that every static initialiser is definetly 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 triggerred by the evaluation rules.

| *NewC*: *Env*⊢ *B* »⟨*NewC C*⟩» (|*nrm*=*B*,*brk*=λ *l. UNIV*|)

| *NewA*: *Env*⊢ *B* »⟨*e*⟩» *A*
$\Longrightarrow$
Env⊢ *B* »⟨*New T*[*e*]⟩» *A*

| *Cast*: *Env*⊢ *B* »⟨*e*⟩» *A*
$\Longrightarrow$
Env⊢ *B* »⟨*Cast T e*⟩» *A*

| *Inst*: *Env*⊢ *B* »⟨*e*⟩» *A*
$\Longrightarrow$
Env⊢ *B* »⟨*e InstOf T*⟩» *A*

| *Lit*: *Env*⊢ *B* »⟨*Lit v*⟩» (|*nrm*=*B*,*brk*=λ *l. UNIV*|)

| *UnOp*: *Env*⊢ *B* »⟨*e*⟩» *A*
$\Longrightarrow$
Env⊢ *B* »⟨*UnOp unop e*⟩» *A*

| *CondAnd*: [[*Env*⊢ *B* »⟨*e1*⟩» *E1*; *Env*⊢ (*B* ∪ *assigns-if True e1*) »⟨*e2*⟩» *E2*;
nrm A = *B* ∪ (*assigns-if True* (*BinOp CondAnd e1 e2*) ∩
assigns-if False (*BinOp CondAnd e1 e2*));
brk A = (λ *l. UNIV*)]]
$\Longrightarrow$
Env⊢ *B* »⟨*BinOp CondAnd e1 e2*⟩» *A*

| *CondOr*: [[*Env*⊢ *B* »⟨*e1*⟩» *E1*; *Env*⊢ (*B* ∪ *assigns-if False e1*) »⟨*e2*⟩» *E2*;
nrm A = *B* ∪ (*assigns-if True* (*BinOp CondOr e1 e2*) ∩
assigns-if False (*BinOp CondOr e1 e2*));

$brk\ A = (\lambda\ l.\ UNIV)\ ]\!]$
$\Longrightarrow$
$Env\vdash\ B\ »\langle BinOp\ CondOr\ e1\ e2\rangle»\ A$

$|\ BinOp$: $[\![Env\vdash\ B\ »\langle e1\rangle»\ E1;\ Env\vdash\ nrm\ E1\ »\langle e2\rangle»\ A;$
$binop \neq CondAnd;\ binop \neq CondOr]\!]$
$\Longrightarrow$
$Env\vdash\ B\ »\langle BinOp\ binop\ e1\ e2\rangle»\ A$

$|\ Super$: $This \in B$
$\Longrightarrow$
$Env\vdash\ B\ »\langle Super\rangle»\ (\!|nrm{=}B,brk{=}\lambda\ l.\ UNIV|\!)$

$|\ AccLVar$: $[\![vn \in B;$
$nrm\ A = B;\ brk\ A = (\lambda\ k.\ UNIV)]\!]$
$\Longrightarrow$
$Env\vdash\ B\ »\langle Acc\ (LVar\ vn)\rangle»\ A$
— To properly access a local variable we have to test the definite assignment here. The variable must occur in the set B

$|\ Acc$: $[\![\forall\ vn.\ v \neq LVar\ vn;$
$Env\vdash\ B\ »\langle v\rangle»\ A]\!]$
$\Longrightarrow$
$Env\vdash\ B\ »\langle Acc\ v\rangle»\ A$

$|\ AssLVar$: $[\![Env\vdash\ B\ »\langle e\rangle»\ E;\ nrm\ A = nrm\ E \cup \{vn\};\ brk\ A = brk\ E]\!]$
$\Longrightarrow$
$Env\vdash\ B\ »\langle (LVar\ vn) := e\rangle»\ A$

$|\ Ass$: $[\![\forall\ vn.\ v \neq LVar\ vn;\ Env\vdash\ B\ »\langle v\rangle»\ V;\ Env\vdash\ nrm\ V\ »\langle e\rangle»\ A]\!]$
$\Longrightarrow$
$Env\vdash\ B\ »\langle v := e\rangle»\ A$

$|\ CondBool$: $[\![Env\vdash(c\ ?\ e1 : e2)::-(PrimT\ Boolean);$
$Env\vdash\ B\ »\langle c\rangle»\ C;$
$Env\vdash\ (B \cup assigns\text{-}if\ True\ \ c)\ »\langle e1\rangle»\ E1;$
$Env\vdash\ (B \cup assigns\text{-}if\ False\ c)\ »\langle e2\rangle»\ E2;$
$nrm\ A = B \cup (assigns\text{-}if\ True\ \ (c\ ?\ e1 : e2) \cap$
$assigns\text{-}if\ False\ (c\ ?\ e1 : e2));$
$brk\ A = (\lambda\ l.\ UNIV)]\!]$
$\Longrightarrow$
$Env\vdash\ B\ »\langle c\ ?\ e1 : e2\rangle»\ A$

$|\ Cond$: $[\![\neg\ Env\vdash(c\ ?\ e1 : e2)::-(PrimT\ Boolean);$
$Env\vdash\ B\ »\langle c\rangle»\ C;$
$Env\vdash\ (B \cup assigns\text{-}if\ True\ \ c)\ »\langle e1\rangle»\ E1;$
$Env\vdash\ (B \cup assigns\text{-}if\ False\ c)\ »\langle e2\rangle»\ E2;$
$nrm\ A = nrm\ E1 \cap nrm\ E2;\ brk\ A = (\lambda\ l.\ UNIV)]\!]$
$\Longrightarrow$
$Env\vdash\ B\ »\langle c\ ?\ e1 : e2\rangle»\ A$

$|\ Call$: $[\![Env\vdash\ B\ »\langle e\rangle»\ E;\ Env\vdash\ nrm\ E\ »\langle args\rangle»\ A]\!]$
$\Longrightarrow$
$Env\vdash\ B\ »\langle \{accC,statT,mode\}e{\cdot}mn(\{pTs\}args)\rangle»\ A$

— The interplay of *Call*, *Methd* and *Body*: Why rules for *Methd* and *Body* at all? Note that a Java source program will not include bare *Methd* or *Body* terms. These terms are just introduced during evaluation. So definite assignment of *Call* does not consider *Methd* or *Body* at all. So for definite assignment alone we could omit the rules for *Methd* and *Body*. But since evaluation of the method invocation is split up into three rules we must ensure that we have enough information about the call even in the *Body* term to make sure that we can proof type safety. Also we must be able transport this information from *Call* to *Methd* and then further to *Body* during evaluation to establish the definite assignment of *Methd* during evaluation of *Call*, and of *Body* during evaluation of *Methd*. This is necessary since definite assignment will be a precondition for each induction hypothesis coming out of the evaluation rules, and therefor we have to establish the definite assignment of the sub-evaluation during the type-safety proof. Note that well-typedness is also a precondition for type-safety and so we can omit some assertion that are already ensured by well-typedness.

| *Methd*: ⟦*methd* (*prg Env*) *D sig* = *Some m*;
Env⊢ *B* »⟨*Body* (*declclass m*) (*stmt* (*mbody* (*mthd m*)))⟩» *A*
⟧
⟹
Env⊢ *B* »⟨*Methd D sig*⟩» *A*

| *Body*: ⟦*Env*⊢ *B* »⟨*c*⟩» *C*; *jumpNestingOkS* {*Ret*} *c*; *Result* ∈ *nrm C*;
nrm A = *B*; *brk A* = (λ *l*. *UNIV*)⟧
⟹
Env⊢ *B* »⟨*Body D c*⟩» *A*

— Note that *A* is not correlated to *C*. If the body statement returns abruptly with return, evaluation of *Body* will absorb this return and complete normally. So we cannot trivially get the assigned variables of the body statement since it has not completed normally or with a break. If the body completes normally we guarantee that the result variable is set with this rule. But if the body completes abruptly with a return we can't guarantee that the result variable is set here, since definite assignment only talks about normal completion and breaks. So for a return the *Jump* rule ensures that the result variable is set and then this information must be carried over to the *Body* rule by the conformance predicate of the state.

| *LVar*: *Env*⊢ *B* »⟨*LVar vn*⟩» ⦇*nrm*=*B*, *brk*=λ *l*. *UNIV*⦈

| *FVar*: *Env*⊢ *B* »⟨*e*⟩» *A*
⟹
Env⊢ *B* »⟨{*accC*,*statDeclC*,*stat*}*e*..*fn*⟩» *A*

| *AVar*: ⟦*Env*⊢ *B* »⟨*e1*⟩» *E1*; *Env*⊢ *nrm E1* »⟨*e2*⟩» *A*⟧
⟹
Env⊢ *B* »⟨*e1*.[*e2*]⟩» *A*

| *Nil*: *Env*⊢ *B* »⟨[]::*expr list*⟩» ⦇*nrm*=*B*, *brk*=λ *l*. *UNIV*⦈

| *Cons*: ⟦*Env*⊢ *B* »⟨*e*::*expr*⟩» *E*; *Env*⊢ *nrm E* »⟨*es*⟩» *A*⟧
⟹
Env⊢ *B* »⟨*e*#*es*⟩» *A*

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

inductive-cases *da-elim-cases* [*cases set*]:
Env⊢ *B* »⟨*Skip*⟩» *A*
Env⊢ *B* »*In1r Skip*» *A*
Env⊢ *B* »⟨*Expr e*⟩» *A*
Env⊢ *B* »*In1r* (*Expr e*)» *A*
Env⊢ *B* »⟨*l*• *c*⟩» *A*
Env⊢ *B* »*In1r* (*l*• *c*)» *A*
Env⊢ *B* »⟨*c1*;; *c2*⟩» *A*
Env⊢ *B* »*In1r* (*c1*;; *c2*)» *A*
Env⊢ *B* »⟨*If*(*e*) *c1 Else c2*⟩» *A*
Env⊢ *B* »*In1r* (*If*(*e*) *c1 Else c2*)» *A*
Env⊢ *B* »⟨*l*• *While*(*e*) *c*⟩» *A*
Env⊢ *B* »*In1r* (*l*• *While*(*e*) *c*)» *A*

Env⊢ B »⟨Jmp jump⟩» A
Env⊢ B »In1r (Jmp jump)» A
Env⊢ B »⟨Throw e⟩» A
Env⊢ B »In1r (Throw e)» A
Env⊢ B »⟨Try c1 Catch(C vn) c2⟩» A
Env⊢ B »In1r (Try c1 Catch(C vn) c2)» A
Env⊢ B »⟨c1 Finally c2⟩» A
Env⊢ B »In1r (c1 Finally c2)» A
Env⊢ B »⟨Init C⟩» A
Env⊢ B »In1r (Init C)» A
Env⊢ B »⟨NewC C⟩» A
Env⊢ B »In1l (NewC C)» A
Env⊢ B »⟨New T[e]⟩» A
Env⊢ B »In1l (New T[e])» A
Env⊢ B »⟨Cast T e⟩» A
Env⊢ B »In1l (Cast T e)» A
Env⊢ B »⟨e InstOf T⟩» A
Env⊢ B »In1l (e InstOf T)» A
Env⊢ B »⟨Lit v⟩» A
Env⊢ B »In1l (Lit v)» A
Env⊢ B »⟨UnOp unop e⟩» A
Env⊢ B »In1l (UnOp unop e)» A
Env⊢ B »⟨BinOp binop e1 e2⟩» A
Env⊢ B »In1l (BinOp binop e1 e2)» A
Env⊢ B »⟨Super⟩» A
Env⊢ B »In1l (Super)» A
Env⊢ B »⟨Acc v⟩» A
Env⊢ B »In1l (Acc v)» A
Env⊢ B »⟨v := e⟩» A
Env⊢ B »In1l (v := e)» A
Env⊢ B »⟨c ? e1 : e2⟩» A
Env⊢ B »In1l (c ? e1 : e2)» A
Env⊢ B »⟨{accC,statT,mode}e·mn({pTs}args)⟩» A
Env⊢ B »In1l ({accC,statT,mode}e·mn({pTs}args))» A
Env⊢ B »⟨Methd C sig⟩» A
Env⊢ B »In1l (Methd C sig)» A
Env⊢ B »⟨Body D c⟩» A
Env⊢ B »In1l (Body D c)» A
Env⊢ B »⟨LVar vn⟩» A
Env⊢ B »In2 (LVar vn)» A
Env⊢ B »⟨{accC,statDeclC,stat}e..fn⟩» A
Env⊢ B »In2 ({accC,statDeclC,stat}e..fn)» A
Env⊢ B »⟨e1.[e2]⟩» A
Env⊢ B »In2 (e1.[e2])» A
Env⊢ B »⟨[]::expr list⟩» A
Env⊢ B »In3 ([]::expr list)» A
Env⊢ B »⟨e#es⟩» A
Env⊢ B »In3 (e#es)» A
declare *inj-term-sym-simps* [*simp del*]
declare *assigns-if.simps* [*simp*]
declare *split-paired-All* [*simp*] *split-paired-Ex* [*simp*]
⟨*ML*⟩

lemma *da-Skip*: A = (|nrm=B,brk=λ l. UNIV|) ⟹ Env⊢ B »⟨Skip⟩» A
⟨*proof*⟩

lemma *da-NewC*: A = (|nrm=B,brk=λ l. UNIV|) ⟹ Env⊢ B »⟨NewC C⟩» A

⟨*proof*⟩

lemma *da-Lit*: $A = (\!|nrm{=}B,brk{=}\lambda\ l.\ UNIV|\!) \Longrightarrow Env\vdash\ B\ »\langle Lit\ v\rangle»\ A$
⟨*proof*⟩

lemma *da-Super*: $[\![This \in B;A = (\!|nrm{=}B,brk{=}\lambda\ l.\ UNIV|\!)]\!] \Longrightarrow Env\vdash\ B\ »\langle Super\rangle»\ A$
⟨*proof*⟩

lemma *da-Init*: $A = (\!|nrm{=}B,brk{=}\lambda\ l.\ UNIV|\!) \Longrightarrow Env\vdash\ B\ »\langle Init\ C\rangle»\ A$
⟨*proof*⟩

lemma *assignsE-subseteq-assigns-ifs*:
assumes *boolEx*: $E\vdash e{::}{-}PrimT\ Boolean$ (**is** *?Boolean e*)
shows $assignsE\ e \subseteq assigns\text{-}if\ True\ e \cap assigns\text{-}if\ False\ e$ (**is** *?Incl e*)
⟨*proof*⟩

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

lemma *rmlab-same-label1* [*simp*]: $l{=}l' \Longrightarrow (rmlab\ l\ A)\ l' = UNIV$
⟨*proof*⟩

lemma *rmlab-other-label* [*simp*]: $l{\neq}l'\Longrightarrow (rmlab\ l\ A)\ l' = A\ l'$
⟨*proof*⟩

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

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

lemma *da-monotone*:
assumes *da*: $Env\vdash\ B\ »t»\ A$ **and**
$B \subseteq B'$ **and**
da': $Env\vdash\ B'\ »t»\ A'$
shows $(nrm\ A \subseteq nrm\ A') \wedge (\forall\ l.\ (brk\ A\ l \subseteq brk\ A'\ l))$
⟨*proof*⟩

lemma *da-weaken*:
assumes *da*: $Env\vdash\ B\ »t»\ A$ **and** $B \subseteq B'$
shows $\exists\ A'.\ Env \vdash B'\ »t»\ A'$
⟨*proof*⟩

corollary *da-weakenE* [*consumes 2*]:
assumes *da*: $Env\vdash\ B\ »t»\ A$ **and**
B': $B \subseteq B'$ **and**
ex-mono: $\bigwedge A'.\ [\![Env\vdash\ B'\ »t»\ A';\ nrm\ A \subseteq nrm\ A';$
$\bigwedge l.\ brk\ A\ l \subseteq brk\ A'\ l]\!] \Longrightarrow P$
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-fdecl :: *prog* ⇒ *pname* ⇒ *fdecl* ⇒ *bool*
where *wf-fdecl G P* = (λ(*fn*,*f*). *is-acc-type G P* (*type f*))

lemma *wf-fdecl-def2*: ⋀*fd*. *wf-fdecl G P fd* = *is-acc-type G P* (*type* (*snd fd*))
⟨*proof*⟩

well-formed method declarations

A method head is wellformed if:

- the signature and the method head agree in the number of parameters
- all types of the parameters are visible
- the result type is visible
- the parameter names are unique

definition
wf-mhead :: *prog* ⇒ *pname* ⇒ *sig* ⇒ *mhead* ⇒ *bool* **where**
wf-mhead G P = (λ *sig mh. length* (*parTs sig*) = *length* (*pars mh*) ∧
(∀ *T*∈*set* (*parTs sig*). *is-acc-type G P T*) ∧
is-acc-type G P (*resTy mh*) ∧
distinct (*pars mh*))

A method declaration is wellformed if:

- the method head is wellformed
- the names of the local variables are unique
- the types of the local variables must be accessible
- the local variables don't shadow the parameters
- the class of the method is defined
- the body statement is welltyped with respect to the modified environment of local names, were the local variables, the parameters the special result variable (Res) and This are assoziated with there types.

definition
callee-lcl :: *qtname* ⇒ *sig* ⇒ *methd* ⇒ *lenv* **where**
callee-lcl C sig m =
(λ*k*. (*case k of*
EName e
⇒ (*case e of*
VNam v
⇒((*table-of* (*lcls* (*mbody m*)))(*pars m* [↦] *parTs sig*)) *v*
| *Res* ⇒ *Some* (*resTy m*))
| *This* ⇒ *if is-static m then None else Some* (*Class C*)))

definition
parameters :: *methd* ⇒ *lname set* **where**
parameters m = *set* (*map* (*EName* ∘ *VNam*) (*pars m*)) ∪ (*if* (*static m*) *then* {} *else* {*This*})

definition
wf-mdecl :: *prog* ⇒ *qtname* ⇒ *mdecl* ⇒ *bool* **where**
wf-mdecl G C =
(λ(*sig*,*m*).
wf-mhead G (*pid C*) *sig* (*mhead m*) ∧
unique (*lcls* (*mbody m*)) ∧
(∀ (*vn*,*T*)∈*set* (*lcls* (*mbody m*)). *is-acc-type G* (*pid C*) *T*) ∧
(∀ *pn*∈*set* (*pars m*). *table-of* (*lcls* (*mbody m*)) *pn* = *None*) ∧
jumpNestingOkS {*Ret*} (*stmt* (*mbody m*)) ∧
is-class G C ∧
(|*prg*=*G*,*cls*=*C*,*lcl*=*callee-lcl C sig m*|)⊢(*stmt* (*mbody m*))::√ ∧
(∃ *A*. (|*prg*=*G*,*cls*=*C*,*lcl*=*callee-lcl C sig m*|)
⊢ *parameters m* »⟨*stmt* (*mbody m*)⟩» *A*
∧ *Result* ∈ *nrm A*))

lemma *callee-lcl-VNam-simp* [*simp*]:
callee-lcl C sig m (*EName* (*VNam v*))
= ((*table-of* (*lcls* (*mbody m*)))(*pars m* [↦] *parTs sig*)) *v*
⟨*proof*⟩

lemma *callee-lcl-Res-simp* [*simp*]:

callee-lcl C sig m (EName Res) = Some (resTy m)
⟨proof⟩

lemma callee-lcl-This-simp [simp]:
callee-lcl C sig m (This) = (if is-static m then None else Some (Class C))
⟨proof⟩

lemma callee-lcl-This-static-simp:
is-static m ⟹ callee-lcl C sig m (This) = None
⟨proof⟩

lemma callee-lcl-This-not-static-simp:
¬ is-static m ⟹ callee-lcl C sig m (This) = Some (Class C)
⟨proof⟩

lemma wf-mheadI:
⟦length (parTs sig) = length (pars m); ∀ T∈set (parTs sig). is-acc-type G P T;
  is-acc-type G P (resTy m); distinct (pars m)⟧ ⟹
  wf-mhead G P sig m
⟨proof⟩

lemma wf-mdeclI: ⟦
  wf-mhead G (pid C) sig (mhead m); unique (lcls (mbody m));
  (∀ pn∈set (pars m). table-of (lcls (mbody m)) pn = None);
  ∀ (vn,T)∈set (lcls (mbody m)). is-acc-type G (pid C) T;
  jumpNestingOkS {Ret} (stmt (mbody m));
  is-class G C;
  ⦇prg=G,cls=C,lcl=callee-lcl C sig m⦈⊢(stmt (mbody m))::√;
  (∃ A. ⦇prg=G,cls=C,lcl=callee-lcl C sig m⦈ ⊢ parameters m »⟨stmt (mbody m)⟩» A
       ∧ Result ∈ nrm A)
  ⟧ ⟹
  wf-mdecl G C (sig,m)
⟨proof⟩

lemma wf-mdeclE [consumes 1]:
  ⟦wf-mdecl G C (sig,m);
    ⟦wf-mhead G (pid C) sig (mhead m); unique (lcls (mbody m));
     ∀ pn∈set (pars m). table-of (lcls (mbody m)) pn = None;
     ∀ (vn,T)∈set (lcls (mbody m)). is-acc-type G (pid C) T;
     jumpNestingOkS {Ret} (stmt (mbody m));
     is-class G C;
     ⦇prg=G,cls=C,lcl=callee-lcl C sig m⦈⊢(stmt (mbody m))::√;
   (∃ A. ⦇prg=G,cls=C,lcl=callee-lcl C sig m⦈⊢ parameters m »⟨stmt (mbody m)⟩» A
       ∧ Result ∈ nrm A)
    ⟧ ⟹ P
  ⟧ ⟹ P
⟨proof⟩

lemma wf-mdeclD1:
wf-mdecl G C (sig,m) ⟹
   wf-mhead G (pid C) sig (mhead m) ∧ unique (lcls (mbody m)) ∧
  (∀ pn∈set (pars m). table-of (lcls (mbody m)) pn = None) ∧
  (∀ (vn,T)∈set (lcls (mbody m)). is-acc-type G (pid C) T)
⟨proof⟩

lemma wf-mdecl-bodyD:
wf-mdecl G C (sig,m) ⟹
 (∃ T. ⦇prg=G,cls=C,lcl=callee-lcl C sig m⦈⊢Body C (stmt (mbody m))::−T ∧

  $G\vdash T\preceq(resTy\ m))$
⟨*proof*⟩

lemma *rT-is-acc-type*:
  wf-mhead G P sig m ⟹ *is-acc-type G P* (*resTy m*)
⟨*proof*⟩

well-formed interface declarations

A interface declaration is wellformed if:

- the interface hierarchy is wellstructured
- there is no class with the same name
- the method heads are wellformed and not static and have Public access
- the methods are uniquely named
- all superinterfaces are accessible
- the result type of a method overriding a method of Object widens to the result type of the overridden method. Shadowing static methods is forbidden.
- the result type of a method overriding a set of methods defined in the superinterfaces widens to each of the corresponding result types

definition
  wf-idecl :: *prog* ⇒ *idecl* ⇒ *bool* **where**
 wf-idecl G =
   (λ(*I*,*i*).
     ws-idecl G I (*isuperIfs i*) ∧
     ¬*is-class G I* ∧
     (∀(*sig*,*mh*)∈*set* (*imethods i*). *wf-mhead G* (*pid I*) *sig mh* ∧
                                   ¬*is-static mh* ∧
                                   accmodi mh = *Public*) ∧
     unique (*imethods i*) ∧
     (∀ *J*∈*set* (*isuperIfs i*). *is-acc-iface G* (*pid I*) *J*) ∧
     (*table-of* (*imethods i*)
       hiding (*methd G Object*)
       under (λ *new old*. *accmodi old* ≠ *Private*)
       entails (λ*new old*. *G*⊢*resTy new*≼*resTy old* ∧
                         is-static new = *is-static old*)) ∧
     (*set-option* ∘ *table-of* (*imethods i*)
           hidings Un-tables((λ*J*.(*imethds G J*))‘*set* (*isuperIfs i*))
           entails (λ*new old*. *G*⊢*resTy new*≼*resTy old*)))

lemma *wf-idecl-mhead*: ⟦*wf-idecl G* (*I*,*i*); (*sig*,*mh*)∈*set* (*imethods i*)⟧ ⟹
  wf-mhead G (*pid I*) *sig mh* ∧ ¬*is-static mh* ∧ *accmodi mh* = *Public*
⟨*proof*⟩

lemma *wf-idecl-hidings*:
wf-idecl G (*I*, *i*) ⟹
  (λ*s*. *set-option* (*table-of* (*imethods i*) *s*))
  hidings Un-tables ((λ*J*. *imethds G J*) ‘ *set* (*isuperIfs i*))
  entails λ*new old*. *G*⊢*resTy new*≼*resTy old*

⟨*proof*⟩

lemma *wf-idecl-hiding*:
wf-idecl G (*I*, *i*) $\Longrightarrow$
(*table-of* (*imethods i*)
hiding (*methd G Object*)
under (λ *new old. accmodi old* $\neq$ *Private*)
entails (λ*new old. G*$\vdash$*resTy new*$\preceq$*resTy old* $\wedge$
is-static new = *is-static old*))
⟨*proof*⟩

lemma *wf-idecl-supD*:
⟦*wf-idecl G* (*I*,*i*); *J* $\in$ *set* (*isuperIfs i*)⟧
$\Longrightarrow$ *is-acc-iface G* (*pid I*) *J* $\wedge$ (*J*, *I*) $\notin$ (*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*:
⟦*t entails P*; *t k* = *Some x*⟧ ⟹ *P x*
⟨*proof*⟩

lemma *empty-entails*[*simp*]: *Map.empty entails P*
⟨*proof*⟩

definition
wf-cdecl :: *prog* ⇒ *cdecl* ⇒ *bool* **where**
wf-cdecl G =
(λ(*C*,*c*).
¬*is-iface G C* ∧
(∀ *I*∈*set* (*superIfs c*). *is-acc-iface G* (*pid C*) *I* ∧
(∀ *s*. ∀ *im* ∈ *imethds G I s*.
(∃ *cm* ∈ *methd G C s*: *G*⊢*resTy cm*≼*resTy im* ∧
¬ *is-static cm* ∧
accmodi im ≤ *accmodi cm*))) ∧
(∀ *f*∈*set* (*cfields c*). *wf-fdecl G* (*pid C*) *f*) ∧ *unique* (*cfields c*) ∧
(∀ *m*∈*set* (*methods c*). *wf-mdecl G C m*) ∧ *unique* (*methods c*) ∧
jumpNestingOkS {} (*init c*) ∧
(∃ *A*. ⦇*prg*=*G*,*cls*=*C*,*lcl*=*Map.empty*⦈⊢ {} »⟨*init c*⟩» *A*) ∧
⦇*prg*=*G*,*cls*=*C*,*lcl*=*Map.empty*⦈⊢(*init c*)::√ ∧ *ws-cdecl G C* (*super c*) ∧
(*C* ≠ *Object* ⟶
(*is-acc-class G* (*pid C*) (*super c*) ∧
(*table-of* (*map* (λ (*s*,*m*). (*s*,*C*,*m*)) (*methods c*))
entails (λ *new*. ∀ *old sig*.
(*G*,*sig*⊢*new overrides*$_S$ *old*
⟶ (*G*⊢*resTy new*≼*resTy old* ∧
accmodi old ≤ *accmodi new* ∧
¬*is-static old*)) ∧
(*G*,*sig*⊢*new hides old*
⟶ (*accmodi old* ≤ *accmodi new* ∧
is-static old))))
)))

lemma *wf-cdeclE* [*consumes 1*]:
⟦*wf-cdecl G* (*C*,*c*);
⟦¬*is-iface G C*;
(∀ *I*∈*set* (*superIfs c*). *is-acc-iface G* (*pid C*) *I* ∧
(∀ *s*. ∀ *im* ∈ *imethds G I s*.
(∃ *cm* ∈ *methd G C s*: *G*⊢*resTy cm*≼*resTy im* ∧
¬ *is-static cm* ∧
accmodi im ≤ *accmodi cm*)));
∀ *f*∈*set* (*cfields c*). *wf-fdecl G* (*pid C*) *f*; *unique* (*cfields c*);
∀ *m*∈*set* (*methods c*). *wf-mdecl G C m*; *unique* (*methods c*);
jumpNestingOkS {} (*init c*);
∃ *A*. ⦇*prg*=*G*,*cls*=*C*,*lcl*=*Map.empty*⦈⊢ {} »⟨*init c*⟩» *A*;
⦇*prg*=*G*,*cls*=*C*,*lcl*=*Map.empty*⦈⊢(*init c*)::√;
ws-cdecl G C (*super c*);
(*C* ≠ *Object* ⟶
(*is-acc-class G* (*pid C*) (*super c*) ∧
(*table-of* (*map* (λ (*s*,*m*). (*s*,*C*,*m*)) (*methods c*))
entails (λ *new*. ∀ *old sig*.
(*G*,*sig*⊢*new overrides*$_S$ *old*
⟶ (*G*⊢*resTy new*≼*resTy old* ∧
accmodi old ≤ *accmodi new* ∧
¬*is-static old*)) ∧

$(G,sig \vdash new\ hides\ old$
$\longrightarrow (accmodi\ old \leq accmodi\ new \wedge$
$is\text{-}static\ old))))$
$))]\!] \Longrightarrow P$
$]\!] \Longrightarrow P$
⟨*proof*⟩

lemma *wf-cdecl-unique*:
$wf\text{-}cdecl\ G\ (C,c) \Longrightarrow unique\ (cfields\ c) \wedge unique\ (methods\ c)$
⟨*proof*⟩

lemma *wf-cdecl-fdecl*:
$[\![wf\text{-}cdecl\ G\ (C,c);\ f \in set\ (cfields\ c)]\!] \Longrightarrow wf\text{-}fdecl\ G\ (pid\ C)\ f$
⟨*proof*⟩

lemma *wf-cdecl-mdecl*:
$[\![wf\text{-}cdecl\ G\ (C,c);\ m \in set\ (methods\ c)]\!] \Longrightarrow wf\text{-}mdecl\ G\ C\ m$
⟨*proof*⟩

lemma *wf-cdecl-impD*:
$[\![wf\text{-}cdecl\ G\ (C,c);\ I \in set\ (superIfs\ c)]\!]$
$\Longrightarrow is\text{-}acc\text{-}iface\ G\ (pid\ C)\ I \wedge$
$(\forall s.\ \forall im \in imethds\ G\ I\ s.$
$(\exists cm \in methd\ G\ C\ s:\ G \vdash resTy\ cm \preceq resTy\ im \wedge \neg is\text{-}static\ cm \wedge$
$accmodi\ im \leq accmodi\ cm))$
⟨*proof*⟩

lemma *wf-cdecl-supD*:
$[\![wf\text{-}cdecl\ G\ (C,c);\ C \neq Object]\!] \Longrightarrow$
$is\text{-}acc\text{-}class\ G\ (pid\ C)\ (super\ c) \wedge (super\ c,C) \notin (subcls1\ G)^{+} \wedge$
$(table\text{-}of\ (map\ (\lambda\ (s,m).\ (s,C,m))\ (methods\ c))$
$entails\ (\lambda\ new.\ \forall\ \ old\ sig.$
$(G,sig \vdash new\ overrides_S\ old$
$\longrightarrow (G \vdash resTy\ new \preceq resTy\ old \wedge$
$accmodi\ old \leq accmodi\ new \wedge$
$\neg is\text{-}static\ old)) \wedge$
$(G,sig \vdash new\ hides\ old$
$\longrightarrow (accmodi\ old \leq accmodi\ new \wedge$
$is\text{-}static\ old))))$
⟨*proof*⟩

lemma *wf-cdecl-overrides-SomeD*:
$[\![wf\text{-}cdecl\ G\ (C,c);\ C \neq Object;\ table\text{-}of\ (methods\ c)\ sig = Some\ newM;$
$G,sig \vdash (C,newM)\ overrides_S\ old$
$]\!] \Longrightarrow\ G \vdash resTy\ newM \preceq resTy\ old \wedge$
$accmodi\ old \leq accmodi\ newM \wedge$
$\neg\ is\text{-}static\ old$
⟨*proof*⟩

lemma *wf-cdecl-hides-SomeD*:
$[\![wf\text{-}cdecl\ G\ (C,c);\ C \neq Object;\ table\text{-}of\ (methods\ c)\ sig = Some\ newM;$
$G,sig \vdash (C,newM)\ hides\ old$
$]\!] \Longrightarrow\ accmodi\ old \leq access\ newM \wedge$
$is\text{-}static\ old$
⟨*proof*⟩

lemma *wf-cdecl-wt-init*:
$wf\text{-}cdecl\ G\ (C,\ c) \Longrightarrow (\!|prg=G,cls=C,lcl=Map.empty|\!) \vdash init\ c{::}\surd$

⟨*proof*⟩

well-formed programs

A program declaration is wellformed if:

- the class ObjectC of Object is defined
- every method of Object has an access modifier distinct from Package. This is necessary since every interface automatically inherits from Object. We must know, that every time a Object method is "overriden" by an interface method this is also overriden by the class implementing the the interface (see *implement-dynmethd and class-mheadsD*)
- all standard Exceptions are defined
- all defined interfaces are wellformed
- all defined classes are wellformed

definition
wf-prog :: *prog* ⇒ *bool* **where**
wf-prog G = (*let is* = *ifaces G*; *cs* = *classes G in*
ObjectC ∈ *set cs* ∧
(∀ *m*∈*set Object-mdecls*. *accmodi m* ≠ *Package*) ∧
(∀ *xn*. *SXcptC xn* ∈ *set cs*) ∧
(∀ *i*∈*set is*. *wf-idecl G i*) ∧ *unique is* ∧
(∀ *c*∈*set cs*. *wf-cdecl G c*) ∧ *unique cs*)

lemma *wf-prog-idecl*: ⟦*iface G I* = *Some i*; *wf-prog G*⟧ ⟹ *wf-idecl G* (*I*,*i*)
⟨*proof*⟩

lemma *wf-prog-cdecl*: ⟦*class G C* = *Some c*; *wf-prog G*⟧ ⟹ *wf-cdecl G* (*C*,*c*)
⟨*proof*⟩

lemma *wf-prog-Object-mdecls*:
wf-prog G ⟹ (∀ *m*∈*set Object-mdecls*. *accmodi m* ≠ *Package*)
⟨*proof*⟩

lemma *wf-prog-acc-superD*:
⟦*wf-prog G*; *class G C* = *Some c*; *C* ≠ *Object* ⟧
⟹ *is-acc-class G* (*pid C*) (*super c*)
⟨*proof*⟩

lemma *wf-ws-prog* [*elim*!,*simp*]: *wf-prog G* ⟹ *ws-prog G*
⟨*proof*⟩

lemma *class-Object* [*simp*]:
wf-prog G ⟹
class G Object = *Some* (|*access*=*Public*,*cfields*=[],*methods*=*Object-mdecls*,
init=*Skip*,*super*=*undefined*,*superIfs*=[]|)
⟨*proof*⟩

lemma *methd-Object*[*simp*]: *wf-prog G* ⟹ *methd G Object* =
table-of (*map* (λ(*s*,*m*). (*s*, *Object*, *m*)) *Object-mdecls*)
⟨*proof*⟩

lemma *wf-prog-Object-methd*:
⟦*wf-prog G*; *methd G Object sig* = *Some m*⟧ ⟹ *accmodi m* ≠ *Package*
⟨*proof*⟩

lemma *wf-prog-Object-is-public*[*intro*]:
wf-prog G ⟹ *is-public G Object*
⟨*proof*⟩

lemma *class-SXcpt* [*simp*]:
wf-prog G ⟹
class G (*SXcpt xn*) = *Some* (|*access*=*Public*,*cfields*=[],*methods*=*SXcpt-mdecls*,
init=*Skip*,
super=*if xn* = *Throwable then Object*
else SXcpt Throwable,
superIfs=[]|)
⟨*proof*⟩

lemma *wf-ObjectC* [*simp*]:
wf-cdecl G ObjectC = (¬*is-iface G Object* ∧ *Ball* (*set Object-mdecls*)
(*wf-mdecl G Object*) ∧ *unique Object-mdecls*)
⟨*proof*⟩

lemma *Object-is-class* [*simp*,*elim*!]: *wf-prog G* ⟹ *is-class G Object*
⟨*proof*⟩

lemma *Object-is-acc-class* [*simp*,*elim*!]: *wf-prog G* ⟹ *is-acc-class G S Object*
⟨*proof*⟩

lemma *SXcpt-is-class* [*simp*,*elim*!]: *wf-prog G* ⟹ *is-class G* (*SXcpt xn*)
⟨*proof*⟩

lemma *SXcpt-is-acc-class* [*simp*,*elim*!]:
wf-prog G ⟹ *is-acc-class G S* (*SXcpt xn*)
⟨*proof*⟩

lemma *fields-Object* [*simp*]: *wf-prog G* ⟹ *DeclConcepts.fields G Object* = []
⟨*proof*⟩

lemma *accfield-Object* [*simp*]:
wf-prog G ⟹ *accfield G S Object* = *Map.empty*
⟨*proof*⟩

lemma *fields-Throwable* [*simp*]:
wf-prog G ⟹ *DeclConcepts.fields G* (*SXcpt Throwable*) = []
⟨*proof*⟩

lemma *fields-SXcpt* [*simp*]: *wf-prog G* ⟹ *DeclConcepts.fields G* (*SXcpt xn*) = []
⟨*proof*⟩

lemmas *widen-trans* = *ws-widen-trans* [*OF* - - *wf-ws-prog*, *elim*]
lemma *widen-trans2* [*elim*]: ⟦$G \vdash U \preceq T$; $G \vdash S \preceq U$; *wf-prog G*⟧ ⟹ $G \vdash S \preceq T$
⟨*proof*⟩

lemma *Xcpt-subcls-Throwable* [*simp*]:
wf-prog G ⟹ $G \vdash$*SXcpt xn*$\preceq_C$ *SXcpt Throwable*
⟨*proof*⟩

lemma *unique-fields*:
⟦*is-class G C*; *wf-prog G*⟧ ⟹ *unique* (*DeclConcepts.fields G C*)
⟨*proof*⟩

lemma *fields-mono*:

$[\![$*table-of* (*DeclConcepts.fields G C*) *fn* = *Some f*; $G \vdash D \preceq_C C$;
is-class G D; *wf-prog G*$]\!]$
$\Longrightarrow$ *table-of* (*DeclConcepts.fields G D*) *fn* = *Some f*
⟨*proof*⟩

lemma *fields-is-type* [*elim*]:
$[\![$*table-of* (*DeclConcepts.fields G C*) *m* = *Some f*; *wf-prog G*; *is-class G C*$]\!]$ $\Longrightarrow$
is-type G (*type f*)
⟨*proof*⟩

lemma *imethds-wf-mhead* [*rule-format* (*no-asm*)]:
$[\![$*m* ∈ *imethds G I sig*; *wf-prog G*; *is-iface G I*$]\!]$ $\Longrightarrow$
wf-mhead G (*pid* (*decliface m*)) *sig* (*mthd m*) ∧
¬ *is-static m* ∧ *accmodi m* = *Public*
⟨*proof*⟩

lemma *methd-wf-mdecl*:
$[\![$*methd G C sig* = *Some m*; *wf-prog G*; *class G C* = *Some y*$]\!]$ $\Longrightarrow$
$G \vdash C \preceq_C$ (*declclass m*) ∧ *is-class G* (*declclass m*) ∧
wf-mdecl G (*declclass m*) (*sig*,(*mthd m*))
⟨*proof*⟩

lemma *methd-rT-is-type*:
$[\![$*wf-prog G*;*methd G C sig* = *Some m*;
class G C = *Some y*$]\!]$
$\Longrightarrow$ *is-type G* (*resTy m*)
⟨*proof*⟩

lemma *accmethd-rT-is-type*:
$[\![$*wf-prog G*;*accmethd G S C sig* = *Some m*;
class G C = *Some y*$]\!]$
$\Longrightarrow$ *is-type G* (*resTy m*)
⟨*proof*⟩

lemma *methd-Object-SomeD*:
$[\![$*wf-prog G*;*methd G Object sig* = *Some m*$]\!]$
$\Longrightarrow$ *declclass m* = *Object*
⟨*proof*⟩

lemmas *iface-rec-induct'* = *iface-rec.induct* [*of %x y z. P x y*] **for** *P*

lemma *wf-imethdsD*:
$[\![$*im* ∈ *imethds G I sig*;*wf-prog G*; *is-iface G I*$]\!]$
$\Longrightarrow$ ¬*is-static im* ∧ *accmodi im* = *Public*
⟨*proof*⟩

lemma *wf-prog-hidesD*:
assumes *hides*: *G* ⊢*new hides old* **and** *wf*: *wf-prog G*
shows
accmodi old ≤ *accmodi new* ∧
is-static old
⟨*proof*⟩

Compare this lemma about static overriding $G \vdash$ *new overrides*$_S$ *old* with the definition of dynamic overriding $G \vdash$ *new overrides old*. Conforming result types and restrictions on the access modifiers of the old and the new method are not part of the predicate for static overriding. But they are enshured in a wellfromed program. Dynamic overriding has no restrictions on the access modifiers but enforces confrom result types as precondition. But with some efford we can guarantee the access modifier restriction for dynamic overriding, too. See lemma *wf-prog-dyn-override-prop*.

lemma *wf-prog-stat-overridesD*:
assumes *stat-override*: $G \vdash new\ overrides_S\ old$ **and** *wf*: *wf-prog G*
shows
$G \vdash resTy\ new \preceq resTy\ old\ \wedge$
$accmodi\ old \leq accmodi\ new\ \wedge$
$\neg\ is\text{-}static\ old$
⟨*proof*⟩

lemma *static-to-dynamic-overriding*:
assumes *stat-override*: $G \vdash new\ overrides_S\ old$ **and** *wf* : *wf-prog G*
shows $G \vdash new\ overrides\ old$
⟨*proof*⟩

lemma *non-Package-instance-method-inheritance*:
assumes *old-inheritable*: $G \vdash Method\ old\ inheritable\text{-}in\ (pid\ C)$ **and**
accmodi-old: $accmodi\ old \neq Package$ **and**
instance-method: $\neg\ is\text{-}static\ old$ **and**
subcls: $G \vdash C \prec_C declclass\ old$ **and**
old-declared: $G \vdash Method\ old\ declared\text{-}in\ (declclass\ old)$ **and**
wf: *wf-prog G*
shows $G \vdash Method\ old\ member\text{-}of\ C\ \vee$
$(\exists\ new.\ G \vdash\ new\ overrides_S\ old\ \wedge\ G \vdash Method\ new\ member\text{-}of\ C)$
⟨*proof*⟩

lemma *non-Package-instance-method-inheritance-cases*:
assumes *old-inheritable*: $G \vdash Method\ old\ inheritable\text{-}in\ (pid\ C)$ **and**
accmodi-old: $accmodi\ old \neq Package$ **and**
instance-method: $\neg\ is\text{-}static\ old$ **and**
subcls: $G \vdash C \prec_C declclass\ old$ **and**
old-declared: $G \vdash Method\ old\ declared\text{-}in\ (declclass\ old)$ **and**
wf: *wf-prog G*
obtains (*Inheritance*) $G \vdash Method\ old\ member\text{-}of\ C$
| (*Overriding*) *new* **where** $G \vdash\ new\ overrides_S\ old$ **and** $G \vdash Method\ new\ member\text{-}of\ C$
⟨*proof*⟩

lemma *dynamic-to-static-overriding*:
assumes *dyn-override*: $G \vdash\ new\ overrides\ old$ **and**
accmodi-old: $accmodi\ old \neq Package$ **and**
wf: *wf-prog G*
shows $G \vdash\ new\ overrides_S\ old$
⟨*proof*⟩

lemma *wf-prog-dyn-override-prop*:
assumes *dyn-override*: $G \vdash new\ overrides\ old$ **and**
wf: *wf-prog G*
shows $accmodi\ old \leq accmodi\ new$
⟨*proof*⟩

lemma *overrides-Package-old*:
assumes *dyn-override*: $G \vdash new\ overrides\ old$ **and**
accmodi-new: $accmodi\ new = Package$ **and**
wf: *wf-prog G*
shows $accmodi\ old = Package$
⟨*proof*⟩

lemma *dyn-override-Package*:
assumes *dyn-override*: $G \vdash$ *new overrides old* **and**
accmodi-old: *accmodi old* = *Package* **and**
accmodi-new: *accmodi new* = *Package* **and**
wf: *wf-prog G*
shows *pid* (*declclass old*) = *pid* (*declclass new*)
⟨*proof*⟩

lemma *dyn-override-Package-escape*:
assumes *dyn-override*: $G \vdash$ *new overrides old* **and**
accmodi-old: *accmodi old* = *Package* **and**
outside-pack: *pid* (*declclass old*) $\neq$ *pid* (*declclass new*) **and**
wf: *wf-prog G*
shows $\exists$ *inter*. $G \vdash$ *new overrides inter* $\wedge$ $G \vdash$ *inter overrides old* $\wedge$
pid (*declclass old*) = *pid* (*declclass inter*) $\wedge$
Protected $\leq$ *accmodi inter*
⟨*proof*⟩

lemmas *class-rec-induct′* = *class-rec.induct* [*of %x y z w*. *P x y*] **for** *P*

lemma *declclass-widen*[*rule-format*]:
wf-prog G
$\longrightarrow$ ($\forall$ *c m*. *class G C* = *Some c* $\longrightarrow$ *methd G C sig* = *Some m*
$\longrightarrow$ $G \vdash C \preceq_C$ *declclass m*) (**is** *?P G C*)
⟨*proof*⟩

lemma *declclass-methd-Object*:
⟦*wf-prog G*; *methd G Object sig* = *Some m*⟧ $\Longrightarrow$ *declclass m* = *Object*
⟨*proof*⟩

lemma *methd-declaredD*:
⟦*wf-prog G*; *is-class G C*;*methd G C sig* = *Some m*⟧
$\Longrightarrow$ $G \vdash$(*mdecl* (*sig*,*mthd m*)) *declared-in* (*declclass m*)
⟨*proof*⟩

lemma *methd-rec-Some-cases*:
assumes *methd-C*: *methd G C sig* = *Some m* **and**
ws: *ws-prog G* **and**
clsC: *class G C* = *Some c* **and**
neq-C-Obj: $C \neq$ *Object*
obtains (*NewMethod*) *table-of* (*map* (λ(*s*, *m*). (*s*, *C*, *m*)) (*methods c*)) *sig* = *Some m*
| (*InheritedMethod*) $G \vdash C$ *inherits* (*method sig m*) **and** *methd G* (*super c*) *sig* = *Some m*
⟨*proof*⟩

lemma *methd-member-of*:
assumes *wf*: *wf-prog G*
shows
⟦*is-class G C*; *methd G C sig* = *Some m*⟧ $\Longrightarrow$ $G \vdash$*Methd sig m member-of C*
(**is** *?Class C* $\Longrightarrow$ *?Method C* $\Longrightarrow$ *?MemberOf C*)
⟨*proof*⟩

lemma *current-methd*:
⟦*table-of* (*methods c*) *sig* = *Some new*;
ws-prog G; *class G C* = *Some c*; $C \neq$ *Object*;
methd G (*super c*) *sig* = *Some old*⟧
$\Longrightarrow$ *methd G C sig* = *Some* (*C*,*new*)
⟨*proof*⟩

lemma *wf-prog-staticD*:
assumes *wf*: *wf-prog G* **and**
clsC: *class G C = Some c* **and**
neq-C-Obj: $C \neq$ *Object* **and**
old: *methd G (super c) sig = Some old* **and**
accmodi-old: *Protected* $\leq$ *accmodi old* **and**
new: *table-of (methods c) sig = Some new*
shows *is-static new = is-static old*
⟨*proof*⟩

lemma *inheritable-instance-methd*:
assumes *subclseq-C-D*: $G \vdash C \preceq_C D$ **and**
is-cls-D: *is-class G D* **and**
wf: *wf-prog G* **and**
old: *methd G D sig = Some old* **and**
accmodi-old: *Protected* $\leq$ *accmodi old* **and**
not-static-old: ¬ *is-static old*
shows
∃ *new. methd G C sig = Some new* ∧
(*new = old* ∨ *G,sig*⊢*new overrides*$_S$ *old*)
(**is** (∃ *new.* (*?Constraint C new old*)))
⟨*proof*⟩

lemma *inheritable-instance-methd-cases*:
assumes *subclseq-C-D*: $G \vdash C \preceq_C D$ **and**
is-cls-D: *is-class G D* **and**
wf: *wf-prog G* **and**
old: *methd G D sig = Some old* **and**
accmodi-old: *Protected* $\leq$ *accmodi old* **and**
not-static-old: ¬ *is-static old*
obtains (*Inheritance*) *methd G C sig = Some old*
| (*Overriding*) *new* **where** *methd G C sig = Some new* **and** *G,sig*⊢*new overrides*$_S$ *old*
⟨*proof*⟩

lemma *inheritable-instance-methd-props*:
assumes *subclseq-C-D*: $G \vdash C \preceq_C D$ **and**
is-cls-D: *is-class G D* **and**
wf: *wf-prog G* **and**
old: *methd G D sig = Some old* **and**
accmodi-old: *Protected* $\leq$ *accmodi old* **and**
not-static-old: ¬ *is-static old*
shows
∃ *new. methd G C sig = Some new* ∧
¬ *is-static new* ∧ *G*⊢*resTy new*⪯*resTy old* ∧ *accmodi old* ≤*accmodi new*
(**is** (∃ *new.* (*?Constraint C new old*)))
⟨*proof*⟩

lemma *bexI′*: $x \in A \Longrightarrow P\ x \Longrightarrow \exists x{\in}A.\ P\ x$ ⟨*proof*⟩
lemma *ballE′*: $\forall x{\in}A.\ P\ x \Longrightarrow (x \notin A \Longrightarrow Q) \Longrightarrow (P\ x \Longrightarrow Q) \Longrightarrow Q$ ⟨*proof*⟩

lemma *subint-widen-imethds*:
assumes *irel*: *G*⊢*I*⪯*I J*
and *wf*: *wf-prog G*
and *is-iface*: *is-iface G J*
and *jm*: *jm* ∈ *imethds G J sig*
shows ∃ *im* ∈ *imethds G I sig. is-static im = is-static jm* ∧
accmodi im = accmodi jm ∧

G⊢resTy im≼resTy jm
⟨*proof*⟩

lemma *implmt1-methd*:
⋀*sig*. ⟦*G⊢C⇝1I*; *wf-prog G*; *im ∈ imethds G I sig*⟧ ⟹
∃ *cm* ∈*methd G C sig*: ¬ *is-static cm* ∧ ¬ *is-static im* ∧
G⊢resTy cm≼resTy im ∧
accmodi im = *Public* ∧ *accmodi cm* = *Public*
⟨*proof*⟩

lemma *implmt-methd* [*rule-format* (*no-asm*)]:
⟦*wf-prog G*; *G⊢C⇝I*⟧ ⟹ *is-iface G I* ⟶
(∀ *im* ∈*imethds G I sig*.
∃ *cm*∈*methd G C sig*: ¬*is-static cm* ∧ ¬ *is-static im* ∧
G⊢resTy cm≼resTy im ∧
accmodi im = *Public* ∧ *accmodi cm* = *Public*)
⟨*proof*⟩

lemma *mheadsD* [*rule-format* (*no-asm*)]:
emh ∈ *mheads G S t sig* ⟶ *wf-prog G* ⟶
(∃ *C D m*. *t* = *ClassT C* ∧ *declrefT emh* = *ClassT D* ∧
accmethd G S C sig = *Some m* ∧
(*declclass m* = *D*) ∧ *mhead* (*mthd m*) = (*mhd emh*)) ∨
(∃ *I*. *t* = *IfaceT I* ∧ ((∃ *im*. *im* ∈ *accimethds G* (*pid S*) *I sig* ∧
mthd im = *mhd emh*) ∨
(∃ *m*. *G⊢Iface I accessible-in* (*pid S*) ∧ *accmethd G S Object sig* = *Some m* ∧
accmodi m ≠ *Private* ∧
declrefT emh = *ClassT Object* ∧ *mhead* (*mthd m*) = *mhd emh*))) ∨
(∃ *T m*. *t* = *ArrayT T* ∧ *G⊢Array T accessible-in* (*pid S*) ∧
accmethd G S Object sig = *Some m* ∧ *accmodi m* ≠ *Private* ∧
declrefT emh = *ClassT Object* ∧ *mhead* (*mthd m*) = *mhd emh*)
⟨*proof*⟩

lemma *mheads-cases*:
assumes *emh* ∈ *mheads G S t sig* **and** *wf-prog G*
obtains (*Class-methd*) *C D m* **where**
t = *ClassT C declrefT emh* = *ClassT D accmethd G S C sig* = *Some m*
declclass m = *D mhead* (*mthd m*) = *mhd emh*
| (*Iface-methd*) *I im* **where** *t* = *IfaceT I*
im ∈ *accimethds G* (*pid S*) *I sig mthd im* = *mhd emh*
| (*Iface-Object-methd*) *I m* **where**
t = *IfaceT I G⊢Iface I accessible-in* (*pid S*)
accmethd G S Object sig = *Some m accmodi m* ≠ *Private*
declrefT emh = *ClassT Object mhead* (*mthd m*) = *mhd emh*
| (*Array-Object-methd*) *T m* **where**
t = *ArrayT T G⊢Array T accessible-in* (*pid S*)
accmethd G S Object sig = *Some m accmodi m* ≠ *Private*
declrefT emh = *ClassT Object mhead* (*mthd m*) = *mhd emh*
⟨*proof*⟩

lemma *declclassD*[*rule-format*]:
⟦*wf-prog G*;*class G C* = *Some c*; *methd G C sig* = *Some m*;

class G (declclass m) = Some d⟧
⟹ *table-of (methods d) sig = Some (mthd m)*
⟨*proof*⟩

lemma *dynmethd-Object*:
assumes *statM*: *methd G Object sig = Some statM* **and**
private: *accmodi statM = Private* **and**
is-cls-C: *is-class G C* **and**
wf: *wf-prog G*
shows *dynmethd G Object C sig = Some statM*
⟨*proof*⟩

lemma *wf-imethds-hiding-objmethdsD*:
assumes *old*: *methd G Object sig = Some old* **and**
is-if-I: *is-iface G I* **and**
wf: *wf-prog G* **and**
not-private: *accmodi old ≠ Private* **and**
new: *new ∈ imethds G I sig*
shows *G⊢resTy new≼resTy old ∧ is-static new = is-static old* (**is** *?P new*)
⟨*proof*⟩

Which dynamic classes are valid to look up a member of a distinct static type? We have to distinct class members (named static members in Java) from instance members. Class members are global to all Objects of a class, instance members are local to a single Object instance. If a member is equipped with the static modifier it is a class member, else it is an instance member. The following table gives an overview of the current framework. We assume to have a reference with static type statT and a dynamic class dynC. Between both of these types the widening relation holds *G⊢Class dynC≼statT*. Unfortunately this ordinary widening relation isn't enough to describe the valid lookup classes, since we must cope the special cases of arrays and interfaces,too. If we statically expect an array or inteface 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 con 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). I stems out of the fact, that Object indeed has no non private fields. So interfaces and arrays can actually have no fields at all and a field access would be senseless. (In Java interfaces are allowed to declare new fields but in current Bali not!). So there is no principal reason why we should not allow Objects to declare non private fields. Then we would get the following column:

statT field ——————— NullT / Iface Object Class dynC Array Object

primrec *valid-lookup-cls*:: *prog ⇒ ref-ty ⇒ qtname ⇒ bool ⇒ bool*
(‹-,- ⊢ - *valid′-lookup′-cls′-for* -› [*61,61,61,61*] *60*)
where
G,NullT ⊢ *dynC valid-lookup-cls-for static-membr = False*
| *G,IfaceT I* ⊢ *dynC valid-lookup-cls-for static-membr*
= (*if static-membr*
then dynC=Object
else G⊢Class dynC≼ Iface I)
| *G,ClassT C* ⊢ *dynC valid-lookup-cls-for static-membr = G⊢Class dynC≼ Class C*
| *G,ArrayT T* ⊢ *dynC valid-lookup-cls-for static-membr* = (*dynC=Object*)

lemma *valid-lookup-cls-is-class*:

assumes *dynC*: *G*,*statT* ⊢ *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*:
⟦*emh* ∈ *mheads G S statT sig*;
G,*statT* ⊢ *dynC valid-lookup-cls-for* (*is-static emh*);
isrtype G statT; *wf-prog G*
⟧ ⟹ ∃ *m* ∈ *dynlookup G statT dynC sig*:
is-static m=*is-static emh* ∧ *G*⊢*resTy m*≼*resTy emh*
⟨*proof*⟩
declare *split-paired-All* [*simp*] *split-paired-Ex* [*simp*]
⟨*ML*⟩

lemma *methd-declclass*:
⟦*class G C* = *Some c*; *wf-prog G*; *methd G C sig* = *Some m*⟧
⟹ *methd G* (*declclass m*) *sig* = *Some m*
⟨*proof*⟩

lemma *dynmethd-declclass*:
⟦*dynmethd G statC dynC sig* = *Some m*;
wf-prog G; *is-class G statC*
⟧ ⟹ *methd G* (*declclass m*) *sig* = *Some m*
⟨*proof*⟩

lemma *dynlookup-declC*:
⟦*dynlookup G statT dynC sig* = *Some m*; *wf-prog G*;
is-class G dynC;*isrtype G statT*
⟧ ⟹ *G*⊢*dynC* $\preceq_C$ (*declclass m*) ∧ *is-class G* (*declclass m*)
⟨*proof*⟩

lemma *dynlookup-Array-declclassD* [*simp*]:
⟦*dynlookup G* (*ArrayT T*) *Object sig* = *Some dm*;*wf-prog G*⟧
⟹ *declclass dm* = *Object*
⟨*proof*⟩

declare *split-paired-All* [*simp del*] *split-paired-Ex* [*simp del*]
⟨*ML*⟩

lemma *wt-is-type*: *E*,*dt*⊨*v*::*T* ⟹ *wf-prog* (*prg E*) ⟶
dt=*empty-dt* ⟶ (*case T of*
Inl T ⇒ *is-type* (*prg E*) *T*
| *Inr Ts* ⇒ *Ball* (*set Ts*) (*is-type* (*prg E*)))
⟨*proof*⟩
declare *split-paired-All* [*simp*] *split-paired-Ex* [*simp*]
⟨*ML*⟩

lemma *ty-expr-is-type*:
⟦*E*⊢*e*::−*T*; *wf-prog* (*prg E*)⟧ ⟹ *is-type* (*prg E*) *T*

⟨*proof*⟩
lemma *ty-var-is-type*:
⟦*E*⊢*v*::=*T*; *wf-prog* (*prg E*)⟧ ⟹ *is-type* (*prg E*) *T*
⟨*proof*⟩
lemma *ty-exprs-is-type*:
⟦*E*⊢*es*::≐*Ts*; *wf-prog* (*prg E*)⟧ ⟹ *Ball* (*set Ts*) (*is-type* (*prg E*))
⟨*proof*⟩

lemma *static-mheadsD*:
⟦ *emh* ∈ *mheads G S t sig*; *wf-prog G*; *E*⊢*e*::−*RefT t*; *prg E*=*G* ;
invmode (*mhd emh*) *e* ≠ *IntVir*
⟧ ⟹ ∃ *m*. ((∃ *C*. *t* = *ClassT C* ∧ *accmethd G S C sig* = *Some m*)
∨ (∀ *C*. *t* ≠ *ClassT C* ∧ *accmethd G S Object sig* = *Some m*)) ∧
declrefT emh = *ClassT* (*declclass m*) ∧ *mhead* (*mthd m*) = (*mhd emh*)
⟨*proof*⟩

lemma *wt-MethdI*:
⟦*methd G C sig* = *Some m*; *wf-prog G*;
class G C = *Some c*⟧ ⟹
∃ *T*. ⦇*prg*=*G*,*cls*=(*declclass m*),
lcl=*callee-lcl* (*declclass m*) *sig* (*mthd m*)⦈⊢ *Methd C sig*::−*T* ∧ *G*⊢*T*≼*resTy m*
⟨*proof*⟩

2 accessibility concerns

lemma *mheads-type-accessible*:
⟦*emh* ∈ *mheads G S T sig*; *wf-prog G*⟧
⟹ *G*⊢*RefT T accessible-in* (*pid S*)
⟨*proof*⟩

lemma *static-to-dynamic-accessible-from-aux*:
⟦*G*⊢*m of C accessible-from accC*;*wf-prog G*⟧
⟹ *G*⊢*m in C dyn-accessible-from accC*
⟨*proof*⟩

lemma *static-to-dynamic-accessible-from*:
assumes *stat-acc*: *G*⊢*m of statC accessible-from accC* **and**
subclseq: *G*⊢*dynC* $\preceq_C$ *statC* **and**
wf: *wf-prog G*
shows *G*⊢*m in dynC dyn-accessible-from accC*
⟨*proof*⟩

lemma *static-to-dynamic-accessible-from-static*:
assumes *stat-acc*: *G*⊢*m of statC accessible-from accC* **and**
static: *is-static m* **and**
wf: *wf-prog G*
shows *G*⊢*m in* (*declclass m*) *dyn-accessible-from accC*
⟨*proof*⟩

lemma *dynmethd-member-in*:
assumes *m*: *dynmethd G statC dynC sig* = *Some m* **and**
iscls-statC: *is-class G statC* **and**
wf: *wf-prog G*
shows *G*⊢*Methd sig m member-in dynC*
⟨*proof*⟩

lemma *dynmethd-access-prop*:
assumes *statM*: *methd G statC sig* = *Some statM* **and**

stat-acc: $G \vdash$ *Methd sig statM of statC accessible-from accC* **and**
dynM: *dynmethd G statC dynC sig* = *Some dynM* **and**
wf: *wf-prog G*
shows $G \vdash$ *Methd sig dynM in dynC dyn-accessible-from accC*
⟨*proof*⟩

lemma *implmt-methd-access*:
fixes *accC*::*qtname*
assumes *iface-methd*: *imethds G I sig* $\neq$ {} **and**
implements: $G \vdash$ *dynC* $\rightsquigarrow$ *I* **and**
isif-I: *is-iface G I* **and**
wf: *wf-prog G*
shows $\exists$ *dynM*. *methd G dynC sig* = *Some dynM* $\wedge$
$G \vdash$ *Methd sig dynM in dynC dyn-accessible-from accC*
⟨*proof*⟩

corollary *implmt-dynimethd-access*:
fixes *accC*::*qtname*
assumes *iface-methd*: *imethds G I sig* $\neq$ {} **and**
implements: $G \vdash$ *dynC* $\rightsquigarrow$ *I* **and**
isif-I: *is-iface G I* **and**
wf: *wf-prog G*
shows $\exists$ *dynM*. *dynimethd G I dynC sig* = *Some dynM* $\wedge$
$G \vdash$ *Methd sig dynM in dynC dyn-accessible-from accC*
⟨*proof*⟩

lemma *dynlookup-access-prop*:
assumes *emh*: *emh* $\in$ *mheads G accC statT sig* **and**
dynM: *dynlookup G statT dynC sig* = *Some dynM* **and**
dynC-prop: *G*,*statT* $\vdash$ *dynC valid-lookup-cls-for is-static emh* **and**
isT-statT: *isrtype G statT* **and**
wf: *wf-prog G*
shows $G \vdash$ *Methd sig dynM in dynC dyn-accessible-from accC*
⟨*proof*⟩

lemma *dynlookup-access*:
assumes *emh*: *emh* $\in$ *mheads G accC statT sig* **and**
dynC-prop: *G*,*statT* $\vdash$ *dynC valid-lookup-cls-for* (*is-static emh*) **and**
isT-statT: *isrtype G statT* **and**
wf: *wf-prog G*
shows $\exists$ *dynM*. *dynlookup G statT dynC sig* = *Some dynM* $\wedge$
$G \vdash$ *Methd sig dynM in dynC dyn-accessible-from accC*
⟨*proof*⟩

lemma *stat-overrides-Package-old*:
assumes *stat-override*: $G \vdash$ *new overrides*$_S$ *old* **and**
accmodi-new: *accmodi new* = *Package* **and**
wf: *wf-prog G*
shows *accmodi old* = *Package*
⟨*proof*⟩

Properties of dynamic accessibility

lemma *dyn-accessible-Private*:
assumes *dyn-acc*: $G \vdash$ *m in C dyn-accessible-from accC* **and**
priv: *accmodi m* = *Private*
shows *accC* = *declclass m*
⟨*proof*⟩

dyn-accessible-Package only works with the *wf-prog* assumption. Without it. it is easy to leaf the Package!

lemma *dyn-accessible-Package*:
$[\![G \vdash m\ in\ C\ dyn\text{-}accessible\text{-}from\ accC;\ accmodi\ m = Package;$
$wf\text{-}prog\ G]\!]$
$\Longrightarrow pid\ accC = pid\ (declclass\ m)$
⟨*proof*⟩

For fields we don't need the wellformedness of the program, since there is no overriding

lemma *dyn-accessible-field-Package*:
assumes *dyn-acc*: $G \vdash f\ in\ C\ dyn\text{-}accessible\text{-}from\ accC$ **and**
pack: $accmodi\ f = Package$ **and**
field: $is\text{-}field\ f$
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*: $G \vdash f\ in\ C\ dyn\text{-}accessible\text{-}from\ accC$ **and**
prot: $accmodi\ f = Protected$ **and**
field: $is\text{-}field\ f$ **and**
instance-field: $\neg\ is\text{-}static\ f$ **and**
outside: $pid\ (declclass\ f) \neq pid\ accC$
shows $G\vdash\ C \preceq_C accC$
⟨*proof*⟩

lemma *dyn-accessible-static-field-Protected*:
assumes *dyn-acc*: $G \vdash f\ in\ C\ dyn\text{-}accessible\text{-}from\ accC$ **and**
prot: $accmodi\ f = Protected$ **and**
field: $is\text{-}field\ f$ **and**
static-field: $is\text{-}static\ f$ **and**
outside: $pid\ (declclass\ f) \neq pid\ accC$
shows $G\vdash\ accC \preceq_C declclass\ f\ \wedge\ G\vdash C \preceq_C declclass\ f$
⟨*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 handeled via a general object abstraction
- the heap and the map for class objects are combined into a single table (*recall* (*loc*, *obj*) *table* × (*qtname*, *obj*) *table* ~= (*loc* + *qtname*, *obj*) *table*)

objects

datatype *obj-tag* = — tag for generic object
CInst qtname — class instance
| *Arr ty int* — array with component type and length
— | CStat qtname 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* × *qtname*
(*type*) *vn* <= (*type*) *fspec* + *int*
(*type*) *obj* <= (*type*) (|*tag*::*obj-tag*, *values*::*vn* ⇒ *val option*|)
(*type*) *obj* <= (*type*) (|*tag*::*obj-tag*, *values*::*vn* ⇒ *val option*,...::'*a*|)

definition
the-Arr :: *obj option* ⇒ *ty* × *int* × (*vn*, *val*) *table*
where *the-Arr obj* = (*SOME* (*T*,*k*,*t*). *obj* = *Some* (|*tag*=*Arr T k*,*values*=*t*|))

lemma *the-Arr-Arr* [*simp*]: *the-Arr* (*Some* (|*tag*=*Arr T k*,*values*=*cs*|)) = (*T*,*k*,*cs*)
⟨*proof*⟩

lemma *the-Arr-Arr1* [*simp*,*intro*,*dest*]:
⟦*tag obj* = *Arr T k*⟧ ⟹ *the-Arr* (*Some obj*) = (*T*,*k*,*values obj*)
⟨*proof*⟩

definition
$upd\text{-}obj$ :: $vn \Rightarrow val \Rightarrow obj \Rightarrow obj$
where $upd\text{-}obj\ n\ v = (\lambda obj.\ obj\ (\!|values{:=}(values\ obj)(n{\mapsto}v)|\!))$

lemma *upd-obj-def2* [*simp*]:
$upd\text{-}obj\ n\ v\ obj = obj\ (\!|values{:=}(values\ obj)(n{\mapsto}v)|\!)$
⟨*proof*⟩

definition
$obj\text{-}ty$ :: $obj \Rightarrow ty$ **where**
$obj\text{-}ty\ obj = (case\ tag\ obj\ of$
$CInst\ C \Rightarrow Class\ C$
$|\ Arr\ T\ k \Rightarrow T.[])$

lemma *obj-ty-eq* [*intro*!]: $obj\text{-}ty\ (\!|tag{=}oi,values{=}x|\!) = obj\text{-}ty\ (\!|tag{=}oi,values{=}y|\!)$
⟨*proof*⟩

lemma *obj-ty-eq1* [*intro*!,*dest*]:
$tag\ obj = tag\ obj' \Longrightarrow obj\text{-}ty\ obj = obj\text{-}ty\ obj'$
⟨*proof*⟩

lemma *obj-ty-cong* [*simp*]:
$obj\text{-}ty\ (obj\ (\!|values{:=}vs|\!)) = obj\text{-}ty\ obj$
⟨*proof*⟩

lemma *obj-ty-CInst* [*simp*]:
$obj\text{-}ty\ (\!|tag{=}CInst\ C,values{=}vs|\!) = Class\ C$
⟨*proof*⟩

lemma *obj-ty-CInst1* [*simp*,*intro*!,*dest*]:
$[\![tag\ obj = CInst\ C]\!] \Longrightarrow obj\text{-}ty\ obj = Class\ C$
⟨*proof*⟩

lemma *obj-ty-Arr* [*simp*]:
$obj\text{-}ty\ (\!|tag{=}Arr\ T\ i,values{=}vs|\!) = T.[]$
⟨*proof*⟩

lemma *obj-ty-Arr1* [*simp*,*intro*!,*dest*]:
$[\![tag\ obj = Arr\ T\ i]\!] \Longrightarrow obj\text{-}ty\ obj = T.[]$
⟨*proof*⟩

lemma *obj-ty-widenD*:
$G{\vdash}obj\text{-}ty\ obj{\preceq}RefT\ t \Longrightarrow (\exists\ C.\ tag\ obj = CInst\ C) \lor (\exists\ T\ k.\ tag\ obj = Arr\ T\ k)$
⟨*proof*⟩

definition
$obj\text{-}class$ :: $obj \Rightarrow qtname$ **where**
$obj\text{-}class\ obj = (case\ tag\ obj\ of$
$CInst\ C \Rightarrow C$
$|\ Arr\ T\ k \Rightarrow Object)$

lemma *obj-class-CInst* [*simp*]: $obj\text{-}class\ (\!|tag{=}CInst\ C,values{=}vs|\!) = C$
⟨*proof*⟩

lemma *obj-class-CInst1* [*simp*,*intro*!,*dest*]:
$tag\ obj = CInst\ C \Longrightarrow obj\text{-}class\ obj = C$

⟨*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* ⟹ *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* + *qtname* — generalized object reference

translations
(*type*) *oref* <= (*type*) *loc* + *qtname*

abbreviation (*input*)
Heap :: *loc* ⇒ *oref* **where** *Heap* ≡ *Inl*
abbreviation (*input*)
Stat :: *qtname* ⇒ *oref* **where** *Stat* ≡ *Inr*

definition
fields-table :: *prog* ⇒ *qtname* ⇒ (*fspec* ⇒ *field* ⇒ *bool*) ⇒ (*fspec*, *ty*) *table* **where**
fields-table G C P =
map-option type ∘ *table-of* (*filter* (*case-prod P*) (*DeclConcepts.fields G C*))

lemma *fields-table-SomeI*:
⟦*table-of* (*DeclConcepts.fields G C*) *n* = *Some f*; *P n f*⟧
⟹ *fields-table G C P n* = *Some* (*type f*)
⟨*proof*⟩

lemma *fields-table-SomeD*′: *fields-table G C P fn* = *Some T* ⟹
∃*f*. (*fn*,*f*)∈*set*(*DeclConcepts.fields G C*) ∧ *type f* = *T*
⟨*proof*⟩

lemma *fields-table-SomeD*:
⟦*fields-table G C P fn* = *Some T*; *unique* (*DeclConcepts.fields G C*)⟧ ⟹
∃*f*. *table-of* (*DeclConcepts.fields G C*) *fn* = *Some f* ∧ *type f* = *T*
⟨*proof*⟩

definition
in-bounds :: *int* ⇒ *int* ⇒ *bool* (‹(-/ *in*′*-bounds* -)› [*50*, *51*] *50*)
where *i in-bounds k* = (*0* ≤ *i* ∧ *i* < *k*)

definition
arr-comps :: ′*a* ⇒ *int* ⇒ *int* ⇒ ′*a option*
where *arr-comps T k* = (λ*i*. *if i in-bounds k then Some T else None*)

definition
var-tys :: *prog* ⇒ *obj-tag* ⇒ *oref* ⇒ (*vn*, *ty*) *table* **where**
var-tys G oi r =
(*case r of*
Heap a ⇒ (*case oi of*
CInst C ⇒ *fields-table G C* (λ*n f*. ¬*static f*) (+) *Map.empty*
| *Arr T k* ⇒ *Map.empty* (+) *arr-comps T k*)

  | *Stat C* ⇒ *fields-table G C* (λ*fn f*. *declclassf fn* = *C* ∧ *static f*)
        (+) *Map.empty*)

lemma *var-tys-Some-eq*:
 var-tys G oi r n = *Some T*
 = (*case r of*
   Inl a ⇒ (*case oi of*
     CInst C ⇒ (∃*nt*. *n* = *Inl nt* ∧ *fields-table G C* (λ*n f*.
       ¬*static f*) *nt* = *Some T*)
    | *Arr t k* ⇒ (∃ *i*. *n* = *Inr i* ∧ *i in-bounds k* ∧ *t* = *T*))
  | *Inr C* ⇒ (∃*nt*. *n* = *Inl nt* ∧
    fields-table G C (λ*fn f*. *declclassf fn* = *C* ∧ *static f*) *nt*
    = *Some T*))

⟨*proof*⟩

stores

type-synonym *globs* — global variables: heap and static variables
  = (*oref* , *obj*) *table*
type-synonym *heap*
  = (*loc* , *obj*) *table*

translations
 (*type*) *globs* <= (*type*) (*oref* , *obj*) *table*
 (*type*) *heap* <= (*type*) (*loc* , *obj*) *table*

datatype *st* =
  st globs locals

2 access

definition
 globs :: *st* ⇒ *globs*
 where *globs* = *case-st* (λ*g l*. *g*)

definition
 locals :: *st* ⇒ *locals*
 where *locals* = *case-st* (λ*g l*. *l*)

definition *heap* :: *st* ⇒ *heap* **where**
 heap s = *globs s* ∘ *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* ⇒ *val*
 where *val-this s* == *the* (*locals s This*)

abbreviation *lookup-obj* :: *st* ⇒ *val* ⇒ *obj*

where $lookup\text{-}obj\ s\ a' == the\ (heap\ s\ (the\text{-}Addr\ a'))$

3 memory allocation

definition
$new\text{-}Addr :: heap \Rightarrow loc\ option$ **where**
$new\text{-}Addr\ h = (if\ (\forall a.\ h\ a \neq None)\ then\ None\ else\ Some\ (SOME\ a.\ h\ a = None))$

lemma $new\text{-}AddrD$: $new\text{-}Addr\ h = Some\ a \Longrightarrow h\ a = None$
⟨*proof*⟩

lemma $new\text{-}AddrD2$: $new\text{-}Addr\ h = Some\ a \Longrightarrow \forall b.\ h\ b \neq None \longrightarrow b \neq a$
⟨*proof*⟩

lemma $new\text{-}Addr\text{-}SomeI$: $h\ a = None \Longrightarrow \exists b.\ new\text{-}Addr\ h = Some\ b \wedge h\ b = None$
⟨*proof*⟩

4 initialization

abbreviation $init\text{-}vals :: ('a,\ ty)\ table \Rightarrow ('a,\ val)\ table$
where $init\text{-}vals\ vs == map\text{-}option\ default\text{-}val \circ vs$

lemma $init\text{-}arr\text{-}comps\text{-}base$ $[simp]$: $init\text{-}vals\ (arr\text{-}comps\ T\ 0) = Map.empty$
⟨*proof*⟩

lemma $init\text{-}arr\text{-}comps\text{-}step$ $[simp]$:
$0 < j \Longrightarrow init\text{-}vals\ (arr\text{-}comps\ T\ j\ ) =$
$(init\text{-}vals\ (arr\text{-}comps\ T\ (j - 1)))(j - 1 \mapsto default\text{-}val\ T)$
⟨*proof*⟩

5 update

definition
$gupd :: oref \Rightarrow obj \Rightarrow st \Rightarrow st$ (‹$gupd'(\text{-}\mapsto\text{-}')$› $[10,\ 10]\ 1000$)
where $gupd\ r\ obj = case\text{-}st\ (\lambda g\ l.\ st\ (g(r \mapsto obj))\ l)$

definition
$lupd :: lname \Rightarrow val \Rightarrow st \Rightarrow st$ (‹$lupd'(\text{-}\mapsto\text{-}')$› $[10,\ 10]\ 1000$)
where $lupd\ vn\ v = case\text{-}st\ (\lambda g\ l.\ st\ g\ (l(vn \mapsto v)))$

definition
$upd\text{-}gobj :: oref \Rightarrow vn \Rightarrow val \Rightarrow st \Rightarrow st$
where $upd\text{-}gobj\ r\ n\ v = case\text{-}st\ (\lambda g\ l.\ st\ (chg\text{-}map\ (upd\text{-}obj\ n\ v)\ r\ g)\ l)$

definition
$set\text{-}locals :: locals \Rightarrow st \Rightarrow st$
where $set\text{-}locals\ l = case\text{-}st\ (\lambda g\ l'.\ st\ g\ l)$

definition
$init\text{-}obj :: prog \Rightarrow obj\text{-}tag \Rightarrow oref \Rightarrow st \Rightarrow st$
where $init\text{-}obj\ G\ oi\ r = gupd(r \mapsto (\!|tag{=}oi,\ values{=}init\text{-}vals\ (var\text{-}tys\ G\ oi\ r)|\!))$

abbreviation
$init\text{-}class\text{-}obj :: prog \Rightarrow qtname \Rightarrow st \Rightarrow st$
where $init\text{-}class\text{-}obj\ G\ C == init\text{-}obj\ G\ undefined\ (Inr\ C)$

lemma $gupd\text{-}def2$ $[simp]$: $gupd(r \mapsto obj)\ (st\ g\ l) = st\ (g(r \mapsto obj))\ l$
⟨*proof*⟩

lemma *lupd-def2* [*simp*]: *lupd*(*vn*↦*v*) (*st g l*) = *st g* (*l*(*vn*↦*v*))
⟨*proof*⟩

lemma *globs-gupd* [*simp*]: *globs* (*gupd*(*r*↦*obj*) *s*) = (*globs s*)(*r*↦*obj*)
⟨*proof*⟩

lemma *globs-lupd* [*simp*]: *globs* (*lupd*(*vn*↦*v*) *s*) = *globs s*
⟨*proof*⟩

lemma *locals-gupd* [*simp*]: *locals* (*gupd*(*r*↦*obj*) *s*) = *locals s*
⟨*proof*⟩

lemma *locals-lupd* [*simp*]: *locals* (*lupd*(*vn*↦*v*) *s*) = (*locals s*)(*vn*↦*v*)
⟨*proof*⟩

lemma *globs-upd-gobj-new* [*rule-format* (*no-asm*), *simp*]:
globs s r = *None* ⟶ *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*⟶ *globs* (*upd-gobj r n v s*) = (*globs s*)(*r*↦*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*↦*obj*)) *l*) = (*heap* (*st g l*))(*a*↦*obj*)
⟨*proof*⟩
lemma *heap-stat-upd* [*simp*]: *heap* (*st* (*g*(*Inr C*↦*obj*)) *l*) = *heap* (*st g l*)
⟨*proof*⟩
lemma *heap-local-upd* [*simp*]: *heap* (*st g* (*l*(*vn*↦*v*))) = *heap* (*st g l*)
⟨*proof*⟩

lemma *heap-gupd-Heap* [*simp*]: *heap* (*gupd*(*Heap a*↦*obj*) *s*) = (*heap s*)(*a*↦*obj*)
⟨*proof*⟩
lemma *heap-gupd-Stat* [*simp*]: *heap* (*gupd*(*Stat C*↦*obj*) *s*) = *heap s*
⟨*proof*⟩
lemma *heap-lupd* [*simp*]: *heap* (*lupd*(*vn*↦*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* ⇒ *xcpt*
  where *the-Xcpt* (*Xcpt x*) = *x*

primrec *the-Jump* :: *abrupt* => *jump*
  where *the-Jump* (*Jump j*) = *j*

primrec *the-Loc* :: *xcpt* ⇒ *loc*
  where *the-Loc* (*Loc a*) = *a*

primrec *the-Std* :: *xcpt* ⇒ *xname*
  where *the-Std* (*Std x*) = *x*

definition
  abrupt-if :: *bool* ⇒ *abopt* ⇒ *abopt* ⇒ *abopt*
  where *abrupt-if c x′ x* = (*if c* ∧ (*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* ≠ *None* ⟹ *abrupt-if True x y* ≠ *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* ≠ *None* ⟹ *abrupt-if c x y* = *y*
⟨*proof*⟩

lemma *split-abrupt-if*:
P (*abrupt-if c x′ x*) =
  ((*c* ∧ *x* = *None* ⟶ *P x′*) ∧ (¬ (*c* ∧ *x* = *None*) ⟶ *P x*))
⟨*proof*⟩

abbreviation *raise-if* :: *bool* ⇒ *xname* ⇒ *abopt* ⇒ *abopt*

where *raise-if c xn == abrupt-if c (Some (Xcpt (Std xn)))*

abbreviation *np* :: *val* ⇒ *abopt* ⇒ *abopt*
where *np v == raise-if (v = Null) NullPointer*

abbreviation *check-neg* :: *val* ⇒ *abopt* ⇒ *abopt*
where *check-neg i′ == raise-if (the-Intg i′<0) NegArrSize*

abbreviation *error-if* :: *bool* ⇒ *error* ⇒ *abopt* ⇒ *abopt*
where *error-if c e == abrupt-if c (Some (Error e))*

lemma *raise-if-None* [*simp*]: (*raise-if c x y = None*) = (¬*c* ∧ *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* ⟶ *b*) ∧ *y = None*)
⟨*proof*⟩

lemma *raise-if-SomeD* [*dest*!]:
raise-if c x y = Some z ⟹ *c* ∧ *z*=(*Xcpt* (*Std x*)) ∧ *y*=*None* ∨ (*y*=*Some z*)
⟨*proof*⟩

lemma *error-if-None* [*simp*]: (*error-if c e y = None*) = (¬*c* ∧ *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* ⟶ *b*) ∧ *y = None*)
⟨*proof*⟩

lemma *error-if-SomeD* [*dest*!]:
error-if c e y = Some z ⟹ *c* ∧ *z*=(*Error e*) ∧ *y*=*None* ∨ (*y*=*Some z*)
⟨*proof*⟩

definition
absorb :: *jump* ⇒ *abopt* ⇒ *abopt*
where *absorb j a* = (*if a*=*Some* (*Jump j*) *then None else a*)

lemma *absorb-SomeD* [*dest*!]: *absorb j a = Some x* ⟹ *a = Some x*
⟨*proof*⟩

lemma *absorb-same* [*simp*]: *absorb j* (*Some* (*Jump j*)) = *None*
⟨*proof*⟩

lemma *absorb-other* [*simp*]: *a* ≠ *Some* (*Jump j*) ⟹ *absorb j a = a*
⟨*proof*⟩

lemma *absorb-Some-NoneD*: *absorb j* (*Some abr*) = *None* ⟹ *abr = Jump j*
⟨*proof*⟩

lemma *absorb-Some-JumpD*: *absorb j s = Some* (*Jump j′*) ⟹ *j′*≠*j*
⟨*proof*⟩

full program state

type-synonym
state = abopt × *st* — state including abruption information

translations
  (*type*) *abopt* <= (*type*) *abrupt option*
  (*type*) *state* <= (*type*) *abopt* × *st*

abbreviation
  Norm :: *st* ⇒ *state*
  where *Norm s* == (*None*, *s*)

abbreviation (*input*)
  abrupt :: *state* ⇒ *abopt*
  where *abrupt* == *fst*

abbreviation (*input*)
  store :: *state* ⇒ *st*
  where *store* == *snd*

lemma *single-stateE*: ∀ *Z*. *Z* = (*s*::*state*) ⟹ *False*
⟨*proof*⟩

lemma *state-not-single*: *All* ((=) (*x*::*state*)) ⟹ *R*
⟨*proof*⟩

definition
  normal :: *state* ⇒ *bool*
  where *normal* = (λ*s*. *abrupt s* = *None*)

lemma *normal-def2* [*simp*]: *normal s* = (*abrupt s* = *None*)
⟨*proof*⟩

definition
  heap-free :: *nat* ⇒ *state* ⇒ *bool*
  where *heap-free n* = (λ*s*. *atleast-free* (*heap* (*store s*)) *n*)

lemma *heap-free-def2* [*simp*]: *heap-free n s* = *atleast-free* (*heap* (*store s*)) *n*
⟨*proof*⟩

6 update

definition
  abupd :: (*abopt* ⇒ *abopt*) ⇒ *state* ⇒ *state*
  where *abupd f* = *map-prod f id*

definition
  supd :: (*st* ⇒ *st*) ⇒ *state* ⇒ *state*
  where *supd* = *map-prod id*

lemma *abupd-def2* [*simp*]: *abupd f* (*x*,*s*) = (*f x*,*s*)
⟨*proof*⟩

lemma *abupd-abrupt-if-False* [*simp*]: ⋀ *s*. *abupd* (*abrupt-if False xo*) *s* = *s*
⟨*proof*⟩

lemma *supd-def2* [*simp*]: *supd f* (*x*,*s*) = (*x*,*f s*)
⟨*proof*⟩

lemma *supd-lupd* [*simp*]:
 ⋀ *s*. *supd* (*lupd vn v*) *s* = (*abrupt s*,*lupd vn v* (*store s*))
⟨*proof*⟩

lemma *supd-gupd* [*simp*]:
$\bigwedge$ *s. supd* (*gupd r obj*) *s* = (*abrupt s*,*gupd r obj* (*store s*))
⟨*proof*⟩

lemma *supd-init-obj* [*simp*]:
supd (*init-obj G oi r*) *s* = (*abrupt s*,*init-obj G oi r* (*store s*))
⟨*proof*⟩

lemma *abupd-store-invariant* [*simp*]: *store* (*abupd f s*) = *store s*
⟨*proof*⟩

lemma *supd-abrupt-invariant* [*simp*]: *abrupt* (*supd f s*) = *abrupt s*
⟨*proof*⟩

abbreviation *set-lvars* :: *locals* ⇒ *state* ⇒ *state*
where *set-lvars l* == *supd* (*set-locals l*)

abbreviation *restore-lvars* :: *state* ⇒ *state* ⇒ *state*
where *restore-lvars s′ s* == *set-lvars* (*locals* (*store s′*)) *s*

lemma *set-set-lvars* [*simp*]: $\bigwedge$ *s. set-lvars l* (*set-lvars l′ s*) = *set-lvars l s*
⟨*proof*⟩

lemma *set-lvars-id* [*simp*]: $\bigwedge$ *s. set-lvars* (*locals* (*store s*)) *s* = *s*
⟨*proof*⟩

initialisation test

definition
inited :: *qtname* ⇒ *globs* ⇒ *bool*
where *inited C g* = (*g* (*Stat C*) ≠ *None*)

definition
initd :: *qtname* ⇒ *state* ⇒ *bool*
where *initd C* = *inited C* ∘ *globs* ∘ *store*

lemma *not-inited-empty* [*simp*]: ¬*inited C Map.empty*
⟨*proof*⟩

lemma *inited-gupdate* [*simp*]: *inited C* (*g*(*r*↦*obj*)) = (*inited C g* ∨ *r* = *Stat C*)
⟨*proof*⟩

lemma *inited-init-class-obj* [*intro!*]: *inited C* (*globs* (*init-class-obj G C s*))
⟨*proof*⟩

lemma *not-initedD*: ¬ *inited C g* ⟹ *g* (*Stat C*) = *None*
⟨*proof*⟩

lemma *initedD*: *inited C g* ⟹ ∃ *obj. g* (*Stat C*) = *Some obj*
⟨*proof*⟩

lemma *initd-def2* [*simp*]: *initd C s* = *inited C* (*globs* (*store s*))
⟨*proof*⟩

error-free

definition
error-free :: *state* ⇒ *bool*

where *error-free s* = (¬ (∃ *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* ⟹ *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*]:
¬ (∃ *err*. *x*=*Error err*) ⟹ *error-free* ((*Some x*),*s*)
⟨*proof*⟩

lemma *error-free-abupd-absorb* [*simp*,*intro*]:
error-free s ⟹ *error-free* (*abupd* (*absorb j*) *s*)
⟨*proof*⟩

lemma *error-free-absorb* [*simp*,*intro*]:
error-free (*a*,*s*) ⟹ *error-free* (*absorb j a*, *s*)
⟨*proof*⟩

lemma *error-free-abrupt-if* [*simp*,*intro*]:
⟦*error-free s*; ¬ (∃ *err*. *x*=*Error err*)⟧
⟹ *error-free* (*abupd* (*abrupt-if p* (*Some x*)) *s*)
⟨*proof*⟩

lemma *error-free-abrupt-if1* [*simp*,*intro*]:
⟦*error-free* (*a*,*s*); ¬ (∃ *err*. *x*=*Error err*)⟧
⟹ *error-free* (*abrupt-if p* (*Some x*) *a*, *s*)
⟨*proof*⟩

lemma *error-free-abrupt-if-Xcpt* [*simp*,*intro*]:
error-free s
⟹ *error-free* (*abupd* (*abrupt-if p* (*Some* (*Xcpt x*))) *s*)
⟨*proof*⟩

lemma *error-free-abrupt-if-Xcpt1* [*simp*,*intro*]:
error-free (*a*,*s*)
⟹ *error-free* (*abrupt-if p* (*Some* (*Xcpt x*)) *a*, *s*)
⟨*proof*⟩

lemma *error-free-abrupt-if-Jump* [*simp*,*intro*]:
error-free s
⟹ *error-free* (*abupd* (*abrupt-if p* (*Some* (*Jump j*))) *s*)
⟨*proof*⟩

lemma *error-free-abrupt-if-Jump1* [*simp*,*intro*]:
error-free (*a*,*s*)
⟹ *error-free* (*abrupt-if p* (*Some* (*Jump j*)) *a*, *s*)
⟨*proof*⟩

lemma *error-free-raise-if* [*simp*,*intro*]:
error-free s $\Longrightarrow$ *error-free* (*abupd* (*raise-if p x*) *s*)
⟨*proof*⟩

lemma *error-free-raise-if1* [*simp*,*intro*]:
error-free (*a*,*s*) $\Longrightarrow$ *error-free* ((*raise-if p x a*), *s*)
⟨*proof*⟩

lemma *error-free-supd* [*simp*,*intro*]:
error-free s $\Longrightarrow$ *error-free* (*supd f s*)
⟨*proof*⟩

lemma *error-free-supd1* [*simp*,*intro*]:
error-free (*a*,*s*) $\Longrightarrow$ *error-free* (*a*,*f s*)
⟨*proof*⟩

lemma *error-free-set-lvars* [*simp*,*intro*]:
error-free s $\Longrightarrow$ *error-free* ((*set-lvars l*) *s*)
⟨*proof*⟩

lemma *error-free-set-locals* [*simp*,*intro*]:
error-free (*x*, *s*)
$\Longrightarrow$ *error-free* (*x*, *set-locals l s*′)
⟨*proof*⟩

end

Chapter 15

Eval

1 Operational evaluation (big-step) semantics of Java expressions and statements

theory *Eval* **imports** *State DeclConcepts* **begin**

improvements over Java Specification 1.0:

- dynamic method lookup does not need to consider the return type (cf.15.11.4.4)
- throw raises a NullPointer exception 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.

  ++ faciliates 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* × (*val* ⇒ *state* ⇒ *state*)
type-synonym *vals* = (*val*, *vvar*, *val list*) *sum3*
translations
(*type*) *vvar* <= (*type*) *val* × (*val* ⇒ *state* ⇒ *state*)
(*type*) *vals* <= (*type*) (*val*, *vvar*, *val list*) *sum3*

To avoid redundancy and to reduce the number of rules, there is only one evaluation rule for each syntactic term. This is also true for variables (e.g. see the rules below for *LVar*, *FVar* and *AVar*). So evaluation of a variable must capture both possible further uses: read (rule *Acc*) or write (rule *Ass*) to the variable. Therefor 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* (‹◇›)
where ◇ == *In1 Unit*

abbreviation (*input*)
val-inj-vals (‹⌊-⌋$_e$› *1000*)
where ⌊*e*⌋$_e$ == *In1 e*

abbreviation (*input*)
var-inj-vals (‹⌊-⌋$_v$› *1000*)
where ⌊*v*⌋$_v$ == *In2 v*

abbreviation (*input*)
lst-inj-vals (‹⌊-⌋$_l$› *1000*)
where ⌊*es*⌋$_l$ == *In3 es*

definition *undefined3* :: (*'al* + *'ar*, *'b*, *'c*) *sum3* ⇒ *vals* **where**
undefined3 = *case-sum3* (*In1* ∘ *case-sum* (λ*x*. *undefined*) (λ*x*. *Unit*))
(λ*x*. *In2 undefined*) (λ*x*. *In3 undefined*)

lemma [*simp*]: *undefined3* (*In1l x*) = *In1 undefined*
⟨*proof*⟩

lemma [*simp*]: *undefined3* (*In1r x*) = ◇
⟨*proof*⟩

lemma [*simp*]: *undefined3* (*In2 x*) = *In2 undefined*
⟨*proof*⟩

lemma [*simp*]: *undefined3* (*In3 x*) = *In3 undefined*
⟨*proof*⟩

exception throwing and catching

definition
throw :: *val* ⇒ *abopt* ⇒ *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*)
⟨*proof*⟩

definition
fits :: *prog* ⇒ *st* ⇒ *val* ⇒ *ty* ⇒ *bool* (‹-,-⊢- *fits* -›[*61*,*61*,*61*,*61*]*60*)
where *G*,*s*⊢*a' fits T* = ((∃ *rt*. *T*=*RefT rt*) ⟶ *a'*=*Null* ∨ *G*⊢*obj-ty*(*lookup-obj s a'*)≼*T*)

lemma *fits-Null* [*simp*]: *G*,*s*⊢*Null fits T*
⟨*proof*⟩

lemma *fits-Addr-RefT* [*simp*]:
G,*s*⊢*Addr a fits RefT t* = *G*⊢*obj-ty* (*the* (*heap s a*))≼*RefT t*
⟨*proof*⟩

lemma *fitsD*: ⋀*X*. *G*,*s*⊢*a' fits T* ⟹ (∃ *pt*. *T* = *PrimT pt*) ∨

$(\exists\, t.\ T = RefT\ t) \wedge a' = Null \vee$
$(\exists\, t.\ T = RefT\ t) \wedge a' \neq Null \wedge\ \ G\vdash obj\text{-}ty\ (lookup\text{-}obj\ s\ a')\preceq T$
⟨*proof*⟩

definition
catch :: *prog* ⇒ *state* ⇒ *qtname* ⇒ *bool* (‹-,-⊢*catch* -›[*61*,*61*,*61*]*60*) **where**
G,*s*⊢*catch C* = (∃ *xc*. *abrupt s*=*Some* (*Xcpt xc*) ∧
G,*store s*⊢*Addr* (*the-Loc xc*) *fits Class C*)

lemma *catch-Norm* [*simp*]: ¬*G*,*Norm s*⊢*catch tn*
⟨*proof*⟩

lemma *catch-XcptLoc* [*simp*]:
G,(*Some* (*Xcpt* (*Loc a*)),*s*)⊢*catch C* = *G*,*s*⊢*Addr a fits Class C*
⟨*proof*⟩

lemma *catch-Jump* [*simp*]: ¬*G*,(*Some* (*Jump j*),*s*)⊢*catch tn*
⟨*proof*⟩

lemma *catch-Error* [*simp*]: ¬*G*,(*Some* (*Error e*),*s*)⊢*catch tn*
⟨*proof*⟩

definition
new-xcpt-var :: *vname* ⇒ *state* ⇒ *state* **where**
new-xcpt-var vn = (λ(*x*,*s*). *Norm* (*lupd*(*VName vn*↦*Addr* (*the-Loc* (*the-Xcpt* (*the x*)))) *s*))

lemma *new-xcpt-var-def2* [*simp*]:
new-xcpt-var vn (*x*,*s*) =
Norm (*lupd*(*VName vn*↦*Addr* (*the-Loc* (*the-Xcpt* (*the x*)))) *s*)
⟨*proof*⟩

misc

definition
assign :: (′*a* ⇒ *state* ⇒ *state*) ⇒ ′*a* ⇒ *state* ⇒ *state* **where**
assign f v = (λ(*x*,*s*). *let* (*x*′,*s*′) = (*if x* = *None then f v else id*) (*x*,*s*)
in (*x*′,*if x*′ = *None then s*′ *else s*))

lemma *assign-Norm-Norm* [*simp*]:
f v (*Norm s*) = *Norm s*′ ⟹ *assign f v* (*Norm s*) = *Norm s*′
⟨*proof*⟩

lemma *assign-Norm-Some* [*simp*]:
⟦*abrupt* (*f v* (*Norm s*)) = *Some y*⟧
⟹ *assign f v* (*Norm s*) = (*Some y*,*s*)
⟨*proof*⟩

lemma *assign-Some* [*simp*]:
assign f v (*Some x*,*s*) = (*Some x*,*s*)
⟨*proof*⟩

lemma *assign-Some1* [*simp*]: ¬ *normal s* ⟹ *assign f v s* = *s*
⟨*proof*⟩

lemma *assign-supd* [*simp*]:
assign ($\lambda v.$ *supd* (*f v*)) *v* (*x*,*s*)
= (*x*, *if x* = *None then f v s else s*)
⟨*proof*⟩

lemma *assign-raise-if* [*simp*]:
assign (λv (*x*,*s*). ((*raise-if* (*b s v*) *xcpt*) *x*, *f v s*)) *v* (*x*, *s*) =
(*raise-if* (*b s v*) *xcpt x*, *if x*=*None* $\wedge$ $\neg$*b s v then f v s else s*)
⟨*proof*⟩

definition
init-comp-ty :: *ty* $\Rightarrow$ *stmt*
where *init-comp-ty T* = (*if* ($\exists$ *C*. *T* = *Class C*) *then Init* (*the-Class T*) *else Skip*)

lemma *init-comp-ty-PrimT* [*simp*]: *init-comp-ty* (*PrimT pt*) = *Skip*
⟨*proof*⟩

definition
invocation-class :: *inv-mode* $\Rightarrow$ *st* $\Rightarrow$ *val* $\Rightarrow$ *ref-ty* $\Rightarrow$ *qtname* **where**
invocation-class m s a′ statT =
(*case m of*
Static $\Rightarrow$ *if* ($\exists$ *statC*. *statT* = *ClassT statC*)
then the-Class (*RefT statT*)
else Object
| *SuperM* $\Rightarrow$ *the-Class* (*RefT statT*)
| *IntVir* $\Rightarrow$ *obj-class* (*lookup-obj s a′*))

definition
invocation-declclass :: *prog* $\Rightarrow$ *inv-mode* $\Rightarrow$ *st* $\Rightarrow$ *val* $\Rightarrow$ *ref-ty* $\Rightarrow$ *sig* $\Rightarrow$ *qtname* **where**
invocation-declclass G m s a′ statT sig =
declclass (*the* (*dynlookup G statT*
(*invocation-class m s a′ statT*)
sig))

lemma *invocation-class-IntVir* [*simp*]:
invocation-class IntVir s a′ statT = *obj-class* (*lookup-obj s a′*)
⟨*proof*⟩

lemma *dynclass-SuperM* [*simp*]:
invocation-class SuperM s a′ statT = *the-Class* (*RefT statT*)
⟨*proof*⟩

lemma *invocation-class-Static* [*simp*]:
invocation-class Static s a′ statT = (*if* ($\exists$ *statC*. *statT* = *ClassT statC*)
then the-Class (*RefT statT*)
else Object)
⟨*proof*⟩

definition
init-lvars :: *prog* $\Rightarrow$ *qtname* $\Rightarrow$ *sig* $\Rightarrow$ *inv-mode* $\Rightarrow$ *val* $\Rightarrow$ *val list* $\Rightarrow$ *state* $\Rightarrow$ *state*
where
init-lvars G C sig mode a′ pvs =
(λ(*x*,*s*).
let m = *mthd* (*the* (*methd G C sig*));
l = λ *k*.
(*case k of*

```
          EName e
            ⇒ (case e of
                 VNam v ⇒ (Map.empty ((pars m)[↦]pvs)) v
               | Res    ⇒ None)
         | This
            ⇒ (if mode=Static then None else Some a'))
    in set-lvars l (if mode = Static then x else np a' x,s))
```

lemma *init-lvars-def2*: — better suited for simplification

```
init-lvars G C sig mode a' pvs (x,s) =
  set-lvars
    (λ k.
       (case k of
          EName e
            ⇒ (case e of
                  VNam v
                  ⇒ (Map.empty ((pars (mthd (the (methd G C sig))))[↦]pvs)) v
               | Res ⇒ None)
        | This
            ⇒ (if mode=Static then None else Some a')))
    (if mode = Static then x else np a' x,s)
⟨proof⟩
```

definition

```
  body :: prog ⇒ qtname ⇒ sig ⇒ expr where
 body G C sig =
    (let m = the (methd G C sig)
     in Body (declclass m) (stmt (mbody (mthd m))))
```

lemma *body-def2*: — better suited for simplification

```
body G C sig = Body  (declclass (the (methd G C sig)))
                     (stmt (mbody (mthd (the (methd G C sig)))))
⟨proof⟩
```

variables

definition

```
  lvar :: lname ⇒ st ⇒ vvar
  where lvar vn s = (the (locals s vn), λv. supd (lupd(vn↦v)))
```

definition

```
  fvar :: qtname ⇒ bool ⇒ vname ⇒ val ⇒ state ⇒ vvar × state where
 fvar C stat fn a' s =
   (let (oref,xf) = if stat then (Stat C,id)
                    else (Heap (the-Addr a'),np a');
                  n = Inl (fn,C);
                  f = (λv. supd (upd-gobj oref n v))
    in ((the (values (the (globs (store s) oref)) n),f),abupd xf s))
```

definition

```
  avar :: prog ⇒ val ⇒ val ⇒ state ⇒ vvar × state where
  avar G i' a' s =
    (let   oref = Heap (the-Addr a');
              i = the-Intg i';
              n = Inr i;
       (T,k,cs) = the-Arr (globs (store s) oref);
              f = (λv (x,s). (raise-if (¬G,s⊢v fits T)
```

```
                                    ArrStore x
                          ,upd-gobj oref n v s))
   in ((the (cs n),f),abupd (raise-if (¬i in-bounds k) IndOutBound ∘ np a′) s))
```

lemma *fvar-def2*: — better suited for simplification

```
fvar C stat fn a′ s =
  ((the
     (values
      (the (globs (store s) (if stat then Stat C else Heap (the-Addr a′))))
      (Inl (fn,C)))
   ,(λv. supd (upd-gobj (if stat then Stat C else Heap (the-Addr a′))
                        (Inl (fn,C))
                        v)))
  ,abupd (if stat then id else np a′) s)
```

⟨*proof*⟩

lemma *avar-def2*: — better suited for simplification

```
avar G i′ a′ s =
  ((the ((snd(snd(the-Arr (globs (store s) (Heap (the-Addr a′))))))
         (Inr (the-Intg i′)))
   ,(λv (x,s′). (raise-if (¬G,s′⊢v fits (fst(the-Arr (globs (store s)
                                          (Heap (the-Addr a′))))))
                          ArrStore x
                 ,upd-gobj (Heap (the-Addr a′))
                            (Inr (the-Intg i′)) v s′)))
  ,abupd (raise-if (¬(the-Intg i′) in-bounds (fst(snd(the-Arr (globs (store s)
                   (Heap (the-Addr a′))))))) IndOutBound ∘ np a′)
         s)
```

⟨*proof*⟩

definition

```
check-field-access :: prog ⇒ qtname ⇒ qtname ⇒ vname ⇒ bool ⇒ val ⇒ state ⇒ 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 ⇒ obj-class (the (globs (store s) oref))
              | Stat C ⇒ C;
       f    = (the (table-of (DeclConcepts.fields G dynC) (fn,statDeclC)))
   in abupd
      (error-if (¬ G⊢Field fn (statDeclC,f) in dynC dyn-accessible-from accC)
                AccessViolation)
      s)
```

definition

```
check-method-access :: prog ⇒ qtname ⇒ ref-ty ⇒ inv-mode ⇒  sig ⇒ val ⇒ state ⇒ 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 (¬ G⊢Methd sig dynM in invC dyn-accessible-from accC)
                AccessViolation)
      s)
```

evaluation judgments

inductive

```
halloc :: [prog,state,obj-tag,loc,state]⇒bool (‹-⊢- −halloc ->-→ -›[61,61,61,61,61]60) for G::prog
```

where — allocating objects on the heap, cf. 12.5

Abrupt:
G⊢(Some x,s) −halloc oi≻undefined→ (Some x,s)

| *New*: ⟦new-Addr (heap s) = Some a;
(x,oi′) = (if atleast-free (heap s) (Suc (Suc 0)) then (None,oi)
else (Some (Xcpt (Loc a)),CInst (SXcpt OutOfMemory)))⟧
⟹
G⊢Norm s −halloc oi≻a→ (x,init-obj G oi′ (Heap a) s)

inductive *sxalloc* :: [prog,state,state]⇒bool (‹-⊢- −sxalloc→ -›[61,61,61]60) **for** G::prog
where — allocating exception objects for standard exceptions (other than OutOfMemory)

Norm: G⊢ Norm s −sxalloc→ Norm s

| *Jmp*: G⊢(Some (Jump j), s) −sxalloc→ (Some (Jump j), s)

| *Error*: G⊢(Some (Error e), s) −sxalloc→ (Some (Error e), s)

| *XcptL*: G⊢(Some (Xcpt (Loc a)),s) −sxalloc→ (Some (Xcpt (Loc a)),s)

| *SXcpt*: ⟦G⊢Norm s0 −halloc (CInst (SXcpt xn))≻a→ (x,s1)⟧ ⟹
G⊢(Some (Xcpt (Std xn)),s0) −sxalloc→ (Some (Xcpt (Loc a)),s1)

inductive
eval :: [prog,state,term,vals,state]⇒bool (‹-⊢- −-≻→ ′(-, -′)› [61,61,80,0,0]60)
and *exec* ::[prog,state,stmt ,state]⇒bool(‹-⊢- −-→ -› [61,61,65, 61]60)
and *evar* ::[prog,state,var ,vvar,state]⇒bool(‹-⊢- −-=≻-→ -›[61,61,90,61,61]60)
and *eval′*::[prog,state,expr ,val ,state]⇒bool(‹-⊢- −-−≻-→ -›[61,61,80,61,61]60)
and *evals*::[prog,state,expr list ,
val list ,state]⇒bool(‹-⊢- −-≐≻-→ -›[61,61,61,61,61]60)
for G::prog
where

G⊢s −c → s′ ≡ G⊢s −In1r c≻→ (◇, s′)
| G⊢s −e−≻v → s′ ≡ G⊢s −In1l e≻→ (In1 v, s′)
| G⊢s −e=≻vf→ s′ ≡ G⊢s −In2 e≻→ (In2 vf, s′)
| G⊢s −e≐≻v → s′ ≡ G⊢s −In3 e≻→ (In3 v, s′)

— propagation of abrupt completion

— cf. 14.1, 15.5
| *Abrupt*:
G⊢(Some xc,s) −t≻→ (undefined3 t, (Some xc, s))

— execution of statements

— cf. 14.5
| *Skip*: G⊢Norm s −Skip→ Norm s

— cf. 14.7
| *Expr*: ⟦G⊢Norm s0 −e−≻v→ s1⟧ ⟹
G⊢Norm s0 −Expr e→ s1

| *Lab*: ⟦G⊢Norm s0 −c → s1⟧ ⟹
G⊢Norm s0 −l• c→ abupd (absorb l) s1

— cf. 14.2
| *Comp*: ⟦*G*⊢*Norm s0* −*c1* → *s1*;
G⊢ *s1* −*c2* → *s2*⟧ ⟹
G⊢*Norm s0* −*c1*;; *c2*→ *s2*

— cf. 14.8.2
| *If*: ⟦*G*⊢*Norm s0* −*e*−≻*b*→ *s1*;
G⊢ *s1*−(*if the-Bool b then c1 else c2*)→ *s2*⟧ ⟹
G⊢*Norm s0* −*If*(*e*) *c1 Else c2* → *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*: ⟦*G*⊢*Norm s0* −*e*−≻*b*→ *s1*;
if the-Bool b
then (*G*⊢*s1* −*c*→ *s2* ∧
G⊢(*abupd* (*absorb* (*Cont l*)) *s2*) −*l*· *While*(*e*) *c*→ *s3*)
else s3 = *s1*⟧ ⟹
G⊢*Norm s0* −*l*· *While*(*e*) *c*→ *s3*

| *Jmp*: *G*⊢*Norm s* −*Jmp j*→ (*Some* (*Jump j*), *s*)

— cf. 14.16
| *Throw*: ⟦*G*⊢*Norm s0* −*e*−≻*a′*→ *s1*⟧ ⟹
G⊢*Norm s0* −*Throw e*→ *abupd* (*throw a′*) *s1*

— cf. 14.18.1
| *Try*: ⟦*G*⊢*Norm s0* −*c1*→ *s1*; *G*⊢*s1* −*sxalloc*→ *s2*;
if G,*s2*⊢*catch C then G*⊢*new-xcpt-var vn s2* −*c2*→ *s3 else s3* = *s2*⟧ ⟹
G⊢*Norm s0* −*Try c1 Catch*(*C vn*) *c2*→ *s3*

— cf. 14.18.2
| *Fin*: ⟦*G*⊢*Norm s0* −*c1*→ (*x1*,*s1*);
G⊢*Norm s1* −*c2*→ *s2*;
s3=(*if* (∃ *err*. *x1*=*Some* (*Error err*))
then (*x1*,*s1*)
else abupd (*abrupt-if* (*x1*≠*None*) *x1*) *s2*) ⟧
⟹
G⊢*Norm s0* −*c1 Finally c2*→ *s3*
— cf. 12.4.2, 8.5
| *Init*: ⟦*the* (*class G C*) = *c*;
if inited C (*globs s0*) *then s3* = *Norm s0*
else (*G*⊢*Norm* (*init-class-obj G C s0*)
−(*if C* = *Object then Skip else Init* (*super c*))→ *s1* ∧
G⊢*set-lvars Map.empty s1* −*init c*→ *s2* ∧ *s3* = *restore-lvars s1 s2*)⟧
⟹
G⊢*Norm s0* −*Init C*→ *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*: ⟦G⊢Norm s0 −Init C→ s1;
G⊢ s1 −halloc (CInst C)≻a→ s2⟧ ⟹
G⊢Norm s0 −NewC C−≻Addr a→ s2

— cf. 15.9.1, 12.4.1
| *NewA*: ⟦G⊢Norm s0 −init-comp-ty T→ s1; G⊢s1 −e−≻i′→ s2;
G⊢abupd (check-neg i′) s2 −halloc (Arr T (the-Intg i′))≻a→ s3⟧ ⟹
G⊢Norm s0 −New T[e]−≻Addr a→ s3

— cf. 15.15
| *Cast*: ⟦G⊢Norm s0 −e−≻v→ s1;
s2 = abupd (raise-if (¬G,store s1⊢v fits T) ClassCast) s1⟧ ⟹
G⊢Norm s0 −Cast T e−≻v→ s2

— cf. 15.19.2
| *Inst*: ⟦G⊢Norm s0 −e−≻v→ s1;
b = (v≠Null ∧ G,store s1⊢v fits RefT T)⟧ ⟹
G⊢Norm s0 −e InstOf T−≻Bool b→ s1

— cf. 15.7.1
| *Lit*: G⊢Norm s −Lit v−≻v→ Norm s

| *UnOp*: ⟦G⊢Norm s0 −e−≻v→ s1⟧
⟹ G⊢Norm s0 −UnOp unop e−≻(eval-unop unop v)→ s1

| *BinOp*: ⟦G⊢Norm s0 −e1−≻v1→ s1;
G⊢s1 −(if need-second-arg binop v1 then (In1l e2) else (In1r Skip))
≻→ (In1 v2, s2)
⟧
⟹ G⊢Norm s0 −BinOp binop e1 e2−≻(eval-binop binop v1 v2)→ s2

— cf. 15.10.2
| *Super*: G⊢Norm s −Super−≻val-this s→ Norm s

— cf. 15.2
| *Acc*: ⟦G⊢Norm s0 −va=≻(v,f)→ s1⟧ ⟹
G⊢Norm s0 −Acc va−≻v→ s1

— cf. 15.25.1
| *Ass*: ⟦G⊢Norm s0 −va=≻(w,f)→ s1;
G⊢ s1 −e−≻v → s2⟧ ⟹
G⊢Norm s0 −va:=e−≻v→ assign f v s2

— cf. 15.24
| *Cond*: ⟦G⊢Norm s0 −e0−≻b→ s1;
G⊢ s1 −(if the-Bool b then e1 else e2)−≻v→ s2⟧ ⟹
G⊢Norm s0 −e0 ? e1 : e2−≻v→ s2

— The interplay of *Call*, *Methd* and *Body*: Method invocation is split up into these three rules:

Call Calculates the target address and evaluates the arguments of the method, and then performs dynamic or static lookup of the method, corresponding to the call mode. Then the *Methd* rule is evaluated on the calculated declaration class of the method invocation.

Methd A syntactic bridge for the folded method body. It is used by the axiomatic semantics to add the proper hypothesis for recursive calls of the method.

Body An extra syntactic entity for the unfolded method body was introduced to properly trigger class initialisation. Without class initialisation we could just evaluate the body statement.

— cf. 15.11.4.1, 15.11.4.2, 15.11.4.4, 15.11.4.5
| *Call*:
$[\![G \vdash Norm\ s0\ -e-\succ a' \rightarrow\ s1;\ G \vdash s1\ -args \dot{=}\succ vs \rightarrow\ s2;$
$D = invocation\text{-}declclass\ G\ mode\ (store\ s2)\ a'\ statT\ (\!|name=mn,parTs=pTs|\!);$
$s3 = init\text{-}lvars\ G\ D\ (\!|name=mn,parTs=pTs|\!)\ mode\ a'\ vs\ s2;$
$s3' = check\text{-}method\text{-}access\ G\ accC\ statT\ mode\ (\!|name=mn,parTs=pTs|\!)\ a'\ s3;$
$G \vdash s3'\ -Methd\ D\ (\!|name=mn,parTs=pTs|\!)-\succ v \rightarrow\ s4]\!]$
$\Longrightarrow$
$G \vdash Norm\ s0\ -\{accC,statT,mode\}e \cdot mn(\{pTs\}args)-\succ v \rightarrow\ (restore\text{-}lvars\ s2\ s4)$
— 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.

| *Methd*: $[\![G \vdash Norm\ s0\ -body\ G\ D\ sig-\succ v \rightarrow\ s1]\!] \Longrightarrow$
$G \vdash Norm\ s0\ -Methd\ D\ sig-\succ v \rightarrow\ s1$

| *Body*: $[\![G \vdash Norm\ s0\ -Init\ D \rightarrow\ s1;\ G \vdash s1\ -c \rightarrow\ s2;$
$s3 = (if\ (\exists\ l.\ abrupt\ s2 = Some\ (Jump\ (Break\ l)) \lor$
$abrupt\ s2 = Some\ (Jump\ (Cont\ l)))$
$then\ abupd\ (\lambda\ x.\ Some\ (Error\ CrossMethodJump))\ s2$
$else\ s2)]\!] \Longrightarrow$
$G \vdash Norm\ s0\ -Body\ D\ c-\succ the\ (locals\ (store\ s2)\ Result)$
$\rightarrow abupd\ (absorb\ Ret)\ s3$
— 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*: $G \vdash Norm\ s\ -LVar\ vn = \succ lvar\ vn\ s \rightarrow\ Norm\ s$

— cf. 15.10.1, 12.4.1
| *FVar*: $[\![G \vdash Norm\ s0\ -Init\ statDeclC \rightarrow\ s1;\ G \vdash s1\ -e-\succ a \rightarrow\ s2;$
$(v,s2') = fvar\ statDeclC\ stat\ fn\ a\ s2;$
$s3 = check\text{-}field\text{-}access\ G\ accC\ statDeclC\ fn\ stat\ a\ s2']\!] \Longrightarrow$
$G \vdash Norm\ s0\ -\{accC,statDeclC,stat\}e..fn = \succ v \rightarrow\ s3$
— 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*: $[\![G \vdash\ Norm\ s0\ -e1-\succ a \rightarrow\ s1;\ G \vdash s1\ -e2-\succ i \rightarrow\ s2;$
$(v,s2') = avar\ G\ i\ a\ s2]\!] \Longrightarrow$
$G \vdash Norm\ s0\ -e1.[e2] = \succ v \rightarrow\ s2'$

— evaluation of expression lists

— cf. 15.11.4.2
| *Nil*:
$G \vdash Norm\ s0\ -[] \dot{=}\succ [] \rightarrow\ Norm\ s0$

— cf. 15.6.4
| *Cons*: $[\![G \vdash Norm\ s0\ -e\ -\succ\ v\ \rightarrow\ s1;$
$G \vdash\ \ s1\ -es \dot{=}\succ vs \rightarrow\ s2]\!] \Longrightarrow$
$G \vdash Norm\ s0\ -e\#es \dot{=}\succ v\#vs \rightarrow\ s2$

⟨*ML*⟩

declare *if-split* [*split del*] *if-split-asm* [*split del*]
option.split [*split del*] *option.split-asm* [*split del*]
inductive-cases *halloc-elim-cases*:
$G\vdash(Some\ xc,s)\ -halloc\ oi\succ a\rightarrow\ s'$
$G\vdash(Norm\quad s)\ -halloc\ oi\succ a\rightarrow\ s'$

inductive-cases *sxalloc-elim-cases*:
$G\vdash\ Norm\qquad s\ \ -sxalloc\rightarrow\ s'$
$G\vdash(Some\ (Jump\ j),s)\ -sxalloc\rightarrow\ s'$
$G\vdash(Some\ (Error\ e),s)\ -sxalloc\rightarrow\ s'$
$G\vdash(Some\ (Xcpt\ (Loc\ a\ )),s)\ -sxalloc\rightarrow\ s'$
$G\vdash(Some\ (Xcpt\ (Std\ xn)),s)\ -sxalloc\rightarrow\ s'$
inductive-cases *sxalloc-cases*: $G\vdash s\ -sxalloc\rightarrow\ s'$

lemma *sxalloc-elim-cases2*: $[\![G\vdash s\ -sxalloc\rightarrow\ s';$
$\bigwedge s.\quad [\![s' = Norm\ s]\!] \Longrightarrow P;$
$\bigwedge j\ s.\ [\![s' = (Some\ (Jump\ j),s)]\!] \Longrightarrow P;$
$\bigwedge e\ s.\ [\![s' = (Some\ (Error\ e),s)]\!] \Longrightarrow P;$
$\bigwedge a\ s.\ [\![s' = (Some\ (Xcpt\ (Loc\ a)),s)]\!] \Longrightarrow P$
$]\!] \Longrightarrow P$
⟨*proof*⟩

declare *not-None-eq* [*simp del*]
declare *split-paired-All* [*simp del*] *split-paired-Ex* [*simp del*]
⟨*ML*⟩

inductive-cases *eval-cases*: $G\vdash s\ -t\succ\rightarrow\ (v,\ s')$

inductive-cases *eval-elim-cases* [*cases set*]:
$G\vdash(Some\ xc,s)\ -t\qquad\succ\rightarrow\ (v,\ s')$
$G\vdash Norm\ s\ -In1r\ Skip\qquad\succ\rightarrow\ (x,\ s')$
$G\vdash Norm\ s\ -In1r\ (Jmp\ j)\qquad\succ\rightarrow\ (x,\ s')$
$G\vdash Norm\ s\ -In1r\ (l\bullet\ c)\qquad\succ\rightarrow\ (x,\ s')$
$G\vdash Norm\ s\ -In3\ \ ([])\qquad\succ\rightarrow\ (v,\ s')$
$G\vdash Norm\ s\ -In3\ \ (e\#es)\qquad\succ\rightarrow\ (v,\ s')$
$G\vdash Norm\ s\ -In1l\ (Lit\ w)\qquad\succ\rightarrow\ (v,\ s')$
$G\vdash Norm\ s\ -In1l\ (UnOp\ unop\ e)\qquad\succ\rightarrow\ (v,\ s')$
$G\vdash Norm\ s\ -In1l\ (BinOp\ binop\ e1\ e2)\qquad\succ\rightarrow\ (v,\ s')$
$G\vdash Norm\ s\ -In2\ \ (LVar\ vn)\qquad\succ\rightarrow\ (v,\ s')$
$G\vdash Norm\ s\ -In1l\ (Cast\ T\ e)\qquad\succ\rightarrow\ (v,\ s')$
$G\vdash Norm\ s\ -In1l\ (e\ InstOf\ T)\qquad\succ\rightarrow\ (v,\ s')$
$G\vdash Norm\ s\ -In1l\ (Super)\qquad\succ\rightarrow\ (v,\ s')$
$G\vdash Norm\ s\ -In1l\ (Acc\ va)\qquad\succ\rightarrow\ (v,\ s')$
$G\vdash Norm\ s\ -In1r\ (Expr\ e)\qquad\succ\rightarrow\ (x,\ s')$
$G\vdash Norm\ s\ -In1r\ (c1;;\ c2)\qquad\succ\rightarrow\ (x,\ s')$
$G\vdash Norm\ s\ -In1l\ (Methd\ C\ sig)\qquad\succ\rightarrow\ (x,\ s')$
$G\vdash Norm\ s\ -In1l\ (Body\ D\ c)\qquad\succ\rightarrow\ (x,\ s')$
$G\vdash Norm\ s\ -In1l\ (e0\ ?\ e1\ :\ e2)\qquad\succ\rightarrow\ (v,\ s')$
$G\vdash Norm\ s\ -In1r\ (If(e)\ c1\ Else\ c2)\qquad\succ\rightarrow\ (x,\ s')$
$G\vdash Norm\ s\ -In1r\ (l\bullet\ While(e)\ c)\qquad\succ\rightarrow\ (x,\ s')$
$G\vdash Norm\ s\ -In1r\ (c1\ Finally\ c2)\qquad\succ\rightarrow\ (x,\ s')$
$G\vdash Norm\ s\ -In1r\ (Throw\ e)\qquad\succ\rightarrow\ (x,\ s')$
$G\vdash Norm\ s\ -In1l\ (NewC\ C)\qquad\succ\rightarrow\ (v,\ s')$

$G \vdash Norm\ s\ -In1l\ (New\ T[e]) \succ\rightarrow (v,\ s')$
$G \vdash Norm\ s\ -In1l\ (Ass\ va\ e) \succ\rightarrow (v,\ s')$
$G \vdash Norm\ s\ -In1r\ (Try\ c1\ Catch(tn\ vn)\ c2) \succ\rightarrow (x,\ s')$
$G \vdash Norm\ s\ -In2\ (\{accC,statDeclC,stat\}e..fn) \succ\rightarrow (v,\ s')$
$G \vdash Norm\ s\ -In2\ (e1.[e2]) \succ\rightarrow (v,\ s')$
$G \vdash Norm\ s\ -In1l\ (\{accC,statT,mode\}e\cdot mn(\{pT\}p)) \succ\rightarrow (v,\ s')$
$G \vdash Norm\ s\ -In1r\ (Init\ C) \succ\rightarrow (x,\ s')$

declare *not-None-eq* [*simp*]
declare *split-paired-All* [*simp*] *split-paired-Ex* [*simp*]
⟨*ML*⟩
declare *if-split* [*split*] *if-split-asm* [*split*]
option.split [*split*] *option.split-asm* [*split*]

lemma *eval-Inj-elim*:
$G \vdash s\ -t \succ\rightarrow (w,s')$
$\Longrightarrow case\ t\ of$
$In1\ ec \Rightarrow (case\ ec\ of$
$Inl\ e \Rightarrow (\exists v.\ w = In1\ v)$
$|\ Inr\ c \Rightarrow w = \diamondsuit)$
$|\ In2\ e \Rightarrow (\exists v.\ w = In2\ v)$
$|\ In3\ e \Rightarrow (\exists v.\ w = In3\ v)$
⟨*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 *eval-expr-eq*: $G \vdash s\ -In1l\ t \succ\rightarrow (w,\ s') = (\exists v.\ w{=}In1\ v \wedge G \vdash s\ -t-\succ v \rightarrow s')$
⟨*proof*⟩

lemma *eval-var-eq*: $G \vdash s\ -In2\ t \succ\rightarrow (w,\ s') = (\exists vf.\ w{=}In2\ vf \wedge G \vdash s\ -t{=}\succ vf \rightarrow s')$
⟨*proof*⟩

lemma *eval-exprs-eq*: $G \vdash s\ -In3\ t \succ\rightarrow (w,\ s') = (\exists vs.\ w{=}In3\ vs \wedge G \vdash s\ -t\doteq\succ vs \rightarrow s')$
⟨*proof*⟩

lemma *eval-stmt-eq*: $G \vdash s\ -In1r\ t \succ\rightarrow (w,\ s') = (w{=}\diamondsuit \wedge G \vdash s\ -t \rightarrow s')$
⟨*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*:

$\bigwedge s\ s'.\ G\vdash s\ -t\succ\rightarrow\ (w,s') \Longrightarrow normal\ s' \longrightarrow normal\ s$
$\langle proof \rangle$

lemma *eval-no-abrupt*:
$G\vdash(x,s)\ -t\succ\rightarrow\ (w,Norm\ s') =$
$(x = None \wedge G\vdash Norm\ s\ -t\succ\rightarrow\ (w,Norm\ s'))$
$\langle proof \rangle$

$\langle ML \rangle$

lemma *eval-abrupt-lemma*:
$G\vdash s\ -t\succ\rightarrow\ (v,s') \Longrightarrow abrupt\ s{=}Some\ xc \longrightarrow s'{=}\ s \wedge v = undefined3\ t$
$\langle proof \rangle$

lemma *eval-abrupt*:
$G\vdash(Some\ xc,s)\ -t\succ\rightarrow\ (w,s') =$
$(s'{=}(Some\ xc,s) \wedge w{=}undefined3\ t \wedge$
$G\vdash(Some\ xc,s)\ -t\succ\rightarrow\ (undefined3\ t,(Some\ xc,s)))$
$\langle proof \rangle$

$\langle ML \rangle$

lemma *LitI*: $G\vdash s\ -Lit\ v-\succ(if\ normal\ s\ then\ v\ else\ undefined)\rightarrow\ s$
$\langle proof \rangle$

lemma *SkipI* [*intro*!]: $G\vdash s\ -Skip\rightarrow\ s$
$\langle proof \rangle$

lemma *ExprI*: $G\vdash s\ -e-\succ v\rightarrow\ s' \Longrightarrow G\vdash s\ -Expr\ e\rightarrow\ s'$
$\langle proof \rangle$

lemma *CompI*: $[\![G\vdash s\ -c1\rightarrow\ s1;\ G\vdash s1\ -c2\rightarrow\ s2]\!] \Longrightarrow G\vdash s\ -c1;;\ c2\rightarrow\ s2$
$\langle proof \rangle$

lemma *CondI*:
$\bigwedge s1.\ [\![G\vdash s\ -e-\succ b\rightarrow\ s1;\ G\vdash s1\ -(if\ the\text{-}Bool\ b\ then\ e1\ else\ e2)-\succ v\rightarrow\ s2]\!] \Longrightarrow$
$G\vdash s\ -e\ ?\ e1\ :\ e2-\succ(if\ normal\ s1\ then\ v\ else\ undefined)\rightarrow\ s2$
$\langle proof \rangle$

lemma *IfI*: $[\![G\vdash s\ -e-\succ v\rightarrow\ s1;\ G\vdash s1\ -(if\ the\text{-}Bool\ v\ then\ c1\ else\ c2)\rightarrow\ s2]\!]$
$\Longrightarrow G\vdash s\ -If(e)\ c1\ Else\ c2\rightarrow\ s2$
$\langle proof \rangle$

lemma *MethdI*: $G\vdash s\ -body\ G\ C\ sig-\succ v\rightarrow\ s'$
$\Longrightarrow G\vdash s\ -Methd\ C\ sig-\succ v\rightarrow\ s'$
$\langle proof \rangle$

lemma *eval-Call*:
$[\![G\vdash Norm\ s0\ -e-\succ a'\rightarrow\ s1;\ G\vdash s1\ -ps\doteq\succ pvs\rightarrow\ s2;$
$D = invocation\text{-}declclass\ G\ mode\ (store\ s2)\ a'\ statT\ (\!|name{=}mn,parTs{=}pTs|\!);$
$s3 = init\text{-}lvars\ G\ D\ (\!|name{=}mn,parTs{=}pTs|\!)\ mode\ a'\ pvs\ s2;$
$s3' = check\text{-}method\text{-}access\ G\ accC\ statT\ mode\ (\!|name{=}mn,parTs{=}pTs|\!)\ a'\ s3;$
$G\vdash s3'-Methd\ D\ (\!|name{=}mn,parTs{=}pTs|\!)-\succ\ v\rightarrow\ s4;$
$s4' = restore\text{-}lvars\ s2\ s4]\!] \Longrightarrow$
$G\vdash Norm\ s0\ -\{accC,statT,mode\}e{\cdot}mn(\{pTs\}ps)-\succ v\rightarrow\ s4'$
$\langle proof \rangle$

lemma *eval-Init*:

⟦*if inited C* (*globs s0*) *then s3* = *Norm s0*
else G⊢*Norm* (*init-class-obj G C s0*)
−(*if C* = *Object then Skip else Init* (*super* (*the* (*class G C*))))→ *s1* ∧
G⊢*set-lvars Map.empty s1* −(*init* (*the* (*class G C*)))→ *s2* ∧
s3 = *restore-lvars s1 s2*⟧ ⟹
G⊢*Norm s0* −*Init C*→ *s3*
⟨*proof*⟩

lemma *init-done*: *initd C s* ⟹ *G*⊢*s* −*Init C*→ *s*
⟨*proof*⟩

lemma *eval-StatRef*:
G⊢*s* −*StatRef rt*−≻(*if abrupt s*=*None then Null else undefined*)→ *s*
⟨*proof*⟩

lemma *SkipD* [*dest*!]: *G*⊢*s* −*Skip*→ *s'* ⟹ *s'* = *s*
⟨*proof*⟩

lemma *Skip-eq* [*simp*]: *G*⊢*s* −*Skip*→ *s'* = (*s* = *s'*)
⟨*proof*⟩

lemma *init-retains-locals* [*rule-format* (*no-asm*)]: *G*⊢*s* −*t*≻→ (*w*,*s'*) ⟹
(∀ *C*. *t*=*In1r* (*Init C*) ⟶ *locals* (*store s*) = *locals* (*store s'*))
⟨*proof*⟩

lemma *halloc-xcpt* [*dest*!]:
⋀*s'*. *G*⊢(*Some xc*,*s*) −*halloc oi*≻*a*→ *s'* ⟹ *s'*=(*Some xc*,*s*)
⟨*proof*⟩

lemma *eval-Methd*:
G⊢*s* −*In1l*(*body G C sig*)≻→ (*w*,*s'*)
⟹ *G*⊢*s* −*In1l*(*Methd C sig*)≻→ (*w*,*s'*)
⟨*proof*⟩

lemma *eval-Body*: ⟦*G*⊢*Norm s0* −*Init D*→ *s1*; *G*⊢*s1* −*c*→ *s2*;
res=*the* (*locals* (*store s2*) *Result*);
s3 = (*if* (∃ *l*. *abrupt s2* = *Some* (*Jump* (*Break l*)) ∨
abrupt s2 = *Some* (*Jump* (*Cont l*)))
then abupd (λ *x*. *Some* (*Error CrossMethodJump*)) *s2*
else s2);
s4=*abupd* (*absorb Ret*) *s3*⟧ ⟹
G⊢*Norm s0* −*Body D c*−≻*res*→*s4*
⟨*proof*⟩

lemma *eval-binop-arg2-indep*:
¬ *need-second-arg binop v1* ⟹ *eval-binop binop v1 x* = *eval-binop binop v1 y*
⟨*proof*⟩

lemma *eval-BinOp-arg2-indepI*:
assumes *eval-e1*: *G*⊢*Norm s0* −*e1*−≻*v1*→ *s1* **and**
no-need: ¬ *need-second-arg binop v1*
shows *G*⊢*Norm s0* −*BinOp binop e1 e2*−≻(*eval-binop binop v1 v2*)→ *s1*
(**is** *?EvalBinOp v2*)
⟨*proof*⟩

single valued

lemma *unique-halloc* [*rule-format* (*no-asm*)]:
$G\vdash s\ -halloc\ oi\succ a \rightarrow s' \Longrightarrow G\vdash s\ -halloc\ oi\succ a' \rightarrow s'' \longrightarrow a' = a \wedge s'' = s'$
⟨*proof*⟩

lemma *single-valued-halloc*:
$single\text{-}valued\ \{((s,oi),(a,s')).\ G\vdash s\ -halloc\ oi\succ a \rightarrow s'\}$
⟨*proof*⟩

lemma *unique-sxalloc* [*rule-format* (*no-asm*)]:
$G\vdash s\ -sxalloc\rightarrow s' \Longrightarrow G\vdash s\ -sxalloc\rightarrow s'' \longrightarrow s'' = s'$
⟨*proof*⟩

lemma *single-valued-sxalloc*: $single\text{-}valued\ \{(s,s').\ G\vdash s\ -sxalloc\rightarrow s'\}$
⟨*proof*⟩

lemma *split-pairD*: $(x,y) = p \Longrightarrow x = fst\ p\ \&\ y = snd\ p$
⟨*proof*⟩

lemma *unique-eval* [*rule-format* (*no-asm*)]:
$G\vdash s\ -t\succ\rightarrow (w,\ s') \Longrightarrow (\forall w'\ s''.\ G\vdash s\ -t\succ\rightarrow (w',\ s'') \longrightarrow w' = w \wedge s'' = s')$
⟨*proof*⟩

lemma *single-valued-eval*:
$single\text{-}valued\ \{((s,\ t),\ (v,\ s')).\ G\vdash s\ -t\succ\rightarrow (v,\ s')\}$
⟨*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*: $\bigwedge fd.\ wf\text{-}fdecl\ G\ P\ fd = is\text{-}acc\text{-}type\ G\ P\ (type\ (snd\ fd))$
⟨*proof*⟩

declare *wf-fdecl-def2* [*iff*]

type and expression names

datatype $tnam' = HasFoo' \mid Base' \mid Ext' \mid Main'$
datatype $vnam' = arr' \mid vee' \mid z' \mid e'$
datatype $label' = lab1'$

axiomatization
  $tnam' :: tnam' \Rightarrow tnam$ **and**
  $vnam' :: vnam' \Rightarrow vname$ **and**
  $label' :: label' \Rightarrow label$
where

  inj-tnam' [*simp*]: $\bigwedge x\ y.\ (tnam'\ x = tnam'\ y) = (x = y)$ **and**
  inj-vnam' [*simp*]: $\bigwedge x\ y.\ (vnam'\ x = vnam'\ y) = (x = y)$ **and**
  inj-label' [*simp*]: $\bigwedge x\ y.\ (label'\ x = label'\ y) = (x = y)$ **and**

  surj-tnam': $\bigwedge n.\ \exists m.\ n = tnam'\ m$ **and**
  surj-vnam': $\bigwedge n.\ \exists m.\ n = vnam'\ m$ **and**
  surj-label': $\bigwedge n.\ \exists m.\ n = label'\ m$

abbreviation
  $HasFoo :: qtname$ **where**
  $HasFoo == (\!|pid=java\text{-}lang,tid=TName\ (tnam'\ HasFoo')|\!)$

abbreviation
  $Base :: qtname$ **where**
  $Base == (\!|pid=java\text{-}lang,tid=TName\ (tnam'\ Base')|\!)$

abbreviation
  $Ext :: qtname$ **where**
  $Ext == (\!|pid=java\text{-}lang,tid=TName\ (tnam'\ Ext')|\!)$

abbreviation
  $Main :: qtname$ **where**
  $Main == (\!|pid=java\text{-}lang,tid=TName\ (tnam'\ Main')|\!)$

abbreviation
  $arr :: vname$ **where**
  $arr == (vnam'\ arr')$

abbreviation
vee :: *vname* **where**
vee == (*vnam′ vee′*)

abbreviation
z :: *vname* **where**
z == (*vnam′ z′*)

abbreviation
e :: *vname* **where**
e == (*vnam′ e′*)

abbreviation
lab1 :: *label* **where**
lab1 == *label′ lab1′*

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 :: *qtname* ⇒ *qtname* ⇒ *var*
where *arr-viewed-from accC C* = {*accC*,*Base*,*True*}*StatRef* (*ClassT C*)..*arr*

definition
BaseCl :: *class* **where**
BaseCl = (|*access*=*Public*,
cfields=[(*arr*, (|*access*=*Public*,*static*=*True* ,*type*=*PrimT Boolean*.[]|)),
(*vee*, (|*access*=*Public*,*static*=*False*,*type*=*Iface HasFoo* |))],
methods=[*Base-foo*],
init=*Expr*(*arr-viewed-from Base Base*
:=*New* (*PrimT Boolean*)[*Lit* (*Intg 2*)]),
super=*Object*,
superIfs=[*HasFoo*]|)

definition
ExtCl :: *class* **where**
ExtCl = (|*access*=*Public*,
cfields=[(*vee*, (|*access*=*Public*,*static*=*False*,*type*= *PrimT Integer*|))],
methods=[*Ext-foo*],
init=*Skip*,
super=*Base*,
superIfs=[]|)

definition
MainCl :: *class* **where**
MainCl = (|*access*=*Public*,
cfields=[],
methods=[],
init=*Skip*,
super=*Object*,
superIfs=[]|)

definition
HasFooInt :: *iface*
where *HasFooInt* = (|*access*=*Public*,*imethods*=[(*foo-sig*, *foo-mhead*)],*isuperIfs*=[]|)

definition
Ifaces ::*idecl list*
where *Ifaces* = [(*HasFoo*,*HasFooInt*)]

definition
Classes ::*cdecl list*
where *Classes* = [(*Base*,*BaseCl*),(*Ext*,*ExtCl*),(*Main*,*MainCl*)]@*standard-classes*

lemmas *table-classes-defs* =
Classes-def standard-classes-def ObjectC-def SXcptC-def

lemma *table-ifaces* [*simp*]: *table-of Ifaces* = *Map.empty*(*HasFoo*↦*HasFooInt*)
⟨*proof*⟩

lemma *table-classes-Object* [*simp*]:
table-of Classes Object = *Some* ⦇*access*=*Public*,*cfields*=[]
,*methods*=*Object-mdecls*
,*init*=*Skip*,*super*=*undefined*,*superIfs*=[]⦈
⟨*proof*⟩

lemma *table-classes-SXcpt* [*simp*]:
table-of Classes (*SXcpt xn*)
= *Some* ⦇*access*=*Public*,*cfields*=[],*methods*=*SXcpt-mdecls*,
init=*Skip*,
super=*if xn* = *Throwable then Object else SXcpt Throwable*,
superIfs=[]⦈
⟨*proof*⟩

lemma *table-classes-HasFoo* [*simp*]: *table-of Classes HasFoo* = *None*
⟨*proof*⟩

lemma *table-classes-Base* [*simp*]: *table-of Classes Base* = *Some BaseCl*
⟨*proof*⟩

lemma *table-classes-Ext* [*simp*]: *table-of Classes Ext* = *Some ExtCl*
⟨*proof*⟩

lemma *table-classes-Main* [*simp*]: *table-of Classes Main* = *Some MainCl*
⟨*proof*⟩

program

abbreviation
tprg :: *prog* **where**
tprg == ⦇*ifaces*=*Ifaces*,*classes*=*Classes*⦈

definition
test :: (*ty*)*list* ⇒ *stmt* **where**
test pTs = (*e*:==*NewC Ext*;;
Try Expr({*Main*,*ClassT Base*,*IntVir*}!!*e*·*foo*({*pTs*}[*Lit Null*]))
Catch((*SXcpt NullPointer*) *z*)
(*lab1*· *While*(*Acc*
(*Acc* (*arr-viewed-from Main Ext*).[*Lit* (*Intg 2*)])) *Skip*))

well-structuredness

lemma *not-Object-subcls-any* [*elim*!]: (*Object*, *C*) ∈ (*subcls1 tprg*)$^+$ ⟹ *R*
⟨*proof*⟩

lemma *not-Throwable-subcls-SXcpt* [*elim*!]:
(*SXcpt Throwable*, *SXcpt xn*) ∈ (*subcls1 tprg*)$^+$ ⟹ *R*
⟨*proof*⟩

lemma *not-SXcpt-n-subcls-SXcpt-n* [*elim*!]:
(*SXcpt xn*, *SXcpt xn*) ∈ (*subcls1 tprg*)$^+$ ⟹ *R*
⟨*proof*⟩

lemma *not-Base-subcls-Ext* [*elim*!]: (*Base*, *Ext*) ∈ (*subcls1 tprg*)$^+$ ⟹ *R*
⟨*proof*⟩

lemma *not-TName-n-subcls-TName-n* [*rule-format* (*no-asm*), *elim*!]:
(⦇*pid*=*java-lang*,*tid*=*TName tn*⦈, ⦇*pid*=*java-lang*,*tid*=*TName tn*⦈)
∈ (*subcls1 tprg*)$^+$ ⟶ *R*
⟨*proof*⟩

lemma *ws-idecl-HasFoo*: *ws-idecl tprg HasFoo* []
⟨*proof*⟩

lemma *ws-cdecl-Object*: *ws-cdecl tprg Object any*
⟨*proof*⟩

lemma *ws-cdecl-Throwable*: *ws-cdecl tprg* (*SXcpt Throwable*) *Object*
⟨*proof*⟩

lemma *ws-cdecl-SXcpt*: *ws-cdecl tprg* (*SXcpt xn*) (*SXcpt Throwable*)
⟨*proof*⟩

lemma *ws-cdecl-Base*: *ws-cdecl tprg Base Object*
⟨*proof*⟩

lemma *ws-cdecl-Ext*: *ws-cdecl tprg Ext Base*
⟨*proof*⟩

lemma *ws-cdecl-Main*: *ws-cdecl tprg Main Object*
⟨*proof*⟩

lemmas *ws-cdecls* = *ws-cdecl-SXcpt ws-cdecl-Object ws-cdecl-Throwable*
ws-cdecl-Base ws-cdecl-Ext ws-cdecl-Main

declare *not-Object-subcls-any* [*rule del*]
not-Throwable-subcls-SXcpt [*rule del*]
not-SXcpt-n-subcls-SXcpt-n [*rule del*]
not-Base-subcls-Ext [*rule del*] *not-TName-n-subcls-TName-n* [*rule del*]

lemma *ws-idecl-all*:
G=*tprg* ⟹ (∀ (*I*,*i*)∈*set Ifaces*. *ws-idecl G I* (*isuperIfs i*))
⟨*proof*⟩

lemma *ws-cdecl-all*: *G*=*tprg* ⟹ (∀ (*C*,*c*)∈*set Classes*. *ws-cdecl G C* (*super c*))
⟨*proof*⟩

lemma *ws-tprg*: *ws-prog tprg*
⟨*proof*⟩

misc program properties (independent of well-structuredness)

lemma *single-iface* [*simp*]: *is-iface tprg I* = (*I* = *HasFoo*)
⟨*proof*⟩

lemma *empty-subint1* [*simp*]: *subint1 tprg* = {}
⟨*proof*⟩

lemma *unique-ifaces*: *unique Ifaces*
⟨*proof*⟩

lemma *unique-classes*: *unique Classes*
⟨*proof*⟩

lemma *SXcpt-subcls-Throwable* [*simp*]: *tprg*$\vdash$*SXcpt xn*$\preceq_C$ *SXcpt Throwable*
⟨*proof*⟩

lemma *Ext-subclseq-Base* [*simp*]: *tprg*$\vdash$*Ext* $\preceq_C$ *Base*
⟨*proof*⟩

lemma *Ext-subcls-Base* [*simp*]: *tprg*$\vdash$*Ext* $\prec_C$ *Base*
⟨*proof*⟩

fields and method lookup

lemma *fields-tprg-Object* [*simp*]: *DeclConcepts.fields tprg Object* = []
⟨*proof*⟩

lemma *fields-tprg-Throwable* [*simp*]:
DeclConcepts.fields tprg (*SXcpt Throwable*) = []
⟨*proof*⟩

lemma *fields-tprg-SXcpt* [*simp*]: *DeclConcepts.fields tprg* (*SXcpt xn*) = []
⟨*proof*⟩

lemmas *fields-rec′* = *fields-rec* [*OF - ws-tprg*]

lemma *fields-Base* [*simp*]:
DeclConcepts.fields tprg Base
= [((*arr*,*Base*), (|*access*=*Public*,*static*=*True* ,*type*=*PrimT Boolean*.[]|)),
((*vee*,*Base*), (|*access*=*Public*,*static*=*False*,*type*=*Iface HasFoo* |))]
⟨*proof*⟩

lemma *fields-Ext* [*simp*]:
DeclConcepts.fields tprg Ext
= [((*vee*,*Ext*), (|*access*=*Public*,*static*=*False*,*type*= *PrimT Integer*|))]
@ *DeclConcepts.fields tprg Base*
⟨*proof*⟩

lemmas *imethds-rec′* = *imethds-rec* [*OF - ws-tprg*]
lemmas *methd-rec′* = *methd-rec* [*OF - ws-tprg*]

lemma *imethds-HasFoo* [*simp*]:
imethds tprg HasFoo = *set-option* ∘ *Map.empty*(*foo-sig*↦(*HasFoo*, *foo-mhead*))
⟨*proof*⟩

lemma *methd-tprg-Object* [*simp*]: *methd tprg Object* = *Map.empty*
⟨*proof*⟩

lemma *methd-Base* [*simp*]:
methd tprg Base = *table-of* [(λ(*s*,*m*). (*s*, *Base*, *m*)) *Base-foo*]
⟨*proof*⟩

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

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

lemma *Base-declares-foo*:

tprg⊢*mdecl Base-foo declared-in Base*
⟨*proof*⟩

lemma *foo-sig-not-undeclared-in-Base*:
¬ *tprg*⊢*mid foo-sig undeclared-in Base*
⟨*proof*⟩

lemma *Ext-declares-foo*:
tprg⊢*mdecl Ext-foo declared-in Ext*
⟨*proof*⟩

lemma *foo-sig-not-undeclared-in-Ext*:
¬ *tprg*⊢*mid foo-sig undeclared-in Ext*
⟨*proof*⟩

lemma *Base-foo-not-inherited-in-Ext*:
¬ *tprg* ⊢ *Ext inherits* (*Base*,*mdecl Base-foo*)
⟨*proof*⟩

lemma *Ext-method-inheritance*:
filter-tab (λ*sig m*. *tprg* ⊢ *Ext inherits method sig m*)
(*Map.empty*(*fst* ((λ(*s*, *m*). (*s*, *Base*, *m*)) *Base-foo*)↦
snd ((λ(*s*, *m*). (*s*, *Base*, *m*)) *Base-foo*)))
= *Map.empty*
⟨*proof*⟩

lemma *methd-Ext* [*simp*]: *methd tprg Ext* =
table-of [(λ(*s*,*m*). (*s*, *Ext*, *m*)) *Ext-foo*]
⟨*proof*⟩

accessibility

lemma *classesDefined*:
⟦*class tprg C* = *Some c*; *C*≠*Object*⟧ ⟹ ∃ *sc*. *class tprg* (*super c*) = *Some sc*
⟨*proof*⟩

lemma *superclassesBase* [*simp*]: *superclasses tprg Base*={*Object*}
⟨*proof*⟩

lemma *superclassesExt* [*simp*]: *superclasses tprg Ext*={*Base*,*Object*}
⟨*proof*⟩

lemma *superclassesMain* [*simp*]: *superclasses tprg Main*={*Object*}
⟨*proof*⟩

lemma *HasFoo-accessible*[*simp*]:*tprg*⊢(*Iface HasFoo*) *accessible-in P*
⟨*proof*⟩

lemma *HasFoo-is-acc-iface*[*simp*]: *is-acc-iface tprg P HasFoo*
⟨*proof*⟩

lemma *HasFoo-is-acc-type*[*simp*]: *is-acc-type tprg P* (*Iface HasFoo*)
⟨*proof*⟩

lemma *Base-accessible*[*simp*]:*tprg*⊢(*Class Base*) *accessible-in P*
⟨*proof*⟩

lemma *Base-is-acc-class*[*simp*]: *is-acc-class tprg P Base*

⟨*proof*⟩

lemma *Base-is-acc-type*[*simp*]: *is-acc-type tprg P* (*Class Base*)
⟨*proof*⟩

lemma *Ext-accessible*[*simp*]:*tprg*⊢(*Class Ext*) *accessible-in P*
⟨*proof*⟩

lemma *Ext-is-acc-class*[*simp*]: *is-acc-class tprg P Ext*
⟨*proof*⟩

lemma *Ext-is-acc-type*[*simp*]: *is-acc-type tprg P* (*Class Ext*)
⟨*proof*⟩

lemma *accmethd-tprg-Object* [*simp*]: *accmethd tprg S Object* = *Map.empty*
⟨*proof*⟩

lemma *snd-special-simp*: *snd* ((λ(*s*, *m*). (*s*, *a*, *m*)) *x*) = (*a*,*snd x*)
⟨*proof*⟩

lemma *fst-special-simp*: *fst* ((λ(*s*, *m*). (*s*, *a*, *m*)) *x*) = *fst x*
⟨*proof*⟩

lemma *foo-sig-undeclared-in-Object*:
tprg⊢*mid foo-sig undeclared-in Object*
⟨*proof*⟩

lemma *unique-sig-Base-foo*:
tprg⊢ *mdecl* (*sig*, *snd Base-foo*) *declared-in Base* ⟹ *sig*=*foo-sig*
⟨*proof*⟩

lemma *Base-foo-no-override*:
tprg,*sig*⊢(*Base*,(*snd Base-foo*)) *overrides old* ⟹ *P*
⟨*proof*⟩

lemma *Base-foo-no-stat-override*:
tprg,*sig*⊢(*Base*,(*snd Base-foo*)) $overrides_S$ *old* ⟹ *P*
⟨*proof*⟩

lemma *Base-foo-no-hide*:
tprg,*sig*⊢(*Base*,(*snd Base-foo*)) *hides old* ⟹ *P*
⟨*proof*⟩

lemma *Ext-foo-no-hide*:
tprg,*sig*⊢(*Ext*,(*snd Ext-foo*)) *hides old* ⟹ *P*
⟨*proof*⟩

lemma *unique-sig-Ext-foo*:
tprg⊢ *mdecl* (*sig*, *snd Ext-foo*) *declared-in Ext* ⟹ *sig*=*foo-sig*
⟨*proof*⟩

lemma *Ext-foo-override*:
tprg,*sig*⊢(*Ext*,(*snd Ext-foo*)) *overrides old*
⟹ *old* = (*Base*,(*snd Base-foo*))
⟨*proof*⟩

lemma *Ext-foo-stat-override*:
tprg,*sig*⊢(*Ext*,(*snd Ext-foo*)) $overrides_S$ *old*

$\Longrightarrow$ *old* = (*Base*,(*snd Base-foo*))
⟨*proof*⟩

lemma *Base-foo-member-of-Base*:
tprg⊢(*Base*,*mdecl Base-foo*) *member-of Base*
⟨*proof*⟩

lemma *Base-foo-member-in-Base*:
tprg⊢(*Base*,*mdecl Base-foo*) *member-in Base*
⟨*proof*⟩

lemma *Ext-foo-member-of-Ext*:
tprg⊢(*Ext*,*mdecl Ext-foo*) *member-of Ext*
⟨*proof*⟩

lemma *Ext-foo-member-in-Ext*:
tprg⊢(*Ext*,*mdecl Ext-foo*) *member-in Ext*
⟨*proof*⟩

lemma *Base-foo-permits-acc*:
tprg ⊢ (*Base*, *mdecl Base-foo*) *in Base permits-acc-from S*
⟨*proof*⟩

lemma *Base-foo-accessible* [*simp*]:
tprg⊢(*Base*,*mdecl Base-foo*) *of Base accessible-from S*
⟨*proof*⟩

lemma *Base-foo-dyn-accessible* [*simp*]:
tprg⊢(*Base*,*mdecl Base-foo*) *in Base dyn-accessible-from S*
⟨*proof*⟩

lemma *accmethd-Base* [*simp*]:
accmethd tprg S Base = *methd tprg Base*
⟨*proof*⟩

lemma *Ext-foo-permits-acc*:
tprg ⊢ (*Ext*, *mdecl Ext-foo*) *in Ext permits-acc-from S*
⟨*proof*⟩

lemma *Ext-foo-accessible* [*simp*]:
tprg⊢(*Ext*,*mdecl Ext-foo*) *of Ext accessible-from S*
⟨*proof*⟩

lemma *Ext-foo-dyn-accessible* [*simp*]:
tprg⊢(*Ext*,*mdecl Ext-foo*) *in Ext dyn-accessible-from S*
⟨*proof*⟩

lemma *Ext-foo-overrides-Base-foo*:
tprg⊢(*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*]⦈

$= Some\ (Ext,snd\ Ext\text{-}foo)$
⟨*proof*⟩

lemma *Base-fields-accessible*[*simp*]:
accfield tprg S Base
$= table\text{-}of((map\ (\lambda((n,d),f).(n,(d,f))))\ (DeclConcepts.fields\ tprg\ Base))$
⟨*proof*⟩

lemma *arr-member-of-Base*:
$tprg\vdash(Base,\ fdecl\ (arr,$
$(\!|access = Public,\ static = True,\ type = PrimT\ Boolean.[]|\!)))$
member-of Base
⟨*proof*⟩

lemma *arr-member-in-Base*:
$tprg\vdash(Base,\ fdecl\ (arr,$
$(\!|access = Public,\ static = True,\ type = PrimT\ Boolean.[]|\!)))$
member-in Base
⟨*proof*⟩

lemma *arr-member-of-Ext*:
$tprg\vdash(Base,\ fdecl\ (arr,$
$(\!|access = Public,\ static = True,\ type = PrimT\ Boolean.[]|\!)))$
member-of Ext
⟨*proof*⟩

lemma *arr-member-in-Ext*:
$tprg\vdash(Base,\ fdecl\ (arr,$
$(\!|access = Public,\ static = True,\ type = PrimT\ Boolean.[]|\!)))$
member-in Ext
⟨*proof*⟩

lemma *Ext-fields-accessible*[*simp*]:
accfield tprg S Ext
$= table\text{-}of((map\ (\lambda((n,d),f).(n,(d,f))))\ (DeclConcepts.fields\ tprg\ Ext))$
⟨*proof*⟩

lemma *arr-Base-dyn-accessible* [*simp*]:
$tprg\vdash(Base,\ fdecl\ (arr,\ (\!|access{=}Public,static{=}True\ ,type{=}PrimT\ Boolean.[]|\!)))$
in Base dyn-accessible-from S
⟨*proof*⟩

lemma *arr-Ext-dyn-accessible*[*simp*]:
$tprg\vdash(Base,\ fdecl\ (arr,\ (\!|access{=}Public,static{=}True\ ,type{=}PrimT\ Boolean.[]|\!)))$
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* ⟹ *Ball* (*set Ifaces*) (*wf-idecl p*)
⟨*proof*⟩

lemma *wf-cdecl-all-standard-classes*:
Ball (*set standard-classes*) (*wf-cdecl tprg*)
⟨*proof*⟩

lemma *wf-cdecl-all*: *p*=*tprg* ⟹ *Ball* (*set Classes*) (*wf-cdecl p*)
⟨*proof*⟩

theorem *wf-tprg*: *wf-prog tprg*
⟨*proof*⟩

max spec

lemma *appl-methds-Base-foo*:
appl-methds tprg S (*ClassT Base*) (|*name*=*foo*, *parTs*=[*NT*]|) =
{((*ClassT Base*, (|*access*=*Public*,*static*=*False*,*pars*=[*z*],*resT*=*Class Base*|))
,[*Class Base*])}
⟨*proof*⟩

lemma *max-spec-Base-foo*: *max-spec tprg S* (*ClassT Base*) (|*name*=*foo*,*parTs*=[*NT*]|) =

{((*ClassT Base*, (|*access*=*Public*,*static*=*False*,*pars*=[*z*],*resT*=*Class Base*|))
  , [*Class Base*])}
⟨*proof*⟩

well-typedness

schematic-goal *wt-test*: (|*prg*=*tprg*,*cls*=*Main*,*lcl*=*Map.empty*(*VName e*↦*Class Base*)|)⊢*test ?pTs*::√
⟨*proof*⟩

definite assignment

schematic-goal *da-test*: (|*prg*=*tprg*,*cls*=*Main*,*lcl*=*Map.empty*(*VName e*↦*Class Base*)|)
    ⊢{} »⟨*test ?pTs*⟩» (|*nrm*={*VName e*},*brk*=λ *l*. *UNIV*|)
⟨*proof*⟩

execution

lemma *alloc-one*: ⋀*a obj*. ⟦*the* (*new-Addr h*) = *a*; *atleast-free h* (*Suc n*)⟧ ⟹
  new-Addr h = *Some a* ∧ *atleast-free* (*h*(*a*↦*obj*)) *n*
⟨*proof*⟩

declare *fvar-def2* [*simp*] *avar-def2* [*simp*] *init-lvars-def2* [*simp*]
declare *init-obj-def* [*simp*] *var-tys-def* [*simp*] *fields-table-def* [*simp*]
declare *BaseCl-def* [*simp*] *ExtCl-def* [*simp*] *Ext-foo-def* [*simp*]
    Base-foo-defs [*simp*]

⟨*ML*⟩
lemmas *eval-Is* = *eval-Init eval-StatRef AbruptIs eval-intros*

axiomatization
  a :: *loc* **and**
  b :: *loc* **and**
  c :: *loc*

abbreviation *one* == *Suc 0*
abbreviation *two* == *Suc one*
abbreviation *three* == *Suc two*
abbreviation *four* == *Suc three*

abbreviation
  obj-a == (|*tag*=*Arr* (*PrimT Boolean*) *2*
        ,*values*= *Map.empty*(*Inr 0*↦*Bool False*, *Inr 1*↦*Bool False*)|)

abbreviation
  obj-b == (|*tag*=*CInst Ext*
        ,*values*=(*Map.empty*(*Inl* (*vee*, *Base*)↦*Null*, *Inl* (*vee*, *Ext*)↦*Intg 0*))|)

abbreviation
  obj-c == (|*tag*=*CInst* (*SXcpt NullPointer*),*values*=*CONST Map.empty*|)

abbreviation *arr-N* == *Map.empty*(*Inl* (*arr*, *Base*)↦*Null*)
abbreviation *arr-a* == *Map.empty*(*Inl* (*arr*, *Base*)↦*Addr a*)

abbreviation
  globs1 == *Map.empty*(*Inr Ext*   ↦(|*tag*=*undefined*, *values*=*Map.empty*|),
        Inr Base ↦(|*tag*=*undefined*, *values*=*arr-N*|),
        Inr Object↦(|*tag*=*undefined*, *values*=*Map.empty*|))

abbreviation

globs2 == *Map.empty*(*Inr Ext* ↦(|*tag*=*undefined*, *values*=*Map.empty*|),
Inr Object↦(|*tag*=*undefined*, *values*=*Map.empty*|),
Inl a↦*obj-a*,
Inr Base ↦(|*tag*=*undefined*, *values*=*arr-a*|))

abbreviation *globs3* == *globs2*(*Inl b*↦*obj-b*)
abbreviation *globs8* == *globs3*(*Inl c*↦*obj-c*)
abbreviation *locs3* == *Map.empty*(*VName e*↦*Addr b*)
abbreviation *locs8* == *locs3*(*VName z*↦*Addr c*)

abbreviation *s0* == *st Map.empty Map.empty*
abbreviation *s0'* == *Norm s0*
abbreviation *s1* == *st globs1 Map.empty*
abbreviation *s1'* == *Norm s1*
abbreviation *s2* == *st globs2 Map.empty*
abbreviation *s2'* == *Norm s2*
abbreviation *s3* == *st globs3 locs3*
abbreviation *s3'* == *Norm s3*
abbreviation *s7'* == (*Some* (*Xcpt* (*Std NullPointer*)), *s3*)
abbreviation *s8* == *st globs8 locs8*
abbreviation *s8'* == *Norm s8*
abbreviation *s9'* == (*Some* (*Xcpt* (*Std IndOutBound*)), *s8*)

declare *prod.inject* [*simp del*]
schematic-goal *exec-test*:
⟦*the* (*new-Addr* (*heap s1*)) = *a*;
the (*new-Addr* (*heap ?s2*)) = *b*;
the (*new-Addr* (*heap ?s3*)) = *c*⟧ ⟹
atleast-free (*heap s0*) *four* ⟹
tprg⊢*s0'* −*test* [*Class Base*]→ *?s9'*
⟨*proof*⟩
declare *prod.inject* [*simp*]

end

Chapter 17

Conform

1 Conformance notions for the type soundness proof for Java

theory *Conform* **imports** *State* **begin**

design issues:

- lconf allows for (arbitrary) inaccessible values
- "conforms" does not directly imply that the dynamic types of all objects on the heap are indeed existing classes. Yet this can be inferred for all referenced objs.

type-synonym $env' = prog \times (lname, ty)\ table$

extension of global store

definition $gext :: st \Rightarrow st \Rightarrow bool$ (‹-≤|-› [71,71] 70) **where**
$s{\le}|s' \equiv \forall r.\ \forall obj{\in}globs\ s\ r{:}\ \exists obj'{\in}globs\ s'\ r{:}\ tag\ obj'{=}\ tag\ obj$

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*:
$[\![s{\le}|s';\ globs\ s\ r = Some\ obj]\!]$
$\Longrightarrow \exists obj'.\ globs\ s'\ r = Some\ obj' \wedge tag\ obj' = tag\ obj$
⟨*proof*⟩

lemma *rev-gext-objD*:
$[\![globs\ s\ r = Some\ obj;\ s{\le}|s']\!]$
$\Longrightarrow \exists obj'.\ globs\ s'\ r = Some\ obj' \wedge tag\ obj' = tag\ obj$
⟨*proof*⟩

lemma *init-class-obj-inited*:
$init\text{-}class\text{-}obj\ G\ C\ s1{\le}|s2 \Longrightarrow inited\ C\ (globs\ s2)$
⟨*proof*⟩

lemma *gext-refl* [*intro*!, *simp*]: $s{\le}|s$
⟨*proof*⟩

lemma *gext-gupd* [*simp*, *elim*!]: $\bigwedge s.\ globs\ s\ r = None \Longrightarrow s{\le}|gupd(r{\mapsto}x)s$
⟨*proof*⟩

lemma *gext-new* [*simp*, *elim*!]: $\bigwedge s.\ globs\ s\ r = None \Longrightarrow s{\le}|init\text{-}obj\ G\ oi\ r\ s$
⟨*proof*⟩

lemma *gext-trans* [*elim*]: $\bigwedge X.\ [\![s{\leq}|s';\ s'{\leq}|s'']\!] \Longrightarrow s{\leq}|s''$
⟨*proof*⟩

lemma *gext-upd-gobj* [*intro*!]: $s{\leq}|upd\text{-}gobj\ r\ n\ v\ s$
⟨*proof*⟩

lemma *gext-cong1* [*simp*]: $set\text{-}locals\ l\ s1{\leq}|s2 = s1{\leq}|s2$
⟨*proof*⟩

lemma *gext-cong2* [*simp*]: $s1{\leq}|set\text{-}locals\ l\ s2 = s1{\leq}|s2$
⟨*proof*⟩

lemma *gext-lupd1* [*simp*]: $lupd(vn{\mapsto}v)s1{\leq}|s2 = s1{\leq}|s2$
⟨*proof*⟩

lemma *gext-lupd2* [*simp*]: $s1{\leq}|lupd(vn{\mapsto}v)s2 = s1{\leq}|s2$
⟨*proof*⟩

lemma *inited-gext*: $[\![inited\ C\ (globs\ s);\ s{\leq}|s']\!] \Longrightarrow inited\ C\ (globs\ s')$
⟨*proof*⟩

value conformance

definition *conf* :: $prog \Rightarrow st \Rightarrow val \Rightarrow ty \Rightarrow bool$ (‹-,-⊢-::≼-› [71,71,71,71] 70)
where $G,s{\vdash}v{::}{\preceq}T = (\exists\,T'{\in}typeof\ (\lambda a.\ map\text{-}option\ obj\text{-}ty\ (heap\ s\ a))\ v{:}G{\vdash}T'{\preceq}T)$

lemma *conf-cong* [*simp*]: $G,set\text{-}locals\ l\ s{\vdash}v{::}{\preceq}T = G,s{\vdash}v{::}{\preceq}T$
⟨*proof*⟩

lemma *conf-lupd* [*simp*]: $G,lupd(vn{\mapsto}va)s{\vdash}v{::}{\preceq}T = G,s{\vdash}v{::}{\preceq}T$
⟨*proof*⟩

lemma *conf-PrimT* [*simp*]: $\forall\,dt.\ typeof\ dt\ v = Some\ (PrimT\ t) \Longrightarrow G,s{\vdash}v{::}{\preceq}PrimT\ t$
⟨*proof*⟩

lemma *conf-Boolean*: $G,s{\vdash}v{::}{\preceq}PrimT\ Boolean \Longrightarrow \exists\ b.\ v{=}Bool\ b$
⟨*proof*⟩

lemma *conf-litval* [*rule-format* (*no-asm*)]:
$typeof\ (\lambda a.\ None)\ v = Some\ T \longrightarrow G,s{\vdash}v{::}{\preceq}T$
⟨*proof*⟩

lemma *conf-Null* [*simp*]: $G,s{\vdash}Null{::}{\preceq}T = G{\vdash}NT{\preceq}T$
⟨*proof*⟩

lemma *conf-Addr*:
$G,s{\vdash}Addr\ a{::}{\preceq}T = (\exists\,obj.\ heap\ s\ a = Some\ obj \wedge G{\vdash}obj\text{-}ty\ obj{\preceq}T)$
⟨*proof*⟩

lemma *conf-AddrI*: $[\![heap\ s\ a = Some\ obj;\ G{\vdash}obj\text{-}ty\ obj{\preceq}T]\!] \Longrightarrow G,s{\vdash}Addr\ a{::}{\preceq}T$
⟨*proof*⟩

lemma *defval-conf* [*rule-format* (*no-asm*), *elim*]:
$is\text{-}type\ G\ T \longrightarrow G,s{\vdash}default\text{-}val\ T{::}{\preceq}T$
⟨*proof*⟩

lemma *conf-widen* [*rule-format* (*no-asm*), *elim*]:
$G \vdash T \preceq T' \Longrightarrow G,s \vdash x :: \preceq T \longrightarrow ws\text{-}prog\ G \longrightarrow G,s \vdash x :: \preceq T'$
⟨*proof*⟩

lemma *conf-gext* [*rule-format* (*no-asm*), *elim*]:
$G,s \vdash v :: \preceq T \longrightarrow s \leq | s' \longrightarrow G,s' \vdash v :: \preceq T$
⟨*proof*⟩

lemma *conf-list-widen* [*rule-format* (*no-asm*)]:
$ws\text{-}prog\ G \Longrightarrow$
$\forall Ts\ Ts'.\ list\text{-}all2\ (conf\ G\ s)\ vs\ Ts$
$\longrightarrow\ G \vdash Ts[\preceq]\ Ts' \longrightarrow list\text{-}all2\ (conf\ G\ s)\ vs\ Ts'$
⟨*proof*⟩

lemma *conf-RefTD* [*rule-format* (*no-asm*)]:
$G,s \vdash a' :: \preceq RefT\ T$
$\longrightarrow a' = Null \lor (\exists a\ obj\ T'.\ a' = Addr\ a \land heap\ s\ a = Some\ obj \land$
$obj\text{-}ty\ obj = T' \land G \vdash T' \preceq RefT\ T)$
⟨*proof*⟩

value list conformance

definition
lconf :: *prog* ⇒ *st* ⇒ (′*a*, *val*) *table* ⇒ (′*a*, *ty*) *table* ⇒ *bool* (‹-,-⊢-[::≼]-› [*71*,*71*,*71*,*71*] *70*)
where $G,s \vdash vs[::\preceq]Ts = (\forall n.\ \forall T \in Ts\ n:\ \exists v \in vs\ n:\ G,s \vdash v :: \preceq T)$

lemma *lconfD*: $[\![G,s \vdash vs[::\preceq]Ts;\ Ts\ n = Some\ T]\!] \Longrightarrow G,s \vdash (the\ (vs\ n)) :: \preceq T$
⟨*proof*⟩

lemma *lconf-cong* [*simp*]: $\bigwedge s.\ G,set\text{-}locals\ x\ s \vdash l[::\preceq]L = G,s \vdash l[::\preceq]L$
⟨*proof*⟩

lemma *lconf-lupd* [*simp*]: $G,lupd(vn \mapsto v)s \vdash l[::\preceq]L = G,s \vdash l[::\preceq]L$
⟨*proof*⟩

lemma *lconf-new*: $[\![L\ vn = None;\ G,s \vdash l[::\preceq]L]\!] \Longrightarrow G,s \vdash l(vn \mapsto v)[::\preceq]L$
⟨*proof*⟩

lemma *lconf-upd*: $[\![G,s \vdash l[::\preceq]L;\ G,s \vdash v :: \preceq T;\ L\ vn = Some\ T]\!] \Longrightarrow$
$G,s \vdash l(vn \mapsto v)[::\preceq]L$
⟨*proof*⟩

lemma *lconf-ext*: $[\![G,s \vdash l[::\preceq]L;\ G,s \vdash v :: \preceq T]\!] \Longrightarrow G,s \vdash l(vn \mapsto v)[::\preceq]L(vn \mapsto T)$
⟨*proof*⟩

lemma *lconf-map-sum* [*simp*]:
$G,s \vdash l1\ (+)\ l2[::\preceq]L1\ (+)\ L2 = (G,s \vdash l1[::\preceq]L1 \land G,s \vdash l2[::\preceq]L2)$
⟨*proof*⟩

lemma *lconf-ext-list* [*rule-format* (*no-asm*)]:
$\bigwedge X.\ [\![G,s \vdash l[::\preceq]L]\!] \Longrightarrow$
$\forall vs\ Ts.\ distinct\ vns \longrightarrow length\ Ts = length\ vns$
$\longrightarrow list\text{-}all2\ (conf\ G\ s)\ vs\ Ts \longrightarrow G,s \vdash l(vns[\mapsto]vs)[::\preceq]L(vns[\mapsto]Ts)$
⟨*proof*⟩

lemma *lconf-deallocL*: $[\![G,s\vdash l[::\preceq]L(vn\mapsto T);\ L\ vn = None]\!] \Longrightarrow G,s\vdash l[::\preceq]L$
⟨*proof*⟩

lemma *lconf-gext* [*elim*]: $[\![G,s\vdash l[::\preceq]L;\ s\leq|s']\!] \Longrightarrow G,s'\vdash l[::\preceq]L$
⟨*proof*⟩

lemma *lconf-empty* [*simp, intro*!]: $G,s\vdash vs[::\preceq]Map.empty$
⟨*proof*⟩

lemma *lconf-init-vals* [*intro*!]:
$\forall n.\ \forall T\in fs\ n{:}is\text{-}type\ G\ T \Longrightarrow G,s\vdash init\text{-}vals\ fs[::\preceq]fs$
⟨*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* ⇒ *st* ⇒ (′*a*, *val*) *table* ⇒ (′*a*, *ty*) *table* ⇒ *bool* (‹-,-⊢-[∼::≼]-› [*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*: $[\![G,s\vdash vs[\sim::\preceq]Ts;\ Ts\ n = Some\ T;\ vs\ n = Some\ v]\!] \Longrightarrow G,s\vdash v::\preceq T$
⟨*proof*⟩

lemma *wlconf-cong* [*simp*]: $\bigwedge s.\ G,set\text{-}locals\ x\ s\vdash l[\sim::\preceq]L = G,s\vdash l[\sim::\preceq]L$
⟨*proof*⟩

lemma *wlconf-lupd* [*simp*]: $G,lupd(vn\mapsto v)s\vdash l[\sim::\preceq]L = G,s\vdash l[\sim::\preceq]L$
⟨*proof*⟩

lemma *wlconf-upd*: $[\![G,s\vdash l[\sim::\preceq]L;\ G,s\vdash v::\preceq T;\ L\ vn = Some\ T]\!] \Longrightarrow$
$G,s\vdash l(vn\mapsto v)[\sim::\preceq]L$
⟨*proof*⟩

lemma *wlconf-ext*: $[\![G,s\vdash l[\sim::\preceq]L;\ G,s\vdash v::\preceq T]\!] \Longrightarrow G,s\vdash l(vn\mapsto v)[\sim::\preceq]L(vn\mapsto T)$
⟨*proof*⟩

lemma *wlconf-map-sum* [*simp*]:
$G,s\vdash l1\ (+)\ l2[\sim::\preceq]L1\ (+)\ L2 = (G,s\vdash l1[\sim::\preceq]L1 \wedge G,s\vdash l2[\sim::\preceq]L2)$
⟨*proof*⟩

lemma *wlconf-ext-list* [*rule-format* (*no-asm*)]:
$\bigwedge X.\ [\![G,s\vdash l[\sim::\preceq]L]\!] \Longrightarrow$
$\forall vs\ Ts.\ distinct\ vns \longrightarrow length\ Ts = length\ vns$
$\longrightarrow list\text{-}all2\ (conf\ G\ s)\ vs\ Ts \longrightarrow G,s\vdash l(vns[\mapsto]vs)[\sim::\preceq]L(vns[\mapsto]Ts)$
⟨*proof*⟩

lemma *wlconf-deallocL*: $[\![G,s\vdash l[\sim::\preceq]L(vn\mapsto T);\ L\ vn = None]\!] \Longrightarrow G,s\vdash l[\sim::\preceq]L$
⟨*proof*⟩

lemma *wlconf-gext* [*elim*]: $[\![G,s\vdash l[\sim::\preceq]L;\ s\leq|s']\!] \Longrightarrow G,s'\vdash l[\sim::\preceq]L$
⟨*proof*⟩

lemma *wlconf-empty* [*simp*, *intro*!]: $G,s\vdash vs[\sim::\preceq]Map.empty$
⟨*proof*⟩

lemma *wlconf-empty-vals*: $G,s\vdash Map.empty[\sim::\preceq]ts$
⟨*proof*⟩

lemma *wlconf-init-vals* [*intro*!]:
$\forall n.\ \forall T\in fs\ n{:}is\text{-}type\ G\ T \Longrightarrow G,s\vdash init\text{-}vals\ fs[\sim::\preceq]fs$
⟨*proof*⟩

lemma *lconf-wlconf*:
$G,s\vdash l[::\preceq]L \Longrightarrow G,s\vdash l[\sim::\preceq]L$
⟨*proof*⟩

object conformance

definition
oconf :: *prog* ⇒ *st* ⇒ *obj* ⇒ *oref* ⇒ *bool* (‹-,-⊢-::≼√-› [*71*,*71*,*71*,*71*] *70*) **where**
$(G,s\vdash obj::\preceq\surd r) = (G,s\vdash values\ obj[::\preceq]var\text{-}tys\ G\ (tag\ obj)\ r \wedge$
(*case r of*
Heap a ⇒ *is-type G* (*obj-ty obj*)
| *Stat C* ⇒ *True*))

lemma *oconf-is-type*: $G,s\vdash obj::\preceq\surd Heap\ a \Longrightarrow is\text{-}type\ G\ (obj\text{-}ty\ obj)$
⟨*proof*⟩

lemma *oconf-lconf*: $G,s\vdash obj::\preceq\surd r \Longrightarrow G,s\vdash values\ obj[::\preceq]var\text{-}tys\ G\ (tag\ obj)\ r$
⟨*proof*⟩

lemma *oconf-cong* [*simp*]: $G,set\text{-}locals\ l\ s\vdash obj::\preceq\surd r = G,s\vdash obj::\preceq\surd r$
⟨*proof*⟩

lemma *oconf-init-obj-lemma*:
$[\![\bigwedge C\ c.\ class\ G\ C = Some\ c \Longrightarrow unique\ (DeclConcepts.fields\ G\ C);$
$\bigwedge C\ c\ f\ fld.\ [\![class\ G\ C = Some\ c;$
$table\text{-}of\ (DeclConcepts.fields\ G\ C)\ f = Some\ fld\ ]\!]$
$\Longrightarrow is\text{-}type\ G\ (type\ fld);$
(*case r of*
Heap a ⇒ *is-type G* (*obj-ty obj*)
| *Stat C* ⇒ *is-class G C*)
$]\!] \Longrightarrow G,s\vdash obj\ (\!|values{:=}init\text{-}vals\ (var\text{-}tys\ G\ (tag\ obj)\ r)|\!)::\preceq\surd r$
⟨*proof*⟩

state conformance

definition
conforms :: *state* ⇒ *env'* ⇒ *bool* (‹-::≼-› [*71*,*71*] *70*) **where**
$xs::\preceq E =$
(*let* (*G*, *L*) = *E*; *s* = *snd xs*; *l* = *locals s in*
$(\forall r.\ \forall obj\in globs\ s\ r{:}\ G,s\vdash obj\ ::\preceq\surd r) \wedge G,s\vdash l\ [\sim::\preceq]L \wedge$
$(\forall a.\ fst\ xs{=}Some(Xcpt\ (Loc\ a)) \longrightarrow G,s\vdash Addr\ a::\preceq\ Class\ (SXcpt\ Throwable)) \wedge$
$(fst\ xs{=}Some(Jump\ Ret) \longrightarrow l\ Result \neq None))$

conforms

lemma *conforms-globsD*:
$[\![(x,\ s)::\preceq(G,\ L);\ globs\ s\ r = Some\ obj]\!] \Longrightarrow G,s\vdash obj::\preceq\surd r$
⟨*proof*⟩

lemma *conforms-localD*: $(x,\ s)::\preceq(G,\ L) \Longrightarrow G,s\vdash locals\ s[\sim::\preceq]L$
⟨*proof*⟩

lemma *conforms-XcptLocD*: $[\![(x,\ s)::\preceq(G,\ L);\ x = Some\ (Xcpt\ (Loc\ a))]\!] \Longrightarrow$
$G,s\vdash Addr\ a::\preceq\ Class\ (SXcpt\ Throwable)$
⟨*proof*⟩

lemma *conforms-RetD*: $[\![(x,\ s)::\preceq(G,\ L);\ x = Some\ (Jump\ Ret)]\!] \Longrightarrow$
$(locals\ s)\ Result \neq None$
⟨*proof*⟩

lemma *conforms-RefTD*:
$[\![G,s\vdash a'::\preceq RefT\ t;\ a' \neq Null;\ (x,s)\ ::\preceq(G,\ L)]\!] \Longrightarrow$
$\exists\,a\ obj.\ a' = Addr\ a \wedge globs\ s\ (Inl\ a) = Some\ obj \wedge$
$G\vdash obj\text{-}ty\ obj\preceq RefT\ t \wedge is\text{-}type\ G\ (obj\text{-}ty\ obj)$
⟨*proof*⟩

lemma *conforms-Jump* [*iff*]:
$j{=}Ret \longrightarrow locals\ s\ Result \neq None$
$\Longrightarrow ((Some\ (Jump\ j),\ s)::\preceq(G,\ L)) = (Norm\ s::\preceq(G,\ L))$
⟨*proof*⟩

lemma *conforms-StdXcpt* [*iff*]:
$((Some\ (Xcpt\ (Std\ xn)),\ s)::\preceq(G,\ L)) = (Norm\ s::\preceq(G,\ L))$
⟨*proof*⟩

lemma *conforms-Err* [*iff*]:
$((Some\ (Error\ e),\ s)::\preceq(G,\ L)) = (Norm\ s::\preceq(G,\ L))$
⟨*proof*⟩

lemma *conforms-raise-if* [*iff*]:
$((raise\text{-}if\ c\ xn\ x,\ s)::\preceq(G,\ L)) = ((x,\ s)::\preceq(G,\ L))$
⟨*proof*⟩

lemma *conforms-error-if* [*iff*]:
$((error\text{-}if\ c\ err\ x,\ s)::\preceq(G,\ L)) = ((x,\ s)::\preceq(G,\ L))$
⟨*proof*⟩

lemma *conforms-NormI*: $(x,\ s)::\preceq(G,\ L) \Longrightarrow Norm\ s::\preceq(G,\ L)$
⟨*proof*⟩

lemma *conforms-absorb* [*rule-format*]:
$(a,\ b)::\preceq(G,\ L) \longrightarrow (absorb\ j\ a,\ b)::\preceq(G,\ L)$
⟨*proof*⟩

lemma *conformsI*: $[\![\forall\,r.\ \forall\,obj{\in}globs\ s\ r:\ G,s\vdash obj::\preceq\surd r;$
$G,s\vdash locals\ s[\sim::\preceq]L;$
$\forall\,a.\ x = Some\ (Xcpt\ (Loc\ a)) \longrightarrow G,s\vdash Addr\ a::\preceq\ Class\ (SXcpt\ Throwable);$
$x = Some\ (Jump\ Ret)\longrightarrow locals\ s\ Result \neq None]\!] \Longrightarrow$
$(x,\ s)::\preceq(G,\ L)$
⟨*proof*⟩

lemma *conforms-xconf*: $[\![(x,\ s)::\preceq(G,L);$

$\forall a.\ x' = Some\ (Xcpt\ (Loc\ a)) \longrightarrow G,s\vdash Addr\ a{::}\preceq\ Class\ (SXcpt\ Throwable);$
$x' = Some\ (Jump\ Ret) \longrightarrow locals\ s\ Result \neq None]\!] \Longrightarrow$
$(x',s){::}\preceq(G,L)$
⟨*proof*⟩

lemma *conforms-lupd*:
$[\![(x,\ s){::}\preceq(G,\ L);\ L\ vn = Some\ T;\ G,s\vdash v{::}\preceq T]\!] \Longrightarrow (x,\ lupd(vn\mapsto v)s){::}\preceq(G,\ L)$
⟨*proof*⟩

lemmas *conforms-allocL-aux* = *conforms-localD* [*THEN wlconf-ext*]

lemma *conforms-allocL*:
$[\![(x,\ s){::}\preceq(G,\ L);\ G,s\vdash v{::}\preceq T]\!] \Longrightarrow (x,\ lupd(vn\mapsto v)s){::}\preceq(G,\ L(vn\mapsto T))$
⟨*proof*⟩

lemmas *conforms-deallocL-aux* = *conforms-localD* [*THEN wlconf-deallocL*]

lemma *conforms-deallocL*: $\bigwedge s.[\![s{::}\preceq(G,\ L(vn\mapsto T));\ L\ vn = None]\!] \Longrightarrow s{::}\preceq(G,L)$
⟨*proof*⟩

lemma *conforms-gext*: $[\![(x,\ s){::}\preceq(G,L);\ s\leq|s';$
$\forall r.\ \forall obj\in globs\ s'\ r{:}\ G,s'\vdash obj{::}\preceq\surd r;$
$locals\ s'{=}locals\ s]\!] \Longrightarrow (x,s'){::}\preceq(G,L)$
⟨*proof*⟩

lemma *conforms-xgext*:
$[\![(x\ ,s){::}\preceq(G,L);\ (x',\ s'){::}\preceq(G,\ L);\ s'\leq|s;dom\ (locals\ s') \subseteq dom\ (locals\ s)]\!]$
$\Longrightarrow (x',s){::}\preceq(G,L)$
⟨*proof*⟩

lemma *conforms-gupd*: $\bigwedge obj.\ [\![(x,\ s){::}\preceq(G,\ L);\ G,s\vdash obj{::}\preceq\surd r;\ s\leq|gupd(r\mapsto obj)s]\!]$
$\Longrightarrow\ (x,\ gupd(r\mapsto obj)s){::}\preceq(G,\ L)$
⟨*proof*⟩

lemma *conforms-upd-gobj*: $[\![(x,s){::}\preceq(G,\ L);\ globs\ s\ r = Some\ obj;$
$var\text{-}tys\ G\ (tag\ obj)\ r\ n = Some\ T;\ G,s\vdash v{::}\preceq T]\!] \Longrightarrow (x,upd\text{-}gobj\ r\ n\ v\ s){::}\preceq(G,L)$
⟨*proof*⟩

lemma *conforms-set-locals*:
$[\![(x,s){::}\preceq(G,\ L');\ G,s\vdash l[\sim{::}\preceq]L;\ x{=}Some\ (Jump\ Ret) \longrightarrow l\ Result \neq None]\!]$
$\Longrightarrow (x,set\text{-}locals\ l\ s){::}\preceq(G,L)$
⟨*proof*⟩

lemma *conforms-locals*:
$[\![(a,b){::}\preceq(G,\ L);\ L\ x = Some\ T;locals\ b\ x \neq None]\!]$
$\Longrightarrow G,b\vdash the\ (locals\ b\ x){::}\preceq T$
⟨*proof*⟩

lemma *conforms-return*:
$\bigwedge s'.\ [\![(x,s){::}\preceq(G,\ L);\ (x',s'){::}\preceq(G,\ L');\ s\leq|s';x'{\neq}Some\ (Jump\ Ret)]\!] \Longrightarrow$
$(x',set\text{-}locals\ (locals\ s)\ s'){::}\preceq(G,\ L)$
⟨*proof*⟩

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*: $G \vdash s0 \ -sxalloc\rightarrow \ s1$ **and**
no-jmp: *abrupt s0* $\neq$ *Some* (*Jump j*)
shows *abrupt s1* $\neq$ *Some* (*Jump j*)
⟨*proof*⟩

lemma *sxalloc-no-jump'*:
assumes *sxalloc*: $G \vdash s0 \ -sxalloc\rightarrow \ s1$ **and**
jump: *abrupt s1* = *Some* (*Jump j*)
shows *abrupt s0* = *Some* (*Jump j*)
⟨*proof*⟩

lemma *halloc-no-jump*:
assumes *halloc*: $G \vdash s0 \ -halloc\ oi \succ a\rightarrow \ s1$ **and**
no-jmp: *abrupt s0* $\neq$ *Some* (*Jump j*)
shows *abrupt s1* $\neq$ *Some* (*Jump j*)
⟨*proof*⟩

lemma *halloc-no-jump'*:
assumes *halloc*: $G \vdash s0 \ -halloc\ oi \succ a\rightarrow \ s1$ **and**
jump: *abrupt s1* = *Some* (*Jump j*)
shows *abrupt s0* = *Some* (*Jump j*)
⟨*proof*⟩

lemma *Body-no-jump*:
assumes *eval*: $G \vdash s0 \ -Body\ D\ c-\succ v\rightarrow s1$ **and**
jump: *abrupt s0* $\neq$ *Some* (*Jump j*)
shows *abrupt s1* $\neq$ *Some* (*Jump j*)
⟨*proof*⟩

lemma *Methd-no-jump*:
assumes *eval*: $G \vdash s0 \ -Methd\ D\ sig-\succ v\rightarrow \ s1$ **and**
jump: *abrupt s0* $\neq$ *Some* (*Jump j*)
shows *abrupt s1* $\neq$ *Some* (*Jump j*)
⟨*proof*⟩

lemma *jumpNestingOkS-mono*:
assumes *jumpNestingOk-l'*: *jumpNestingOkS jmps' c*

and *subset*: $jmps' \subseteq jmps$
shows *jumpNestingOkS jmps c*
⟨*proof*⟩

corollary *jumpNestingOk-mono*:
assumes *jmpOk*: *jumpNestingOk jmps′ t*
and *subset*: $jmps' \subseteq jmps$
shows *jumpNestingOk jmps t*
⟨*proof*⟩

lemma *assign-abrupt-propagation*:
assumes *f-ok*: $abrupt\ (f\ n\ s) \neq x$
and *ass*: $abrupt\ (assign\ f\ n\ s) = x$
shows $abrupt\ s = x$
⟨*proof*⟩

lemma *wt-init-comp-ty′*:
is-acc-type (*prg Env*) (*pid* (*cls Env*)) *T* $\Longrightarrow$ *Env*⊢*init-comp-ty T*::$\surd$
⟨*proof*⟩

lemma *fvar-upd-no-jump*:
assumes *upd*: *upd* = *snd* (*fst* (*fvar statDeclC stat fn a s′*))
and *noJmp*: *abrupt s* ≠ *Some* (*Jump j*)
shows *abrupt* (*upd val s*) ≠ *Some* (*Jump j*)
⟨*proof*⟩

lemma *avar-state-no-jump*:
assumes *jmp*: *abrupt* (*snd* (*avar G i a s*)) = *Some* (*Jump j*)
shows *abrupt s* = *Some* (*Jump j*)
⟨*proof*⟩

lemma *avar-upd-no-jump*:
assumes *upd*: *upd* = *snd* (*fst* (*avar G i a s′*))
and *noJmp*: *abrupt s* ≠ *Some* (*Jump j*)
shows *abrupt* (*upd val s*) ≠ *Some* (*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 exeample, a break can't leave its enclosing loop, an return cant leave its enclosing method. To proove this, the method call is critical. Allthough 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 enshure this we need a kind of conformance of the state, like in the typesafety proof. But then we will redo the typesafty 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 typesafty 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 conlcude, 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 enshure that for all classinitialisations and methods the nesting of jumps is wellformed, too.

theorem *jumpNestingOk-eval*:
assumes *eval*: *G*⊢ *s0* −*t*≻→ (*v*,*s1*)

and *jmpOk*: *jumpNestingOk jmps t*
and *wt*: *Env⊢t::T*
and *wf*: *wf-prog G*
and *G*: *prg Env = G*
and *no-jmp*: *∀ j. abrupt s0 = Some (Jump j) ⟶ j ∈ jmps*
(**is** *?Jmp jmps s0*)
shows (*∀ j. fst s1 = Some (Jump j) ⟶ j ∈ jmps*) ∧
(*normal s1* ⟶
(∀ *w upd. v=In2 (w,upd)*
⟶ (∀ *s j val.*
abrupt s ≠ Some (Jump j) ⟶
abrupt (upd val s) ≠ Some (Jump j)))))
(**is** *?Jmp jmps s1 ∧ ?Upd v s1*)
⟨*proof*⟩

lemmas *jumpNestingOk-evalE = jumpNestingOk-eval [THEN conjE,rule-format]*

lemma *jumpNestingOk-eval-no-jump*:
assumes *eval*: *prg Env⊢ s0 −t≻→ (v,s1)* **and**
jmpOk: *jumpNestingOk {} t* **and**
no-jmp: *abrupt s0 ≠ Some (Jump j)* **and**
wt: *Env⊢t::T* **and**
wf: *wf-prog (prg Env)*
shows *abrupt s1 ≠ Some (Jump j)* ∧
(*normal s1 ⟶ v=In2 (w,upd)*
⟶ *abrupt s ≠ Some (Jump j′)*
⟶ *abrupt (upd val s) ≠ 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⊢s0 −e−≻v→ s1* **and**
no-jmp: *abrupt s0 ≠ Some (Jump j)* **and**
wt: *Env⊢e::−T* **and**
wf: *wf-prog (prg Env)*
shows *abrupt s1 ≠ Some (Jump j)*
⟨*proof*⟩

corollary *eval-var-no-jump*:
assumes *eval*: *prg Env⊢s0 −var=≻(w,upd)→ s1* **and**
no-jmp: *abrupt s0 ≠ Some (Jump j)* **and**
wt: *Env⊢var::=T* **and**
wf: *wf-prog (prg Env)*
shows *abrupt s1 ≠ Some (Jump j)* ∧
(*normal s1* ⟶
(*abrupt s ≠ Some (Jump j′)*
⟶ *abrupt (upd val s) ≠ Some (Jump j′)*)))
⟨*proof*⟩

lemmas *eval-var-no-jumpE = eval-var-no-jump [THEN conjE,rule-format]*

corollary *eval-statement-no-jump*:
assumes *eval*: *prg Env⊢s0 −c→ s1* **and**
jmpOk: *jumpNestingOkS {} c* **and**
no-jmp: *abrupt s0 ≠ Some (Jump j)* **and**
wt: *Env⊢c::√* **and**

wf: *wf-prog* (*prg Env*)
shows *abrupt s1* $\neq$ *Some* (*Jump j*)
⟨*proof*⟩

corollary *eval-expression-list-no-jump*:
assumes *eval*: *prg Env*⊢*s0* −*es*≐≻*v*→ *s1* **and**
no-jmp: *abrupt s0* $\neq$ *Some* (*Jump j*) **and**
wt: *Env*⊢*es*::≐*T* **and**
wf: *wf-prog* (*prg Env*)
shows *abrupt s1* $\neq$ *Some* (*Jump j*)
⟨*proof*⟩

lemma *union-subseteq-elim* [*elim*]: $[\![A \cup B \subseteq C; [\![A \subseteq C; B \subseteq C]\!] \Longrightarrow P]\!] \Longrightarrow P$
⟨*proof*⟩

lemma *dom-locals-halloc-mono*:
assumes *halloc*: *G*⊢*s0*−*halloc oi*≻*a*→*s1*
shows *dom* (*locals* (*store s0*)) ⊆ *dom* (*locals* (*store s1*))
⟨*proof*⟩

lemma *dom-locals-sxalloc-mono*:
assumes *sxalloc*: *G*⊢*s0*−*sxalloc*→*s1*
shows *dom* (*locals* (*store s0*)) ⊆ *dom* (*locals* (*store s1*))
⟨*proof*⟩

lemma *dom-locals-assign-mono*:
assumes *f-ok*: *dom* (*locals* (*store s*)) ⊆ *dom* (*locals* (*store* (*f n s*)))
shows *dom* (*locals* (*store s*)) ⊆ *dom* (*locals* (*store* (*assign f n s*)))
⟨*proof*⟩

lemma *dom-locals-lvar-mono*:
dom (*locals* (*store s*)) ⊆ *dom* (*locals* (*store* (*snd* (*lvar vn s′*) *val s*)))
⟨*proof*⟩

lemma *dom-locals-fvar-vvar-mono*:
dom (*locals* (*store s*))
⊆ *dom* (*locals* (*store* (*snd* (*fst* (*fvar statDeclC stat fn a s′*)) *val s*)))
⟨*proof*⟩

lemma *dom-locals-fvar-mono*:
dom (*locals* (*store s*))
⊆ *dom* (*locals* (*store* (*snd* (*fvar statDeclC stat fn a s*))))
⟨*proof*⟩

lemma *dom-locals-avar-vvar-mono*:
dom (*locals* (*store s*))
⊆ *dom* (*locals* (*store* (*snd* (*fst* (*avar G i a s′*)) *val s*)))
⟨*proof*⟩

lemma *dom-locals-avar-mono*:
dom (*locals* (*store s*))
⊆ *dom* (*locals* (*store* (*snd* (*avar G i a s*))))
⟨*proof*⟩

Since assignments are modelled as functions from states to states, we must take into account these functions. They appear only in the assignment rule and as result from evaluating a variable. Thats 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. Thats why the evaluation rules for variables result in a pair consisting of a value and an update function. Of course we could also think of a pair of a value and a reference in the store, instead of the generic update function. But as only array updates can cause a special exception (if the types mismatch) and not array reads we then have to introduce two different rules to handle array reads and updates

lemma *dom-locals-eval-mono*:
  assumes *eval*: $G\vdash s0 -t\succ\rightarrow (v,s1)$
  shows $dom\ (locals\ (store\ s0)) \subseteq dom\ (locals\ (store\ s1)) \wedge$
  $(\forall\ vv.\ v{=}In2\ vv \wedge normal\ s1$
  $\longrightarrow (\forall\ s\ val.\ dom\ (locals\ (store\ s))$
  $\subseteq dom\ (locals\ (store\ ((snd\ vv)\ val\ s)))))$
⟨*proof*⟩

lemma *dom-locals-eval-mono-elim*:
  assumes *eval*: $G\vdash s0 -t\succ\rightarrow (v,s1)$
  obtains $dom\ (locals\ (store\ s0)) \subseteq dom\ (locals\ (store\ s1))$ **and**
  $\bigwedge vv\ s\ val.\ [\![v{=}In2\ vv;\ normal\ s1]\!]$
  $\Longrightarrow dom\ (locals\ (store\ s))$
  $\subseteq dom\ (locals\ (store\ ((snd\ vv)\ val\ s)))$
  ⟨*proof*⟩

lemma *halloc-no-abrupt*:
  assumes *halloc*: $G\vdash s0 -halloc\ oi\succ a\rightarrow s1$ **and**
  normal: *normal s1*
  shows *normal s0*
⟨*proof*⟩

lemma *sxalloc-mono-no-abrupt*:
  assumes *sxalloc*: $G\vdash s0 -sxalloc\rightarrow s1$ **and**
  normal: *normal s1*
  shows *normal s0*
⟨*proof*⟩

lemma *union-subseteqI*: $[\![A \cup B \subseteq C;\ A' \subseteq A;\ B' \subseteq B]\!] \Longrightarrow A' \cup B' \subseteq C$
  ⟨*proof*⟩

lemma *union-subseteqIl*: $[\![A \cup B \subseteq C;\ A' \subseteq A]\!] \Longrightarrow A' \cup B \subseteq C$
  ⟨*proof*⟩

lemma *union-subseteqIr*: $[\![A \cup B \subseteq C;\ B' \subseteq B]\!] \Longrightarrow A \cup B' \subseteq C$
  ⟨*proof*⟩

lemma *subseteq-union-transl* [*trans*]: $[\![A \subseteq B;\ B \cup C \subseteq D]\!] \Longrightarrow A \cup C \subseteq D$
  ⟨*proof*⟩

lemma *subseteq-union-transr* [*trans*]: $[\![A \subseteq B;\ C \cup B \subseteq D]\!] \Longrightarrow A \cup C \subseteq D$
  ⟨*proof*⟩

lemma *union-subseteq-weaken*: $[\![A \cup B \subseteq C;\ [\![A \subseteq C;\ B \subseteq C]\!] \Longrightarrow P\ ]\!] \Longrightarrow P$
  ⟨*proof*⟩

lemma *assigns-good-approx*:

assumes
eval: $G\vdash s0 -t\succ\rightarrow (v,s1)$ **and**
normal: *normal s1*
shows *assigns t* $\subseteq$ *dom* (*locals* (*store s1*))
⟨*proof*⟩

corollary *assignsE-good-approx*:
assumes
eval: *prg Env*$\vdash s0 -e-\succ v\rightarrow s1$ **and**
normal: *normal s1*
shows *assignsE e* $\subseteq$ *dom* (*locals* (*store s1*))
⟨*proof*⟩

corollary *assignsV-good-approx*:
assumes
eval: *prg Env*$\vdash s0 -v=\succ vf\rightarrow s1$ **and**
normal: *normal s1*
shows *assignsV v* $\subseteq$ *dom* (*locals* (*store s1*))
⟨*proof*⟩

corollary *assignsEs-good-approx*:
assumes
eval: *prg Env*$\vdash s0 -es\doteq\succ vs\rightarrow s1$ **and**
normal: *normal s1*
shows *assignsEs es* $\subseteq$ *dom* (*locals* (*store s1*))
⟨*proof*⟩

lemma *constVal-eval*:
assumes *const*: *constVal e* = *Some c* **and**
eval: $G\vdash$*Norm s0* $-e-\succ v\rightarrow s$
shows $v = c \wedge$ *normal s*
⟨*proof*⟩

lemmas *constVal-eval-elim* = *constVal-eval* [*THEN conjE*]

lemma *eval-unop-type*:
typeof dt (*eval-unop unop v*) = *Some* (*PrimT* (*unop-type unop*))
⟨*proof*⟩

lemma *eval-binop-type*:
typeof dt (*eval-binop binop v1 v2*) = *Some* (*PrimT* (*binop-type binop*))
⟨*proof*⟩

lemma *constVal-Boolean*:
assumes *const*: *constVal e* = *Some c* **and**
wt: *Env*$\vdash$*e*::$-$*PrimT Boolean*
shows *typeof empty-dt c* = *Some* (*PrimT Boolean*)
⟨*proof*⟩

lemma *assigns-if-good-approx*:
assumes
eval: *prg Env*$\vdash s0 -e-\succ b\rightarrow s1$ **and**
normal: *normal s1* **and**
bool: *Env*$\vdash$ *e*::$-$*PrimT Boolean*
shows *assigns-if* (*the-Bool b*) *e* $\subseteq$ *dom* (*locals* (*store s1*))
⟨*proof*⟩

lemma *assigns-if-good-approx'*:
assumes *eval*: $G\vdash s0 -e-\succ b\rightarrow s1$

and *normal*: *normal s1*
and *bool*: (|*prg*=*G*,*cls*=*C*,*lcl*=*L*|)⊢*e*::− (*PrimT Boolean*)
shows *assigns-if* (*the-Bool b*) *e* ⊆ *dom* (*locals* (*store s1*))
⟨*proof*⟩

lemma *subset-Intl*: *A* ⊆ *C* ⟹ *A* ∩ *B* ⊆ *C*
⟨*proof*⟩

lemma *subset-Intr*: *B* ⊆ *C* ⟹ *A* ∩ *B* ⊆ *C*
⟨*proof*⟩

lemma *da-good-approx*:
assumes *eval*: *prg Env*⊢*s0* −*t*≻→ (*v*,*s1*) **and**
wt: *Env*⊢*t*::*T* (**is** *?Wt Env t T*) **and**
da: *Env*⊢ *dom* (*locals* (*store s0*)) »*t*» *A* (**is** *?Da Env s0 t A*) **and**
wf: *wf-prog* (*prg Env*)
shows (*normal s1* ⟶ (*nrm A* ⊆ *dom* (*locals* (*store s1*)))) ∧
(∀ *l*. *abrupt s1* = *Some* (*Jump* (*Break l*)) ∧ *normal s0*
⟶ (*brk A l* ⊆ *dom* (*locals* (*store s1*)))) ∧
(*abrupt s1* = *Some* (*Jump Ret*) ∧ *normal s0*
⟶ *Result* ∈ *dom* (*locals* (*store s1*)))
(**is** *?NormalAssigned s1 A* ∧ *?BreakAssigned s0 s1 A* ∧ *?ResAssigned s0 s1*)
⟨*proof*⟩

lemma *da-good-approxE*:
assumes
prg Env⊢*s0* −*t*≻→ (*v*, *s1*) **and** *Env*⊢*t*::*T* **and**
Env⊢ *dom* (*locals* (*store s0*)) »*t*» *A* **and** *wf-prog* (*prg Env*)
obtains
normal s1 ⟹ *nrm A* ⊆ *dom* (*locals* (*store s1*)) **and**
⋀ *l*. ⟦*abrupt s1* = *Some* (*Jump* (*Break l*)); *normal s0*⟧
⟹ *brk A l* ⊆ *dom* (*locals* (*store s1*)) **and**
⟦*abrupt s1* = *Some* (*Jump Ret*);*normal s0*⟧⟹*Result* ∈ *dom* (*locals* (*store s1*))
⟨*proof*⟩

lemma *da-good-approxE'*:
assumes *eval*: *G*⊢*s0* −*t*≻→ (*v*, *s1*)
and *wt*: (|*prg*=*G*,*cls*=*C*,*lcl*=*L*|)⊢*t*::*T*
and *da*: (|*prg*=*G*,*cls*=*C*,*lcl*=*L*|)⊢ *dom* (*locals* (*store s0*)) »*t*» *A*
and *wf*: *wf-prog G*
obtains *normal s1* ⟹ *nrm A* ⊆ *dom* (*locals* (*store s1*)) **and**
⋀ *l*. ⟦*abrupt s1* = *Some* (*Jump* (*Break l*)); *normal s0*⟧
⟹ *brk A l* ⊆ *dom* (*locals* (*store s1*)) **and**
⟦*abrupt s1* = *Some* (*Jump Ret*);*normal s0*⟧
⟹ *Result* ∈ *dom* (*locals* (*store s1*))
⟨*proof*⟩

declare [[*simproc add*: *wt-expr wt-var wt-exprs wt-stmt*]]

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*: $G \vdash s0\ -halloc\ oi \succ a \rightarrow\ s1$ **and**
error-free-s0: *error-free s0*
shows *error-free s1*
⟨*proof*⟩

lemma *error-free-sxalloc*:
assumes *sxalloc*: $G \vdash s0\ -sxalloc \rightarrow\ s1$ **and** *error-free-s0*: *error-free s0*
shows *error-free s1*
⟨*proof*⟩

lemma *error-free-check-field-access-eq*:
error-free (*check-field-access G accC statDeclC fn stat a s*)
$\Longrightarrow$ (*check-field-access G accC statDeclC fn stat a s*) = *s*
⟨*proof*⟩

lemma *error-free-check-method-access-eq*:
error-free (*check-method-access G accC statT mode sig a′ s*)
$\Longrightarrow$ (*check-method-access G accC statT mode sig a′ s*) = *s*
⟨*proof*⟩

lemma *error-free-FVar-lemma*:
error-free s
$\Longrightarrow$ *error-free* (*abupd* (*if stat then id else np a*) *s*)
⟨*proof*⟩

lemma *error-free-init-lvars* [*simp*,*intro*]:
error-free s $\Longrightarrow$
error-free (*init-lvars G C sig mode a pvs s*)
⟨*proof*⟩

lemma *error-free-LVar-lemma*:
error-free s $\Longrightarrow$ *error-free* (*assign* ($\lambda v.$ *supd lupd*($vn \mapsto v$)) *w s*)
⟨*proof*⟩

lemma *error-free-throw* [*simp*,*intro*]:

error-free s $\Longrightarrow$ *error-free* (*abupd* (*throw x*) *s*)
⟨*proof*⟩

result conformance

definition
assign-conforms :: *st* ⇒ (*val* ⇒ *state* ⇒ *state*) ⇒ *ty* ⇒ *env'* ⇒ *bool* (‹-≤|-≼-::≼-› [*71*,*71*,*71*,*71*] *70*)
where
s≤|*f*≼*T*::≼*E* =
((∀ *s' w*. *Norm s'*::≼*E* ⟶ *fst E*,*s'*⊢*w*::≼*T* ⟶ *s*≤|*s'* ⟶ *assign f w* (*Norm s'*)::≼*E*) ∧
(∀ *s' w*. *error-free s'* ⟶ (*error-free* (*assign f w s'*))))

definition
rconf :: *prog* ⇒ *lenv* ⇒ *st* ⇒ *term* ⇒ *vals* ⇒ *tys* ⇒ *bool* (‹-,-,-⊢-≻-::≼-› [*71*,*71*,*71*,*71*,*71*,*71*] *70*)
where
G,*L*,*s*⊢*t*≻*v*::≼*T* =
(*case T of*
Inl T ⇒ *if* (∃ *var*. *t*=*In2 var*)
then (∀ *n*. (*the-In2 t*) = *LVar n*
⟶ (*fst* (*the-In2 v*) = *the* (*locals s n*)) ∧
(*locals s n* ≠ *None* ⟶ *G*,*s*⊢*fst* (*the-In2 v*)::≼*T*)) ∧
(¬ (∃ *n*. *the-In2 t*=*LVar n*) ⟶ (*G*,*s*⊢*fst* (*the-In2 v*)::≼*T*))∧
(*s*≤|*snd* (*the-In2 v*)≼*T*::≼(*G*,*L*))
else G,*s*⊢*the-In1 v*::≼*T*
| *Inr Ts* ⇒ *list-all2* (*conf G s*) (*the-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 ∃ *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 form 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*⊢*In1 ec*≻*In1 v* ::≼*Inl T* = *G*,*s*⊢*v*::≼*T*
⟨*proof*⟩

lemma *rconf-In2-no-LVar* [*simp*]:
∀ *n*. *va*≠*LVar n* $\Longrightarrow$
G,*L*,*s*⊢*In2 va*≻*In2 vf*::≼*Inl T* = (*G*,*s*⊢*fst vf*::≼*T* ∧ *s*≤|*snd vf*≼*T*::≼(*G*,*L*))
⟨*proof*⟩

lemma *rconf-In2-LVar* [*simp*]:
va=*LVar n* $\Longrightarrow$
G,*L*,*s*⊢*In2 va*≻*In2 vf*::≼*Inl T*
= ((*fst vf* = *the* (*locals s n*)) ∧
(*locals s n* ≠ *None* ⟶ *G*,*s*⊢*fst vf*::≼*T*) ∧ *s*≤|*snd vf*≼*T*::≼(*G*,*L*))
⟨*proof*⟩

lemma *rconf-In3* [*simp*]:

$G,L,s\vdash In3\ es\succ In3\ vs::\preceq Inr\ Ts = list\text{-}all2\ (\lambda v\ T.\ G,s\vdash v::\preceq T)\ vs\ Ts$
⟨proof⟩

fits and conf

lemma *conf-fits*: $G,s\vdash v::\preceq T \Longrightarrow G,s\vdash v\ fits\ T$
⟨proof⟩

lemma *fits-conf*:
$[\![G,s\vdash v::\preceq T;\ G\vdash T\preceq?\ T';\ G,s\vdash v\ fits\ T';\ ws\text{-}prog\ G]\!] \Longrightarrow G,s\vdash v::\preceq T'$
⟨proof⟩

lemma *fits-Array*:
$[\![G,s\vdash v::\preceq T;\ G\vdash T'.[]\preceq T.[];\ G,s\vdash v\ fits\ T';\ ws\text{-}prog\ G]\!] \Longrightarrow G,s\vdash v::\preceq T'$
⟨proof⟩

gext

lemma *halloc-gext*: $\bigwedge s1\ s2.\ G\vdash s1\ -halloc\ oi\succ a\rightarrow\ s2 \Longrightarrow snd\ s1\le|snd\ s2$
⟨proof⟩

lemma *sxalloc-gext*: $\bigwedge s1\ s2.\ G\vdash s1\ -sxalloc\rightarrow\ s2 \Longrightarrow snd\ s1\le|snd\ s2$
⟨proof⟩

lemma *eval-gext-lemma* [*rule-format* (*no-asm*)]:
$G\vdash s\ -t\succ\rightarrow\ (w,s') \Longrightarrow snd\ s\le|snd\ s' \wedge (case\ w\ of$
$\quad In1\ v \Rightarrow True$
$\mid In2\ vf \Rightarrow normal\ s \longrightarrow (\forall\ v\ x\ s.\ s\le|snd\ (assign\ (snd\ vf)\ v\ (x,s)))$
$\mid In3\ vs \Rightarrow True)$
⟨proof⟩

lemma *evar-gext-f*:
$G\vdash Norm\ s1\ -e=\succ vf\ \rightarrow\ s2 \Longrightarrow s\le|snd\ (assign\ (snd\ vf)\ v\ (x,s))$
⟨proof⟩

lemmas *eval-gext* = *eval-gext-lemma* [*THEN conjunct1*]

lemma *eval-gext'*: $G\vdash(x1,s1)\ -t\succ\rightarrow\ (w,(x2,s2)) \Longrightarrow s1\le|s2$
⟨proof⟩

lemma *init-yields-initd*: $G\vdash Norm\ s1\ -Init\ C\rightarrow\ s2 \Longrightarrow initd\ C\ s2$
⟨proof⟩

Lemmas

lemma *obj-ty-obj-class1*:
$[\![wf\text{-}prog\ G;\ is\text{-}type\ G\ (obj\text{-}ty\ obj)]\!] \Longrightarrow is\text{-}class\ G\ (obj\text{-}class\ obj)$
⟨proof⟩

lemma *oconf-init-obj*:
$[\![wf\text{-}prog\ G;$
$(case\ r\ of\ Heap\ a \Rightarrow is\text{-}type\ G\ (obj\text{-}ty\ obj) \mid Stat\ C \Rightarrow is\text{-}class\ G\ C)$
$]\!] \Longrightarrow G,s\vdash obj\ (\!|values:=init\text{-}vals\ (var\text{-}tys\ G\ (tag\ obj)\ r)|\!)::\preceq\surd r$
⟨proof⟩

lemma *conforms-newG*: $[\![globs\ s\ oref = None;\ (x,\ s)::\preceq(G,L);$
$wf\text{-}prog\ G;\ case\ oref\ of\ Heap\ a \Rightarrow is\text{-}type\ G\ (obj\text{-}ty\ (\!|tag=oi,values=vs|\!))$
$\mid Stat\ C \Rightarrow is\text{-}class\ G\ C]\!] \Longrightarrow$
$(x,\ init\text{-}obj\ G\ oi\ oref\ s)::\preceq(G,\ L)$

⟨*proof*⟩

lemma *conforms-init-class-obj*:
⟦(*x*,*s*)::≼(*G*, *L*); *wf-prog G*; *class G C*=*Some y*; ¬ *inited C* (*globs s*)⟧ ⟹
(*x*,*init-class-obj G C s*)::≼(*G*, *L*)
⟨*proof*⟩

lemma *fst-init-lvars*[*simp*]:
fst (*init-lvars G C sig* (*invmode m e*) *a′ pvs* (*x*,*s*)) =
(*if is-static m then x else* (*np a′*) *x*)
⟨*proof*⟩

lemma *halloc-conforms*: ⋀*s1*. ⟦*G*⊢*s1* −*halloc oi*≻*a*→ *s2*; *wf-prog G*; *s1*::≼(*G*, *L*);
is-type G (*obj-ty* (|*tag*=*oi*,*values*=*fs*|))⟧ ⟹ *s2*::≼(*G*, *L*)
⟨*proof*⟩

lemma *halloc-type-sound*:
⋀*s1*. ⟦*G*⊢*s1* −*halloc oi*≻*a*→ (*x*,*s*); *wf-prog G*; *s1*::≼(*G*, *L*);
T = *obj-ty* (|*tag*=*oi*,*values*=*fs*|); *is-type G T*⟧ ⟹
(*x*,*s*)::≼(*G*, *L*) ∧ (*x* = *None* ⟶ *G*,*s*⊢*Addr a*::≼*T*)
⟨*proof*⟩

lemma *sxalloc-type-sound*:
⋀*s1 s2*. ⟦*G*⊢*s1* −*sxalloc*→ *s2*; *wf-prog G*⟧ ⟹
case fst s1 of
None ⇒ *s2* = *s1*
| *Some abr* ⇒ (*case abr of*
Xcpt x ⇒ (∃ *a*. *fst s2* = *Some*(*Xcpt* (*Loc a*)) ∧
(∀ *L*. *s1*::≼(*G*,*L*) ⟶ *s2*::≼(*G*,*L*)))
| *Jump j* ⇒ *s2* = *s1*
| *Error e* ⇒ *s2* = *s1*)
⟨*proof*⟩

lemma *wt-init-comp-ty*:
is-acc-type G (*pid C*) *T* ⟹ (|*prg*=*G*,*cls*=*C*,*lcl*=*L*|)⊢*init-comp-ty T*::√
⟨*proof*⟩

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

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

definition
DynT-prop :: [*prog*,*inv-mode*,*qtname*,*ref-ty*] ⇒ *bool* (‹-⊢-→-≼-›[*71*,*71*,*71*,*71*]*70*)
where
G⊢*mode*→*D*≼*t* = (*mode* = *IntVir* ⟶ *is-class G D* ∧
(*if* (∃ *T*. *t*=*ArrayT T*) *then D*=*Object else G*⊢*Class D*≼*RefT t*))

lemma *DynT-propI*:
⟦(*x*,*s*)::≼(*G*, *L*); *G*,*s*⊢*a′*::≼*RefT statT*; *wf-prog G*; *mode* = *IntVir* ⟶ *a′* ≠ *Null*⟧
⟹ *G*⊢*mode*→*invocation-class mode s a′ statT*≼*statT*
⟨*proof*⟩

lemma *invocation-methd*:
⟦*wf-prog G*; *statT* ≠ *NullT*;
(∀ *statC*. *statT* = *ClassT statC* ⟶ *is-class G statC*);
(∀ *I*. *statT* = *IfaceT I* ⟶ *is-iface G I* ∧ *mode* ≠ *SuperM*);

$(\forall \quad T.\ statT = ArrayT\ T \longrightarrow mode \neq SuperM)$;
$G \vdash mode \rightarrow invocation\text{-}class\ mode\ s\ a'\ statT \preceq statT$;
$dynlookup\ G\ statT\ (invocation\text{-}class\ mode\ s\ a'\ statT)\ sig = Some\ m\ \rrbracket$
$\Longrightarrow methd\ G\ (invocation\text{-}declclass\ G\ mode\ s\ a'\ statT\ sig)\ sig = Some\ m$
⟨*proof*⟩

lemma *DynT-mheadsD*:
$\llbracket G \vdash invmode\ sm\ e \rightarrow invC \preceq statT$;
$wf\text{-}prog\ G$; $(\!|prg=G, cls=C, lcl=L|\!) \vdash e :: -RefT\ statT$;
$(statDeclT, sm) \in mheads\ G\ C\ statT\ sig$;
$invC = invocation\text{-}class\ (invmode\ sm\ e)\ s\ a'\ statT$;
$declC = invocation\text{-}declclass\ G\ (invmode\ sm\ e)\ s\ a'\ statT\ sig$
$\rrbracket \Longrightarrow$
$\exists\ dm.$
$methd\ G\ declC\ sig = Some\ dm \wedge dynlookup\ G\ statT\ invC\ sig = Some\ dm\ \wedge$
$G \vdash resTy\ (mthd\ dm) \preceq resTy\ sm \wedge$
$wf\text{-}mdecl\ G\ declC\ (sig,\ mthd\ dm) \wedge$
$declC = declclass\ dm \wedge$
$is\text{-}static\ dm = is\text{-}static\ sm \wedge$
$is\text{-}class\ G\ invC \wedge is\text{-}class\ G\ declC\ \wedge G \vdash invC \preceq_C declC\ \wedge$
$(if\ invmode\ sm\ e = IntVir$
$then\ (\forall\ statC.\ statT = ClassT\ statC \longrightarrow G \vdash invC \preceq_C statC)$
$else\ (\ (\exists\ statC.\ statT = ClassT\ statC \wedge G \vdash statC \preceq_C declC)$
$\vee (\forall\ statC.\ statT \neq ClassT\ statC \wedge declC = Object)) \wedge$
$statDeclT = ClassT\ (declclass\ dm))$
⟨*proof*⟩

corollary *DynT-mheadsE* [*consumes 7*]:
— Same as *DynT-mheadsD* but better suited for application in typesafety proof
assumes *invC-compatible*: $G \vdash mode \rightarrow invC \preceq statT$
and *wf*: $wf\text{-}prog\ G$
and *wt-e*: $(\!|prg=G, cls=C, lcl=L|\!) \vdash e :: -RefT\ statT$
and *mheads*: $(statDeclT, sm) \in mheads\ G\ C\ statT\ sig$
and *mode*: $mode = invmode\ sm\ e$
and *invC*: $invC = invocation\text{-}class\ mode\ s\ a'\ statT$
and *declC*: $declC = invocation\text{-}declclass\ G\ mode\ s\ a'\ statT\ sig$
and *dm*: $\bigwedge\ dm.\ \llbracket methd\ G\ declC\ sig = Some\ dm$;
$dynlookup\ G\ statT\ invC\ sig = Some\ dm$;
$G \vdash resTy\ (mthd\ dm) \preceq resTy\ sm$;
$wf\text{-}mdecl\ G\ declC\ (sig,\ mthd\ dm)$;
$declC = declclass\ dm$;
$is\text{-}static\ dm = is\text{-}static\ sm$;
$is\text{-}class\ G\ invC$; $is\text{-}class\ G\ declC$; $G \vdash invC \preceq_C declC$;
$(if\ invmode\ sm\ e = IntVir$
$then\ (\forall\ statC.\ statT = ClassT\ statC \longrightarrow G \vdash invC \preceq_C statC)$
$else\ (\ (\exists\ statC.\ statT = ClassT\ statC \wedge G \vdash statC \preceq_C declC)$
$\vee (\forall\ statC.\ statT \neq ClassT\ statC \wedge declC = Object)) \wedge$
$statDeclT = ClassT\ (declclass\ dm))\rrbracket \Longrightarrow P$
shows P
⟨*proof*⟩

lemma *DynT-conf*: $\llbracket G \vdash invocation\text{-}class\ mode\ s\ a'\ statT \preceq_C declC$; $wf\text{-}prog\ G$;
$isrtype\ G\ (statT)$;
$G,s \vdash a' :: \preceq RefT\ statT$; $mode = IntVir \longrightarrow a' \neq Null$;
$mode \neq IntVir \longrightarrow \quad (\exists\ statC.\ statT = ClassT\ statC \wedge G \vdash statC \preceq_C declC)$
$\vee\ (\forall\ statC.\ statT \neq ClassT\ statC \wedge declC = Object)\rrbracket$
$\Longrightarrow G,s \vdash a' :: \preceq\ Class\ declC$
⟨*proof*⟩

lemma *Ass-lemma*:
$[\![$ $G \vdash Norm\ s0\ -var{=}\succ(w,\ f)\to\ Norm\ s1$; $G \vdash Norm\ s1\ -e{-}\succ v\to\ Norm\ s2$;
  $G,s2 \vdash v{::}\preceq eT$;$s1 \leq| s2 \longrightarrow assign\ f\ v\ (Norm\ s2){::}\preceq(G,\ L)]\!]$
$\Longrightarrow assign\ f\ v\ (Norm\ s2){::}\preceq(G,\ L)\ \wedge$
  $(normal\ (assign\ f\ v\ (Norm\ s2)) \longrightarrow G,store\ (assign\ f\ v\ (Norm\ s2)) \vdash v{::}\preceq eT)$
⟨*proof*⟩

lemma *Throw-lemma*: $[\![G \vdash tn \preceq_C\ SXcpt\ Throwable$; $wf\text{-}prog\ G$; $(x1,s1){::}\preceq(G,\ L)$;
  $x1 = None \longrightarrow G,s1 \vdash a'{::}\preceq\ Class\ tn]\!] \Longrightarrow (throw\ a'\ x1,\ s1){::}\preceq(G,\ L)$
⟨*proof*⟩

lemma *Try-lemma*: $[\![G \vdash obj\text{-}ty\ (the\ (globs\ s1'\ (Heap\ a))) \preceq\ Class\ tn$;
$(Some\ (Xcpt\ (Loc\ a)),\ s1'){::}\preceq(G,\ L)$; $wf\text{-}prog\ G]\!]$
$\Longrightarrow Norm\ (lupd(vn \mapsto Addr\ a)\ s1'){::}\preceq(G,\ L(vn \mapsto Class\ tn))$
⟨*proof*⟩

lemma *Fin-lemma*:
$[\![G \vdash Norm\ s1\ -c2\to\ (x2,s2)$; $wf\text{-}prog\ G$; $(Some\ a,\ s1){::}\preceq(G,\ L)$; $(x2,s2){::}\preceq(G,\ L)$;
  $dom\ (locals\ s1) \subseteq dom\ (locals\ s2)]\!]$
$\Longrightarrow (abrupt\text{-}if\ True\ (Some\ a)\ x2,\ s2){::}\preceq(G,\ L)$
⟨*proof*⟩

lemma *FVar-lemma1*:
$[\![table\text{-}of\ (DeclConcepts.fields\ G\ statC)\ (fn,\ statDeclC) = Some\ f$;
  $x2 = None \longrightarrow G,s2 \vdash a{::}\preceq\ Class\ statC$; $wf\text{-}prog\ G$; $G \vdash statC \preceq_C\ statDeclC$;
  $statDeclC \neq Object$;
  $class\ G\ statDeclC = Some\ y$; $(x2,s2){::}\preceq(G,\ L)$; $s1 \leq| s2$;
  $inited\ statDeclC\ (globs\ s1)$;
  $(if\ static\ f\ then\ id\ else\ np\ a)\ x2 = None]\!]$
$\Longrightarrow$
  $\exists obj.\ globs\ s2\ (if\ static\ f\ then\ Inr\ statDeclC\ else\ Inl\ (the\text{-}Addr\ a))$
    $= Some\ obj\ \wedge$
  $var\text{-}tys\ G\ (tag\ obj)\ (if\ static\ f\ then\ Inr\ statDeclC\ else\ Inl(the\text{-}Addr\ a))$
    $(Inl(fn,statDeclC)) = Some\ (type\ f)$
⟨*proof*⟩

lemma *FVar-lemma2*: *error-free state*
  $\Longrightarrow$ *error-free*
    (*assign*
      ($\lambda v.$ *supd*
        (*upd-gobj*
          (*if static field then Inr statDeclC*
           else Inl (*the-Addr a*))
          (*Inl* (*fn*, *statDeclC*)) *v*))
      w state)
⟨*proof*⟩

declare *split-paired-All* [*simp del*] *split-paired-Ex* [*simp del*]
declare *if-split* [*split del*] *if-split-asm* [*split del*]
  option.split [*split del*] *option.split-asm* [*split del*]
⟨*ML*⟩

lemma *FVar-lemma*:
$[\![((v,\ f),\ Norm\ s2') = fvar\ statDeclC\ (static\ field)\ fn\ a\ (x2,\ s2)$;
  $G \vdash statC \preceq_C\ statDeclC$;
  $table\text{-}of\ (DeclConcepts.fields\ G\ statC)\ (fn,\ statDeclC) = Some\ field$;
  $wf\text{-}prog\ G$;
  $x2 = None \longrightarrow G,s2 \vdash a{::}\preceq Class\ statC$;

statDeclC ≠ *Object*; *class G statDeclC* = *Some y*;
(*x2*, *s2*)::≼(*G*, *L*); *s1*≤|*s2*; *inited statDeclC* (*globs s1*)⟧ ⟹
G,*s2*′⊢*v*::≼*type field* ∧ *s2*′≤|*f*≼*type field*::≼(*G*, *L*)
⟨*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*⟩

lemma *AVar-lemma1*: ⟦*globs s* (*Inl a*) = *Some obj*;*tag obj*=*Arr ty i*;
the-Intg i′ *in-bounds i*; *wf-prog G*; *G*⊢*ty*.[]≼*Tb*.[]; *Norm s*::≼(*G*, *L*)
⟧ ⟹ *G*,*s*⊢*the* ((*values obj*) (*Inr* (*the-Intg i*′)))::≼*Tb*
⟨*proof*⟩

lemma *obj-split*: ∃ *t vs*. *obj* = (|*tag*=*t*,*values*=*vs*|)
⟨*proof*⟩

lemma *AVar-lemma2*: *error-free state*
⟹ *error-free*
(*assign*
(λ*v* (*x*, *s*′).
((*raise-if* (¬ *G*,*s*′⊢*v fits T*) *ArrStore*) *x*,
upd-gobj (*Inl a*) (*Inr* (*the-Intg i*)) *v s*′))
w state)
⟨*proof*⟩

lemma *AVar-lemma*: ⟦*wf-prog G*; *G*⊢(*x1*, *s1*) −*e2*−≻*i*→ (*x2*, *s2*);
((*v*,*f*), *Norm s2*′) = *avar G i a* (*x2*, *s2*); *x1* = *None* ⟶ *G*,*s1*⊢*a*::≼*Ta*.[];
(*x2*, *s2*)::≼(*G*, *L*); *s1*≤|*s2*⟧ ⟹ *G*,*s2*′⊢*v*::≼*Ta* ∧ *s2*′≤|*f*≼*Ta*::≼(*G*, *L*)
⟨*proof*⟩

Call

lemma *conforms-init-lvars-lemma*: ⟦*wf-prog G*;
wf-mhead G P sig mh;
list-all2 (*conf G s*) *pvs pTsa*; *G*⊢*pTsa*[≼](*parTs sig*)⟧ ⟹
G,*s*⊢*Map.empty* (*pars mh*[↦]*pvs*)
[∼::≼](*table-of lvars*)(*pars mh*[↦]*parTs sig*)
⟨*proof*⟩

lemma *lconf-map-lname* [*simp*]:
G,*s*⊢(*case-lname l1 l2*)[::≼](*case-lname L1 L2*)
=
(*G*,*s*⊢*l1*[::≼]*L1* ∧ *G*,*s*⊢(λ*x*::*unit* . *l2*)[::≼](λ*x*::*unit*. *L2*))
⟨*proof*⟩

lemma *wlconf-map-lname* [*simp*]:
G,*s*⊢(*case-lname l1 l2*)[∼::≼](*case-lname L1 L2*)
=
(*G*,*s*⊢*l1*[∼::≼]*L1* ∧ *G*,*s*⊢(λ*x*::*unit* . *l2*)[∼::≼](λ*x*::*unit*. *L2*))
⟨*proof*⟩

lemma *lconf-map-ename* [*simp*]:
G,*s*⊢(*case-ename l1 l2*)[::≼](*case-ename L1 L2*)
=
(*G*,*s*⊢*l1*[::≼]*L1* ∧ *G*,*s*⊢(λ*x*::*unit*. *l2*)[::≼](λ*x*::*unit*. *L2*))
⟨*proof*⟩

```
lemma wlconf-map-ename [simp]:
  G,s⊢(case-ename l1 l2)[∼::≼](case-ename L1 L2)
   =
  (G,s⊢l1[∼::≼]L1 ∧ G,s⊢(λx::unit. l2)[∼::≼](λx::unit. L2))
⟨proof⟩
```

```
lemma defval-conf1 [rule-format (no-asm), elim]:
  is-type G T ⟶ (∃ v∈Some (default-val T): G,s⊢v::≼T)
⟨proof⟩

lemma np-no-jump: x≠Some (Jump j) ⟹ (np a') x ≠ Some (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⊢pTsa[≼](parTs sig);
  (x, s)::≼(G, L);
  methd G declC sig = Some dm;
  isrtype G statT;
  G⊢invC≼C declC;
  G,s⊢a'::≼RefT statT;
  invmode (mhd sm) e = IntVir ⟶ a' ≠ Null;
  invmode (mhd sm) e ≠ IntVir ⟶
      (∃ statC. statT=ClassT statC ∧ G⊢statC≼C declC)
    ∨ (∀ statC. statT≠ClassT statC ∧ declC=Object);
  invC  = invocation-class (invmode (mhd sm) e) s a' statT;
  declC = invocation-declclass G (invmode (mhd sm) e) s a' statT sig;
  x≠Some (Jump Ret)
 ⟧ ⟹
  init-lvars G declC sig (invmode (mhd sm) e) a'
  pvs (x,s)::≼(G,λ k.
                (case k of
                   EName e ⇒ (case e of
                                VNam v
                                ⇒ ((table-of (lcls (mbody (mthd dm))))
                                      (pars (mthd dm)[↦]parTs sig)) v
                              | Res ⇒ Some (resTy (mthd dm)))
                 | This ⇒ if (is-static (mthd sm))
                              then None else Some (Class declC)))
⟨proof⟩
declare split-paired-All [simp] split-paired-Ex [simp]
declare 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: ¬ stat ⟶ a≠Null and
   conform-a: G,(store s)⊢a::≼ Class statC and
```

conform-s: $s::\preceq(G, L)$ **and**
normal-s: *normal s* **and**
wt-e: $(|prg=G,cls=accC,lcl=L|)\vdash e::-Class\ statC$ **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 (¬ is-static f)*
shows *table-of (DeclConcepts.fields G dynC) (fn,declclass f) = Some (fld f)∧*
G⊢Field fn f in dynC dyn-accessible-from accC
⟨*proof*⟩

lemma *error-free-field-access*:
assumes *accfield*: *accfield G accC statC fn = Some (statDeclC, f)* **and**
wt-e: $(|prg = G, cls = accC, lcl = L|)\vdash e::-Class\ statC$ **and**
eval-init: *G⊢Norm s0 −Init statDeclC→ s1* **and**
eval-e: *G⊢s1 −e−≻a→ s2* **and**
conf-s2: $s2::\preceq(G, L)$ **and**
conf-a: *normal s2* ⟹ *G, store s2⊢a::≼Class statC* **and**
fvar: *(v,s2′)=fvar statDeclC (is-static f) fn a s2* **and**
wf: *wf-prog G*
shows *check-field-access G accC statDeclC fn (is-static f) a s2′ = s2′*
⟨*proof*⟩

lemma *call-access-ok*:
assumes *invC-prop*: *G⊢invmode statM e→invC≼statT*
and *wf*: *wf-prog G*
and *wt-e*: $(|prg=G,cls=C,lcl=L|)\vdash e::-RefT\ statT$
and *statM*: *(statDeclT,statM) ∈ mheads G accC statT sig*
and *invC*: *invC = invocation-class (invmode statM e) s a statT*
shows *∃ dynM. dynlookup G statT invC sig = Some dynM ∧*
G⊢Methd sig dynM in invC dyn-accessible-from accC
⟨*proof*⟩

lemma *error-free-call-access*:
assumes
eval-args: *G⊢s1 −args≐≻vs→ s2* **and**
wt-e: $(|prg = G, cls = accC, lcl = L|)\vdash e::-(RefT\ statT)$ **and**
statM: $max\text{-}spec\ G\ accC\ statT\ (|name = mn, parTs = pTs|)$
$= \{((statDeclT, statM), pTs')\}$ **and**
conf-s2: $s2::\preceq(G, L)$ **and**
conf-a: *normal s1* ⟹ *G, store s1⊢a::≼RefT statT* **and**
invProp: *normal s3* ⟹
G⊢invmode statM e→invC≼statT **and**
s3: $s3=init\text{-}lvars\ G\ invDeclC\ (|name = mn, parTs = pTs'|)$
(invmode statM e) a vs s2 **and**
invC: *invC = invocation-class (invmode statM e) (store s2) a statT***and**
invDeclC: *invDeclC = invocation-declclass G (invmode statM e) (store s2)*
$a\ statT\ (|name = mn, parTs = pTs'|)$ **and**
wf: *wf-prog G*
shows $check\text{-}method\text{-}access\ G\ accC\ statT\ (invmode\ statM\ e)\ (|name=mn,parTs=pTs'|)\ a\ s3$
$= s3$
⟨*proof*⟩

lemma *map-upds-eq-length-append-simp*:
$\bigwedge tab\ qs.\ length\ ps = length\ qs \Longrightarrow\ tab(ps[\mapsto]qs@zs) = tab(ps[\mapsto]qs)$
⟨*proof*⟩

lemma *map-upds-upd-eq-length-simp*:
$\bigwedge tab\ qs\ x\ y.\ length\ ps = length\ qs$

$\Longrightarrow tab(ps[\mapsto]qs,\ x\mapsto y) = tab(ps@[x][\mapsto]qs@[y])$
⟨*proof*⟩

lemma *map-upd-cong*: $tab{=}tab' \Longrightarrow tab(x\mapsto y) = tab'(x\mapsto y)$
⟨*proof*⟩

lemma *map-upd-cong-ext*: $tab\ z{=}tab'\ z \Longrightarrow (tab(x\mapsto y))\ z = (tab'(x\mapsto y))\ z$
⟨*proof*⟩

lemma *map-upds-cong*: $tab{=}tab' \Longrightarrow tab(xs[\mapsto]ys) = tab'(xs[\mapsto]ys)$
⟨*proof*⟩

lemma *map-upds-cong-ext*:
$\bigwedge tab\ tab'\ ys.\ tab\ z{=}tab'\ z \Longrightarrow (tab(xs[\mapsto]ys))\ z = (tab'(xs[\mapsto]ys))\ z$
⟨*proof*⟩

lemma *map-upd-override*: $(tab(x\mapsto y))\ x = (tab'(x\mapsto y))\ x$
⟨*proof*⟩

lemma *map-upds-eq-length-suffix*: $\bigwedge tab\ qs.$
$length\ ps = length\ qs \Longrightarrow tab(ps@xs[\mapsto]qs) = tab(ps[\mapsto]qs,\ xs[\mapsto][])$
⟨*proof*⟩

lemma *map-upds-upds-eq-length-prefix-simp*:
$\bigwedge tab\ qs.\ length\ ps = length\ qs$
$\Longrightarrow tab(ps[\mapsto]qs,\ xs[\mapsto]ys) = tab(ps@xs[\mapsto]qs@ys)$
⟨*proof*⟩

lemma *map-upd-cut-irrelevant*:
$[\![(tab(x\mapsto y))\ vn = Some\ el;\ (tab'(x\mapsto y))\ vn = None]\!]$
$\Longrightarrow tab\ vn = Some\ el$
⟨*proof*⟩

lemma *map-upd-Some-expand*:
$[\![tab\ vn = Some\ z]\!]$
$\Longrightarrow \exists\ z.\ (tab(x\mapsto y))\ vn = Some\ z$
⟨*proof*⟩

lemma *map-upds-Some-expand*:
$\bigwedge tab\ ys\ z.\ [\![tab\ vn = Some\ z]\!]$
$\Longrightarrow \exists\ z.\ (tab(xs[\mapsto]ys))\ vn = Some\ z$
⟨*proof*⟩

lemma *map-upd-Some-swap*:
$(tab(r\mapsto w,\ u\mapsto v))\ vn = Some\ z \Longrightarrow \exists\ z.\ (tab(u\mapsto v,\ r\mapsto w))\ vn = Some\ z$
⟨*proof*⟩

lemma *map-upd-None-swap*:
$(tab(r\mapsto w,\ u\mapsto v))\ vn = None \Longrightarrow (tab(u\mapsto v,\ r\mapsto w))\ vn = None$
⟨*proof*⟩

lemma *map-eq-upd-eq*: $tab\ vn = tab'\ vn \Longrightarrow (tab(x\mapsto y))\ vn = (tab'(x\mapsto y))\ vn$
⟨*proof*⟩

lemma *map-upd-in-expansion-map-swap*:

$[\![(tab(x \mapsto y))\ vn = Some\ z; tab\ vn \neq Some\ z]\!]$
$\Longrightarrow\ (tab'(x \mapsto y))\ vn = Some\ z$
⟨proof⟩

lemma *map-upds-in-expansion-map-swap*:
$\bigwedge tab\ tab'\ ys\ z.\ [\![(tab(xs[\mapsto]ys))\ vn = Some\ z; tab\ vn \neq Some\ z]\!]$
$\Longrightarrow\ (tab'(xs[\mapsto]ys))\ vn = Some\ z$
⟨proof⟩

lemma *map-upds-Some-swap*:
assumes *r-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*:
$\bigwedge\ tab\ tab'\ ys.\ [\![(tab(xs[\mapsto]ys))\ vn = Some\ el;\ (tab'(xs[\mapsto]ys))\ vn = None]\!]$
$\Longrightarrow tab\ vn = Some\ el$
⟨proof⟩

lemma *dom-vname-split*:
$dom\ (case\text{-}lname\ (case\text{-}ename\ (tab(x \mapsto y,\ xs[\mapsto]ys))\ a)\ b)$
$= dom\ (case\text{-}lname\ (case\text{-}ename\ (tab(x \mapsto y))\ a)\ b)\ \cup$
$dom\ (case\text{-}lname\ (case\text{-}ename\ (tab(xs[\mapsto]ys))\ a)\ b)$
(**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*: $\bigwedge\ tab\ ys.\ length\ xs = length\ ys$
$\Longrightarrow dom\ (tab(xs[\mapsto]ys)) = dom\ tab \cup set\ xs$
⟨proof⟩

lemma *dom-case-ename-None-simp*:
$dom\ (case\text{-}ename\ vname\text{-}tab\ None) = VNam\ `\ (dom\ vname\text{-}tab)$
⟨proof⟩

lemma *dom-case-ename-Some-simp*:
$dom\ (case\text{-}ename\ vname\text{-}tab\ (Some\ a)) = VNam\ `\ (dom\ vname\text{-}tab) \cup \{Res\}$
⟨proof⟩

lemma *dom-case-lname-None-simp*:
$dom\ (case\text{-}lname\ ename\text{-}tab\ None) = EName\ `\ (dom\ ename\text{-}tab)$
⟨proof⟩

lemma *dom-case-lname-Some-simp*:
$dom\ (case\text{-}lname\ ename\text{-}tab\ (Some\ a)) = EName\ `\ (dom\ ename\text{-}tab) \cup \{This\}$

⟨*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 ‘ g ‘ A = ($f \circ g$) ‘ A
⟨*proof*⟩

lemma *dom-locals-init-lvars*:
assumes *m*: *m*=(*mthd* (*the* (*methd G C sig*)))
assumes *len*: *length* (*pars m*) = *length pvs*
shows *dom* (*locals* (*store* (*init-lvars G C sig* (*invmode m e*) *a pvs s*)))
= *parameters m*
⟨*proof*⟩

lemma *da-e2-BinOp*:
assumes *da*: (|*prg*=*G*,*cls*=*accC*,*lcl*=*L*|)
⊢*dom* (*locals* (*store s0*)) »⟨*BinOp binop e1 e2*⟩$_e$» *A*
and *wt-e1*: (|*prg*=*G*,*cls*=*accC*,*lcl*=*L*|)⊢*e1*::−*e1T*
and *wt-e2*: (|*prg*=*G*,*cls*=*accC*,*lcl*=*L*|)⊢*e2*::−*e2T*
and *wt-binop*: *wt-binop G binop e1T e2T*
and *conf-s0*: *s0*::⪯(*G*,*L*)
and *normal-s1*: *normal s1*
and *eval-e1*: *G*⊢*s0* −*e1*−≻*v1*→ *s1*
and *conf-v1*: *G*,*store s1*⊢*v1*::⪯*e1T*
and *wf*: *wf-prog G*
shows ∃ *E2*. (|*prg*=*G*,*cls*=*accC*,*lcl*=*L*|)⊢ *dom* (*locals* (*store s1*))
»(*if need-second-arg binop v1 then* ⟨*e2*⟩$_e$ *else* ⟨*Skip*⟩$_s$)» *E2*
⟨*proof*⟩

main proof of type safety

lemma *eval-type-sound*:
assumes *eval*: *G*⊢*s0* −*t*≻→ (*v*,*s1*)
and *wt*: (|*prg*=*G*,*cls*=*accC*,*lcl*=*L*|)⊢*t*::*T*
and *da*: (|*prg*=*G*,*cls*=*accC*,*lcl*=*L*|)⊢*dom* (*locals* (*store s0*))»*t*»*A*
and *wf*: *wf-prog G*
and *conf-s0*: *s0*::⪯(*G*,*L*)
shows *s1*::⪯(*G*,*L*) ∧ (*normal s1* ⟶ *G*,*L*,*store s1*⊢*t*≻*v*::⪯*T*) ∧
(*error-free s0* = *error-free s1*)
⟨*proof*⟩

corollary *eval-type-soundE* [*consumes 5*]:
assumes *eval*: *G*⊢*s0* −*t*≻→ (*v*, *s1*)
and *conf*: *s0*::⪯(*G*, *L*)
and *wt*: (|*prg* = *G*, *cls* = *accC*, *lcl* = *L*|)⊢*t*::*T*
and *da*: (|*prg* = *G*, *cls* = *accC*, *lcl* = *L*|)⊢ *dom* (*locals* (*snd s0*)) »*t*» *A*
and *wf*: *wf-prog G*
and *elim*: ⟦*s1*::⪯(*G*, *L*); *normal s1* ⟹ *G*,*L*,*snd s1*⊢*t*≻*v*::⪯*T*;
error-free s0 = *error-free s1*⟧ ⟹ *P*
shows *P*
⟨*proof*⟩

corollary *eval-ts*:

$[\![G\vdash s\ -e-\succ v\ \to\ s';\ \textit{wf-prog}\ G;\ s::\preceq(G,L);\ (\!|prg=G,cls=C,lcl=L|\!)\vdash e::-T;$
$(\!|prg=G,cls=C,lcl=L|\!)\vdash dom\ (locals\ (store\ s))\gg In1l\ e\gg A]\!]$
$\Longrightarrow\ s'::\preceq(G,L)\ \wedge\ (normal\ s'\ \longrightarrow\ G,store\ s'\vdash v::\preceq T)\ \wedge$
$(\textit{error-free}\ s\ =\ \textit{error-free}\ s')$
⟨*proof*⟩

corollary *evals-ts*:
$[\![G\vdash s\ -es\doteq\succ vs\to\ s';\ \textit{wf-prog}\ G;\ s::\preceq(G,L);\ (\!|prg=G,cls=C,lcl=L|\!)\vdash es::\doteq Ts;$
$(\!|prg=G,cls=C,lcl=L|\!)\vdash dom\ (locals\ (store\ s))\gg In3\ es\gg A]\!]$
$\Longrightarrow\ s'::\preceq(G,L)\ \wedge\ (normal\ s'\ \longrightarrow\ \textit{list-all2}\ (conf\ G\ (store\ s'))\ vs\ Ts)\ \wedge$
$(\textit{error-free}\ s\ =\ \textit{error-free}\ s')$
⟨*proof*⟩

corollary *evar-ts*:
$[\![G\vdash s\ -v=\succ vf\to\ s';\ \textit{wf-prog}\ G;\ s::\preceq(G,L);\ (\!|prg=G,cls=C,lcl=L|\!)\vdash v::=T;$
$(\!|prg=G,cls=C,lcl=L|\!)\vdash dom\ (locals\ (store\ s))\gg In2\ v\gg A]\!]\ \Longrightarrow$
$s'::\preceq(G,L)\ \wedge\ (normal\ s'\ \longrightarrow\ G,L,(store\ s')\vdash In2\ v\succ In2\ vf::\preceq Inl\ T)\ \wedge$
$(\textit{error-free}\ s\ =\ \textit{error-free}\ s')$
⟨*proof*⟩

theorem *exec-ts*:
$[\![G\vdash s\ -c\to\ s';\ \textit{wf-prog}\ G;\ s::\preceq(G,L);\ (\!|prg=G,cls=C,lcl=L|\!)\vdash c::\surd;$
$(\!|prg=G,cls=C,lcl=L|\!)\vdash dom\ (locals\ (store\ s))\gg In1r\ c\gg A]\!]$
$\Longrightarrow s'::\preceq(G,L)\ \wedge\ (\textit{error-free}\ s\ \longrightarrow\ \textit{error-free}\ s')$
⟨*proof*⟩

lemma *wf-eval-Fin*:
assumes *wf*: $\textit{wf-prog}\ G$
and *wt-c1*: $(\!|prg\ =\ G,\ cls\ =\ C,\ lcl\ =\ L|\!)\vdash In1r\ c1::Inl\ (PrimT\ Void)$
and *da-c1*: $(\!|prg=G,cls=C,lcl=L|\!)\vdash dom\ (locals\ (store\ (Norm\ s0)))\gg In1r\ c1\gg A$
and *conf-s0*: $Norm\ s0::\preceq(G,\ L)$
and *eval-c1*: $G\vdash Norm\ s0\ -c1\to\ (x1,s1)$
and *eval-c2*: $G\vdash Norm\ s1\ -c2\to\ s2$
and *s3*: $s3=abupd\ (\textit{abrupt-if}\ (x1\neq None)\ x1)\ s2$
shows $G\vdash Norm\ s0\ -c1\ Finally\ c2\to\ s3$
⟨*proof*⟩

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*: $G\vdash s0\ -t\succ\to\ (v,s1)$
and *wt*: $(\!|prg=G,cls=accC,lcl=L|\!)\vdash t::T$
and *da*: $(\!|prg=G,cls=accC,lcl=L|\!)\vdash dom\ (locals\ (store\ s0))\gg t\gg A$
and *wf*: $\textit{wf-prog}\ G$
and *abrupt*: $\bigwedge\ s\ t\ abr\ L\ accC\ T\ A.$
$[\![(\!|prg=G,cls=accC,lcl=L|\!)\vdash t::T;$
$(\!|prg=G,cls=accC,lcl=L|\!)\vdash dom\ (locals\ (store\ (Some\ abr,s)))\gg t\gg A$
$]\!]\ \Longrightarrow\ P\ L\ accC\ (Some\ abr,\ s)\ t\ (undefined3\ t)\ (Some\ abr,\ s)$

and *skip*: ⋀ *s L accC*. *P L accC* (*Norm s*) $\langle Skip\rangle_s$ ◇ (*Norm s*)
and *expr*: ⋀ *e s0 s1 v L accC eT E*.
⟦⦇*prg*=*G*,*cls*=*accC*,*lcl*=*L*⦈⊢*e*::−*eT*;
⦇*prg*=*G*,*cls*=*accC*,*lcl*=*L*⦈
⊢*dom* (*locals* (*store* ((*Norm s0*)::*state*)))»$\langle e\rangle_e$»*E*;
P L accC (*Norm s0*) $\langle e\rangle_e$ $\lfloor v\rfloor_e$ *s1*⟧
⟹ *P L accC* (*Norm s0*) $\langle Expr\ e\rangle_s$ ◇ *s1*
and *lab*: ⋀ *c l s0 s1 L accC C*.
⟦⦇*prg*=*G*,*cls*=*accC*,*lcl*=*L*⦈⊢*c*::√;
⦇*prg*=*G*,*cls*=*accC*, *lcl*=*L*⦈
⊢*dom* (*locals* (*store* ((*Norm s0*)::*state*)))»$\langle c\rangle_s$»*C*;
P L accC (*Norm s0*) $\langle c\rangle_s$ ◇ *s1*⟧
⟹ *P L accC* (*Norm s0*) $\langle l\cdot\ c\rangle_s$ ◇ (*abupd* (*absorb l*) *s1*)
and *comp*: ⋀ *c1 c2 s0 s1 s2 L accC C1*.
⟦*G*⊢*Norm s0* −*c1* → *s1*;*G*⊢*s1* −*c2* → *s2*;
⦇*prg*=*G*,*cls*=*accC*,*lcl*=*L*⦈⊢*c1*::√;
⦇*prg*=*G*,*cls*=*accC*,*lcl*=*L*⦈⊢*c2*::√;
⦇*prg*=*G*,*cls*=*accC*,*lcl*=*L*⦈⊢
dom (*locals* (*store* ((*Norm s0*)::*state*))) »$\langle c1\rangle_s$» *C1*;
P L accC (*Norm s0*) $\langle c1\rangle_s$ ◇ *s1*;
⋀ *Q*. ⟦*normal s1*;
⋀ *C2*.⟦⦇*prg*=*G*,*cls*=*accC*,*lcl*=*L*⦈
⊢*dom* (*locals* (*store s1*)) »$\langle c2\rangle_s$» *C2*;
P L accC s1 $\langle c2\rangle_s$ ◇ *s2*⟧ ⟹ *Q*
⟧ ⟹ *Q*
⟧⟹ *P L accC* (*Norm s0*) $\langle c1;;\ c2\rangle_s$ ◇ *s2*
and *if*: ⋀ *b c1 c2 e s0 s1 s2 L accC E*.
⟦*G*⊢*Norm s0* −*e*−≻*b*→ *s1*;
G⊢*s1* −(*if the-Bool b then c1 else c2*)→ *s2*;
⦇*prg*=*G*,*cls*=*accC*,*lcl*=*L*⦈⊢*e*::−*PrimT Boolean*;
⦇*prg*=*G*, *cls*=*accC*, *lcl*=*L*⦈⊢(*if the-Bool b then c1 else c2*)::√;
⦇*prg*=*G*,*cls*=*accC*,*lcl*=*L*⦈⊢
dom (*locals* (*store* ((*Norm s0*)::*state*))) »$\langle e\rangle_e$» *E*;
P L accC (*Norm s0*) $\langle e\rangle_e$ $\lfloor b\rfloor_e$ *s1*;
⋀ *Q*. ⟦*normal s1*;
⋀ *C*. ⟦⦇*prg*=*G*,*cls*=*accC*,*lcl*=*L*⦈⊢ (*dom* (*locals* (*store s1*)))
»$\langle if\ the\text{-}Bool\ b\ then\ c1\ else\ c2\rangle_s$» *C*;
P L accC s1 $\langle if\ the\text{-}Bool\ b\ then\ c1\ else\ c2\rangle_s$ ◇ *s2*
⟧ ⟹ *Q*
⟧ ⟹ *Q*
⟧ ⟹ *P L accC* (*Norm s0*) $\langle If(e)\ c1\ Else\ c2\rangle_s$ ◇ *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 *evarn* :: [*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

  $G\vdash s\ -c\ \ -n\rightarrow\ \ s' \equiv G\vdash s\ -In1r\ \ c\succ-n\rightarrow (\Diamond\ ,\ s')$
| $G\vdash s\ -e-\succ v\ \ -n\rightarrow\ \ s' \equiv G\vdash s\ -In1l\ e\succ-n\rightarrow (In1\ v\ ,\ s')$
| $G\vdash s\ -e{=}\succ vf\ -n\rightarrow\ \ s' \equiv G\vdash s\ -In2\ \ e\succ-n\rightarrow (In2\ vf,\ s')$
| $G\vdash s\ -e\doteq\succ v\ \ -n\rightarrow\ \ s' \equiv G\vdash s\ -In3\ \ e\succ-n\rightarrow (In3\ v\ ,\ s')$

— propagation of abrupt completion

| *Abrupt*: $G\vdash(Some\ xc,s)\ -t\succ-n\rightarrow (undefined3\ t,(Some\ xc,s))$

— evaluation of variables

| *LVar*: $G\vdash Norm\ s\ -LVar\ vn{=}\succ lvar\ vn\ s{-}n\rightarrow\ Norm\ s$

| *FVar*: $[\![G\vdash Norm\ s0\ -Init\ statDeclC-n\rightarrow\ s1;\ G\vdash s1\ -e-\succ a-n\rightarrow\ s2;$
$(v,s2') = fvar\ statDeclC\ stat\ fn\ a\ s2;$
$s3 = check\text{-}field\text{-}access\ G\ accC\ statDeclC\ fn\ stat\ a\ s2']\!]\Longrightarrow$
$G\vdash Norm\ s0\ -\{accC,statDeclC,stat\}e..fn{=}\succ v-n\rightarrow\ s3$

| *AVar*: $[\![G\vdash\ Norm\ s0\ -e1-\succ a-n\rightarrow\ s1\ ;\ G\vdash s1\ -e2-\succ i-n\rightarrow\ s2;$
$(v,s2') = avar\ G\ i\ a\ s2]\!]\Longrightarrow$
$G\vdash Norm\ s0\ -e1.[e2]{=}\succ v-n\rightarrow\ s2'$

— evaluation of expressions

| *NewC*: $[\![G\vdash Norm\ s0\ -Init\ C-n\rightarrow\ s1;$
$G\vdash\quad s1\ -halloc\ (CInst\ C)\succ a\rightarrow\ s2]\!]\Longrightarrow$
$G\vdash Norm\ s0\ -NewC\ C-\succ Addr\ a-n\rightarrow\ s2$

| *NewA*: $[\![G\vdash Norm\ s0\ -init\text{-}comp\text{-}ty\ T-n\rightarrow\ s1;\ G\vdash s1\ -e-\succ i'-n\rightarrow\ s2;$
$G\vdash abupd\ (check\text{-}neg\ i')\ s2\ -halloc\ (Arr\ T\ (the\text{-}Intg\ i'))\succ a\rightarrow\ s3]\!]\Longrightarrow$
$G\vdash Norm\ s0\ -New\ T[e]-\succ Addr\ a-n\rightarrow\ s3$

| *Cast*: $[\![G\vdash Norm\ s0\ -e-\succ v-n\rightarrow\ s1;$
$s2 = abupd\ (raise\text{-}if\ (\neg G,snd\ s1\vdash v\ fits\ T)\ ClassCast)\ s1]\!]\Longrightarrow$
$G\vdash Norm\ s0\ -Cast\ T\ e-\succ v-n\rightarrow\ s2$

| *Inst*: $[\![G\vdash Norm\ s0\ -e-\succ v-n\rightarrow\ s1;$
$b = (v\neq Null\ \wedge\ G,store\ s1\vdash v\ fits\ RefT\ T)]\!]\Longrightarrow$
$G\vdash Norm\ s0\ -e\ InstOf\ T-\succ Bool\ b-n\rightarrow\ s1$

| *Lit*: $G\vdash Norm\ s\ -Lit\ v-\succ v-n\rightarrow\ Norm\ s$

| *UnOp*: $[\![G\vdash Norm\ s0\ -e-\succ v-n\rightarrow\ s1]\!]$
$\Longrightarrow\ G\vdash Norm\ s0\ -UnOp\ unop\ e-\succ(eval\text{-}unop\ unop\ v)-n\rightarrow\ s1$

| *BinOp*: $[\![G\vdash Norm\ s0\ -e1-\succ v1-n\rightarrow\ s1;$
$G\vdash s1\ -(if\ need\text{-}second\text{-}arg\ binop\ v1\ then\ (In1l\ e2)\ else\ (In1r\ Skip))$
$\succ-n\rightarrow\ (In1\ v2,s2)]\!]$
$\Longrightarrow\ G\vdash Norm\ s0\ -BinOp\ binop\ e1\ e2-\succ(eval\text{-}binop\ binop\ v1\ v2)-n\rightarrow\ s2$

| *Super*: $G\vdash Norm\ s\ -Super-\succ val\text{-}this\ s-n\rightarrow\ Norm\ s$

| *Acc*: $[\![G\vdash Norm\ s0\ -va{=}\succ(v,f)-n\rightarrow\ s1]\!]\Longrightarrow$
$G\vdash Norm\ s0\ -Acc\ va-\succ v-n\rightarrow\ s1$

| *Ass*: $[\![G\vdash Norm\ s0\ -va{=}\succ(w,f)-n\rightarrow\ s1;$
$G\vdash\quad s1\ -e-\succ v\quad -n\rightarrow\ s2]\!]\Longrightarrow$
$G\vdash Norm\ s0\ -va{:=}e-\succ v-n\rightarrow\ assign\ f\ v\ s2$

| *Cond*: $[\![G\vdash Norm\ s0\ -e0-\succ b-n\rightarrow\ s1;$
$G\vdash\quad s1\ -(if\ the\text{-}Bool\ b\ then\ e1\ else\ e2)-\succ v-n\rightarrow\ s2]\!]\Longrightarrow$
$G\vdash Norm\ s0\ -e0\ ?\ e1\ :\ e2-\succ v-n\rightarrow\ s2$

| *Call*:
$[\![G\vdash Norm\ s0\ -e-\succ a'-n\rightarrow\ s1;\ G\vdash s1\ -args\doteq\succ vs-n\rightarrow\ s2;$
$D = invocation\text{-}declclass\ G\ mode\ (store\ s2)\ a'\ statT\ (\!|name{=}mn,parTs{=}pTs|\!);$
$s3{=}init\text{-}lvars\ G\ D\ (\!|name{=}mn,parTs{=}pTs|\!)\ mode\ a'\ vs\ s2;$
$s3' = check\text{-}method\text{-}access\ G\ accC\ statT\ mode\ (\!|name{=}mn,parTs{=}pTs|\!)\ a'\ s3;$

G⊢s3′−Methd D (|name=mn,parTs=pTs|)−≻v−n→ s4
⟧
⟹
G⊢Norm s0 −{accC,statT,mode}e·mn({pTs}args)−≻v−n→ (restore-lvars s2 s4)

| Methd:⟦G⊢Norm s0 −body G D sig−≻v−n→ s1⟧ ⟹
G⊢Norm s0 −Methd D sig−≻v−Suc n→ s1

| Body: ⟦G⊢Norm s0−Init D−n→ s1; G⊢s1 −c−n→ s2;
s3 = (if (∃ l. abrupt s2 = Some (Jump (Break l)) ∨
abrupt s2 = Some (Jump (Cont l)))
then abupd (λ x. Some (Error CrossMethodJump)) s2
else s2)⟧⟹
G⊢Norm s0 −Body D c
−≻the (locals (store s2) Result)−n→abupd (absorb Ret) s3

— evaluation of expression lists

| Nil:
G⊢Norm s0 −[]≐≻[]−n→ Norm s0

| Cons: ⟦G⊢Norm s0 −e −≻ v −n→ s1;
G⊢ s1 −es≐≻vs−n→ s2⟧ ⟹
G⊢Norm s0 −e#es≐≻v#vs−n→ s2

— execution of statements

| Skip: G⊢Norm s −Skip−n→ Norm s

| Expr: ⟦G⊢Norm s0 −e−≻v−n→ s1⟧ ⟹
G⊢Norm s0 −Expr e−n→ s1

| Lab: ⟦G⊢Norm s0 −c −n→ s1⟧ ⟹
G⊢Norm s0 −l• c−n→ abupd (absorb l) s1

| Comp: ⟦G⊢Norm s0 −c1 −n→ s1;
G⊢ s1 −c2 −n→ s2⟧ ⟹
G⊢Norm s0 −c1;; c2−n→ s2

| If: ⟦G⊢Norm s0 −e−≻b−n→ s1;
G⊢ s1−(if the-Bool b then c1 else c2)−n→ s2⟧ ⟹
G⊢Norm s0 −If(e) c1 Else c2 −n→ s2

| Loop: ⟦G⊢Norm s0 −e−≻b−n→ s1;
if the-Bool b
then (G⊢s1 −c−n→ s2 ∧
G⊢(abupd (absorb (Cont l)) s2) −l• While(e) c−n→ s3)
else s3 = s1⟧ ⟹
G⊢Norm s0 −l• While(e) c−n→ s3

| Jmp: G⊢Norm s −Jmp j−n→ (Some (Jump j), s)

| Throw:⟦G⊢Norm s0 −e−≻a′−n→ s1⟧ ⟹
G⊢Norm s0 −Throw e−n→ abupd (throw a′) s1

| Try: ⟦G⊢Norm s0 −c1−n→ s1; G⊢s1 −sxalloc→ s2;
if G,s2⊢catch tn then G⊢new-xcpt-var vn s2 −c2−n→ s3 else s3 = s2⟧
⟹

```
              G⊢Norm s0 −Try c1 Catch(tn vn) c2−n→ s3

| Fin: ⟦G⊢Norm s0 −c1−n→ (x1,s1);
        G⊢Norm s1 −c2−n→ s2;
        s3=(if (∃ err. x1=Some (Error err))
             then (x1,s1)
             else abupd (abrupt-if (x1≠None) x1) s2)⟧ ⟹
             G⊢Norm s0 −c1 Finally c2−n→ s3

| Init: ⟦the (class G C) = c;
        if inited C (globs s0) then s3 = Norm s0
        else (G⊢Norm (init-class-obj G C s0)
                −(if C = Object then Skip else Init (super c))−n→ s1 ∧
              G⊢set-lvars Map.empty s1 −init c−n→ s2 ∧
              s3 = restore-lvars s1 s2)⟧
        ⟹
              G⊢Norm s0 −Init C−n→ 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*]
⟨*ML*⟩

inductive-cases *evaln-cases*: G⊢s −t≻−n→ (v, s′)

inductive-cases *evaln-elim-cases*:

```
G⊢(Some xc, s) −t                              ≻−n→ (v, s′)
G⊢Norm s −In1r Skip                            ≻−n→ (x, s′)
G⊢Norm s −In1r (Jmp j)                         ≻−n→ (x, s′)
G⊢Norm s −In1r (l• c)                          ≻−n→ (x, s′)
G⊢Norm s −In3 ([])                             ≻−n→ (v, s′)
G⊢Norm s −In3 (e#es)                           ≻−n→ (v, s′)
G⊢Norm s −In1l (Lit w)                         ≻−n→ (v, s′)
G⊢Norm s −In1l (UnOp unop e)                   ≻−n→ (v, s′)
G⊢Norm s −In1l (BinOp binop e1 e2)             ≻−n→ (v, s′)
G⊢Norm s −In2 (LVar vn)                        ≻−n→ (v, s′)
G⊢Norm s −In1l (Cast T e)                      ≻−n→ (v, s′)
G⊢Norm s −In1l (e InstOf T)                    ≻−n→ (v, s′)
G⊢Norm s −In1l (Super)                         ≻−n→ (v, s′)
G⊢Norm s −In1l (Acc va)                        ≻−n→ (v, s′)
G⊢Norm s −In1r (Expr e)                        ≻−n→ (x, s′)
G⊢Norm s −In1r (c1;; c2)                       ≻−n→ (x, s′)
G⊢Norm s −In1l (Methd C sig)                   ≻−n→ (x, s′)
G⊢Norm s −In1l (Body D c)                      ≻−n→ (x, s′)
G⊢Norm s −In1l (e0 ? e1 : e2)                  ≻−n→ (v, s′)
G⊢Norm s −In1r (If(e) c1 Else c2)              ≻−n→ (x, s′)
G⊢Norm s −In1r (l• While(e) c)                 ≻−n→ (x, s′)
G⊢Norm s −In1r (c1 Finally c2)                 ≻−n→ (x, s′)
G⊢Norm s −In1r (Throw e)                       ≻−n→ (x, s′)
G⊢Norm s −In1l (NewC C)                        ≻−n→ (v, s′)
G⊢Norm s −In1l (New T[e])                      ≻−n→ (v, s′)
G⊢Norm s −In1l (Ass va e)                      ≻−n→ (v, s′)
G⊢Norm s −In1r (Try c1 Catch(tn vn) c2)        ≻−n→ (x, s′)
G⊢Norm s −In2 ({accC,statDeclC,stat}e..fn)     ≻−n→ (v, s′)
G⊢Norm s −In2 (e1.[e2])                        ≻−n→ (v, s′)
```

$G\vdash Norm\ s\ -In1l\ (\{accC,statT,mode\}e{\cdot}mn(\{pT\}p))\ \succ\!-n\!\to\ (v,\ s')$
$G\vdash Norm\ s\ -In1r\ (Init\ C)\ \succ\!-n\!\to\ (x,\ s')$

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*: $G\vdash s\ -t\succ\!-n\!\to\ (w,s') \Longrightarrow case\ t\ of\ In1\ ec \Rightarrow$
$(case\ ec\ of\ Inl\ e \Rightarrow (\exists\, v.\ w = In1\ v)\ |\ Inr\ c \Rightarrow w = \diamondsuit)$
$|\ In2\ e \Rightarrow (\exists\, v.\ w = In2\ v)\ |\ In3\ e \Rightarrow (\exists\, v.\ w = In3\ v)$
⟨*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*: $G\vdash s\ -In1l\ t\succ\!-n\!\to\ (w,\ s') = (\exists\, v.\ w{=}In1\ v \wedge G\vdash s\ -t\!-\!\succ v\ -n\!\to\ s')$
⟨*proof*⟩

lemma *evaln-var-eq*: $G\vdash s\ -In2\ t\succ\!-n\!\to\ (w,\ s') = (\exists\, vf.\ w{=}In2\ vf \wedge G\vdash s\ -t{=}\!\succ vf\!-n\!\to\ s')$
⟨*proof*⟩

lemma *evaln-exprs-eq*: $G\vdash s\ -In3\ t\succ\!-n\!\to\ (w,\ s') = (\exists\, vs.\ w{=}In3\ vs \wedge G\vdash s\ -t\doteq\!\succ vs\!-n\!\to\ s')$
⟨*proof*⟩

lemma *evaln-stmt-eq*: $G\vdash s\ -In1r\ t\succ\!-n\!\to\ (w,\ s') = (w{=}\diamondsuit \wedge G\vdash s\ -t\ -n\!\to\ s')$
⟨*proof*⟩

⟨*ML*⟩
declare *evaln-AbruptIs* [*intro*!]

lemma *evaln-Callee*: $G\vdash Norm\ s{-}In1l\ (Callee\ l\ e)\succ\!-n\!\to\ (v,s') = False$
⟨*proof*⟩

lemma *evaln-InsInitE*: $G\vdash Norm\ s{-}In1l\ (InsInitE\ c\ e)\succ\!-n\!\to\ (v,s') = False$
⟨*proof*⟩

lemma *evaln-InsInitV*: $G\vdash Norm\ s{-}In2\ (InsInitV\ c\ w)\succ\!-n\!\to\ (v,s') = False$
⟨*proof*⟩

lemma *evaln-FinA*: $G\vdash Norm\ s{-}In1r\ (FinA\ a\ c)\succ\!-n\!\to\ (v,s') = False$
⟨*proof*⟩

lemma *evaln-abrupt-lemma*: $G\vdash s\ -e\succ\!-n\!\to\ (v,s') \Longrightarrow$
$fst\ s = Some\ xc \longrightarrow s' = s \wedge v = undefined3\ e$
⟨*proof*⟩

lemma *evaln-abrupt*:
$\bigwedge s'.\ G\vdash(Some\ xc,s)\ -e\succ\!-n\!\to\ (w,s') = (s' = (Some\ xc,s) \wedge$
$w{=}undefined3\ e \wedge G\vdash(Some\ xc,s)\ -e\succ\!-n\!\to\ (undefined3\ e,(Some\ xc,s)))$
⟨*proof*⟩

⟨*ML*⟩

lemma *evaln-LitI*: $G\vdash s\ -Lit\ v\!-\!\succ(if\ normal\ s\ then\ v\ else\ undefined)\!-n\!\to\ s$
⟨*proof*⟩

lemma *CondI*:
$\bigwedge s1.$ ⟦*G*⊢*s* −*e*−≻*b*−*n*→ *s1*; *G*⊢*s1* −(*if the-Bool b then e1 else e2*)−≻*v*−*n*→ *s2*⟧ ⟹
G⊢*s* −*e* ? *e1* : *e2*−≻(*if normal s1 then v else undefined*)−*n*→ *s2*
⟨*proof*⟩

lemma *evaln-SkipI* [*intro*!]: *G*⊢*s* −*Skip*−*n*→ *s*
⟨*proof*⟩

lemma *evaln-ExprI*: *G*⊢*s* −*e*−≻*v*−*n*→ *s'* ⟹ *G*⊢*s* −*Expr e*−*n*→ *s'*
⟨*proof*⟩

lemma *evaln-CompI*: ⟦*G*⊢*s* −*c1*−*n*→ *s1*; *G*⊢*s1* −*c2*−*n*→ *s2*⟧ ⟹ *G*⊢*s* −*c1*;; *c2*−*n*→ *s2*
⟨*proof*⟩

lemma *evaln-IfI*:
⟦*G*⊢*s* −*e*−≻*v*−*n*→ *s1*; *G*⊢*s1* −(*if the-Bool v then c1 else c2*)−*n*→ *s2*⟧ ⟹
G⊢*s* −*If*(*e*) *c1 Else c2*−*n*→ *s2*
⟨*proof*⟩

lemma *evaln-SkipD* [*dest*!]: *G*⊢*s* −*Skip*−*n*→ *s'* ⟹ *s'* = *s*
⟨*proof*⟩

lemma *evaln-Skip-eq* [*simp*]: *G*⊢*s* −*Skip*−*n*→ *s'* = (*s* = *s'*)
⟨*proof*⟩

evaln implies eval

lemma *evaln-eval*:
assumes *evaln*: *G*⊢*s0* −*t*≻−*n*→ (*v*,*s1*)
shows *G*⊢*s0* −*t*≻→ (*v*,*s1*)
⟨*proof*⟩

lemma *Suc-le-D-lemma*: ⟦*Suc n* <= *m'*; ($\bigwedge$*m*. *n* <= *m* ⟹ *P* (*Suc m*)) ⟧ ⟹ *P m'*
⟨*proof*⟩

lemma *evaln-nonstrict* [*rule-format* (*no-asm*), *elim*]:
G⊢*s* −*t*≻−*n*→ (*w*, *s'*) ⟹ ∀ *m*. *n*≤*m* ⟶ *G*⊢*s* −*t*≻−*m*→ (*w*, *s'*)
⟨*proof*⟩

lemmas *evaln-nonstrict-Suc* = *evaln-nonstrict* [*OF* - *le-refl* [*THEN le-SucI*]]

lemma *evaln-max2*: ⟦*G*⊢*s1* −*t1*≻−*n1*→ (*w1*, *s1'*); *G*⊢*s2* −*t2*≻−*n2*→ (*w2*, *s2'*)⟧ ⟹
G⊢*s1* −*t1*≻−*max n1 n2*→ (*w1*, *s1'*) ∧ *G*⊢*s2* −*t2*≻−*max n1 n2*→ (*w2*, *s2'*)
⟨*proof*⟩

corollary *evaln-max2E* [*consumes 2*]:
⟦*G*⊢*s1* −*t1*≻−*n1*→ (*w1*, *s1'*); *G*⊢*s2* −*t2*≻−*n2*→ (*w2*, *s2'*);
⟦*G*⊢*s1* −*t1*≻−*max n1 n2*→ (*w1*, *s1'*);*G*⊢*s2* −*t2*≻−*max n1 n2*→ (*w2*, *s2'*) ⟧ ⟹ *P* ⟧ ⟹ *P*
⟨*proof*⟩

lemma *evaln-max3*:
⟦*G*⊢*s1* −*t1*≻−*n1*→ (*w1*, *s1'*); *G*⊢*s2* −*t2*≻−*n2*→ (*w2*, *s2'*); *G*⊢*s3* −*t3*≻−*n3*→ (*w3*, *s3'*)⟧ ⟹
G⊢*s1* −*t1*≻−*max* (*max n1 n2*) *n3*→ (*w1*, *s1'*) ∧
G⊢*s2* −*t2*≻−*max* (*max n1 n2*) *n3*→ (*w2*, *s2'*) ∧
G⊢*s3* −*t3*≻−*max* (*max n1 n2*) *n3*→ (*w3*, *s3'*)
⟨*proof*⟩

corollary *evaln-max3E*:

⟦*G*⊢*s1* −*t1*≻−*n1*→ (*w1*, *s1'*); *G*⊢*s2* −*t2*≻−*n2*→ (*w2*, *s2'*); *G*⊢*s3* −*t3*≻−*n3*→ (*w3*, *s3'*);
  ⟦*G*⊢*s1* −*t1*≻−*max* (*max n1 n2*) *n3*→ (*w1*, *s1'*);
   G⊢*s2* −*t2*≻−*max* (*max n1 n2*) *n3*→ (*w2*, *s2'*);
   G⊢*s3* −*t3*≻−*max* (*max n1 n2*) *n3*→ (*w3*, *s3'*)
  ⟧ ⟹ *P*
 ⟧ ⟹ *P*
⟨*proof*⟩

lemma *le-max3I1*: (*n2*::*nat*) ≤ *max n1* (*max n2 n3*)
⟨*proof*⟩

lemma *le-max3I2*: (*n3*::*nat*) ≤ *max n1* (*max n2 n3*)
⟨*proof*⟩

declare [[*simproc del*: *wt-expr wt-var wt-exprs wt-stmt*]]

eval implies evaln

lemma *eval-evaln*:
 assumes *eval*: *G*⊢*s0* −*t*≻→ (*v*,*s1*)
 shows ∃ *n*. *G*⊢*s0* −*t*≻−*n*→ (*v*,*s1*)
⟨*proof*⟩

end

Chapter 21

Trans

theory *Trans* **imports** *Evaln* **begin**

definition
  groundVar :: *var* ⇒ *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* ⇒ *expr*
  where *Ref a* == *Lit* (*Addr a*)

abbreviation
SKIP :: *expr*
where *SKIP* == *Lit Unit*

inductive
step :: [*prog*,*term* × *state*,*term* × *state*] ⇒ *bool* (‹-⊢- ↦1 -›[61,82,82] 81)
for *G* :: *prog*
where

Abrupt: $[\![\forall v.\ t \neq \langle Lit\ v\rangle;$
$\forall\ t.\ t \neq \langle l\cdot\ Skip\rangle;$
$\forall\ C\ vn\ c.\ \ t \neq \langle Try\ Skip\ Catch(C\ vn)\ c\rangle;$
$\forall\ x\ c.\ t \neq \langle Skip\ Finally\ c\rangle \wedge xc \neq Xcpt\ x;$
$\forall\ a\ c.\ t \neq \langle FinA\ a\ c\rangle]\!]$
$\Longrightarrow$
$G\vdash(t,Some\ xc,s) \mapsto 1\ (\langle Lit\ undefined\rangle,Some\ xc,s)$

| *InsInitE*: $[\![G\vdash(\langle c\rangle,Norm\ s) \mapsto 1\ (\langle c'\rangle,\ s')]\!]$
$\Longrightarrow$
$G\vdash(\langle InsInitE\ c\ e\rangle,Norm\ s) \mapsto 1\ (\langle InsInitE\ c'\ e\rangle,\ s')$

| *NewC*: $G\vdash(\langle NewC\ C\rangle,Norm\ s) \mapsto 1\ (\langle InsInitE\ (Init\ C)\ (NewC\ C)\rangle,\ Norm\ s)$
| *NewCInited*: $[\![G\vdash\ Norm\ s\ -halloc\ (CInst\ C)\succ a\rightarrow\ s']\!]$
$\Longrightarrow$
$G\vdash(\langle InsInitE\ Skip\ (NewC\ C)\rangle,Norm\ s) \mapsto 1\ (\langle Ref\ a\rangle,\ s')$

| *NewA*:
$G\vdash(\langle New\ T[e]\rangle,Norm\ s) \mapsto 1\ (\langle InsInitE\ (init\text{-}comp\text{-}ty\ T)\ (New\ T[e])\rangle,Norm\ s)$
| *InsInitNewAIdx*:
$[\![G\vdash(\langle e\rangle,Norm\ s) \mapsto 1\ (\langle e'\rangle,\ s')]\!]$
$\Longrightarrow$
$G\vdash(\langle InsInitE\ Skip\ (New\ T[e])\rangle,Norm\ s) \mapsto 1\ (\langle InsInitE\ Skip\ (New\ T[e'])\rangle,s')$
| *InsInitNewA*:
$[\![G\vdash abupd\ (check\text{-}neg\ i)\ (Norm\ s)\ -halloc\ (Arr\ T\ (the\text{-}Intg\ i))\succ a\rightarrow\ s'\ ]\!]$
$\Longrightarrow$
$G\vdash(\langle InsInitE\ Skip\ (New\ T[Lit\ i])\rangle,Norm\ s) \mapsto 1\ (\langle Ref\ a\rangle,s')$

| *CastE*:
$[\![G\vdash(\langle e\rangle,Norm\ s) \mapsto 1\ (\langle e'\rangle,s')]\!]$
$\Longrightarrow$
$G\vdash(\langle Cast\ T\ e\rangle,None,s) \mapsto 1\ (\langle Cast\ T\ e'\rangle,s')$
| *Cast*:
$[\![s' = abupd\ (raise\text{-}if\ (\neg G,s\vdash v\ fits\ T)\ \ ClassCast)\ (Norm\ s)]\!]$
$\Longrightarrow$
$G\vdash(\langle Cast\ T\ (Lit\ v)\rangle,Norm\ s) \mapsto 1\ (\langle Lit\ v\rangle,s')$

| *InstE*: $[\![G\vdash(\langle e\rangle,Norm\ s) \mapsto 1\ (\langle e'::expr\rangle,s')]\!]$
$\Longrightarrow$
$G\vdash(\langle e\ InstOf\ T\rangle,Norm\ s) \mapsto 1\ (\langle e'\rangle,s')$
| *Inst*: $[\![b = (v\neq Null \wedge G,s\vdash v\ fits\ RefT\ T)]\!]$
$\Longrightarrow$
$G\vdash(\langle(Lit\ v)\ InstOf\ T\rangle,Norm\ s) \mapsto 1\ (\langle Lit\ (Bool\ b)\rangle,s')$

| *UnOpE*: $[\![G\vdash(\langle e\rangle,Norm\ s) \mapsto 1\ (\langle e'\rangle,s')\ ]\!]$
$\Longrightarrow$
$G\vdash(\langle UnOp\ unop\ e\rangle,Norm\ s) \mapsto 1\ (\langle UnOp\ unop\ e'\rangle,s')$
| *UnOp*: $G\vdash(\langle UnOp\ unop\ (Lit\ v)\rangle,Norm\ s) \mapsto 1\ (\langle Lit\ (eval\text{-}unop\ unop\ v)\rangle,Norm\ s)$

| *BinOpE1*: $[\![G\vdash(\langle e1\rangle,Norm\ s) \mapsto 1\ (\langle e1'\rangle,s')\ ]\!]$
$\Longrightarrow$
$G\vdash(\langle BinOp\ binop\ e1\ e2\rangle,Norm\ s) \mapsto 1\ (\langle BinOp\ binop\ e1'\ e2\rangle,s')$
| *BinOpE2*: $[\![need\text{-}second\text{-}arg\ binop\ v1;\ G\vdash(\langle e2\rangle,Norm\ s) \mapsto 1\ (\langle e2'\rangle,s')\ ]\!]$
$\Longrightarrow$
$G\vdash(\langle BinOp\ binop\ (Lit\ v1)\ e2\rangle,Norm\ s)$
$\mapsto 1\ (\langle BinOp\ binop\ (Lit\ v1)\ e2'\rangle,s')$
| *BinOpTerm*: $[\![\neg\ need\text{-}second\text{-}arg\ binop\ v1]\!]$
$\Longrightarrow$
$G\vdash(\langle BinOp\ binop\ (Lit\ v1)\ e2\rangle,Norm\ s)$
$\mapsto 1\ (\langle Lit\ v1\rangle,Norm\ s)$
| *BinOp*: $G\vdash(\langle BinOp\ binop\ (Lit\ v1)\ (Lit\ v2)\rangle,Norm\ s)$
$\mapsto 1\ (\langle Lit\ (eval\text{-}binop\ binop\ v1\ v2)\rangle,Norm\ s)$

| *Super*: $G\vdash(\langle Super\rangle,Norm\ s) \mapsto 1\ (\langle Lit\ (val\text{-}this\ s)\rangle,Norm\ s)$

| *AccVA*: $[\![G\vdash(\langle va\rangle,Norm\ s) \mapsto 1\ (\langle va'\rangle,s')\ ]\!]$
$\Longrightarrow$
$G\vdash(\langle Acc\ va\rangle,Norm\ s) \mapsto 1\ (\langle Acc\ va'\rangle,s')$
| *Acc*: $[\![groundVar\ va;\ ((v,vf),s') = the\text{-}var\ G\ (Norm\ s)\ va]\!]$
$\Longrightarrow$
$G\vdash(\langle Acc\ va\rangle,Norm\ s) \mapsto 1\ (\langle Lit\ v\rangle,s')$

| *AssVA*: $[\![G\vdash(\langle va\rangle,Norm\ s) \mapsto 1\ (\langle va'\rangle,s')]\!]$
$\Longrightarrow$
$G\vdash(\langle va{:=}e\rangle,Norm\ s) \mapsto 1\ (\langle va'{:=}e\rangle,s')$
| *AssE*: $[\![groundVar\ va;\ G\vdash(\langle e\rangle,Norm\ s) \mapsto 1\ (\langle e'\rangle,s')]\!]$
$\Longrightarrow$
$G\vdash(\langle va{:=}e\rangle,Norm\ s) \mapsto 1\ (\langle va{:=}e'\rangle,s')$
| *Ass*: $[\![groundVar\ va;\ ((w,f),s') = the\text{-}var\ G\ (Norm\ s)\ va]\!]$
$\Longrightarrow$
$G\vdash(\langle va{:=}(Lit\ v)\rangle,Norm\ s) \mapsto 1\ (\langle Lit\ v\rangle,assign\ f\ v\ s')$

| *CondC*: $[\![G\vdash(\langle e0\rangle,Norm\ s) \mapsto 1\ (\langle e0'\rangle,s')]\!]$
$\Longrightarrow$
$G\vdash(\langle e0?\ e1{:}e2\rangle,Norm\ s) \mapsto 1\ (\langle e0'?\ e1{:}e2\rangle,s')$
| *Cond*: $G\vdash(\langle Lit\ b?\ e1{:}e2\rangle,Norm\ s) \mapsto 1\ (\langle if\ the\text{-}Bool\ b\ then\ e1\ else\ e2\rangle,Norm\ s)$

| *CallTarget*: $[\![G\vdash(\langle e\rangle,Norm\ s) \mapsto 1\ (\langle e'\rangle,s')]\!]$
$\Longrightarrow$

G⊢(⟨{accC,statT,mode}e·mn({pTs}args)⟩,Norm s)
↦1 (⟨{accC,statT,mode}e′·mn({pTs}args)⟩,s′)
| *CallArgs*: ⟦G⊢(⟨args⟩,Norm s) ↦1 (⟨args′⟩,s′)⟧
⟹
G⊢(⟨{accC,statT,mode}Lit a·mn({pTs}args)⟩,Norm s)
↦1 (⟨{accC,statT,mode}Lit a·mn({pTs}args′)⟩,s′)
| *Call*: ⟦groundExprs args; vs = map the-val args;
D = invocation-declclass G mode s a statT ⦇name=mn,parTs=pTs⦈;
s′=init-lvars G D ⦇name=mn,parTs=pTs⦈ mode a′ vs (Norm s)⟧
⟹
G⊢(⟨{accC,statT,mode}Lit a·mn({pTs}args)⟩,Norm s)
↦1 (⟨Callee (locals s) (Methd D ⦇name=mn,parTs=pTs⦈)⟩,s′)

| *Callee*: ⟦G⊢(⟨e⟩,Norm s) ↦1 (⟨e′::expr⟩,s′)⟧
⟹
G⊢(⟨Callee lcls-caller e⟩,Norm s) ↦1 (⟨e′⟩,s′)

| *CalleeRet*: G⊢(⟨Callee lcls-caller (Lit v)⟩,Norm s)
↦1 (⟨Lit v⟩,(set-lvars lcls-caller (Norm s)))

| *Methd*: G⊢(⟨Methd D sig⟩,Norm s) ↦1 (⟨body G D sig⟩,Norm s)

| *Body*: G⊢(⟨Body D c⟩,Norm s) ↦1 (⟨InsInitE (Init D) (Body D c)⟩,Norm s)

| *InsInitBody*:
⟦G⊢(⟨c⟩,Norm s) ↦1 (⟨c′⟩,s′)⟧
⟹
G⊢(⟨InsInitE Skip (Body D c)⟩,Norm s) ↦1(⟨InsInitE Skip (Body D c′)⟩,s′)
| *InsInitBodyRet*:
G⊢(⟨InsInitE Skip (Body D Skip)⟩,Norm s)
↦1 (⟨Lit (the ((locals s) Result))⟩,abupd (absorb Ret) (Norm s))

| *FVar*: ⟦¬ inited statDeclC (globs s)⟧
⟹
G⊢(⟨{accC,statDeclC,stat}e..fn⟩,Norm s)
↦1 (⟨InsInitV (Init statDeclC) ({accC,statDeclC,stat}e..fn)⟩,Norm s)
| *InsInitFVarE*:
⟦G⊢(⟨e⟩,Norm s) ↦1 (⟨e′⟩,s′)⟧
⟹
G⊢(⟨InsInitV Skip ({accC,statDeclC,stat}e..fn)⟩,Norm s)
↦1 (⟨InsInitV Skip ({accC,statDeclC,stat}e′..fn)⟩,s′)
| *InsInitFVar*:
G⊢(⟨InsInitV Skip ({accC,statDeclC,stat}Lit a..fn)⟩,Norm s)
↦1 (⟨{accC,statDeclC,stat}Lit a..fn⟩,Norm s)

— Notice, that we do not have literal values for *vars*. The rules for accessing variables (*Acc*) and assigning to variables (*Ass*), test this with the predicate *groundVar*. After initialisation is done and the *FVar* is evaluated, we can't just throw away the *InsInitFVar* term and return a literal value, as in the cases of *New* or *NewC*. Instead we just return the evaluated *FVar* and test for initialisation in the rule *FVar*.

| *AVarE1*: ⟦G⊢(⟨e1⟩,Norm s) ↦1 (⟨e1′⟩,s′)⟧
⟹
G⊢(⟨e1.[e2]⟩,Norm s) ↦1 (⟨e1′.[e2]⟩,s′)

| *AVarE2*: G⊢(⟨e2⟩,Norm s) ↦1 (⟨e2′⟩,s′)
⟹
G⊢(⟨Lit a.[e2]⟩,Norm s) ↦1 (⟨Lit a.[e2′]⟩,s′)

— *Nil* is fully evaluated

| *ConsHd*: $[\![G\vdash(\langle e{::}expr\rangle,Norm\ s) \mapsto 1\ (\langle e'{::}expr\rangle,s')]\!]$
$\Longrightarrow$
$G\vdash(\langle e\#es\rangle,Norm\ s) \mapsto 1\ (\langle e'\#es\rangle,s')$

| *ConsTl*: $[\![G\vdash(\langle es\rangle,Norm\ s) \mapsto 1\ (\langle es'\rangle,s')]\!]$
$\Longrightarrow$
$G\vdash(\langle(Lit\ v)\#es\rangle,Norm\ s) \mapsto 1\ (\langle(Lit\ v)\#es'\rangle,s')$

| *Skip*: $G\vdash(\langle Skip\rangle,Norm\ s) \mapsto 1\ (\langle SKIP\rangle,Norm\ s)$

| *ExprE*: $[\![G\vdash(\langle e\rangle,Norm\ s) \mapsto 1\ (\langle e'\rangle,s')]\!]$
$\Longrightarrow$
$G\vdash(\langle Expr\ e\rangle,Norm\ s) \mapsto 1\ (\langle Expr\ e'\rangle,s')$
| *Expr*: $G\vdash(\langle Expr\ (Lit\ v)\rangle,Norm\ s) \mapsto 1\ (\langle Skip\rangle,Norm\ s)$

| *LabC*: $[\![G\vdash(\langle c\rangle,Norm\ s) \mapsto 1\ (\langle c'\rangle,s')]\!]$
$\Longrightarrow$
$G\vdash(\langle l\bullet\ c\rangle,Norm\ s) \mapsto 1\ (\langle l\bullet\ c'\rangle,s')$
| *Lab*: $G\vdash(\langle l\bullet\ Skip\rangle,s) \mapsto 1\ (\langle Skip\rangle,\ abupd\ (absorb\ l)\ s)$

| *CompC1*: $[\![G\vdash(\langle c1\rangle,Norm\ s) \mapsto 1\ (\langle c1'\rangle,s')]\!]$
$\Longrightarrow$
$G\vdash(\langle c1;;\ c2\rangle,Norm\ s) \mapsto 1\ (\langle c1';;\ c2\rangle,s')$

| *Comp*: $G\vdash(\langle Skip;;\ c2\rangle,Norm\ s) \mapsto 1\ (\langle c2\rangle,Norm\ s)$

| *IfE*: $[\![G\vdash(\langle e\rangle\ ,Norm\ s) \mapsto 1\ (\langle e'\rangle,s')]\!]$
$\Longrightarrow$
$G\vdash(\langle If(e)\ s1\ Else\ s2\rangle,Norm\ s) \mapsto 1\ (\langle If(e')\ s1\ Else\ s2\rangle,s')$
| *If*: $G\vdash(\langle If(Lit\ v)\ s1\ Else\ s2\rangle,Norm\ s)$
$\mapsto 1\ (\langle if\ the\text{-}Bool\ v\ then\ s1\ else\ s2\rangle,Norm\ s)$

| *Loop*: $G\vdash(\langle l\bullet\ While(e)\ c\rangle,Norm\ s)$
$\mapsto 1\ (\langle If(e)\ (Cont\ l\bullet c;;\ l\bullet\ While(e)\ c)\ Else\ Skip\rangle,Norm\ s)$

| *Jmp*: $G\vdash(\langle Jmp\ j\rangle,Norm\ s) \mapsto 1\ (\langle Skip\rangle,(Some\ (Jump\ j),\ s))$

| *ThrowE*: $[\![G\vdash(\langle e\rangle,Norm\ s) \mapsto 1\ (\langle e'\rangle,s')]\!]$
$\Longrightarrow$
$G\vdash(\langle Throw\ e\rangle,Norm\ s) \mapsto 1\ (\langle Throw\ e'\rangle,s')$
| *Throw*: $G\vdash(\langle Throw\ (Lit\ a)\rangle,Norm\ s) \mapsto 1\ (\langle Skip\rangle,abupd\ (throw\ a)\ (Norm\ s))$

| *TryC1*: $[\![G\vdash(\langle c1\rangle,Norm\ s) \mapsto 1\ (\langle c1'\rangle,s')]\!]$
$\Longrightarrow$
$G\vdash(\langle Try\ c1\ Catch(C\ vn)\ c2\rangle,\ Norm\ s) \mapsto 1\ (\langle Try\ c1'\ Catch(C\ vn)\ c2\rangle,s')$

| *Try*: $[\![G\vdash s \ -sxalloc\rightarrow \ s']\!]$
$\Longrightarrow$
$G\vdash(\langle Try\ Skip\ Catch(C\ vn)\ c2\rangle,\ s)$
$\mapsto 1\ (if\ G,s'\vdash catch\ C\ then\ (\langle c2\rangle,new\text{-}xcpt\text{-}var\ vn\ s')$
$else\ (\langle Skip\rangle,s'))$

| *FinC1*: $[\![G\vdash(\langle c1\rangle,Norm\ s) \mapsto 1\ (\langle c1'\rangle,s')]\!]$
$\Longrightarrow$
$G\vdash(\langle c1\ Finally\ c2\rangle,Norm\ s) \mapsto 1\ (\langle c1'\ Finally\ c2\rangle,s')$

| *Fin*: $G\vdash(\langle Skip\ Finally\ c2\rangle,(a,s)) \mapsto 1\ (\langle FinA\ a\ c2\rangle,Norm\ s)$

| *FinAC*: $[\![G\vdash(\langle c\rangle,s) \mapsto 1\ (\langle c'\rangle,s')]\!]$
$\Longrightarrow$
$G\vdash(\langle FinA\ a\ c\rangle,s) \mapsto 1\ (\langle FinA\ a\ c'\rangle,s')$
| *FinA*: $G\vdash(\langle FinA\ a\ Skip\rangle,s) \mapsto 1\ (\langle Skip\rangle,abupd\ (abrupt\text{-}if\ (a\neq None)\ a)\ s)$

| *Init1*: $[\![inited\ C\ (globs\ s)]\!]$
$\Longrightarrow$
$G\vdash(\langle Init\ C\rangle,Norm\ s) \mapsto 1\ (\langle Skip\rangle,Norm\ s)$
| *Init*: $[\![the\ (class\ G\ C)=c;\ \neg\ inited\ C\ (globs\ s)]\!]$
$\Longrightarrow$
$G\vdash(\langle Init\ C\rangle,Norm\ s)$
$\mapsto 1\ (\langle(if\ C = Object\ then\ Skip\ else\ (Init\ (super\ c)));;$
$Expr\ (Callee\ (locals\ s)\ (InsInitE\ (init\ c)\ SKIP))\rangle$
$,Norm\ (init\text{-}class\text{-}obj\ G\ C\ s))$
— *InsInitE* is just used as trick to embed the statement *init c* into an expression
| *InsInitESKIP*:
$G\vdash(\langle InsInitE\ Skip\ SKIP\rangle,Norm\ s) \mapsto 1\ (\langle SKIP\rangle,Norm\ s)$

abbreviation
stepn:: $[prog,\ term \times state,nat,term \times state] \Rightarrow bool$ (‹-⊢- ↦- -›[61,82,82] 81)
where $G\vdash p \mapsto n\ p' \equiv (p,p') \in \{(x,\ y).\ step\ G\ x\ y\}^{\frown\frown}n$

abbreviation
steptr:: $[prog,term \times state,term \times state] \Rightarrow bool$ (‹-⊢- ↦∗ -›[61,82,82] 81)
where $G\vdash p \mapsto *\ p' \equiv (p,p') \in \{(x,\ y).\ step\ G\ x\ y\}^*$

end

Chapter 22

AxSem

1 Axiomatic semantics of Java expressions and statements (see also Eval.thy)

theory *AxSem* **imports** *Evaln TypeSafe* **begin**

design issues:

- a strong version of validity for triples with premises, namely one that takes the recursive depth needed to complete execution, enables correctness proof
- auxiliary variables are handled first-class (-> Thomas Kleymann)
- expressions not flattened to elementary assignments (as usual for axiomatic semantics) but treated first-class => explicit result value handling
- intermediate values not on triple, but on assertion level (with result entry)
- multiple results with semantical substitution mechnism not requiring a stack
- because of dynamic method binding, terms need to be dependent on state. this is also useful for conditional expressions and statements
- result values in triples exactly as in eval relation (also for xcpt states)
- validity: additional assumption of state conformance and well-typedness, which is required for soundness and thus rule hazard required of completeness

restrictions:

- all triples in a derivation are of the same type (due to weak polymorphism)

type-synonym *res* = *vals* — result entry

abbreviation (*input*)
  Val **where** *Val x* == *In1 x*

abbreviation (*input*)
  Var **where** *Var x* == *In2 x*

abbreviation (*input*)
  Vals **where** *Vals x* == *In3 x*

syntax
  -Val :: [*pttrn*] => *pttrn* (‹*Val*:-› [*951*] *950*)
  -Var :: [*pttrn*] => *pttrn* (‹*Var*:-› [*951*] *950*)

-Vals :: [*pttrn*] => *pttrn* (‹*Vals*:-› [*951*] *950*)

translations
λ*Val*:*v* . *b* == (λv. *b*) $\circ$ *CONST the-In1*
λ*Var*:*v* . *b* == (λv. *b*) $\circ$ *CONST the-In2*
λ*Vals*:*v*. *b* == (λv. *b*) $\circ$ *CONST the-In3*

— relation on result values, state and auxiliary variables
type-synonym *'a assn* = *res* $\Rightarrow$ *state* $\Rightarrow$ *'a* $\Rightarrow$ *bool*
translations
(*type*) *'a assn* <= (*type*) *vals* $\Rightarrow$ *state* $\Rightarrow$ *'a* $\Rightarrow$ *bool*

definition
assn-imp :: *'a assn* $\Rightarrow$ *'a assn* $\Rightarrow$ *bool* (**infixr** ‹$\Rightarrow$› *25*)
where ($P \Rightarrow Q$) = ($\forall$ *Y s Z*. *P Y s Z* $\longrightarrow$ *Q Y s Z*)

lemma *assn-imp-def2* [*iff*]: ($P \Rightarrow Q$) = ($\forall$ *Y s Z*. *P Y s Z* $\longrightarrow$ *Q Y s Z*)
⟨*proof*⟩

assertion transformers

2 peek-and

definition
peek-and :: *'a assn* $\Rightarrow$ (*state* $\Rightarrow$ *bool*) $\Rightarrow$ *'a assn* (**infixl** ‹$\wedge$.› *13*)
where ($P \wedge$. p) = (λ*Y s Z*. *P Y s Z* $\wedge$ *p s*)

lemma *peek-and-def2* [*simp*]: *peek-and P p Y s* = (λZ. (*P Y s Z* $\wedge$ *p s*))
⟨*proof*⟩

lemma *peek-and-Not* [*simp*]: ($P \wedge$. (λs. $\neg$ *f s*)) = ($P \wedge$. *Not* $\circ$ *f*)
⟨*proof*⟩

lemma *peek-and-and* [*simp*]: *peek-and* (*peek-and P p*) *p* = *peek-and P p*
⟨*proof*⟩

lemma *peek-and-commut*: ($P \wedge$. $p \wedge$. q) = ($P \wedge$. $q \wedge$. p)
⟨*proof*⟩

abbreviation
Normal :: *'a assn* $\Rightarrow$ *'a assn*
where *Normal P* == $P \wedge$. *normal*

lemma *peek-and-Normal* [*simp*]: *peek-and* (*Normal P*) *p* = *Normal* (*peek-and P p*)
⟨*proof*⟩

3 assn-supd

definition
assn-supd :: *'a assn* $\Rightarrow$ (*state* $\Rightarrow$ *state*) $\Rightarrow$ *'a assn* (**infixl** ‹;.› *13*)
where (*P* ;. *f*) = (λ*Y s' Z*. $\exists$ *s*. *P Y s Z* $\wedge$ *s'* = *f s*)

lemma *assn-supd-def2* [*simp*]: *assn-supd P f Y s' Z* = ($\exists$ *s*. *P Y s Z* $\wedge$ *s'* = *f s*)
⟨*proof*⟩

4 supd-assn

definition
supd-assn :: (*state* $\Rightarrow$ *state*) $\Rightarrow$ *'a assn* $\Rightarrow$ *'a assn* (**infixr** ‹.;› *13*)
where (*f* .; *P*) = (λ*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* ⟹ *Q Y s Z*
⟨*proof*⟩

lemma *supd-assn-supdI* [*elim*]: *Q Y s Z* ⟹ (*f* .; (*Q* ;. *f*)) *Y s Z*
⟨*proof*⟩

5 subst-res

definition
subst-res :: ′*a assn* ⇒ *res* ⇒ ′*a assn* (‹-←-› [*60*,*61*] *60*)
where *P*←*w* = (λ*Y*. *P w*)

lemma *subst-res-def2* [*simp*]: (*P*←*w*) *Y* = *P w*
⟨*proof*⟩

lemma *subst-subst-res* [*simp*]: *P*←*w*←*v* = *P*←*w*
⟨*proof*⟩

lemma *peek-and-subst-res* [*simp*]: (*P* ∧. *p*)←*w* = (*P*←*w* ∧. *p*)
⟨*proof*⟩

6 subst-Bool

definition
subst-Bool :: ′*a assn* ⇒ *bool* ⇒ ′*a assn* (‹-←=-› [*60*,*61*] *60*)
where *P*←=*b* = (λ*Y s Z*. ∃*v*. *P* (*Val v*) *s Z* ∧ (*normal s* ⟶ *the-Bool v*=*b*))

lemma *subst-Bool-def2* [*simp*]:
(*P*←=*b*) *Y s Z* = (∃*v*. *P* (*Val v*) *s Z* ∧ (*normal s* ⟶ *the-Bool v*=*b*))
⟨*proof*⟩

lemma *subst-Bool-the-BoolI*: *P* (*Val b*) *s Z* ⟹ (*P*←=*the-Bool b*) *Y s Z*
⟨*proof*⟩

7 peek-res

definition
peek-res :: (*res* ⇒ ′*a assn*) ⇒ ′*a assn*
where *peek-res Pf* = (λ*Y*. *Pf Y Y*)

syntax
-peek-res :: *pttrn* ⇒ ′*a assn* ⇒ ′*a assn* (‹λ-:. -› [*0*,*3*] *3*)
syntax-consts
-peek-res == *peek-res*
translations
λ*w*:. *P* == *CONST peek-res* (λ*w*. *P*)

lemma *peek-res-def2* [*simp*]: *peek-res P Y* = *P Y Y*
⟨*proof*⟩

lemma *peek-res-subst-res* [*simp*]: *peek-res P*←*w* = *P w*←*w*
⟨*proof*⟩

lemma *peek-subst-res-allI*:
$(\bigwedge a.\ T\ a\ (P\ (f\ a){\leftarrow}f\ a)) \Longrightarrow \forall\, a.\ T\ a\ (\mathit{peek\text{-}res}\ P{\leftarrow}f\ a)$
⟨*proof*⟩

8 ign-res

definition
ign-res :: *'a assn* ⇒ *'a assn* (‹-↓› [*1000*] *1000*)
where $P{\downarrow} = (\lambda Y\ s\ Z.\ \exists\, Y.\ P\ Y\ s\ Z)$

lemma *ign-res-def2* [*simp*]: $P{\downarrow}\ Y\ s\ Z = (\exists\, Y.\ P\ Y\ s\ Z)$
⟨*proof*⟩

lemma *ign-ign-res* [*simp*]: $P{\downarrow}{\downarrow} = P{\downarrow}$
⟨*proof*⟩

lemma *ign-subst-res* [*simp*]: $P{\downarrow}{\leftarrow}w = P{\downarrow}$
⟨*proof*⟩

lemma *peek-and-ign-res* [*simp*]: $(P \wedge.\ p){\downarrow} = (P{\downarrow} \wedge.\ p)$
⟨*proof*⟩

9 peek-st

definition
peek-st :: (*st* ⇒ *'a assn*) ⇒ *'a assn*
where *peek-st* $P = (\lambda Y\ s.\ P\ (\mathit{store}\ s)\ Y\ s)$

syntax
-peek-st :: *pttrn* ⇒ *'a assn* ⇒ *'a assn* (‹λ-.. -› [*0*,*3*] *3*)
syntax-consts
-peek-st == *peek-st*
translations
$\lambda s..\ P$ == *CONST peek-st* $(\lambda s.\ P)$

lemma *peek-st-def2* [*simp*]: $(\lambda s..\ Pf\ s)\ Y\ s = Pf\ (\mathit{store}\ s)\ Y\ s$
⟨*proof*⟩

lemma *peek-st-triv* [*simp*]: $(\lambda s..\ P) = P$
⟨*proof*⟩

lemma *peek-st-st* [*simp*]: $(\lambda s..\ \lambda s'..\ P\ s\ s') = (\lambda s..\ P\ s\ s)$
⟨*proof*⟩

lemma *peek-st-split* [*simp*]: $(\lambda s..\ \lambda Y\ s'.\ P\ s\ Y\ s') = (\lambda Y\ s.\ P\ (\mathit{store}\ s)\ Y\ s)$
⟨*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*]: $(\lambda s..(\mathit{Normal}\ (P\ s))) = \mathit{Normal}\ (\lambda s..\ P\ s)$
⟨*proof*⟩

10 ign-res-eq

definition
ign-res-eq :: *'a assn* ⇒ *res* ⇒ *'a assn* (‹-↓=-› [*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 $(vf\ ..;\ P) = (\lambda Y\ s.\ let\ (v,s') = vf\ s\ in\ P\ (Var\ v)\ s')$

lemma *RefVar-def2* [*simp*]: $(vf\ ..;\ P)\ Y\ s =$
$P\ (Var\ (fst\ (vf\ s)))\ (snd\ (vf\ s))$
⟨*proof*⟩

12 allocation

definition
$Alloc :: prog \Rightarrow obj\text{-}tag \Rightarrow 'a\ assn \Rightarrow 'a\ assn$
where $Alloc\ G\ otag\ P = (\lambda Y\ s\ Z.\ \forall s'\ a.\ G\vdash s\ {-}halloc\ otag{\succ}a{\rightarrow}\ s' \longrightarrow P\ (Val\ (Addr\ a))\ s'\ Z)$

definition
$SXAlloc :: prog \Rightarrow 'a\ assn \Rightarrow 'a\ assn$
where $SXAlloc\ G\ P = (\lambda Y\ s\ Z.\ \forall s'.\ G\vdash s\ {-}sxalloc{\rightarrow}\ s' \longrightarrow P\ Y\ s'\ Z)$

lemma *Alloc-def2* [*simp*]: $Alloc\ G\ otag\ P\ Y\ s\ Z =$
$(\forall s'\ a.\ G\vdash s\ {-}halloc\ otag{\succ}a{\rightarrow}\ s' \longrightarrow P\ (Val\ (Addr\ a))\ s'\ Z)$
⟨*proof*⟩

lemma *SXAlloc-def2* [*simp*]:
$SXAlloc\ G\ P\ Y\ s\ Z = (\forall s'.\ G\vdash s\ {-}sxalloc{\rightarrow}\ s' \longrightarrow P\ Y\ s'\ Z)$
⟨*proof*⟩

validity

definition
$type\text{-}ok :: prog \Rightarrow term \Rightarrow state \Rightarrow bool$ **where**
$type\text{-}ok\ G\ t\ s =$
$(\exists L\ T\ C\ A.\ (normal\ s \longrightarrow (\!|prg{=}G,cls{=}C,lcl{=}L|\!)\vdash t{::}T\ \wedge$
$(\!|prg{=}G,cls{=}C,lcl{=}L|\!)\vdash dom\ (locals\ (store\ s))\text{»}t\text{»}A\ )$
$\wedge\ s{::}{\preceq}(G,L))$

datatype *'a triple = triple ('a assn) term ('a assn)*
(‹{(1-)}/ -≻/ {(1-)}› [*3*,*65*,*3*] *75*)
type-synonym *'a triples = 'a triple set*

abbreviation
var-triple :: [*'a assn, var* ,*'a assn*] ⇒ *'a triple*
(‹{(1-)}/ -=≻/ {(1-)}› [*3*,*80*,*3*] *75*)

where {*P*} *e*=≻ {*Q*} == {*P*} *In2 e*≻ {*Q*}

abbreviation
expr-triple :: [′*a assn*, *expr* ,′*a assn*] ⇒ ′*a triple*
(‹{(*1*-)}/ -−≻/ {(*1*-)}› [*3*,*80*,*3*] *75*)
where {*P*} *e*−≻ {*Q*} == {*P*} *In1l e*≻ {*Q*}

abbreviation
exprs-triple :: [′*a assn*, *expr list* ,′*a assn*] ⇒ ′*a triple*
(‹{(*1*-)}/ -≐≻/ {(*1*-)}› [*3*,*65*,*3*] *75*)
where {*P*} *e*≐≻ {*Q*} == {*P*} *In3 e*≻ {*Q*}

abbreviation
stmt-triple :: [′*a assn*, *stmt*, ′*a assn*] ⇒ ′*a triple*
(‹{(*1*-)}/ .-./ {(*1*-)}› [*3*,*65*,*3*] *75*)
where {*P*} .*c*. {*Q*} == {*P*} *In1r c*≻ {*Q*}

notation (*ASCII*)
triple (‹{(*1*-)}/ ->/ {(*1*-)}› [*3*,*65*,*3*]*75*) **and**
var-triple (‹{(*1*-)}/ -=>/ {(*1*-)}› [*3*,*80*,*3*] *75*) **and**
expr-triple (‹{(*1*-)}/ -->/ {(*1*-)}› [*3*,*80*,*3*] *75*) **and**
exprs-triple (‹{(*1*-)}/ -#>/ {(*1*-)}› [*3*,*65*,*3*] *75*)

lemma *inj-triple*: *inj* (λ(*P*,*t*,*Q*). {*P*} *t*≻ {*Q*})
⟨*proof*⟩

lemma *triple-inj-eq*: ({*P*} *t*≻ {*Q*} = {*P*′} *t*′≻ {*Q*′}) = (*P*=*P*′ ∧ *t*=*t*′ ∧ *Q*=*Q*′)
⟨*proof*⟩

definition *mtriples* :: (′*c* ⇒ ′*sig* ⇒ ′*a assn*) ⇒ (′*c* ⇒ ′*sig* ⇒ *expr*) ⇒
(′*c* ⇒ ′*sig* ⇒ ′*a assn*) ⇒ (′*c* × ′*sig*) *set* ⇒ ′*a triples* (‹{{(*1*-)}/ -−≻/ {(*1*-)} | -}›[*3*,*65*,*3*,*65*]*75*)
where
{{*P*} *tf*−≻ {*Q*} | *ms*} = (λ(*C*,*sig*). {*Normal*(*P C sig*)} *tf C sig*−≻ {*Q C sig*})‘*ms*

definition
triple-valid :: *prog* ⇒ *nat* ⇒ ′*a triple* ⇒ *bool* (‹-|=-:-› [*61*,*0*, *58*] *57*)
where
G|=*n*:*t* =
(*case t of* {*P*} *t*≻ {*Q*} ⇒
∀ *Y s Z*. *P Y s Z* ⟶ *type-ok G t s* ⟶
(∀ *Y*′ *s*′. *G*⊢*s* −*t*≻−*n*→ (*Y*′,*s*′) ⟶ *Q Y*′ *s*′ *Z*))

abbreviation
triples-valid:: *prog* ⇒ *nat* ⇒ ′*a triples* ⇒ *bool* (‹-||=-:-› [*61*,*0*, *58*] *57*)
where *G*||=*n*:*ts* == *Ball ts* (*triple-valid G n*)

notation (*ASCII*)
triples-valid (‹-||=-:-› [*61*,*0*, *58*] *57*)

definition
ax-valids :: *prog* ⇒ ′*b triples* ⇒ ′*a triples* ⇒ *bool* (‹-,-||=-› [*61*,*58*,*58*] *57*)
where (*G*,*A*||=*ts*) = (∀ *n*. *G*||=*n*:*A* ⟶ *G*||=*n*:*ts*)

abbreviation
ax-valid :: *prog* ⇒ ′*b triples* ⇒ ′*a triple* ⇒ *bool* (‹-,-|=-› [*61*,*58*,*58*] *57*)
where *G*,*A* |=*t* == *G*,*A*||={*t*}

notation (*ASCII*)

ax-valid (‹-,-|=-› [61,58,58] 57)

lemma *triple-valid-def2*: G⊨n:{P} t≻ {Q} =
(∀ Y s Z. P Y s Z
⟶ (∃ L. (normal s ⟶ (∃ C T A. (|prg=G,cls=C,lcl=L|)⊢t::T ∧
(|prg=G,cls=C,lcl=L|)⊢dom (locals (store s))»t»A)) ∧
s::≼(G,L))
⟶ (∀ Y′ s′. G⊢s −t≻−n→ (Y′,s′)⟶ 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 :: *prog* ⇒ ′*a triples* ⇒ ′*a triples* ⇒ *bool* (‹-,-|⊢-› [61,58,58] 57)
and *ax-deriv* :: *prog* ⇒ ′*a triples* ⇒ ′*a triple* ⇒ *bool* (‹-,-⊢-› [61,58,58] 57)
for G :: *prog*
where

G,A ⊢t ≡ G,A|⊢{t}

| *empty*: G,A|⊢{}
| *insert*:⟦G,A⊢t; G,A|⊢ts⟧ ⟹
G,A|⊢insert t ts

| *asm*: ts⊆A ⟹ G,A|⊢ts

| *weaken*:⟦G,A|⊢ts′; ts ⊆ ts′⟧ ⟹ G,A|⊢ts

| *conseq*:∀ Y s Z . P Y s Z ⟶ (∃ P′ Q′. G,A⊢{P′} t≻ {Q′} ∧ (∀ Y′ s′.
(∀ Y Z′. P′ Y s Z′ ⟶ Q′ Y′ s′ Z′) ⟶
Q Y′ s′ Z))
⟹ G,A⊢{P } t≻ {Q }

| *hazard*:G,A⊢{P ∧. Not ∘ type-ok G t} t≻ {Q}

| *Abrupt*: G,A⊢{P←(undefined3 t) ∧. Not ∘ normal} t≻ {P}

— variables
| *LVar*: G,A⊢{Normal (λs.. P← Var (lvar vn s))} LVar vn=≻ {P}

| *FVar*: ⟦G,A⊢{Normal P} .Init C. {Q};
G,A⊢{Q} e−≻ {λVal:a:. fvar C stat fn a ..; R}⟧ ⟹
G,A⊢{Normal P} {accC,C,stat}e..fn=≻ {R}

| *AVar*: ⟦G,A⊢{Normal P} e1−≻ {Q};
∀ a. G,A⊢{Q← Val a} e2−≻ {λVal:i:. avar G i a ..; R}⟧ ⟹
G,A⊢{Normal P} e1.[e2]=≻ {R}
— expressions

| *NewC*: ⟦G,A⊢{Normal P} .Init C. {Alloc G (CInst C) Q}⟧ ⟹
G,A⊢{Normal P} NewC C−≻ {Q}

| *NewA*: ⟦G,A⊢{Normal P} .init-comp-ty T. {Q}; G,A⊢{Q} e−≻

{λVal:i:. abupd (check-neg i) .; Alloc G (Arr T (the-Intg i)) R}⟧ ⟹
G,A⊢{Normal P} New T[e]−≻ {R}

| Cast: ⟦G,A⊢{Normal P} e−≻ {λVal:v:. λs..
abupd (raise-if (¬G,s⊢v fits T) ClassCast) .; Q←Val v}⟧ ⟹
G,A⊢{Normal P} Cast T e−≻ {Q}

| Inst: ⟦G,A⊢{Normal P} e−≻ {λVal:v:. λs..
Q←Val (Bool (v≠Null ∧ G,s⊢v fits RefT T))}⟧ ⟹
G,A⊢{Normal P} e InstOf T−≻ {Q}

| Lit: G,A⊢{Normal (P←Val v)} Lit v−≻ {P}

| UnOp: ⟦G,A⊢{Normal P} e−≻ {λVal:v:. Q←Val (eval-unop unop v)}⟧
⟹
G,A⊢{Normal P} UnOp unop e−≻ {Q}

| BinOp:
⟦G,A⊢{Normal P} e1−≻ {Q};
∀v1. G,A⊢{Q←Val v1}
(if need-second-arg binop v1 then (In1l e2) else (In1r Skip))≻
{λVal:v2:. R←Val (eval-binop binop v1 v2)}⟧
⟹
G,A⊢{Normal P} BinOp binop e1 e2−≻ {R}

| Super: G,A⊢{Normal (λs.. P←Val (val-this s))} Super−≻ {P}

| Acc: ⟦G,A⊢{Normal P} va=≻ {λVar:(v,f):. Q←Val v}⟧ ⟹
G,A⊢{Normal P} Acc va−≻ {Q}

| Ass: ⟦G,A⊢{Normal P} va=≻ {Q};
∀vf. G,A⊢{Q←Var vf} e−≻ {λVal:v:. assign (snd vf) v .; R}⟧ ⟹
G,A⊢{Normal P} va:=e−≻ {R}

| Cond: ⟦G,A ⊢{Normal P} e0−≻ {P′};
∀b. G,A⊢{P′←=b} (if b then e1 else e2)−≻ {Q}⟧ ⟹
G,A⊢{Normal P} e0 ? e1 : e2−≻ {Q}

| Call:
⟦G,A⊢{Normal P} e−≻ {Q}; ∀a. G,A⊢{Q←Val a} args≐≻ {R a};
∀a vs invC declC l. G,A⊢{(R a←Vals vs ∧.
(λs. declC=invocation-declclass G mode (store s) a statT ⦇name=mn,parTs=pTs⦈ ∧
invC = invocation-class mode (store s) a statT ∧
l = locals (store s)) ;.
init-lvars G declC ⦇name=mn,parTs=pTs⦈ mode a vs) ∧.
(λs. normal s ⟶ G⊢mode→invC≼statT)}
Methd declC ⦇name=mn,parTs=pTs⦈−≻ {set-lvars l .; S}⟧ ⟹
G,A⊢{Normal P} {accC,statT,mode}e·mn({pTs}args)−≻ {S}

| Methd:⟦G,A∪ {{P} Methd−≻ {Q} | ms} |⊢ {{P} body G−≻ {Q} | ms}⟧ ⟹
G,A|⊢{{P} Methd−≻ {Q} | ms}

| Body: ⟦G,A⊢{Normal P} .Init D. {Q};
G,A⊢{Q} .c. {λs.. abupd (absorb Ret) .; R←(In1 (the (locals s Result)))}⟧
⟹
G,A⊢{Normal P} Body D c−≻ {R}

— expression lists

| *Nil*: $G,A\vdash\{Normal\ (P\leftarrow Vals\ [])\}\ []\dot{=}\succ\ \{P\}$

| *Cons*: $[\![G,A\vdash\{Normal\ P\}\ e-\succ\ \{Q\};$
$\forall\, v.\ G,A\vdash\{Q\leftarrow Val\ v\}\ es\dot{=}\succ\ \{\lambda Vals{:}vs{:}.\ R\leftarrow Vals\ (v\#vs)\}]\!]\Longrightarrow$
$G,A\vdash\{Normal\ P\}\ e\#es\dot{=}\succ\ \{R\}$

— statements

| *Skip*: $G,A\vdash\{Normal\ (P\leftarrow\diamondsuit)\}\ .Skip.\ \{P\}$

| *Expr*: $[\![G,A\vdash\{Normal\ P\}\ e-\succ\ \{Q\leftarrow\diamondsuit\}]\!]\Longrightarrow$
$G,A\vdash\{Normal\ P\}\ .Expr\ e.\ \{Q\}$

| *Lab*: $[\![G,A\vdash\{Normal\ P\}\ .c.\ \{abupd\ (absorb\ l)\ .;\ Q\}]\!]\Longrightarrow$
$G,A\vdash\{Normal\ P\}\ .l\bullet\ c.\ \{Q\}$

| *Comp*: $[\![G,A\vdash\{Normal\ P\}\ .c1.\ \{Q\};$
$G,A\vdash\{Q\}\ .c2.\ \{R\}]\!]\Longrightarrow$
$G,A\vdash\{Normal\ P\}\ .c1;;c2.\ \{R\}$

| *If*: $[\![G,A\vdash\{Normal\ P\}\ e-\succ\ \{P'\};$
$\forall\, b.\ G,A\vdash\{P'\leftarrow=b\}\ .(if\ b\ then\ c1\ else\ c2).\ \{Q\}]\!]\Longrightarrow$
$G,A\vdash\{Normal\ P\}\ .If(e)\ c1\ Else\ c2.\ \{Q\}$

| *Loop*: $[\![G,A\vdash\{P\}\ e-\succ\ \{P'\};$
$G,A\vdash\{Normal\ (P'\leftarrow=True)\}\ .c.\ \{abupd\ (absorb\ (Cont\ l))\ .;\ P\}]\!]\Longrightarrow$
$G,A\vdash\{P\}\ .l\bullet\ While(e)\ c.\ \{(P'\leftarrow=False)\downarrow=\diamondsuit\}$

| *Jmp*: $G,A\vdash\{Normal\ (abupd\ (\lambda a.\ (Some\ (Jump\ j)))\ .;\ P\leftarrow\diamondsuit)\}\ .Jmp\ j.\ \{P\}$

| *Throw*: $[\![G,A\vdash\{Normal\ P\}\ e-\succ\ \{\lambda Val{:}a{:}.\ abupd\ (throw\ a)\ .;\ Q\leftarrow\diamondsuit\}]\!]\Longrightarrow$
$G,A\vdash\{Normal\ P\}\ .Throw\ e.\ \{Q\}$

| *Try*: $[\![G,A\vdash\{Normal\ P\}\ .c1.\ \{SXAlloc\ G\ Q\};$
$G,A\vdash\{Q\ \wedge.\ (\lambda s.\ \ G,s\vdash catch\ C)\ ;.\ new\text{-}xcpt\text{-}var\ vn\}\ .c2.\ \{R\};$
$(Q\ \wedge.\ (\lambda s.\ \neg G,s\vdash catch\ C))\Rightarrow R]\!]\Longrightarrow$
$G,A\vdash\{Normal\ P\}\ .Try\ c1\ Catch(C\ vn)\ c2.\ \{R\}$

| *Fin*: $[\![G,A\vdash\{Normal\ P\}\ .c1.\ \{Q\};$
$\forall\, x.\ G,A\vdash\{Q\ \wedge.\ (\lambda s.\ x = fst\ s)\ ;.\ abupd\ (\lambda x.\ None)\}$
$.c2.\ \{abupd\ (abrupt\text{-}if\ (x\neq None)\ x)\ .;\ R\}]\!]\Longrightarrow$
$G,A\vdash\{Normal\ P\}\ .c1\ Finally\ c2.\ \{R\}$

| *Done*: $G,A\vdash\{Normal\ (P\leftarrow\diamondsuit\ \wedge.\ initd\ C)\}\ .Init\ C.\ \{P\}$

| *Init*: $[\![the\ (class\ G\ C) = c;$
$G,A\vdash\{Normal\ ((P\ \wedge.\ Not\circ initd\ C)\ ;.\ supd\ (init\text{-}class\text{-}obj\ G\ C))\}$
$.(if\ C = Object\ then\ Skip\ else\ Init\ (super\ c)).\ \{Q\};$
$\forall\, l.\ G,A\vdash\{Q\ \wedge.\ (\lambda s.\ l = locals\ (store\ s))\ ;.\ set\text{-}lvars\ Map.empty\}$
$.init\ c.\ \{set\text{-}lvars\ l\ .;\ R\}]\!]\Longrightarrow$
$G,A\vdash\{Normal\ (P\ \wedge.\ Not\circ initd\ C)\}\ .Init\ C.\ \{R\}$

— Some dummy rules for the intermediate terms *Callee*, *InsInitE*, *InsInitV*, *FinA* only used by the smallstep semantics.
| *InsInitV*: $G,A\vdash\{Normal\ P\}\ InsInitV\ c\ v{=}\succ\ \{Q\}$
| *InsInitE*: $G,A\vdash\{Normal\ P\}\ InsInitE\ c\ e-\succ\ \{Q\}$
| *Callee*: $G,A\vdash\{Normal\ P\}\ Callee\ l\ e-\succ\ \{Q\}$
| *FinA*: $G,A\vdash\{Normal\ P\}\ .FinA\ a\ c.\ \{Q\}$

definition
$adapt\text{-}pre :: {'a}\ assn \Rightarrow {'a}\ assn \Rightarrow {'a}\ assn \Rightarrow {'a}\ assn$
where $adapt\text{-}pre\ P\ Q\ Q' = (\lambda Y\ s\ Z.\ \forall Y'\ s'.\ \exists Z'.\ P\ Y\ s\ Z' \wedge (Q\ Y'\ s'\ Z' \longrightarrow Q'\ Y'\ s'\ Z))$

rules derived by induction

lemma *cut-valid*: $[\![G,A'|\models ts;\ G,A|\models A']\!] \Longrightarrow G,A|\models ts$
⟨*proof*⟩

lemma *ax-thin* [*rule-format* (*no-asm*)]:
$G,(A'::{'a}\ triple\ set)|\vdash(ts::{'a}\ triple\ set) \Longrightarrow \forall A.\ A' \subseteq A \longrightarrow G,A|\vdash ts$
⟨*proof*⟩

lemma *ax-thin-insert*: $G,(A::{'a}\ triple\ set)\vdash(t::{'a}\ triple) \Longrightarrow G,insert\ x\ A\vdash t$
⟨*proof*⟩

lemma *subset-mtriples-iff*:
$ts \subseteq \{\{P\}\ mb{-}{\succ}\ \{Q\}\ |\ ms\} = (\exists ms'.\ ms' \subseteq ms \wedge\ ts = \{\{P\}\ mb{-}{\succ}\ \{Q\}\ |\ ms'\})$
⟨*proof*⟩

lemma *weaken*:
$G,(A::{'a}\ triple\ set)|\vdash(ts'::{'a}\ triple\ set) \Longrightarrow \forall ts.\ ts \subseteq ts' \longrightarrow G,A|\vdash ts$
⟨*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*: $[\![G,(A::{'a}\ triple\ set)\vdash\{P'::{'a}\ assn\}\ t{\succ}\ \{Q'\};$
$\forall Y\ s\ Z.\ P\ Y\ s\ Z \longrightarrow (\forall Y'\ s'.\ (\forall Y\ Z'.\ P'\ Y\ s\ Z' \longrightarrow Q'\ Y'\ s'\ Z') \longrightarrow$
$Q\ Y'\ s'\ Z)]\!]$
$\Longrightarrow\ G,A\vdash\{P\ ::{'a}\ assn\}\ t{\succ}\ \{Q\ \}$
⟨*proof*⟩
lemma *conseq12'*: $[\![G,(A::{'a}\ triple\ set)\vdash\{P'::{'a}\ assn\}\ t{\succ}\ \{Q'\};\ \forall s\ Y'\ s'.$
$(\forall Y\ Z.\ P'\ Y\ s\ Z \longrightarrow Q'\ Y'\ s'\ Z) \longrightarrow$
$(\forall Y\ Z.\ P\ \ Y\ s\ Z \longrightarrow Q\ \ Y'\ s'\ Z)]\!]$
$\Longrightarrow\ G,A\vdash\{P::{'a}\ assn\ \}\ t{\succ}\ \{Q\ \}$
⟨*proof*⟩

lemma *conseq12-from-conseq12'*: $[\![G,(A::{'a}\ triple\ set)\vdash\{P'::{'a}\ assn\}\ t{\succ}\ \{Q'\};$
$\forall Y\ s\ Z.\ P\ Y\ s\ Z \longrightarrow (\forall Y'\ s'.\ (\forall Y\ Z'.\ P'\ Y\ s\ Z' \longrightarrow Q'\ Y'\ s'\ Z') \longrightarrow$
$Q\ Y'\ s'\ Z)]\!]$
$\Longrightarrow\ G,A\vdash\{P::{'a}\ assn\}\ t{\succ}\ \{Q\ \}$
⟨*proof*⟩

lemma *conseq1*: $[\![G,(A::{'a}\ triple\ set)\vdash\{P'::{'a}\ assn\}\ t{\succ}\ \{Q\};\ P \Rightarrow P']\!]$
$\Longrightarrow G,A\vdash\{P::{'a}\ assn\}\ t{\succ}\ \{Q\}$
⟨*proof*⟩

lemma *conseq2*: $[\![G,(A::{'a}\ triple\ set)\vdash\{P::{'a}\ assn\}\ t{\succ}\ \{Q'\};\ Q' \Rightarrow Q]\!]$
$\Longrightarrow G,A\vdash\{P::{'a}\ assn\}\ t{\succ}\ \{Q\}$
⟨*proof*⟩

lemma *ax-escape*:
$\llbracket\forall\, Y\ s\ Z.\ P\ Y\ s\ Z$
$\longrightarrow G,(A{::}'a\ triple\ set)\vdash\{\lambda Y'\ s'\ (Z'{::}'a).\ (Y',s') = (Y,s)\}$
$t\succ$
$\{\lambda Y\ s\ Z'.\ Q\ Y\ s\ Z\}$
$\rrbracket \Longrightarrow\ G,A\vdash\{P{::}'a\ assn\}\ t\succ\ \{Q{::}'a\ assn\}$
⟨*proof*⟩

lemma *ax-constant*: $\llbracket\ C \Longrightarrow G,(A{::}'a\ triple\ set)\vdash\{P{::}'a\ assn\}\ t\succ\ \{Q\}\rrbracket$
$\Longrightarrow G,A\vdash\{\lambda Y\ s\ Z.\ C \wedge P\ Y\ s\ Z\}\ t\succ\ \{Q\}$
⟨*proof*⟩

lemma *ax-impossible* [*intro*]:
$G,(A{::}'a\ triple\ set)\vdash\{\lambda Y\ s\ Z.\ False\}\ t\succ\ \{Q{::}'a\ assn\}$
⟨*proof*⟩

lemma *ax-nochange-lemma*: $\llbracket P\ Y\ s;\ All\ ((=)\ w)\rrbracket \Longrightarrow P\ w\ s$
⟨*proof*⟩

lemma *ax-nochange*:
$G,(A{::}(res \times state)\ triple\ set)\vdash\{\lambda Y\ s\ Z.\ (Y,s){=}Z\}\ t\succ\ \{\lambda Y\ s\ Z.\ (Y,s){=}Z\}$
$\Longrightarrow G,A\vdash\{P{::}(res \times state)\ assn\}\ t\succ\ \{P\}$
⟨*proof*⟩

lemma *ax-trivial*: $G,(A{::}'a\ triple\ set)\vdash\{P{::}'a\ assn\}\ \ t\succ\ \{\lambda Y\ s\ Z.\ True\}$
⟨*proof*⟩

lemma *ax-disj*:
$\llbracket G,(A{::}'a\ triple\ set)\vdash\{P1{::}'a\ assn\}\ t\succ\ \{Q1\};\ G,A\vdash\{P2{::}'a\ assn\}\ t\succ\ \{Q2\}\rrbracket$
$\Longrightarrow\ G,A\vdash\{\lambda Y\ s\ Z.\ P1\ Y\ s\ Z \vee P2\ Y\ s\ Z\}\ t\succ\ \{\lambda Y\ s\ Z.\ Q1\ Y\ s\ Z \vee Q2\ Y\ s\ Z\}$
⟨*proof*⟩

lemma *ax-supd-shuffle*:
$(\exists\, Q.\ G,(A{::}'a\ triple\ set)\vdash\{P{::}'a\ assn\}\ .c1.\ \{Q\} \wedge G,A\vdash\{Q\ ;.\ f\}\ .c2.\ \{R\}) =$
$(\exists\, Q'.\ G,A\vdash\{P\}\ .c1.\ \{f\ .;\ Q'\} \wedge G,A\vdash\{Q'\}\ .c2.\ \{R\})$
⟨*proof*⟩

lemma *ax-cases*:
$\llbracket G,(A{::}'a\ triple\ set)\vdash\{P\ \wedge.\ \qquad C\}\ t\succ\ \{Q{::}'a\ assn\};$
$G,A\vdash\{P\ \wedge.\ Not \circ C\}\ t\succ\ \{Q\}\rrbracket \Longrightarrow G,A\vdash\{P\}\ t\succ\ \{Q\}$
⟨*proof*⟩

lemma *ax-adapt*: $G,(A{::}'a\ triple\ set)\vdash\{P{::}'a\ assn\}\ t\succ\ \{Q\}$
$\Longrightarrow G,A\vdash\{adapt\text{-}pre\ P\ Q\ Q'\}\ t\succ\ \{Q'\}$
⟨*proof*⟩

lemma *adapt-pre-adapts*: $G,(A{::}'a\ triple\ set)\models\{P{::}'a\ assn\}\ t\succ\ \{Q\}$
$\longrightarrow G,A\models\{adapt\text{-}pre\ P\ Q\ Q'\}\ t\succ\ \{Q'\}$
⟨*proof*⟩

lemma *adapt-pre-weakest*:
$\forall G\ (A::'a\ triple\ set)\ t.\ G,A\models\{P\}\ t\succ\ \{Q\} \longrightarrow G,A\models\{P'\}\ t\succ\ \{Q'\} \Longrightarrow$
$P' \Rightarrow adapt\text{-}pre\ P\ Q\ (Q'::'a\ assn)$
⟨*proof*⟩

lemma *peek-and-forget1-Normal*:
$G,(A::'a\ triple\ set)\vdash\{Normal\ P\}\ t\succ\ \{Q::'a\ assn\}$
$\Longrightarrow G,A\vdash\{Normal\ (P \wedge.\ p)\}\ t\succ\ \{Q\}$
⟨*proof*⟩

lemma *peek-and-forget1*:
$G,(A::'a\ triple\ set)\vdash\{P::'a\ assn\}\ t\succ\ \{Q\}$
$\Longrightarrow G,A\vdash\{P \wedge.\ p\}\ t\succ\ \{Q\}$
⟨*proof*⟩

lemmas *ax-NormalD* = *peek-and-forget1* [*of - - - - - normal*]

lemma *peek-and-forget2*:
$G,(A::'a\ triple\ set)\vdash\{P::'a\ assn\}\ t\succ\ \{Q \wedge.\ p\}$
$\Longrightarrow G,A\vdash\{P\}\ t\succ\ \{Q\}$
⟨*proof*⟩

lemma *ax-subst-Val-allI*:
$\forall v.\ G,(A::'a\ triple\ set)\vdash\{(P'\ \ \ v\ )\leftarrow Val\ v\}\ t\succ\ \{(Q\ v)::'a\ assn\}$
$\Longrightarrow\ \forall v.\ G,A\vdash\{(\lambda w:.\ P'\ (the\text{-}In1\ w))\leftarrow Val\ v\}\ t\succ\ \{Q\ v\}$
⟨*proof*⟩

lemma *ax-subst-Var-allI*:
$\forall v.\ G,(A::'a\ triple\ set)\vdash\{(P'\ \ \ v\ )\leftarrow Var\ v\}\ t\succ\ \{(Q\ v)::'a\ assn\}$
$\Longrightarrow\ \forall v.\ G,A\vdash\{(\lambda w:.\ P'\ (the\text{-}In2\ w))\leftarrow Var\ v\}\ t\succ\ \{Q\ v\}$
⟨*proof*⟩

lemma *ax-subst-Vals-allI*:
$(\forall v.\ G,(A::'a\ triple\ set)\vdash\{(\ \ P'\ \ \ v\ )\leftarrow Vals\ v\}\ t\succ\ \{(Q\ v)::'a\ assn\})$
$\Longrightarrow\ \forall v.\ G,A\vdash\{(\lambda w:.\ P'\ (the\text{-}In3\ w))\leftarrow Vals\ v\}\ t\succ\ \{Q\ v\}$
⟨*proof*⟩

alternative axioms

lemma *ax-Lit2*:
$G,(A::'a\ triple\ set)\vdash\{Normal\ P::'a\ assn\}\ Lit\ v-\succ\ \{Normal\ (P\downarrow=Val\ v)\}$
⟨*proof*⟩
lemma *ax-Lit2-test-complete*:
$G,(A::'a\ triple\ set)\vdash\{Normal\ (P\leftarrow Val\ v)::'a\ assn\}\ Lit\ v-\succ\ \{P\}$
⟨*proof*⟩

lemma *ax-LVar2*: $G,(A::'a\ triple\ set)\vdash\{Normal\ P::'a\ assn\}\ LVar\ vn{=}\succ\ \{Normal\ (\lambda s..\ P\downarrow=Var\ (lvar\ vn\ s))\}$
⟨*proof*⟩

lemma *ax-Super2*: $G,(A::'a\ triple\ set)\vdash$
$\{Normal\ P::'a\ assn\}\ Super-\succ\ \{Normal\ (\lambda s..\ P\downarrow=Val\ (val\text{-}this\ s))\}$
⟨*proof*⟩

lemma *ax-Nil2*:
$G,(A::'a\ triple\ set)\vdash\{Normal\ P::'a\ assn\}\ []\doteq\succ\ \{Normal\ (P\downarrow=Vals\ [])\}$
⟨*proof*⟩

misc derived structural rules

lemma *ax-finite-mtriples-lemma*: ⟦*F* ⊆ *ms*; *finite ms*; ∀ (*C*,*sig*)∈*ms*.
  G,(*A*::′*a triple set*)⊢{*Normal* (*P C sig*)::′*a assn*} *mb C sig*−≻ {*Q C sig*}⟧ ⟹
    G,*A*|⊢{{*P*} *mb*−≻ {*Q*} | *F*}
⟨*proof*⟩
lemmas *ax-finite-mtriples* = *ax-finite-mtriples-lemma* [*OF subset-refl*]

lemma *ax-derivs-insertD*:
G,(*A*::′*a triple set*)|⊢*insert* (*t*::′*a triple*) *ts* ⟹ *G*,*A*⊢*t* ∧ *G*,*A*|⊢*ts*
⟨*proof*⟩

lemma *ax-methods-spec*:
⟦*G*,(*A*::′*a triple set*)|⊢*case-prod f* ‘ *ms*; (*C*,*sig*) ∈ *ms*⟧⟹ *G*,*A*⊢((*f C sig*)::′*a triple*)
⟨*proof*⟩

lemma *ax-finite-pointwise-lemma* [*rule-format*]: ⟦*F* ⊆ *ms*; *finite ms*⟧ ⟹
 ((∀ (*C*,*sig*)∈*F*. *G*,(*A*::′*a triple set*)⊢(*f C sig*::′*a triple*)) ⟶ (∀ (*C*,*sig*)∈*ms*. *G*,*A*⊢(*g C sig*::′*a triple*))) ⟶
   G,*A*|⊢*case-prod f* ‘ *F* ⟶ *G*,*A*|⊢*case-prod g* ‘ *F*
⟨*proof*⟩
lemmas *ax-finite-pointwise* = *ax-finite-pointwise-lemma* [*OF subset-refl*]

lemma *ax-no-hazard*:
 G,(*A*::′*a triple set*)⊢{*P* ∧. *type-ok G t*} *t*≻ {*Q*::′*a assn*} ⟹ *G*,*A*⊢{*P*} *t*≻ {*Q*}
⟨*proof*⟩

lemma *ax-free-wt*:
(∃ *T L C*. (|*prg*=*G*,*cls*=*C*,*lcl*=*L*|)⊢*t*::*T*)
 ⟶ *G*,(*A*::′*a triple set*)⊢{*Normal P*} *t*≻ {*Q*::′*a assn*} ⟹
 G,*A*⊢{*Normal P*} *t*≻ {*Q*}
⟨*proof*⟩

⟨*ML*⟩
declare *ax-Abrupts* [*intro*!]

lemmas *ax-Normal-cases* = *ax-cases* [*of - - - normal*]

lemma *ax-Skip* [*intro*!]: *G*,(*A*::′*a triple set*)⊢{*P*←◇} .*Skip*. {*P*::′*a assn*}
⟨*proof*⟩
lemmas *ax-SkipI* = *ax-Skip* [*THEN conseq1*]

derived rules for methd call

lemma *ax-Call-known-DynT*:
⟦*G*⊢*IntVir*→*C*⪯*statT*;
 ∀ *a vs l*. *G*,*A*⊢{(*R a*←*Vals vs* ∧. (λ*s*. *l* = *locals* (*store s*)) ;.
 init-lvars G C (|*name*=*mn*,*parTs*=*pTs*|) *IntVir a vs*)}
   Methd C (|*name*=*mn*,*parTs*=*pTs*|)−≻ {*set-lvars l* .; *S*};
 ∀ *a*. *G*,*A*⊢{*Q*←*Val a*} *args*≐≻
    {*R a* ∧. (λ*s*. *C* = *obj-class* (*the* (*heap* (*store s*) (*the-Addr a*))) ∧
          C = *invocation-declclass*
             G IntVir (*store s*) *a statT* (|*name*=*mn*,*parTs*=*pTs*|))};
    G,(*A*::′*a triple set*)⊢{*Normal P*} *e*−≻ {*Q*::′*a assn*}⟧
 ⟹ *G*,*A*⊢{*Normal P*} {*accC*,*statT*,*IntVir*}*e*·*mn*({*pTs*}*args*)−≻ {*S*}
⟨*proof*⟩

lemma *ax-Call-Static*:

$[\![\forall a\ vs\ l.\ G,A\vdash\{R\ a\leftarrow Vals\ vs\ \wedge.\ (\lambda s.\ l = locals\ (store\ s))\ ;.$
$init\text{-}lvars\ G\ C\ (\!|name{=}mn,parTs{=}pTs|\!)\ Static\ any\text{-}Addr\ vs\}$
$Methd\ C\ (\!|name{=}mn,parTs{=}pTs|\!){-}\!\succ\ \{set\text{-}lvars\ l\ .;\ S\};$
$G,A\vdash\{Normal\ P\}\ e{-}\!\succ\ \{Q\};$
$\forall\ a.\ G,(A{::}'a\ triple\ set)\vdash\{Q\leftarrow Val\ a\}\ args\dot{=}\!\succ\ \{(R{::}val \Rightarrow\ 'a\ assn)\ \ a$
$\wedge.\ (\lambda\ s.\ C{=}invocation\text{-}declclass$
$G\ Static\ (store\ s)\ a\ statT\ (\!|name{=}mn,parTs{=}pTs|\!))\}$
$]\!]\ \Longrightarrow\ G,A\vdash\{Normal\ P\}\ \{accC,statT,Static\}e{\cdot}mn(\{pTs\}args){-}\!\succ\ \{S\}$
$\langle proof\rangle$

lemma *ax-Methd1*:
$[\![G,A\cup\{\{P\}\ Methd{-}\!\succ\ \{Q\}\ |\ ms\}|\!\vdash\ \{\{P\}\ body\ G{-}\!\succ\ \{Q\}\ |\ ms\};\ (C,sig)\in ms]\!]\Longrightarrow$
$G,A\vdash\{Normal\ (P\ C\ sig)\}\ Methd\ C\ sig{-}\!\succ\ \{Q\ C\ sig\}$
$\langle proof\rangle$

lemma *ax-MethdN*:
$G,insert(\{Normal\ P\}\ Methd\ \ C\ sig{-}\!\succ\ \{Q\})\ A\vdash$
$\{Normal\ P\}\ body\ G\ C\ sig{-}\!\succ\ \{Q\}\Longrightarrow$
$G,A\vdash\{Normal\ P\}\ Methd\ \ \ C\ sig{-}\!\succ\ \{Q\}$
$\langle proof\rangle$

lemma *ax-StatRef*:
$G,(A{::}'a\ triple\ set)\vdash\{Normal\ (P\leftarrow Val\ Null)\}\ StatRef\ rt{-}\!\succ\ \{P{::}'a\ assn\}$
$\langle proof\rangle$

rules derived from Init and Done

lemma *ax-InitS*: $[\![the\ (class\ G\ C) = c;\ C \neq Object;$
$\forall l.\ G,A\vdash\{Q\ \wedge.\ (\lambda s.\ l = locals\ (store\ s))\ ;.\ set\text{-}lvars\ Map.empty\}$
$.init\ c.\ \{set\text{-}lvars\ l\ .;\ R\};$
$G,A\vdash\{Normal\ ((P\ \wedge.\ Not\circ initd\ C)\ ;.\ supd\ (init\text{-}class\text{-}obj\ G\ C))\}$
$.Init\ (super\ c).\ \{Q\}]\!]\Longrightarrow$
$G,(A{::}'a\ triple\ set)\vdash\{Normal\ (P\ \wedge.\ Not\circ initd\ C)\}\ .Init\ C.\ \{R{::}'a\ assn\}$
$\langle proof\rangle$

lemma *ax-Init-Skip-lemma*:
$\forall l.\ G,(A{::}'a\ triple\ set)\vdash\{P\leftarrow\Diamond\ \wedge.\ (\lambda s.\ l = locals\ (store\ s))\ ;.\ set\text{-}lvars\ l'\}$
$.Skip.\ \{(set\text{-}lvars\ l\ .;\ P){::}'a\ assn\}$
$\langle proof\rangle$

lemma *ax-triv-InitS*: $[\![the\ (class\ G\ C) = c;init\ c = Skip;\ C\neq Object;$
$P\leftarrow\Diamond\ \Rightarrow\ (supd\ (init\text{-}class\text{-}obj\ G\ C)\ .;\ P);$
$G,A\vdash\{Normal\ (P\ \wedge.\ initd\ C)\}\ .Init\ (super\ c).\ \{(P\ \wedge.\ initd\ C)\leftarrow\Diamond\}]\!]\Longrightarrow$
$G,(A{::}'a\ triple\ set)\vdash\{Normal\ P\leftarrow\Diamond\}\ .Init\ C.\ \{(P\ \wedge.\ initd\ C){::}'a\ assn\}$
$\langle proof\rangle$

lemma *ax-Init-Object*: $wf\text{-}prog\ G\Longrightarrow G,(A{::}'a\ triple\ set)\vdash$
$\{Normal\ ((supd\ (init\text{-}class\text{-}obj\ G\ Object)\ .;\ P\leftarrow\Diamond)\ \wedge.\ Not\circ initd\ Object)\}$
$.Init\ Object.\ \{(P\ \wedge.\ initd\ Object){::}'a\ assn\}$
$\langle proof\rangle$

lemma *ax-triv-Init-Object*: $[\![wf\text{-}prog\ G;$
$(P{::}'a\ assn)\Rightarrow(supd\ (init\text{-}class\text{-}obj\ G\ Object)\ .;\ P)]\!]\Longrightarrow$
$G,(A{::}'a\ triple\ set)\vdash\{Normal\ P\leftarrow\Diamond\}\ .Init\ Object.\ \{P\ \wedge.\ initd\ Object\}$
$\langle proof\rangle$

introduction rules for Alloc and SXAlloc

lemma *ax-SXAlloc-Normal*:

$G,(A::'a\ triple\ set)\vdash\{P::'a\ assn\}\ .c.\ \{Normal\ Q\}$
$\Longrightarrow G,A\vdash\{P\}\ .c.\ \{SXAlloc\ G\ Q\}$
⟨*proof*⟩

lemma *ax-Alloc*:
$G,(A::'a\ triple\ set)\vdash\{P::'a\ assn\}\ t\succ$
$\{Normal\ (\lambda Y\ (x,s)\ Z.\ (\forall a.\ new\text{-}Addr\ (heap\ s) = Some\ a \longrightarrow$
$Q\ (Val\ (Addr\ a))\ (Norm(init\text{-}obj\ G\ (CInst\ C)\ (Heap\ a)\ s))\ Z))\ \wedge.$
$heap\text{-}free\ (Suc\ (Suc\ 0))\}$
$\Longrightarrow G,A\vdash\{P\}\ t\succ\ \{Alloc\ G\ (CInst\ C)\ Q\}$
⟨*proof*⟩

lemma *ax-Alloc-Arr*:
$G,(A::'a\ triple\ set)\vdash\{P::'a\ assn\}\ t\succ$
$\{\lambda Val{:}i{:}.\ Normal\ (\lambda Y\ (x,s)\ Z.\ \neg the\text{-}Intg\ i<0\ \wedge$
$(\forall a.\ new\text{-}Addr\ (heap\ s) = Some\ a \longrightarrow$
$Q\ (Val\ (Addr\ a))\ (Norm\ (init\text{-}obj\ G\ (Arr\ T\ (the\text{-}Intg\ i))\ (Heap\ a)\ s))\ Z))\ \wedge.$
$heap\text{-}free\ (Suc\ (Suc\ 0))\}$
$\Longrightarrow$
$G,A\vdash\{P\}\ t\succ\ \{\lambda Val{:}i{:}.\ abupd\ (check\text{-}neg\ i)\ .;\ Alloc\ G\ (Arr\ T(the\text{-}Intg\ i))\ Q\}$
⟨*proof*⟩

lemma *ax-SXAlloc-catch-SXcpt*:
$[\![G,(A::'a\ triple\ set)\vdash\{P::'a\ assn\}\ t\succ$
$\{(\lambda Y\ (x,s)\ Z.\ x{=}Some\ (Xcpt\ (Std\ xn))\ \wedge$
$(\forall a.\ new\text{-}Addr\ (heap\ s) = Some\ a \longrightarrow$
$Q\ Y\ (Some\ (Xcpt\ (Loc\ a)),init\text{-}obj\ G\ (CInst\ (SXcpt\ xn))\ (Heap\ a)\ s)\ Z))$
$\wedge.\ heap\text{-}free\ (Suc\ (Suc\ 0))\}]\!]$
$\Longrightarrow$
$G,A\vdash\{P\}\ t\succ\ \{SXAlloc\ G\ (\lambda Y\ s\ Z.\ Q\ Y\ s\ Z\ \wedge\ G,s\vdash catch\ SXcpt\ xn)\}$
⟨*proof*⟩

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* ⇒ *nat* ⇒ *'a triple* ⇒ *bool* (‹-⊨-::-›[*61*,*0*, *58*] *57*)
where
G⊨n::t =
(*case t of* {P} t≻ {Q} ⇒
∀ Y s Z. P Y s Z ⟶ (∀ L. s::≼(G,L)
⟶ (∀ T C A. (*normal s* ⟶ ((|*prg*=G,*cls*=C,*lcl*=L|)⊢t::T ∧
(|*prg*=G,*cls*=C,*lcl*=L|)⊢*dom* (*locals* (*store s*))»t»A)) ⟶
(∀ Y' s'. G⊢s −t≻−n→ (Y',s') ⟶ Q Y' s' Z ∧ s'::≼(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* ⇒ *'a triples* ⇒ *'a triples* ⇒ *bool* (‹-,-|⊨::-› [*61*,*58*,*58*] *57*)
where G,A|⊨::ts = (∀ n. (∀ t∈A. G⊨n::t) ⟶ (∀ t∈ts. G⊨n::t))

lemma *triple-valid2-def2*: G⊨n::{P} t≻ {Q} =
(∀ Y s Z. P Y s Z ⟶ (∀ Y' s'. G⊢s −t≻−n→ (Y',s')⟶
(∀ L. s::≼(G,L) ⟶ (∀ T C A. (*normal s* ⟶ ((|*prg*=G,*cls*=C,*lcl*=L|)⊢t::T ∧
(|*prg*=G,*cls*=C,*lcl*=L|)⊢*dom* (*locals* (*store s*))»t»A)) ⟶
Q Y' s' Z ∧ s'::≼(G,L)))))
⟨*proof*⟩

lemma *triple-valid2-eq* [*rule-format* (*no-asm*)]:
wf-prog G ==> *triple-valid2* G = *triple-valid* G
⟨*proof*⟩

lemma *ax-valids2-eq*: *wf-prog* G ⟹ G,A|⊨::ts = G,A|⊨ts
⟨*proof*⟩

lemma *triple-valid2-Suc* [*rule-format* (*no-asm*)]: G⊨*Suc* n::t ⟶ G⊨n::t
⟨*proof*⟩

lemma *Methd-triple-valid2-0*: G⊨*0*::{*Normal P*} *Methd C sig*−≻ {Q}
⟨*proof*⟩

lemma *Methd-triple-valid2-SucI*:
⟦*G*⊨*n*::{*Normal P*} *body G C sig*−≻{*Q*}⟧
⟹ *G*⊨*Suc n*::{*Normal P*} *Methd C sig*−≻ {*Q*}
⟨*proof*⟩

lemma *triples-valid2-Suc*:
Ball ts (*triple-valid2 G* (*Suc n*)) ⟹ *Ball ts* (*triple-valid2 G n*)
⟨*proof*⟩

lemma *G*|⊨*n*:*insert t A* = (*G*⊨*n*:*t* ∧ *G*|⊨*n*:*A*)
⟨*proof*⟩

soundness

lemma *Methd-sound*:
assumes *recursive*: *G*,*A*∪ {{*P*} *Methd*−≻ {*Q*} | *ms*}|⊨::{{*P*} *body G*−≻ {*Q*} | *ms*}
shows *G*,*A*|⊨::{{*P*} *Methd*−≻ {*Q*} | *ms*}
⟨*proof*⟩

lemma *valids2-inductI*: ∀ *s t n Y′ s′*. *G*⊢*s*−*t*≻−*n*→ (*Y′*,*s′*) ⟶ *t* = *c* ⟶
Ball A (*triple-valid2 G n*) ⟶ (∀ *Y Z*. *P Y s Z* ⟶
(∀ *L*. *s*::≼(*G*,*L*) ⟶
(∀ *T C A*. (*normal s* ⟶ ((|*prg*=*G*,*cls*=*C*,*lcl*=*L*|)⊢*t*::*T*) ∧
(|*prg*=*G*,*cls*=*C*,*lcl*=*L*|)⊢*dom* (*locals* (*store s*))»*t*»*A*) ⟶
Q Y′ s′ Z ∧ *s′*::≼(*G*, *L*)))) ⟹
G,*A*|⊨::{ {*P*} *c*≻ {*Q*}}
⟨*proof*⟩

lemma *da-good-approx-evalnE* [*consumes 4*]:
assumes *evaln*: *G*⊢*s0* −*t*≻−*n*→ (*v*, *s1*)
and *wt*: (|*prg*=*G*,*cls*=*C*,*lcl*=*L*|)⊢*t*::*T*
and *da*: (|*prg*=*G*,*cls*=*C*,*lcl*=*L*|)⊢ *dom* (*locals* (*store s0*)) »*t*» *A*
and *wf*: *wf-prog G*
and *elim*: ⟦*normal s1* ⟹ *nrm A* ⊆ *dom* (*locals* (*store s1*));
⋀ *l*. ⟦*abrupt s1* = *Some* (*Jump* (*Break l*)); *normal s0*⟧
⟹ *brk A l* ⊆ *dom* (*locals* (*store s1*));
⟦*abrupt s1* = *Some* (*Jump Ret*);*normal s0*⟧
⟹*Result* ∈ *dom* (*locals* (*store s1*))
⟧ ⟹ *P*
shows *P*
⟨*proof*⟩

lemma *validI*:
assumes *I*: ⋀ *n s0 L accC T C v s1 Y Z*.
⟦∀ *t*∈*A*. *G*⊨*n*::*t*; *s0*::≼(*G*,*L*);
normal s0 ⟹ (|*prg*=*G*,*cls*=*accC*,*lcl*=*L*|)⊢*t*::*T*;
normal s0 ⟹ (|*prg*=*G*,*cls*=*accC*,*lcl*=*L*|)⊢*dom* (*locals* (*store s0*))»*t*»*C*;
G⊢*s0* −*t*≻−*n*→ (*v*,*s1*); *P Y s0 Z*⟧ ⟹ *Q v s1 Z* ∧ *s1*::≼(*G*,*L*)
shows *G*,*A*|⊨::{ {*P*} *t*≻ {*Q*} }
⟨*proof*⟩

declare [[*simproc add*: *wt-expr wt-var wt-exprs wt-stmt*]]

lemma *valid-stmtI*:
assumes *I*: ⋀ *n s0 L accC C s1 Y Z*.
⟦∀ *t*∈*A*. *G*⊨*n*::*t*; *s0*::≼(*G*,*L*);

normal s0 ⟹ ⦇*prg*=*G*,*cls*=*accC*,*lcl*=*L*⦈⊢*c*::√;
normal s0 ⟹ ⦇*prg*=*G*,*cls*=*accC*,*lcl*=*L*⦈⊢*dom* (*locals* (*store s0*))»$\langle c\rangle_s$»*C*;
G⊢*s0* −*c*−*n*→ *s1*; *P Y s0 Z*⟧ ⟹ *Q* ♢ *s1 Z* ∧ *s1*::≼(*G*,*L*)
shows *G*,*A*|⊨::{ {*P*} $\langle c\rangle_s$≻ {*Q*} }
⟨*proof*⟩

lemma *valid-stmt-NormalI*:
assumes *I*: ⋀ *n s0 L accC C s1 Y Z*.
⟦∀ *t*∈*A*. *G*⊨*n*::*t*; *s0*::≼(*G*,*L*); *normal s0*; ⦇*prg*=*G*,*cls*=*accC*,*lcl*=*L*⦈⊢*c*::√;
⦇*prg*=*G*,*cls*=*accC*,*lcl*=*L*⦈⊢*dom* (*locals* (*store s0*))»$\langle c\rangle_s$»*C*;
G⊢*s0* −*c*−*n*→ *s1*; (*Normal P*) *Y s0 Z*⟧ ⟹ *Q* ♢ *s1 Z* ∧ *s1*::≼(*G*,*L*)
shows *G*,*A*|⊨::{ {*Normal P*} $\langle c\rangle_s$≻ {*Q*} }
⟨*proof*⟩

lemma *valid-var-NormalI*:
assumes *I*: ⋀ *n s0 L accC T C vf s1 Y Z*.
⟦∀ *t*∈*A*. *G*⊨*n*::*t*; *s0*::≼(*G*,*L*); *normal s0*;
⦇*prg*=*G*,*cls*=*accC*,*lcl*=*L*⦈⊢*t*::=*T*;
⦇*prg*=*G*,*cls*=*accC*,*lcl*=*L*⦈⊢*dom* (*locals* (*store s0*))»$\langle t\rangle_v$»*C*;
G⊢*s0* −*t*=≻*vf*−*n*→ *s1*; (*Normal P*) *Y s0 Z*⟧
⟹ *Q* (*In2 vf*) *s1 Z* ∧ *s1*::≼(*G*,*L*)
shows *G*,*A*|⊨::{ {*Normal P*} $\langle t\rangle_v$≻ {*Q*} }
⟨*proof*⟩

lemma *valid-expr-NormalI*:
assumes *I*: ⋀ *n s0 L accC T C v s1 Y Z*.
⟦∀ *t*∈*A*. *G*⊨*n*::*t*; *s0*::≼(*G*,*L*); *normal s0*;
⦇*prg*=*G*,*cls*=*accC*,*lcl*=*L*⦈⊢*t*::−*T*;
⦇*prg*=*G*,*cls*=*accC*,*lcl*=*L*⦈⊢*dom* (*locals* (*store s0*))»$\langle t\rangle_e$»*C*;
G⊢*s0* −*t*−≻*v*−*n*→ *s1*; (*Normal P*) *Y s0 Z*⟧
⟹ *Q* (*In1 v*) *s1 Z* ∧ *s1*::≼(*G*,*L*)
shows *G*,*A*|⊨::{ {*Normal P*} $\langle t\rangle_e$≻ {*Q*} }
⟨*proof*⟩

lemma *valid-expr-list-NormalI*:
assumes *I*: ⋀ *n s0 L accC T C vs s1 Y Z*.
⟦∀ *t*∈*A*. *G*⊨*n*::*t*; *s0*::≼(*G*,*L*); *normal s0*;
⦇*prg*=*G*,*cls*=*accC*,*lcl*=*L*⦈⊢*t*::≐*T*;
⦇*prg*=*G*,*cls*=*accC*,*lcl*=*L*⦈⊢*dom* (*locals* (*store s0*))»$\langle t\rangle_l$»*C*;
G⊢*s0* −*t*≐≻*vs*−*n*→ *s1*; (*Normal P*) *Y s0 Z*⟧
⟹ *Q* (*In3 vs*) *s1 Z* ∧ *s1*::≼(*G*,*L*)
shows *G*,*A*|⊨::{ {*Normal P*} $\langle t\rangle_l$≻ {*Q*} }
⟨*proof*⟩

lemma *validE* [*consumes 5*]:
assumes *valid*: *G*,*A*|⊨::{ {*P*} *t*≻ {*Q*} }
and *P*: *P Y s0 Z*
and *valid-A*: ∀ *t*∈*A*. *G*⊨*n*::*t*
and *conf*: *s0*::≼(*G*,*L*)
and *eval*: *G*⊢*s0* −*t*≻−*n*→ (*v*,*s1*)
and *wt*: *normal s0* ⟹ ⦇*prg*=*G*,*cls*=*accC*,*lcl*=*L*⦈⊢*t*::*T*
and *da*: *normal s0* ⟹ ⦇*prg*=*G*,*cls*=*accC*,*lcl*=*L*⦈⊢*dom* (*locals* (*store s0*))»*t*»*C*
and *elim*: ⟦*Q v s1 Z*; *s1*::≼(*G*,*L*)⟧ ⟹ *concl*
shows *concl*
⟨*proof*⟩

lemma *all-empty*: (∀ *x*. *P*) = *P*
⟨*proof*⟩

corollary *evaln-type-sound*:
  assumes *evaln*: $G\vdash s0\ -t\succ-n\rightarrow (v,s1)$ **and**
    wt: $(\!|prg=G,cls=accC,lcl=L|\!)\vdash t::T$ **and**
    da: $(\!|prg=G,cls=accC,lcl=L|\!)\vdash dom\ (locals\ (store\ s0))\ »t»\ A$ **and**
   conf-s0: $s0::\preceq(G,L)$ **and**
    wf: *wf-prog G*
  shows $s1::\preceq(G,L) \wedge\ (normal\ s1 \longrightarrow G,L,store\ s1\vdash t\succ v::\preceq T) \wedge$
   $(error\text{-}free\ s0 = error\text{-}free\ s1)$
⟨*proof*⟩

corollary *dom-locals-evaln-mono-elim* [*consumes 1*]:
  assumes
  evaln: $G\vdash\ s0\ -t\succ-n\rightarrow (v,s1)$ **and**
   hyps: $[\![dom\ (locals\ (store\ s0)) \subseteq dom\ (locals\ (store\ s1));$
    $\bigwedge\ vv\ s\ val.\ [\![v=In2\ vv;\ normal\ s1]\!]$
      $\Longrightarrow dom\ (locals\ (store\ s))$
       $\subseteq dom\ (locals\ (store\ ((snd\ vv)\ val\ s)))]\!] \Longrightarrow P$
 shows *P*
⟨*proof*⟩

lemma *evaln-no-abrupt*:
  $\bigwedge s\ s'.\ [\![G\vdash s\ -t\succ-n\rightarrow (w,s');\ normal\ s']\!] \Longrightarrow normal\ s$
⟨*proof*⟩

declare *inj-term-simps* [*simp*]
lemma *ax-sound2*:
  assumes *wf*: *wf-prog G*
   and *deriv*: $G,A|\vdash ts$
  shows $G,A|\models::ts$
⟨*proof*⟩
declare *inj-term-simps* [*simp del*]

theorem *ax-sound*:
 $wf\text{-}prog\ G \Longrightarrow G,(A::'a\ triple\ set)|\vdash(ts::'a\ triple\ set) \Longrightarrow G,A|\models ts$
⟨*proof*⟩

lemma *sound-valid2-lemma*:
$[\![\forall\ v\ n.\ Ball\ A\ (triple\text{-}valid2\ G\ n) \longrightarrow P\ v\ n;\ Ball\ A\ (triple\text{-}valid2\ G\ n)]\!]$
 $\Longrightarrow P\ v\ n$
⟨*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 (-> Thomas Kleymann)

set of not yet initialzed classes

definition
nyinitcls :: *prog* ⇒ *state* ⇒ *qtname set*
where *nyinitcls G s* = {*C*. *is-class G C* ∧ ¬ *initd C s*}

lemma *nyinitcls-subset-class*: *nyinitcls G s* ⊆ {*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*) ≤ *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*)) ≤ *n* ⟹ *card* (*nyinitcls G* (*y*, *s*)) ≤ *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*: ⟦*nyinitcls G s* = {}; *is-class G C*⟧ ⟹ *initd C s*
⟨*proof*⟩

lemma *card-Suc-lemma*:
⟦*card* (*insert a A*) ≤ *Suc n*; *a*∉*A*; *finite A*⟧ ⟹ *card A* ≤ *n*
⟨*proof*⟩

lemma *nyinitcls-le-SucD*:
⟦*card* (*nyinitcls G* (*x*,*s*)) ≤ *Suc n*; ¬*inited C* (*globs s*); *class G C*=*Some y*⟧ ⟹
card (*nyinitcls G* (*x*,*init-class-obj G C s*)) ≤ *n*
⟨*proof*⟩

lemma *inited-gext′*: ⟦*s*≤|*s′*;*inited C* (*globs s*)⟧ ⟹ *inited C* (*globs s′*)
⟨*proof*⟩

lemma *nyinitcls-gext*: *snd s*≤|*snd s′* ⟹ *nyinitcls G s′* ⊆ *nyinitcls G s*
⟨*proof*⟩

lemma *card-nyinitcls-gext*:
⟦*snd s*≤|*snd s′*; *card* (*nyinitcls G s*) ≤ *n*⟧⟹ *card* (*nyinitcls G s′*) ≤ *n*
⟨*proof*⟩

init-le

definition
init-le :: *prog* ⇒ *nat* ⇒ *state* ⇒ *bool* (‹-⊢*init*≤-› [*51*,*51*] *50*)
where *G*⊢*init*≤*n* = (λ*s*. *card* (*nyinitcls G s*) ≤ *n*)

lemma *init-le-def2* [*simp*]: (*G*⊢*init*≤*n*) *s* = (*card* (*nyinitcls G s*)≤*n*)
⟨*proof*⟩

lemma *All-init-leD*:
∀ *n*::*nat*. *G*,(*A*::′*a triple set*)⊢{*P* ∧. *G*⊢*init*≤*n*} *t*≻ {*Q*::′*a assn*}
⟹ *G*,*A*⊢{*P*} *t*≻ {*Q*}
⟨*proof*⟩

Most General Triples and Formulas

definition
remember-init-state :: *state assn* (‹≐›)
where ≐ ≡ λ*Y s Z*. *s* = *Z*

lemma *remember-init-state-def2* [*simp*]: ≐ *Y* = (=)
⟨*proof*⟩

definition
MGF ::[*state assn*, *term*, *prog*] ⇒ *state triple* (‹{-} -≻ {-→}›[*3*,*65*,*3*]*62*)
where {*P*} *t*≻ {*G*→} = {*P*} *t*≻ {λ*Y s′ s*. *G*⊢*s* −*t*≻→ (*Y*,*s′*)}

definition
MGFn :: [*nat*, *term*, *prog*] ⇒ *state triple* (‹{=:-} -≻ {-→}›[*3*,*65*,*3*]*62*)
where {=:*n*} *t*≻ {*G*→} = {≐ ∧. *G*⊢*init*≤*n*} *t*≻ {*G*→}

lemma *MGF-valid*: *wf-prog G* ⟹ *G*,{}⊨{≐} *t*≻ {*G*→}
⟨*proof*⟩

lemma *MGF-res-eq-lemma* [*simp*]:
$(\forall Y'\ Y\ s.\ Y = Y' \wedge P\ s \longrightarrow Q\ s) = (\forall s.\ P\ s \longrightarrow Q\ s)$
$\langle proof \rangle$

lemma *MGFn-def2*:
$G,A\vdash\{=:n\}\ t\succ\ \{G\rightarrow\} = G,A\vdash\{\doteq \wedge.\ G\vdash init\leq n\}$
$t\succ\ \{\lambda Y\ s'\ s.\ G\vdash s\ -t\succ\rightarrow\ (Y,s')\}$
$\langle proof \rangle$

lemma *MGF-MGFn-iff*:
$G,(A::state\ triple\ set)\vdash\{\doteq\}\ t\succ\ \{G\rightarrow\} = (\forall n.\ G,A\vdash\{=:n\}\ t\succ\ \{G\rightarrow\})$
$\langle proof \rangle$

lemma *MGFnD*:
$G,(A::state\ triple\ set)\vdash\{=:n\}\ t\succ\ \{G\rightarrow\} \Longrightarrow$
$G,A\vdash\{(\lambda Y'\ s'\ s.\ s' = s \quad \wedge P\ s) \wedge.\ G\vdash init\leq n\}$
$t\succ\ \{(\lambda Y'\ s'\ s.\ G\vdash s-t\succ\rightarrow(Y',s') \wedge P\ s) \wedge.\ G\vdash init\leq n\}$
$\langle proof \rangle$
lemmas $MGFnD' = MGFnD$ [*of - - - - $\lambda x.$ True*]

To derive the most general formula, we can always assume a normal state in the precondition, since abrupt cases can be handled uniformally by the abrupt rule.

lemma *MGFNormalI*: $G,A\vdash\{Normal\ \doteq\}\ t\succ\ \{G\rightarrow\} \Longrightarrow$
$G,(A::state\ triple\ set)\vdash\{\doteq::state\ assn\}\ t\succ\ \{G\rightarrow\}$
$\langle proof \rangle$

lemma *MGFNormalD*:
$G,(A::state\ triple\ set)\vdash\{\doteq\}\ t\succ\ \{G\rightarrow\} \Longrightarrow G,A\vdash\{Normal\ \doteq\}\ t\succ\ \{G\rightarrow\}$
$\langle proof \rangle$

Additionally to *MGFNormalI*, we also expand the definition of the most general formula here

lemma *MGFn-NormalI*:
$G,(A::state\ triple\ set)\vdash\{Normal((\lambda Y'\ s'\ s.\ s'{=}s \wedge normal\ s) \wedge.\ G\vdash init\leq n)\}t\succ$
$\{\lambda Y\ s'\ s.\ G\vdash s\ -t\succ\rightarrow\ (Y,s')\} \Longrightarrow G,A\vdash\{=:n\}t\succ\{G\rightarrow\}$
$\langle proof \rangle$

To derive the most general formula, we can restrict ourselves to welltyped terms, since all others can be uniformally handled by the hazard rule.

lemma *MGFn-free-wt*:
$(\exists T\ L\ C.\ (\!|prg{=}G,cls{=}C,lcl{=}L|\!)\vdash t::T)$
$\longrightarrow G,(A::state\ triple\ set)\vdash\{=:n\}\ t\succ\ \{G\rightarrow\}$
$\Longrightarrow G,A\vdash\{=:n\}\ t\succ\ \{G\rightarrow\}$
$\langle proof \rangle$

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 uniformally handled by the hazard rule.

lemma *MGFn-free-wt-NormalConformI*:
$(\forall\ T\ L\ C\ .\ (\!|prg{=}G,cls{=}C,lcl{=}L|\!)\vdash t::T$
$\longrightarrow G,(A::state\ triple\ set)$
$\vdash\{Normal((\lambda Y'\ s'\ s.\ s'{=}s \wedge normal\ s) \wedge.\ G\vdash init\leq n) \wedge.\ (\lambda\ s.\ s::\preceq(G,\ L))\}$
$t\succ$
$\{\lambda Y\ s'\ s.\ G\vdash s\ -t\succ\rightarrow\ (Y,s')\})$
$\Longrightarrow G,A\vdash\{=:n\}t\succ\{G\rightarrow\}$
$\langle proof \rangle$

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 definetly assigned with respect to this state. All type violations can be uniformally handled by the hazard rule.

lemma *MGFn-free-wt-da-NormalConformI*:
$(\forall\ T\ L\ C\ B.\ (\!|prg{=}G,cls{=}C,lcl{=}L|\!)\vdash t{::}\ T$
$\longrightarrow G,(A{::}state\ triple\ set)$
$\vdash\{Normal((\lambda Y'\ s'\ s.\ s'{=}s \wedge normal\ s) \wedge.\ G\vdash init{\le}n) \wedge.\ (\lambda\ s.\ s{::}{\preceq}(G,\ L))$
$\wedge.\ (\lambda\ s.\ (\!|prg{=}G,cls{=}C,lcl{=}L|\!)\vdash dom\ (locals\ (store\ s))\gg t\gg B)\}$
$t\succ$
$\{\lambda Y\ s'\ s.\ G\vdash s\ {-}t{\succ}{\rightarrow}\ (Y,s')\})$
$\Longrightarrow G,A\vdash\{{=}{:}n\}t\succ\{G{\rightarrow}\}$
⟨*proof*⟩

main lemmas

lemma *MGFn-Init*:
assumes *mgf-hyp*: $\forall m.\ Suc\ m{\le}n \longrightarrow (\forall t.\ G,A\vdash\{{=}{:}m\}\ t\succ\ \{G{\rightarrow}\})$
shows $G,(A{::}state\ triple\ set)\vdash\{{=}{:}n\}\ \langle Init\ C\rangle_s\succ\ \{G{\rightarrow}\}$
⟨*proof*⟩
lemmas *MGFn-InitD* = *MGFn-Init* [*THEN MGFnD*, *THEN ax-NormalD*]

lemma *MGFn-Call*:
assumes *mgf-methds*:
$\forall C\ sig.\ G,(A{::}state\ triple\ set)\vdash\{{=}{:}n\}\ \langle(Methd\ C\ sig)\rangle_e\succ\ \{G{\rightarrow}\}$
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 $G,A\vdash\{{=}{:}n\}\ \langle\{accC,statT,mode\}e{\cdot}mn(\{pTs'\}ps)\rangle_e\succ\ \{G{\rightarrow}\}$
⟨*proof*⟩

lemma *eval-expression-no-jump'*:
assumes *eval*: $G\vdash s0\ {-}e{-}{\succ}v{\rightarrow}\ s1$
and *no-jmp*: *abrupt s0* $\neq$ *Some* (*Jump j*)
and *wt*: $(\!|prg{=}G,\ cls{=}C,lcl{=}L|\!)\vdash e{::}{-}T$
and *wf*: *wf-prog G*
shows *abrupt s1* $\neq$ *Some* (*Jump j*)
⟨*proof*⟩

To derive the most general formula for the loop statement, we need to come up with a proper loop invariant, which intuitively states that we are currently inside the evaluation of the loop. To define such an invariant, we unroll the loop in iterated evaluations of the expression and evaluations of the loop body.

definition
unroll :: *prog* ⇒ *label* ⇒ *expr* ⇒ *stmt* ⇒ (*state* × *state*) *set* **where**
$unroll\ G\ l\ e\ c = \{(s,t).\ \exists\ v\ s1\ s2.$
$G\vdash s\ {-}e{-}{\succ}v{\rightarrow}\ s1 \wedge the\text{-}Bool\ v \wedge normal\ s1\ \wedge$
$G\vdash s1\ {-}c{\rightarrow}\ s2 \wedge t{=}(abupd\ (absorb\ (Cont\ l))\ s2)\}$

lemma *unroll-while*:
assumes *unroll*: $(s,\ t) \in (unroll\ G\ l\ e\ c)^*$
and *eval-e*: $G\vdash t\ {-}e{-}{\succ}v{\rightarrow}\ s'$
and *normal-termination*: $normal\ s' \longrightarrow \neg\ the\text{-}Bool\ v$
and *wt*: $(\!|prg{=}G,cls{=}C,lcl{=}L|\!)\vdash e{::}{-}T$
and *wf*: *wf-prog G*
shows $G\vdash s\ {-}l{\cdot}\ While(e)\ c{\rightarrow}\ s'$
⟨*proof*⟩

lemma *MGFn-Loop*:
assumes *mfg-e*: *G*,(*A*::*state triple set*)⊢{=:*n*} ⟨*e*⟩_e≻ {*G*→}
and *mfg-c*: *G*,*A*⊢{=:*n*} ⟨*c*⟩_s≻ {*G*→}
and *wf*: *wf-prog G*
shows *G*,*A*⊢{=:*n*} ⟨*l*· *While*(*e*) *c*⟩_s≻ {*G*→}
⟨*proof*⟩

lemma *MGFn-FVar*:
fixes *A* :: *state triple set*
assumes *mgf-init*: *G*,*A*⊢{=:*n*} ⟨*Init statDeclC*⟩_s≻ {*G*→}
and *mgf-e*: *G*,*A*⊢{=:*n*} ⟨*e*⟩_e≻ {*G*→}
and *wf*: *wf-prog G*
shows *G*,*A*⊢{=:*n*} ⟨{*accC*,*statDeclC*,*stat*}*e*..*fn*⟩_v≻ {*G*→}
⟨*proof*⟩

lemma *MGFn-Fin*:
assumes *wf*: *wf-prog G*
and *mgf-c1*: *G*,*A*⊢{=:*n*} ⟨*c1*⟩_s≻ {*G*→}
and *mgf-c2*: *G*,*A*⊢{=:*n*} ⟨*c2*⟩_s≻ {*G*→}
shows *G*,(*A*::*state triple set*)⊢{=:*n*} ⟨*c1 Finally c2*⟩_s≻ {*G*→}
⟨*proof*⟩

lemma *Body-no-break*:
assumes *eval-init*: *G*⊢*Norm s0* −*Init D*→ *s1*
and *eval-c*: *G*⊢*s1* −*c*→ *s2*
and *jmpOk*: *jumpNestingOkS* {*Ret*} *c*
and *wt-c*: ⦇*prg*=*G*, *cls*=*C*, *lcl*=*L*⦈⊢*c*::√
and *clsD*: *class G D*=*Some d*
and *wf*: *wf-prog G*
shows ∀ *l*. *abrupt s2* ≠ *Some* (*Jump* (*Break l*)) ∧
abrupt s2 ≠ *Some* (*Jump* (*Cont l*))
⟨*proof*⟩

lemma *MGFn-Body*:
assumes *wf*: *wf-prog G*
and *mgf-init*: *G*,*A*⊢{=:*n*} ⟨*Init D*⟩_s≻ {*G*→}
and *mgf-c*: *G*,*A*⊢{=:*n*} ⟨*c*⟩_s≻ {*G*→}
shows *G*,(*A*::*state triple set*)⊢{=:*n*} ⟨*Body D c*⟩_e≻ {*G*→}
⟨*proof*⟩

lemma *MGFn-lemma*:
assumes *mgf-methds*:
⋀ *n*. ∀ *C sig*. *G*,(*A*::*state triple set*)⊢{=:*n*} ⟨*Methd C sig*⟩_e≻ {*G*→}
and *wf*: *wf-prog G*
shows ⋀ *t*. *G*,*A*⊢{=:*n*} *t*≻ {*G*→}
⟨*proof*⟩

lemma *MGF-asm*:
⟦∀ *C sig*. *is-methd G C sig* ⟶ *G*,*A*⊢{≐} *In1l* (*Methd C sig*)≻ {*G*→}; *wf-prog G*⟧
⟹ *G*,(*A*::*state triple set*)⊢{≐} *t*≻ {*G*→}
⟨*proof*⟩

nested version

lemma *nesting-lemma′* [*rule-format* (*no-asm*)]:
assumes *ax-derivs-asm*: ⋀*A ts*. *ts* ⊆ *A* ⟹ *P A ts*
and *MGF-nested-Methd*: ⋀*A pn*. ∀ *b*∈*bdy pn*. *P* (*insert* (*mgf-call pn*) *A*) {*mgf b*}
⟹ *P A* {*mgf-call pn*}

and *MGF-asm*: $\bigwedge A\ t.\ \forall\, pn{\in}U.\ P\ A\ \{mgf\text{-}call\ pn\} \Longrightarrow P\ A\ \{mgf\ t\}$
and *finU*: *finite U*
and *uA*: $uA = mgf\text{-}call\text{`}U$
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-derivs-asm*: $\bigwedge A\ ts.\ ts \subseteq A \Longrightarrow P\ A\ ts$
and *MGF-nested-Methd*: $\bigwedge A\ pn.\ \forall\, b{\in}bdy\ pn.\ P\ (insert\ (mgf\ (f\ pn))\ A)\ \{mgf\ b\}$
$\Longrightarrow P\ A\ \{mgf\ (f\ pn)\}$
and *MGF-asm*: $\bigwedge A\ t.\ \forall\, pn{\in}U.\ P\ A\ \{mgf\ (f\ pn)\} \Longrightarrow P\ A\ \{mgf\ t\}$
and *finU*: *finite U*
shows $P\ \{\}\ \{mgf\ t\}$
⟨*proof*⟩

lemma *MGF-nested-Methd*: $[\![$
$G, insert\ (\{Normal\ \doteq\}\ \langle Methd\ \ C\ sig\rangle_e \succ\{G\rightarrow\})\ A$
$\vdash\{Normal\ \doteq\}\ \langle body\ G\ C\ sig\rangle_e \succ\{G\rightarrow\}$
$]\!] \Longrightarrow\ G,A\vdash\{Normal\ \doteq\}\ \ \langle Methd\ \ C\ sig\rangle_e \succ\{G\rightarrow\}$
⟨*proof*⟩

lemma *MGF-deriv*: $wf\text{-}prog\ G \Longrightarrow G,(\{\}::state\ triple\ set)\vdash\{\doteq\}\ t\succ\ \{G\rightarrow\}$
⟨*proof*⟩

simultaneous version

lemma *MGF-simult-Methd-lemma*: $finite\ ms \Longrightarrow$
$G,A \cup (\lambda(C,sig).\ \{Normal\ \doteq\}\ \langle Methd\ \ C\ sig\rangle_e\succ\ \{G\rightarrow\})\ \text{`}\ ms$
$|\vdash(\lambda(C,sig).\ \{Normal\ \doteq\}\ \langle body\ G\ C\ sig\rangle_e\succ\ \{G\rightarrow\})\ \text{`}\ ms \Longrightarrow$
$G,A|\vdash(\lambda(C,sig).\ \{Normal\ \doteq\}\ \langle Methd\ \ C\ sig\rangle_e\succ\ \{G\rightarrow\})\ \text{`}\ ms$
⟨*proof*⟩

lemma *MGF-simult-Methd*: $wf\text{-}prog\ G \Longrightarrow$
$G,(\{\}::state\ triple\ set)|\vdash(\lambda(C,sig).\ \{Normal\ \doteq\}\ \langle Methd\ C\ sig\rangle_e\succ\ \{G\rightarrow\})$
$\text{`}\ Collect\ (case\text{-}prod\ (is\text{-}methd\ G))$
⟨*proof*⟩

corollaries

lemma *eval-to-evaln*: $[\![G\vdash s\ -t\succ\rightarrow\ (Y',\ s');type\text{-}ok\ G\ t\ s;\ wf\text{-}prog\ G]\!]$
$\Longrightarrow\ \exists\, n.\ G\vdash s\ -t\succ-n\rightarrow\ (Y',\ s')$
⟨*proof*⟩

lemma *MGF-complete*:
assumes *valid*: $G,\{\}\models\{P\}\ t\succ\ \{Q\}$
and *mgf*: $G,(\{\}::state\ triple\ set)\vdash\{\doteq\}\ t\succ\ \{G\rightarrow\}$
and *wf*: *wf-prog G*
shows $G,(\{\}::state\ triple\ set)\vdash\{P::state\ assn\}\ t\succ\ \{Q\}$
⟨*proof*⟩

theorem *ax-complete*:
assumes *wf*: *wf-prog G*
and *valid*: $G,\{\}\models\{P::state\ assn\}\ t\succ\ \{Q\}$
shows $G,(\{\}::state\ triple\ set)\vdash\{P\}\ t\succ\ \{Q\}$
⟨*proof*⟩

end

Chapter 25

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 = (λ*s*. ∃ *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* (λ*Y s Z*::′*a*. *heap-free four s* ∧ ¬*initd Base s* ∧ ¬ *initd Ext s*)}
.*test* [*Class Base*].
{λ*Y s Z*. *abrupt s* = *Some* (*Xcpt* (*Std IndOutBound*))}
⟨*proof*⟩

lemma *Loop-Xcpt-benchmark*:
Q = (λ*Y* (*x*,*s*) *Z*. *x* ≠ *None* ⟶ *the-Bool* (*the* (*locals s i*))) ⟹
G,({}::′*a triple set*)⊢{*Normal* (λ*Y s Z*::′*a*. *True*)}
.*lab1*• *While*(*Lit* (*Bool True*)) (*If*(*Acc* (*LVar i*)) (*Throw* (*Acc* (*LVar xcpt*))) *Else*

    $(Expr\ (Ass\ (LVar\ i)\ (Acc\ (LVar\ j))))).\ \{Q\}$
$\langle proof \rangle$

end