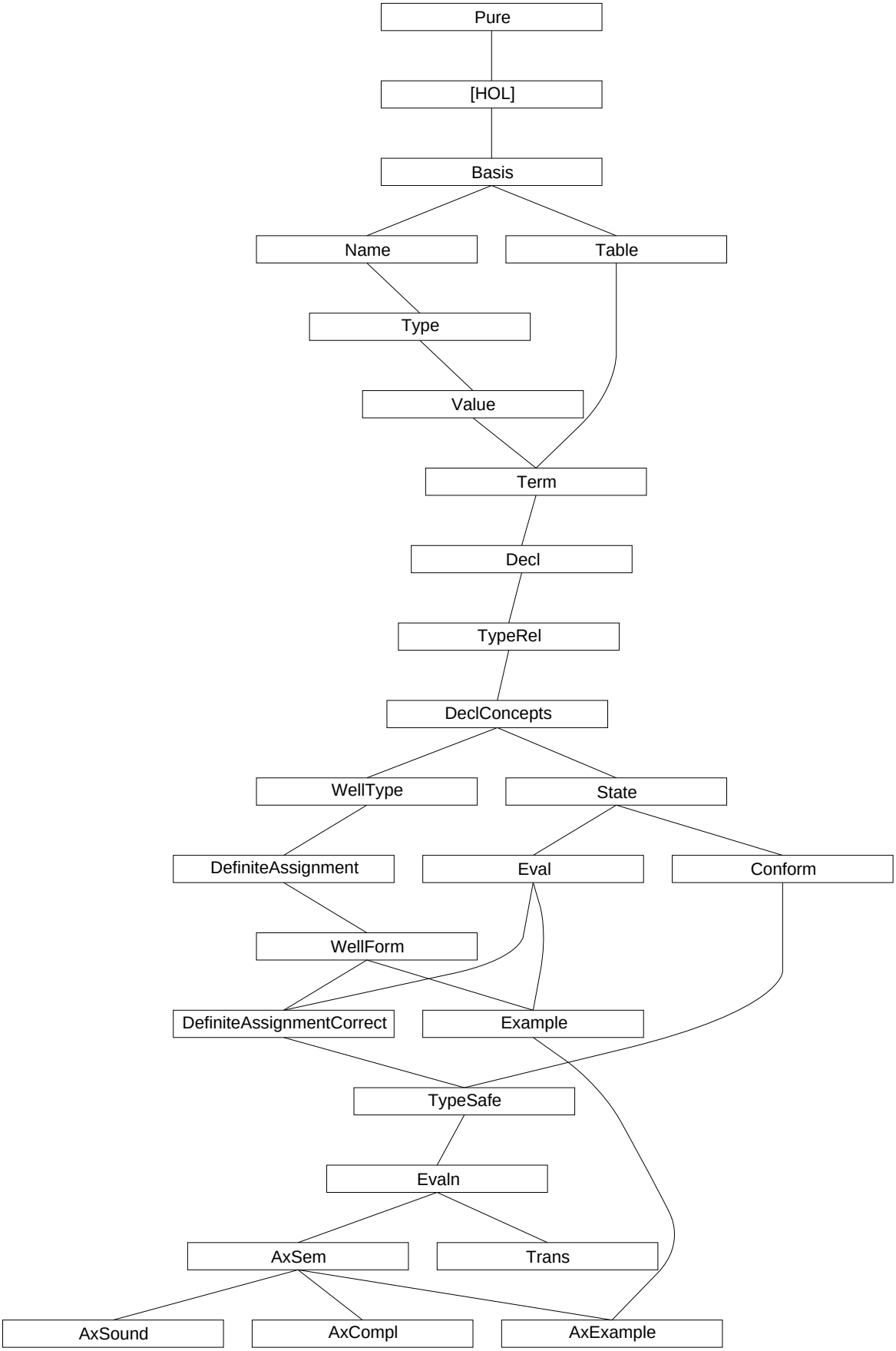
Java Source and Bytecode Formalizations in Isabelle: Bali

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

April 19, 2009

Contents


Pure
[HOL]
Basis
Name
Table
Type
Value
Term
Decl
TypeRel
DeclConcepts
WellType
State
DefiniteAssignment
Eval
Conform
WellForm
DefiniteAssignmentCorrect
Example
TypeSafe
Evaln
AxSem
Trans
AxSound
AxCompl
AxExample

Chapter 1

Overview

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

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

The following features are missing in Bali wrt. JavaCard:

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

- Inner Classes

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

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

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

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

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

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

Term JavaCard terms. Variables, expressions and statements.

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

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

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

WellType Typesystem on the JavaCard term level.

DefiniteAssignment The definite assignment analysis on the JavaCard term level.

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

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

Eval Operational (big step) semantics for JavaCard.

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

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

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

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

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

Trans A smallstep operational semantics for JavaCard.

AxSem An axiomatic semantics (Hoare logic) for JavaCard.

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

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

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

Chapter 2

Basis

1 Definitions extending HOL as logical basis of Bali

theory *Basis* **imports** *Main* **begin**

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

misc

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

declare *split-if-asm* [*split*] *option.split* [*split*] *option.split-asm* [*split*]
⟨*ML*⟩
declare *if-weak-cong* [*cong del*] *option.weak-case-cong* [*cong del*]
declare *length-Suc-conv* [*iff*]

lemma *Collect-split-eq*: {*p*. *P* (*split f p*)} = {(*a*,*b*). *P* (*f a b*)}
⟨*proof*⟩

lemma *subset-insertD*:
A <= *insert x B* ==> *A* <= *B* & *x* ~: *A* | (*EX B'*. *A* = *insert x B'* & *B'* <= *B*)
⟨*proof*⟩

syntax
3 :: *nat* (*3*)
4 :: *nat* (*4*)
translations
3 == *Suc 2*
4 == *Suc 3*

lemma *range-bool-domain*: *range f* = {*f True*, *f False*}
⟨*proof*⟩

lemma *irrefl-trancIl'*: *r*^−*1 Int r*^+ = {} ==> !*x*. (*x*, *x*) ~: *r*^+
⟨*proof*⟩

lemma *trancl-rtrancl-trancl*:
⟦(*x*,*y*)∈*r*^+; (*y*,*z*)∈*r*^*⟧ ⟹ (*x*,*z*)∈*r*^+
⟨*proof*⟩

lemma *rtrancl-into-trancl3*:
⟦(*a*,*b*)∈*r*^*; *a*≠*b*⟧ ⟹ (*a*,*b*)∈*r*^+
⟨*proof*⟩

lemma *rtrancl-into-rtrancl2*:
⟦ (*a*, *b*) ∈ *r*; (*b*, *c*) ∈ *r*^* ⟧ ⟹ (*a*, *c*) ∈ *r*^*
⟨*proof*⟩

lemma *triangle-lemma*:

$[\![\bigwedge a\ b\ c.\ [\![(a,b)\in r;\ (a,c)\in r]\!] \Longrightarrow b=c;\ (a,x)\in r^*;\ (a,y)\in r^*]\!]$
$\Longrightarrow (x,y)\in r^* \vee (y,x)\in r^*$
⟨*proof*⟩

lemma *rtrancl-cases* [*consumes 1*, *case-names Refl Trancl*]:
$[\![(a,b)\in r^*;\ \ a = b \Longrightarrow P;\ (a,b)\in r^+ \Longrightarrow P]\!] \Longrightarrow P$
⟨*proof*⟩

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

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

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

lemma *finite-SetCompr2*: [| *finite* (*Collect P*); !*y*. *P y* −−> *finite* (*range* (*f y*)) |] ==>
finite {*f y x* |*x y*. *P y*}
⟨*proof*⟩

lemma *list-all2-trans*: ∀ *a b c*. *P1 a b* ⟶ *P2 b c* ⟶ *P3 a c* ⟹
∀*xs2 xs3*. *list-all2 P1 xs1 xs2* ⟶ *list-all2 P2 xs2 xs3* ⟶ *list-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*!]:
[| *fst s′* = *x′*; !!*x s*. [| *s′* = (*x*,*s*); *x* = *x′* |] ==> *Q* |] ==> *Q*
⟨*proof*⟩

lemma *fst-in-set-lemma* [*rule-format* (*no-asm*)]: (*x*, *y*) : *set l* −−> *x* : *fst* ‘ *set l*
⟨*proof*⟩

quantifiers

lemma *All-Ex-refl-eq2* [*simp*]:
(!*x*. (*?* *b*. *x* = *f b* & *Q b*) ⟶ *P x*) = (!*b*. *Q b* −−> *P* (*f b*))
⟨*proof*⟩

lemma *ex-ex-miniscope1* [*simp*]:
(*EX w v*. *P w v* & *Q v*) = (*EX v*. (*EX w*. *P w v*) & *Q v*)

⟨*proof*⟩

lemma *ex-miniscope2* [*simp*]:
(*EX v. P v & Q & R v*) = (*Q & (EX v. P v & R v*))
⟨*proof*⟩

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

lemma *All-Ex-refl-eq1* [*simp*]: (!*x.* (*? b. x = f b*) *–––>* *P x*) = (!*b. P* (*f b*))
⟨*proof*⟩

sums

hide *const In0 In1*

syntax
fun-sum :: (*'a => 'c*) => (*'b => 'c*) => ((*'a+'b*) => *'c*) (**infixr** *'(+')80*)
translations
fun-sum == CONST sum-case

consts *the-Inl* :: *'a + 'b ⇒ 'a*
the-Inr :: *'a + 'b ⇒ 'b*
primrec *the-Inl* (*Inl a*) = *a*
primrec *the-Inr* (*Inr b*) = *b*

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

consts *the-In1* :: (*'a, 'b, 'c*) *sum3* ⇒ *'a*
the-In2 :: (*'a, 'b, 'c*) *sum3* ⇒ *'b*
the-In3 :: (*'a, 'b, 'c*) *sum3* ⇒ *'c*
primrec *the-In1* (*In1 a*) = *a*
primrec *the-In2* (*In2 b*) = *b*
primrec *the-In3* (*In3 c*) = *c*

syntax
In1l :: *'al* ⇒ (*'al + 'ar, 'b, 'c*) *sum3*
In1r :: *'ar* ⇒ (*'al + 'ar, 'b, 'c*) *sum3*
translations
In1l e == In1 (*Inl e*)
In1r c == In1 (*Inr c*)

syntax *the-In1l* :: (*'al + 'ar, 'b, 'c*) *sum3* ⇒ *'al*
the-In1r :: (*'al + 'ar, 'b, 'c*) *sum3* ⇒ *'ar*
translations
the-In1l == the-Inl ∘ *the-In1*
the-In1r == the-Inr ∘ *the-In1*

⟨*ML*⟩

translations
option<= (*type*) *Datatype.option*
list <= (*type*) *List.list*
sum3 <= (*type*) *Basis.sum3*

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*)

translations
! *x*:*A*: *P* == ! *x*:*CONST Option.set A*. *P*
? *x*:*A*: *P* == *?* *x*:*CONST Option.set A*. *P*

Special map update

Deemed too special for theory Map.

constdefs
chg-map :: (*'b* => *'b*) => *'a* => (*'a* ~=> *'b*) => (*'a* ~=> *'b*)
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

constdefs
unique :: (*'a* × *'b*) *list* ⇒ *bool*
unique ≡ *distinct* ∘ *map fst*

lemma *uniqueD* [*rule-format* (*no-asm*)]:
unique l--> (!*x y*. (*x*,*y*):*set l* --> (!*x' y'*. (*x'*,*y'*):*set l* --> *x*=*x'*--> *y*=*y'*))
⟨*proof*⟩

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

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

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

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

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

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

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

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

list patterns

consts
lsplit :: [['*a*, '*a list*] => '*b*, '*a list*] => '*b*
defs
lsplit-def: *lsplit* == %*f l*. *f* (*hd l*) (*tl l*)

syntax
-lpttrn :: [*pttrn*,*pttrn*] => *pttrn* (-#/- [*901*,*900*] *900*)
translations
%*y*#*x*#*xs*. *b* == *lsplit* (%*y x*#*xs*. *b*)
%*x*#*xs* . *b* == *lsplit* (%*x xs* . *b*)

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

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

end

Chapter 3

Table

2 Abstract tables and their implementation as lists

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

design issues:

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

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

map of / table of

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

translations
table-of == *map-of*

$(type)'a \rightharpoonup 'b$ <= $(type)'a \Rightarrow 'b$ *Datatype.option*
$(type)('a, 'b)$ *table* <= $(type)'a \rightharpoonup 'b$

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

Conditional Override

constdefs
cond-override::
$('b \Rightarrow 'b \Rightarrow bool) \Rightarrow ('a, 'b)table \Rightarrow ('a, 'b)table \Rightarrow ('a, 'b)\ table$

— when merging tables old and new, only override an entry of table old when the condition cond holds
cond-override cond old new ≡
$\lambda\ k.$
(*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 empty t* = *t*
⟨*proof*⟩

lemma *cond-override-empty2*[*simp*]: *cond-override c t 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

constdefs
filter-tab:: ('*a* ⇒ '*b* ⇒ *bool*) ⇒ ('*a*, '*b*) *table* ⇒ ('*a*, '*b*) *table*
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 empty* = *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* = *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*) ⟹ *finite* (*range* (*filter-tab c t*))
⟨*proof*⟩

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

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

lemma *filter-tab-all-True*:
∀ *k y*. *t k* = *Some y* ⟶ *p k y* ⟹*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*⟧ ⟹ *filter-tab p t k* = *Some v*
⟨*proof*⟩

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

lemma *filter-tab-None*: *t k* = *None* ⟹ *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*⟧ ⟹ *filter-tab C a* = *filter-tab C b*
⟨*proof*⟩

lemma *finite-dom-filter-tab*:
finite (*dom t*) ⟹ *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*⟧ ⟹ *cond k x* ⟶ *cond k y*
⟧ ⟹ ∀ *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*⟧
⟹ (¬ *overC new old* ⟶ ¬ *filterC k new*) ∧
(*overC new old* ⟶ *filterC k old* ⟶ *filterC k new*)
⟧ ⟹
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*) ⟹ ∀ *x*. ∀ *y*∈ *map-of l x*: *P x y*
⟨*proof*⟩

lemma *Ball-set-tableD*:
⟦(∀ (*x*,*y*)∈ *set l*. *P x y*); *x* ∈ *Option.set* (*table-of l xa*)⟧ ⟹ *P xa x*
⟨*proof*⟩

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

lemma *table-of-Some-in-set*:
table-of l k = *Some x* ⟹ (*k*,*x*) ∈ *set l*
⟨*proof*⟩

lemma *set-get-eq*:
unique l ⟹ (*k*, *the* (*table-of l k*)) ∈ *set l* = (*table-of l k* ≠ *None*)
⟨*proof*⟩

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

lemma *table-of-mapconst-SomeI*:
⟦*table-of t k* = *Some y*′; *snd y*=*y*′; *fst y*=*c*⟧ ⟹
table-of (*map* (λ(*k*,*x*). (*k*,*c*,*x*)) *t*) *k* = *Some y*
⟨*proof*⟩

lemma *table-of-mapconst-NoneI*:
⟦*table-of t k* = *None*⟧ ⟹
table-of (*map* (λ(*k*,*x*). (*k*,*c*,*x*)) *t*) *k* = *None*
⟨*proof*⟩

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

lemma *table-of-map-SomeI* [*rule-format* (*no-asm*)]: *table-of t k* = *Some x* ⟶
table-of (*map* (λ(*k*,*x*). (*k*, *f x*)) *t*) *k* = *Some* (*f x*)
⟨*proof*⟩

lemma *table-of-remap-SomeD* [*rule-format* (*no-asm*)]:
table-of (*map* (λ((*k*,*k*′),*x*). (*k*,(*k*′,*x*))) *t*) *k* = *Some* (*k*′,*x*) ⟶
table-of t (*k*, *k*′) = *Some x*
⟨*proof*⟩

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

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

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

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

lemma *table-of-mapkey-SomeD2* [*rule-format* (*no-asm*), *dest*!]:
table-of (*map* ($\lambda(k,x).\ ((k,C),x)$) *t*) *ek* = *Some x*
$\longrightarrow$ *C* = *snd ek* $\wedge$ *table-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* $\vee$ (*table-of xs k* = *None* $\wedge$ *table-of ys k* = *Some z*))
⟨*proof*⟩

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

consts
Un-tables :: ('*a*, '*b*) *tables set* ⇒ ('*a*, '*b*) *tables*
overrides-t :: ('*a*, '*b*) *tables* ⇒ ('*a*, '*b*) *tables* ⇒
('*a*, '*b*) *tables* (**infixl** ⊕⊕ *100*)
hidings-entails:: ('*a*, '*b*) *tables* ⇒ ('*a*, '*c*) *tables* ⇒
('*b* ⇒ '*c* ⇒ *bool*) ⇒ *bool* (- *hidings - entails - 20*)
— variant for unique table:
hiding-entails :: ('*a*, '*b*) *table* ⇒ ('*a*, '*c*) *table* ⇒
('*b* ⇒ '*c* ⇒ *bool*) ⇒ *bool* (- *hiding - entails - 20*)
— 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*)

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

Untables

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

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

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

overrides

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

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

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

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

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

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

hiding entails

lemma *hiding-entailsD*:
$[\![t\ \mathit{hiding}\ s\ \mathit{entails}\ R;\ t\ k = \mathit{Some}\ x;\ s\ k = \mathit{Some}\ y]\!] \Longrightarrow R\ x\ y$
⟨*proof*⟩

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

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

cond hiding entails

lemma *cond-hiding-entailsD*:
$[\![t\ \mathit{hiding}\ s\ \mathit{under}\ C\ \mathit{entails}\ R;\ t\ k = \mathit{Some}\ x;\ s\ k = \mathit{Some}\ y;\ C\ x\ y]\!] \Longrightarrow R\ x\ y$
⟨*proof*⟩

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

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

lemma *hidings-entailsD*: ⟦*t hidings s entails R*; *x* ∈ *t k*; *y* ∈ *s k*⟧ ⟹ *R x y*
⟨*proof*⟩

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

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

consts
atleast-free :: ('*a* ~=> '*b*) => *nat* => *bool*
primrec
atleast-free m 0 = *True*
atleast-free-Suc:
atleast-free m (*Suc n*) = (*? a. m a = None* & (!*b. atleast-free* (*m*(*a*|−>*b*)) *n*))

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

lemma *atleast-free-SucI*:
[| *h a = None*; !*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*)]:
!*m a. m a = None* −−> (!*c. atleast-free* (*m*(*a*|−>*c*)) *n*) −−>
(!*b d. a* ~= *b* −−> *atleast-free* (*m*(*b*|−>*d*)) *n*)
⟨*proof*⟩
declare *fun-upd-apply* [*simp*]

lemma *atleast-free-SucD* [*rule-format* (*no-asm*)]: *atleast-free h* (*Suc n*) ==> *atleast-free* (*h*(*a*|−>*b*)) *n*
⟨*proof*⟩

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

Chapter 4

Name

3 Java names

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

typedecl *tnam* — ordinary type name, i.e. class or interface name
typedecl *pname* — package name
typedecl *mname* — method name
typedecl *vname* — variable or field name
typedecl *label* — label as destination of break or continue

datatype *ename* — expression name
= *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*
syntax
VName :: *vname* ⇒ *lname*
Result :: *lname*

translations
VName n == *EName* (*VNam n*)
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*

axclass *has-pname* < *type*
consts *pname*::′*a*::*has-pname* ⇒ *pname*

instance *pname*::*has-pname* ⟨*proof*⟩

defs (**overloaded**)
pname-pname-def: *pname* (*p*::*pname*) ≡ *p*

axclass *has-tname* < *type*
consts *tname*::′*a*::*has-tname* ⇒ *tname*

instance *tname*::*has-tname* ⟨*proof*⟩

defs (**overloaded**)
tname-tname-def: *tname* (*t*::*tname*) ≡ *t*

axclass *has-qtname* < *type*
consts *qtname*:: 'a::*has-qtname* ⇒ *qtname*

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

defs (**overloaded**)
qtname-qtname-def: *qtname* (*q*::*qtname*) ≡ *q*

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

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

consts
Object :: *qtname*
SXcpt :: *xname* ⇒ *qtname*
defs
Object-def: *Object* ≡ (|*pid* = *java-lang*, *tid* = *Object'*|)
SXcpt-def: *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

4 Java values

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

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

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

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

consts *the-Bool* :: *val* ⇒ *bool*
primrec *the-Bool* (*Bool b*) = *b*
consts *the-Intg* :: *val* ⇒ *int*
primrec *the-Intg* (*Intg i*) = *i*
consts *the-Addr* :: *val* ⇒ *loc*
primrec *the-Addr* (*Addr a*) = *a*

types *dyn-ty* = *loc* ⇒ *ty option*
consts
typeof :: *dyn-ty* ⇒ *val* ⇒ *ty option*
defpval :: *prim-ty* ⇒ *val* — default value for primitive types
default-val :: *ty* ⇒ *val* — default value for all types

primrec *typeof dt Unit* = *Some* (*PrimT Void*)
typeof dt (*Bool b*) = *Some* (*PrimT Boolean*)
typeof dt (*Intg i*) = *Some* (*PrimT Integer*)
typeof dt Null = *Some NT*
typeof dt (*Addr a*) = *dt a*

primrec *defpval Void* = *Unit*
defpval Boolean = *Bool False*
defpval Integer = *Intg 0*
primrec *default-val* (*PrimT pt*) = *defpval pt*
default-val (*RefT r*) = *Null*

end

Chapter 6

Type

5 Java types

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

simplifications:

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

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

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

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

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

syntax
NT :: *ty*
Iface :: *qtname* ⇒ *ty*
Class :: *qtname* ⇒ *ty*
Array :: *ty* ⇒ *ty* (-.[] [*90*] *90*)

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

constdefs
the-Class :: *ty* ⇒ *qtname*
the-Class T ≡ *SOME C*. *T* = *Class C*

lemma *the-Class-eq* [*simp*]: *the-Class* (*Class C*)= *C*
⟨*proof*⟩

end

Chapter 7

Term

6 Java expressions and statements

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

design issues:

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

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

datatype *jump*
= *Break label* — break

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

datatype *xcpt* — exception
= *Loc loc* — location of allocated 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
types
abopt = *abrupt option*

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

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

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

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

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

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

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

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

The boolean operators `&` and `|` strictly evaluate both of their arguments. The conditional operators `&&` and `||` only evaluate the second argument if the value of the whole expression isn't 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 cateched
| *Fin stmt stmt* (- *Finally* - [*79*,*79*]*70*)
| *FinA abopt stmt* — Save abruption of first statement
— technical term for smallstep sem.)
| *Init qtname* — class initialization

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

types *term* = (*expr*+*stmt*,*var*,*expr list*) *sum3*
translations
sig <= (*type*) *mname* × *ty list*
var <= (*type*) *Term.var*
expr <= (*type*) *Term.expr*
stmt <= (*type*) *Term.stmt*
term <= (*type*) (*expr*+*stmt*,*var*,*expr list*) *sum3*

syntax

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

translations

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

constdefs

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

⟨*ML*⟩

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

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

syntax
expr-inj-term:: *expr* ⇒ *term* (⟨-⟩$_e$ *1000*)
stmt-inj-term:: *stmt* ⇒ *term* (⟨-⟩$_s$ *1000*)
var-inj-term:: *var* ⇒ *term* (⟨-⟩$_v$ *1000*)
lst-inj-term:: *expr list* ⇒ *term* (⟨-⟩$_l$ *1000*)

translations
⟨*e*⟩$_e$ ⇀ *In1l e*
⟨*c*⟩$_s$ ⇀ *In1r c*
⟨*v*⟩$_v$ ⇀ *In2 v*
⟨*es*⟩$_l$ ⇀ *In3 es*

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

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

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

instance *stmt*::*inj-term* ⟨*proof*⟩

defs (**overloaded**)
stmt-inj-term-def: ⟨*c*::*stmt*⟩ ≡ *In1r c*

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

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

instance *expr*::*inj-term* ⟨*proof*⟩

defs (**overloaded**)
expr-inj-term-def: ⟨*e*::*expr*⟩ ≡ *In1l e*

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

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

instance *var*::*inj-term* ⟨*proof*⟩

defs (**overloaded**)
var-inj-term-def: ⟨*v*::*var*⟩ ≡ *In2 v*

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

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

instance *list*:: (*type*) *inj-term* ⟨*proof*⟩

defs (**overloaded**)
expr-list-inj-term-def: ⟨*es*::*expr list*⟩ ≡ *In3 es*

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*:
⟦⋀ *v*. *P* ⟨*v*⟩$_v$; ⋀ *e*. *P* ⟨*e*⟩$_e$;⋀ *c*. *P* ⟨*c*⟩$_s$;⋀ *l*. *P* ⟨*l*⟩$_l$⟧
⟹ *P t*
⟨*proof*⟩

Evaluation of unary operations

consts *eval-unop* :: *unop* ⇒ *val* ⇒ *val*
primrec
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

consts *eval-binop* :: *binop* ⇒ *val* ⇒ *val* ⇒ *val*
primrec
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*))

constdefs *need-second-arg* :: *binop* ⇒ *val* ⇒ *bool*
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

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

theory *Decl*
imports *Term Table*
begin

improvements:

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

simplifications:

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

8 Modifier

Access modifier

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

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

instantiation *acc-modi* :: *linorder*
begin

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

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

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*
[*consumes 1*,*case-names Package Protected Public*]:
$Package \leq m \Longrightarrow (\ m = Package \Longrightarrow P\ m) \Longrightarrow (m{=}Protected \Longrightarrow P\ m) \Longrightarrow$
$(m{=}Public \Longrightarrow P\ m) \Longrightarrow P\ m$
⟨*proof*⟩

Static Modifier

types *stat-modi* = *bool*

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

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

translations
decl <= (*type*) (|*access*::*acc-modi*|)
decl <= (*type*) (|*access*::*acc-modi*,. . .::′*a*|)

10 Member (field or method)

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

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

11 Field

record *field* = *member* +
type :: *ty*
translations
field <= (*type*) (|*access*::*acc-modi*, *static*::*bool*, *type*::*ty*|)
field <= (*type*) (|*access*::*acc-modi*, *static*::*bool*, *type*::*ty*,. . .::′*a*|)

types
fdecl
= *vname* × *field*

translations
fdecl <= (*type*) *vname* × *field*

12 Method

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

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

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

types *mdecl* = *sig* × *methd*

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

pars::*vname list*, *resT*::*ty*,*mbody*::*mbody*,...::*'a*⦈
mdecl <= (*type*) *sig* × *methd*

constdefs
mhead::*methd* ⇒ *mhead*
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*

axclass *has-memberid* < *type*
consts
memberid :: *'a*::*has-memberid* ⇒ *memberid*

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

13 Interface

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

record *iface* = *ibody* + — interface
isuperIfs:: *qtname list* — superinterface list
types
idecl — interface declaration, cf. 9.1
= *qtname* × *iface*

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

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

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

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

14 Class

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

record *class* = *cbody* + — class
super :: *qtname* — superclass
superIfs:: *qtname list* — implemented interfaces
types
cdecl — class declaration, cf. 8.1
= *qtname* × *class*

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

constdefs
cbody :: *class* ⇒ *cbody*
cbody c ≡ (|*access*=*access c*, *cfields*=*cfields c*,*methods*=*methods c*,*init*=*init c*|)

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

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

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

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

standard classes

consts

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

defs

ObjectC-def:*ObjectC* ≡ (*Object*,(|*access*=*Public*,*cfields*=[],*methods*=*Object-mdecls*,
init=*Skip*,*super*=*undefined*,*superIfs*=[]|))
SXcptC-def:*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*⟩

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

syntax
iface :: *prog* ⇒ (*qtname*, *iface*) *table*
class :: *prog* ⇒ (*qtname*, *class*) *table*
is-iface :: *prog* ⇒ *qtname* ⇒ *bool*
is-class :: *prog* ⇒ *qtname* ⇒ *bool*

translations
iface G I == *table-of* (*ifaces G*) *I*
class G C == *table-of* (*classes G*) *C*
is-iface G I == *iface G I* ≠ *None*
is-class G C == *class G C* ≠ *None*

is type

consts
is-type :: *prog* ⇒ *ty* ⇒ *bool*
isrtype :: *prog* ⇒ *ref-ty* ⇒ *bool*

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

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

lemma *type-is-iface*: *is-type G (Iface I) ⟹ 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

consts
subint1 :: *prog ⇒ (qtname × qtname) set* — direct subinterface
subcls1 :: *prog ⇒ (qtname × qtname) set* — direct subclass

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

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

syntax (*xsymbols*)
-subcls1 :: *prog ⇒ [qtname, qtname] ⇒ bool* (-⊢-≺$_{C1}$- [*71*,*71*,*71*] *70*)
-subclseq:: *prog ⇒ [qtname, qtname] ⇒ bool* (-⊢-⪯$_C$ - [*71*,*71*,*71*] *70*)
-subcls :: *prog ⇒ [qtname, qtname] ⇒ bool* (-⊢-≺$_C$ - [*71*,*71*,*71*] *70*)

translations
G⊢C ≺$_{C1}$ D == (C,D) ∈ subcls1 G
*G⊢C ⪯$_C$ D == (C,D) ∈(subcls1 G)^**
G⊢C ≺$_C$ D == (C,D) ∈(subcls1 G)^+

lemma *subint1I*: ⟦*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⊢C ≺C D*⟧ ⟹ ∃ *c. class G C = Some c*
⟨*proof*⟩

lemma *no-subcls1-Object*:*G⊢Object≺C1 D* ⟹ *P*
⟨*proof*⟩

lemma *no-subcls-Object*: *G⊢Object≺C D* ⟹ *P*
⟨*proof*⟩

well-structured programs

constdefs
ws-idecl :: *prog* ⇒ *qtname* ⇒ *qtname list* ⇒ *bool*
ws-idecl G I si ≡ ∀ *J*∈*set si. is-iface G J* ∧ (*J*,*I*)∉(*subint1 G*)^+

ws-cdecl :: *prog* ⇒ *qtname* ⇒ *qtname* ⇒ *bool*
ws-cdecl G C sc ≡ *C*≠*Object* ⟶ *is-class G sc* ∧ (*sc*,*C*)∉(*subcls1 G*)^+

ws-prog :: *prog* ⇒ *bool*
ws-prog G ≡ (∀ (*I*,*i*)∈*set* (*ifaces G*). *ws-idecl G I* (*isuperIfs i*)) ∧
(∀ (*C*,*c*)∈*set* (*classes G*). *ws-cdecl G C* (*super c*))

lemma *ws-progI*:
⟦∀ (*I*,*i*)∈*set* (*ifaces G*). ∀ *J*∈*set* (*isuperIfs i*). *is-iface G J* ∧
(*J*,*I*) ∉ (*subint1 G*)^+;
∀ (*C*,*c*)∈*set* (*classes G*). *C*≠*Object* ⟶ *is-class G* (*super c*) ∧
((*super c*),*C*) ∉ (*subcls1 G*)^+
⟧ ⟹ *ws-prog G*
⟨*proof*⟩

lemma *ws-prog-ideclD*:
⟦*iface G I = Some i*; *J*∈*set* (*isuperIfs i*); *ws-prog G*⟧ ⟹
is-iface G J ∧ (*J*,*I*)∉(*subint1 G*)^+
⟨*proof*⟩

lemma *ws-prog-cdeclD*:
⟦*class G C = Some c*; *C*≠*Object*; *ws-prog G*⟧ ⟹
is-class G (*super c*) ∧ (*super c*,*C*)∉(*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*:
$ws\text{-}prog\ G \Longrightarrow (subint1\ G)\hat{}{-1} \cap (subint1\ G)\hat{}{+} = \{\}$
⟨*proof*⟩

lemma *subcls1-irrefl-lemma1*:
$ws\text{-}prog\ G \Longrightarrow (subcls1\ G)\hat{}{-1} \cap (subcls1\ G)\hat{}{+} = \{\}$
⟨*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*, *standard*]
lemmas *subcls1-acyclic* = *subcls1-irrefl-lemma2* [*THEN acyclicI*, *standard*]

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-iface G I*; *ws-prog G*; ⋀*I i*. ⟦*iface G I* = *Some i* ∧
(∀ *J* ∈ *set* (*isuperIfs i*). (*I*,*J*)∈*subint1 G* ∧ *P J* ∧ *is-iface G J*)⟧ ⟹ *P I*
⟧ ⟹ *P I*
⟨*proof*⟩

lemma *ws-subcls1-induct*: ⟦*is-class G C*; *ws-prog G*;
⋀*C c*. ⟦*class G C* = *Some c*;
(*C* ≠ *Object* ⟶ (*C*,(*super c*))∈*subcls1 G* ∧
P (*super c*) ∧ *is-class G* (*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*;
⋀ *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*]:
⟦*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

consts
iface-rec :: *prog* × *qtname* ⇒ (*qtname* ⇒ *iface* ⇒ *′a set* ⇒ *′a*) ⇒ *′a*
class-rec :: *prog* × *qtname* ⇒ *′a* ⇒ (*qtname* ⇒ *class* ⇒ *′a* ⇒ *′a*) ⇒ *′a*

recdef *iface-rec same-fst ws-prog* (λ*G*. (*subint1 G*)ˆ−*1*)
iface-rec (*G*,*I*) =
  (λ*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)
(**hints** *recdef-wf*: *wf-subint1 intro*: *subint1I*)
declare *iface-rec.simps* [*simp del*]

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*⟩

recdef *class-rec same-fst ws-prog* (λ*G*. (*subcls1 G*)ˆ−*1*)
class-rec(*G*,*C*) =
  (λ*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)
(**hints** *recdef-wf*: *wf-subcls1 intro*: *subcls1I*)
declare *class-rec.simps* [*simp del*]

lemma *class-rec*: ⟦*class G C* = *Some c*; *ws-prog G*⟧ ⟹
 class-rec (*G*,*C*) *t f* =
   f C c (*if C* = *Object then t else class-rec* (*G*,*super c*) *t f*)
⟨*proof*⟩

constdefs
imethds:: *prog* ⇒ *qtname* ⇒ (*sig*,*qtname* × *mhead*) *tables*
  — 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*) ⊕⊕
                  (*Option.set* ∘ *table-of* (*map* (λ(*s*,*m*). (*s*,*I*,*m*)) (*imethods i*))))

end

Chapter 9

TypeRel

15 The relations between Java types

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

simplifications:

- subinterface, subclass and widening relation includes identity

improvements over Java Specification 1.0:

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

design issues:

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

consts

implmt1 :: *prog* ⇒ (*qtname* × *qtname*) *set* — direct implementation

syntax

-subint1 :: *prog* => [*qtname*, *qtname*] => *bool* (-|--<:*I1*- [*71*,*71*,*71*] *70*)
-subint :: *prog* => [*qtname*, *qtname*] => *bool* (-|--<=:*I* -[*71*,*71*,*71*] *70*)

@*implmt1* :: *prog* => [*qtname*, *qtname*] => *bool* (-|--~>*1*- [*71*,*71*,*71*] *70*)

syntax (*xsymbols*)

-subint1 :: *prog* ⇒ [*qtname*, *qtname*] ⇒ *bool* (-⊢-≺*I1*- [*71*,*71*,*71*] *70*)
-subint :: *prog* ⇒ [*qtname*, *qtname*] ⇒ *bool* (-⊢-≼*I* - [*71*,*71*,*71*] *70*)

-implmt1 :: *prog* ⇒ [*qtname*, *qtname*] ⇒ *bool* (-⊢-⇝*1*- [*71*,*71*,*71*] *70*)

translations

G⊢*I* ≺*I1* *J* == (*I*,*J*) ∈ *subint1* *G*
G⊢*I* ≼*I* *J* == (*I*,*J*) ∈(*subint1* *G*)^* — cf. 9.1.3

G⊢*C* ⇝*1* *I* == (*C*,*I*) ∈ *implmt1* *G*

subclass and subinterface relations

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

lemma *subcls-direct1*:
⟦*class* *G* *C* = *Some* *c*; *C* ≠ *Object*;*D*=*super* *c*⟧ ⟹ *G*⊢*C*≼$_C$ *D*

⟨*proof*⟩

lemma *subcls1I1*:
⟦*class G C = Some c*; *C ≠ Object*;*D=super c*⟧ ⟹ *G⊢C≺$_{C1}$ D*
⟨*proof*⟩

lemma *subcls-direct2*:
⟦*class G C = Some c*; *C ≠ Object*;*D=super c*⟧ ⟹ *G⊢C≺$_C$ D*
⟨*proof*⟩

lemma *subclseq-trans*: ⟦*G⊢A ⪯$_C$ B*; *G⊢B ⪯$_C$ C*⟧ ⟹ *G⊢A ⪯$_C$ C*
⟨*proof*⟩

lemma *subcls-trans*: ⟦*G⊢A ≺$_C$ B*; *G⊢B ≺$_C$ C*⟧ ⟹ *G⊢A ≺$_C$ C*
⟨*proof*⟩

lemma *SXcpt-subcls-Throwable-lemma*:
⟦*class G* (*SXcpt xn*) = *Some xc*;
super xc = (*if xn* = *Throwable then Object else SXcpt Throwable*)⟧
⟹ *G⊢SXcpt xn⪯$_C$ SXcpt Throwable*
⟨*proof*⟩

lemma *subcls-ObjectI*: ⟦*is-class G C*; *ws-prog G*⟧ ⟹ *G⊢C⪯$_C$ Object*
⟨*proof*⟩

lemma *subclseq-ObjectD* [*dest*!]: *G⊢Object⪯$_C$ C* ⟹ *C* = *Object*
⟨*proof*⟩

lemma *subcls-ObjectD* [*dest*!]: *G⊢Object≺$_C$ C* ⟹ *False*
⟨*proof*⟩

lemma *subcls-ObjectI1* [*intro*!]:
⟦*C ≠ Object*;*is-class G C*;*ws-prog G*⟧ ⟹ *G⊢C ≺$_C$ Object*
⟨*proof*⟩

lemma *subcls-is-class*: (*C*,*D*) ∈ (*subcls1 G*)^+ ⟹ *is-class G C*
⟨*proof*⟩

lemma *subcls-is-class2* [*rule-format* (*no-asm*)]:
G⊢C⪯$_C$ D ⟹ *is-class G D* ⟶ *is-class G C*
⟨*proof*⟩

lemma *single-inheritance*:
⟦*G⊢A ≺$_{C1}$ B*; *G⊢A ≺$_{C1}$ C*⟧ ⟹ *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*: $[\![G\vdash C \prec_{C1} D;\ ws\text{-}prog\ G\ ]\!]$
$\Longrightarrow C \neq D$
⟨*proof*⟩

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

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

lemma *subclseq-cases* [*consumes 1*, *case-names Eq Subcls*]:
$[\![G\vdash C \preceq_C D;\ C = D \Longrightarrow P;\ G\vdash C \prec_C D \Longrightarrow P]\!] \Longrightarrow P$
⟨*proof*⟩

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

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

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

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

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

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

implementation relation

defs

— direct implementation, cf. 8.1.3
implmt1-def: *implmt1* $G \equiv \{(C,I).\ C \neq Object \wedge (\exists\, c{\in}class\ G\ C{:}\ I{\in}set\ (superIfs\ c))\}$

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

inductive — implementation, cf. 8.1.4
implmt :: *prog* ⇒ *qtname* ⇒ *qtname* ⇒ *bool* (-⊢-⇝- [71,71,71] 70)
for G :: *prog*
where
direct: $G{\vdash}C{\rightsquigarrow}1J \Longrightarrow G{\vdash}C{\rightsquigarrow}J$
| *subint*: $[\![G{\vdash}C{\rightsquigarrow}1I;\ G{\vdash}I{\preceq}I\ J]\!] \Longrightarrow G{\vdash}C{\rightsquigarrow}J$
| *subcls1*: $[\![G{\vdash}C{\prec}_{C1}D;\ G{\vdash}D{\rightsquigarrow}J\ ]\!] \Longrightarrow G{\vdash}C{\rightsquigarrow}J$

lemma *implmtD*: $G{\vdash}C{\rightsquigarrow}J \Longrightarrow (\exists\, I.\ G{\vdash}C{\rightsquigarrow}1I \wedge G{\vdash}I{\preceq}I\ J) \vee (\exists\, D.\ G{\vdash}C{\prec}_{C1}D \wedge G{\vdash}D{\rightsquigarrow}J)$
⟨*proof*⟩

lemma *implmt-ObjectE* [*elim*!]: $G{\vdash}Object{\rightsquigarrow}I \Longrightarrow R$
⟨*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*: $G{\vdash}C{\rightsquigarrow}I \Longrightarrow is\text{-}class\ G\ C$
⟨*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*: $G{\vdash}I{\preceq}I\ J \Longrightarrow G{\vdash}Iface\ I{\preceq}\ Iface\ J$ — wid.ref.conv.,cf. 5.1.4
| *int-obj*: $G{\vdash}Iface\ I{\preceq}\ Class\ Object$
| *subcls*: $G{\vdash}C{\preceq}_C\ D \Longrightarrow G{\vdash}Class\ C{\preceq}\ Class\ D$
| *implmt*: $G{\vdash}C{\rightsquigarrow}I \Longrightarrow G{\vdash}Class\ C{\preceq}\ Iface\ I$
| *null*: $G{\vdash}NT{\preceq}\ RefT\ R$
| *arr-obj*: $G{\vdash}T.[]{\preceq}\ Class\ Object$
| *array*: $G{\vdash}RefT\ S{\preceq}RefT\ T \Longrightarrow G{\vdash}RefT\ S.[]{\preceq}\ RefT\ T.[]$

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

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

lemma *widen-PrimT2*: $G\vdash S\preceq PrimT\ x \Longrightarrow \exists\, y.\ S = PrimT\ y$
⟨*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*: $G\vdash PrimT\ x\preceq T \Longrightarrow T = PrimT\ x$
⟨*proof*⟩

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

Specialized versions for booleans also would work for real Java

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

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

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

lemma *widen-RefT2*: $G\vdash S\preceq RefT\ R \Longrightarrow \exists\, t.\ S{=}RefT\ t$
⟨*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*]: $G⊢Class\ C⪯\ Class\ cm = G⊢C⪯_C\ cm$
⟨*proof*⟩

lemma *widen-Class-Iface*: $G⊢Class\ C⪯\ Iface\ I \Longrightarrow G⊢C⇝I$
⟨*proof*⟩

lemma *widen-Class-Iface-eq* [*simp*]: $G⊢Class\ C⪯\ Iface\ I = G⊢C⇝I$
⟨*proof*⟩

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

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

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

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

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

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

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

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

lemma *widen-Object*:$⟦isrtype\ G\ T;ws\text{-}prog\ G⟧ \Longrightarrow G⊢RefT\ T ⪯ Class\ Object$
⟨*proof*⟩

lemma *widen-trans-lemma* [*rule-format* (*no-asm*)]:
$⟦G⊢S⪯U;\ \forall\, C.\ is\text{-}class\ G\ C \longrightarrow G⊢C⪯_C\ Object⟧ \Longrightarrow \forall\, T.\ G⊢U⪯T \longrightarrow G⊢S⪯T$
⟨*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*)]: $[\![G\vdash S\preceq T;$
$\forall I\ J.\ G\vdash I\preceq I\ J \wedge G\vdash J\preceq I\ I \longrightarrow I = J;$
$\forall C\ D.\ G\vdash C\preceq_C D \wedge G\vdash D\preceq_C C \longrightarrow C = D;$
$\forall I\ .\ G\vdash Object\rightsquigarrow I \longrightarrow False]\!] \Longrightarrow G\vdash T\preceq S \longrightarrow S = T$
⟨*proof*⟩

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

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

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

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

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

lemma *widens-Cons* [*simp*]: $G\vdash(S\#Ss)[\preceq](T\#Ts) = (G\vdash S\preceq T \wedge G\vdash Ss[\preceq]Ts)$
⟨*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*: $\neg G\vdash C\rightsquigarrow I \Longrightarrow G\vdash Class\ C\succ Iface\ I$
| *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*: $G⊢PrimT\ pt≻T \Longrightarrow \exists\, t.\ T=PrimT\ t$
⟨*proof*⟩

lemma *narrow-PrimT2*: $G⊢S≻PrimT\ pt \Longrightarrow$
$\exists\, t.\ S=PrimT\ t \wedge G⊢PrimT\ t≼PrimT\ pt$
⟨*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*: $G⊢PrimT\ pt≻T \Longrightarrow T=PrimT\ pt$
⟨*proof*⟩

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

Specialized versions for booleans also would work for real Java

lemma *narrow-Boolean*: $G⊢PrimT\ Boolean≻T \Longrightarrow T=PrimT\ Boolean$
⟨*proof*⟩

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

casting relation

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

lemma *cast-RefT*: $G⊢RefT\ R≼?\ T \Longrightarrow \exists\, t.\ T=RefT\ t$
⟨*proof*⟩

lemma *cast-RefT2*: $G⊢S≼?\ RefT\ R \Longrightarrow \exists\, t.\ S=RefT\ t$
⟨*proof*⟩

lemma *cast-PrimT*: $G⊢PrimT\ pt≼?\ T \Longrightarrow \exists\, t.\ T=PrimT\ t$
⟨*proof*⟩

lemma *cast-PrimT2*: $G⊢S≼?\ PrimT\ pt \Longrightarrow \exists\, t.\ S=PrimT\ t \wedge G⊢PrimT\ t≼PrimT\ pt$
⟨*proof*⟩

lemma *cast-Boolean*:

assumes *bool-cast*: $G \vdash PrimT\ Boolean \preceq ?\ T$
shows $T = PrimT\ Boolean$
⟨*proof*⟩

lemma *cast-Boolean2*:
assumes *bool-cast*: $G \vdash S \preceq ?\ PrimT\ Boolean$
shows $S = PrimT\ Boolean$
⟨*proof*⟩

end

Chapter 10

DeclConcepts

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

theory *DeclConcepts* **imports** *TypeRel* **begin**

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

constdefs
is-public :: *prog* ⇒ *qtname* ⇒ *bool*
is-public G qn ≡ (*case class G qn of*
None ⇒ (*case iface G qn of*
None ⇒ *False*
| *Some iface* ⇒ *access iface* = *Public*)
| *Some class* ⇒ *access class* = *Public*)

17 accessibility of types (cf. 6.6.1)

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

consts *accessible-in* :: *prog* ⇒ *ty* ⇒ *pname* ⇒ *bool*
(- ⊢ - *accessible′-in* - [*61*,*61*,*61*] *60*)
rt-accessible-in:: *prog* ⇒ *ref-ty* ⇒ *pname* ⇒ *bool*
(- ⊢ - *accessible′-in′* - [*61*,*61*,*61*] *60*)
primrec
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*]

constdefs
is-acc-class :: *prog* ⇒ *pname* ⇒ *qtname* ⇒ *bool*
is-acc-class G P C ≡ *is-class G C* ∧ *G*⊢(*Class C*) *accessible-in P*
is-acc-iface :: *prog* ⇒ *pname* ⇒ *qtname* ⇒ *bool*
is-acc-iface G P I ≡ *is-iface G I* ∧ *G*⊢(*Iface I*) *accessible-in P*
is-acc-type :: *prog* ⇒ *pname* ⇒ *ty* ⇒ *bool*
is-acc-type G P T ≡ *is-type G T* ∧ *G*⊢*T accessible-in P*
is-acc-reftype :: *prog* ⇒ *pname* ⇒ *ref-ty* ⇒ *bool*
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 $\Longrightarrow$ *is-type G T* $\wedge$ *G*$\vdash$*T accessible-in P*
⟨*proof*⟩

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

18 accessibility of members

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

Various technical conversion and selection functions

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

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

instance *acc-modi*::*has-accmodi* ⟨*proof*⟩

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

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

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

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

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

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

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

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

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

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

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

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

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

axclass *has-declclass* < *type*
consts *declclass*:: 'a::*has-declclass* ⇒ *qtname*

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

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

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

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

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

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

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

axclass *has-static* < *type*
consts *is-static* :: 'a::*has-static* ⇒ *bool*

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

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

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

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

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

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

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

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

⟨*proof*⟩

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

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

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

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*⟩

constdefs — some mnemotic selectors for various pairs

decliface:: (*qtname* × (*'a*::*type*) *decl-scheme*) ⇒ *qtname*
decliface ≡ *fst* — get the interface component

mbr:: (*qtname* × *memberdecl*) ⇒ *memberdecl*
mbr ≡ *snd* — get the memberdecl component

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

fld:: (*'b* × (*'a*::*type*) *decl-scheme*) ⇒ (*'a*::*type*) *decl-scheme*
— also used for ((*vname* × *qtname*)× *field*)
fld ≡ *snd* — get the field component

constdefs — some mnemotic selectors for (*vname* × *qtname*)
fname:: (*vname* × *'a*) ⇒ *vname* — also used for fdecl
fname ≡ *fst*

declclassf:: (*vname* × *qtname*) ⇒ *qtname*
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*⟩

constdefs — some mnemotic selectors for $(vname \times qtname)$
$fldname :: (vname \times qtname) \Rightarrow vname$
$fldname \equiv fst$

$fldclass :: (vname \times qtname) \Rightarrow qtname$
$fldclass \equiv snd$

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*

constdefs
$methdMembr :: (qtname \times mdecl) \Rightarrow (qtname \times memberdecl)$
$methdMembr\ m \equiv (fst\ m, mdecl\ (snd\ m))$

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

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

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

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

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

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

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

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

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

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

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

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

constdefs
fieldm :: *vname* ⇒ (*qtname* × *field*) ⇒ (*qtname* × *memberdecl*)
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*

constdefs *msig*:: (*qtname* × *mdecl*) ⇒ *sig*
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*

constdefs *qmdecl* :: *sig* ⇒ (*qtname* × *methd*) ⇒ (*qtname* × *mdecl*)
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

axclass *has-resTy* < *type*
consts *resTy*:: *'a*::*has-resTy* ⇒ *ty*

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

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

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

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

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

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

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

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

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

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

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

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

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

inheritable-in

G⊢*m inheritable-in P*: m can be inherited by classes in package P if:

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

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

syntax
Method-inheritable-in::
prog ⇒ (*qtname* × *mdecl*) ⇒ *pname* ⇒ *bool*
(- ⊢*Method* - *inheritable′-in* - [*61*,*61*,*61*] *60*)

translations
G⊢*Method m inheritable-in p* == *G*⊢*methdMembr m inheritable-in p*

syntax
Methd-inheritable-in::
prog ⇒ *sig* ⇒ (*qtname* × *methd*) ⇒ *pname* ⇒ *bool*
(- ⊢*Methd* - - *inheritable′-in* - [*61*,*61*,*61*,*61*] *60*)

translations
G⊢*Methd s m inheritable-in p* == *G*⊢(*method s m*) *inheritable-in p*

declared-in/undeclared-in

constdefs *cdeclaredmethd*:: *prog* ⇒ *qtname* ⇒ (*sig*,*methd*) *table*
cdeclaredmethd G C
≡ (*case class G C of*
None ⇒ λ *sig*. *None*

```
    | Some c ⇒ table-of (methods c)
   )

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


constdefs
declared-in:: prog  ⇒ memberdecl ⇒ qtname ⇒ bool
                                  (-⊢ - declared'-in - [61,61,61] 60)
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)

syntax
method-declared-in:: prog  ⇒ (qtname × mdecl) ⇒ qtname ⇒ bool
                                  (-⊢Method - declared'-in - [61,61,61] 60)
translations
G⊢Method m declared-in C == G⊢mdecl (mthd m) declared-in C

syntax
methd-declared-in:: prog  ⇒ sig  ⇒(qtname × methd) ⇒ qtname ⇒ bool
                                 (-⊢Methd -  - declared'-in - [61,61,61,61] 60)
translations
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⟩

constdefs
undeclared-in:: prog  ⇒ memberid ⇒ qtname ⇒ bool
                                  (-⊢ - undeclared'-in - [61,61,61] 60)

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⊢C ≺C1 S; G⊢(Class S) accessible-in (pid C);G⊢m member-of S
             ⟧ ⟹ G⊢m member-of C
```

Note that in the case of an inherited member only the members of the direct superclass are concerned. If a member of a superclass of the direct superclass isn't inherited in the direct superclass (not

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

syntax
method-member-of:: *prog* ⇒ (*qtname* × *mdecl*) ⇒ *qtname* ⇒ *bool*
(- ⊢*Method* - *member'-of* - [*61*,*61*,*61*] *60*)

translations
G⊢*Method m member-of C* ⇌ *G*⊢(*methdMembr m*) *member-of C*

syntax
methd-member-of:: *prog* ⇒ *sig* ⇒ (*qtname* × *methd*) ⇒ *qtname* ⇒ *bool*
(- ⊢*Methd* - - *member'-of* - [*61*,*61*,*61*,*61*] *60*)

translations
G⊢*Methd s m member-of C* ⇌ *G*⊢(*method s m*) *member-of C*

syntax
fieldm-member-of:: *prog* ⇒ *vname* ⇒ (*qtname* × *field*) ⇒ *qtname* ⇒ *bool*
(- ⊢*Field* - - *member'-of* - [*61*,*61*,*61*] *60*)

translations
G⊢*Field n f member-of C* ⇌ *G*⊢*fieldm n f member-of C*

constdefs
inherits:: *prog* ⇒ *qtname* ⇒ (*qtname* × *memberdecl*) ⇒ *bool*
(- ⊢ - *inherits* - [*61*,*61*,*61*] *60*)
G⊢*C inherits m*
≡ *G*⊢*m inheritable-in* (*pid C*) ∧ *G*⊢*memberid m undeclared-in C* ∧
(∃ *S*. *G*⊢*C* ≺$_{C1}$ *S* ∧ *G*⊢(*Class S*) *accessible-in* (*pid C*) ∧ *G*⊢*m member-of S*)

lemma *inherits-member*: *G*⊢*C inherits m* ⟹ *G*⊢*m member-of C*
⟨*proof*⟩

constdefs *member-in*::*prog* ⇒ (*qtname* × *memberdecl*) ⇒ *qtname* ⇒ *bool*
(- ⊢ - *member'-in* - [*61*,*61*,*61*] *60*)
G⊢*m member-in C* ≡ ∃ *provC*. *G*⊢ *C* ≼$_C$ *provC* ∧ *G* ⊢ *m member-of provC*

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

syntax
method-member-in:: *prog* ⇒ (*qtname* × *mdecl*) ⇒ *qtname* ⇒ *bool*
(- ⊢*Method* - *member'-in* - [*61*,*61*,*61*] *60*)

translations
G⊢*Method m member-in C* ⇌ *G*⊢(*methdMembr m*) *member-in C*

syntax
methd-member-in:: *prog* ⇒ *sig* ⇒ (*qtname* × *methd*) ⇒ *qtname* ⇒ *bool*
(- ⊢*Methd* - - *member'-in* - [*61*,*61*,*61*,*61*] *60*)

translations
G⊢*Methd s m member-in C* ⇌ *G*⊢(*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*: $[\![G \vdash m$ *member-of provC*;$G\vdash C \preceq_C$ *provC*$]\!] \Longrightarrow G\vdash m$ *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: $[\![\neg$ *is-static new*; *msig new* = *msig old*;
$G\vdash$*Method new declared-in* (*declclass new*);
$G\vdash$*Method old declared-in* (*declclass old*);
$G\vdash$*Method old inheritable-in pid* (*declclass new*);
$G\vdash$(*declclass new*) $\prec_{C1}$ *superNew*;
$G \vdash$*Method old member-of superNew*
$]\!] \Longrightarrow G\vdash$*new overrides*$_S$ *old*

| *Indirect*: $[\![G\vdash$*new overrides*$_S$ *inter*; $G\vdash$*inter overrides*$_S$ *old*$]\!]$
$\Longrightarrow G\vdash$*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: $[\![\neg$ *is-static new*; $\neg$ *is-static old*; *accmodi new* $\neq$ *Private*;
msig new = *msig old*;
$G\vdash$(*declclass new*) $\prec_C$ (*declclass old*);
$G\vdash$*Method new declared-in* (*declclass new*);
$G\vdash$*Method old declared-in* (*declclass old*);
$G\vdash$*Method old inheritable-in pid* (*declclass new*);
$G\vdash$*resTy new* $\preceq$ *resTy old*
$]\!] \Longrightarrow G\vdash$*new overrides old*

| *Indirect*: $[\![G\vdash$*new overrides inter*; $G\vdash$*inter overrides old*$]\!]$
$\Longrightarrow G\vdash$*new overrides old*

syntax
sig-stat-overrides::
prog ⇒ *sig* ⇒ (*qtname* × *methd*) ⇒ (*qtname* × *methd*) ⇒ *bool*

(-,-⊢ - $overrides_S$ - [61,61,61,61] 60)

translations
$G,s⊢new\ overrides_S\ old \rightharpoonup G⊢(qmdecl\ s\ new)\ overrides_S\ (qmdecl\ s\ old)$

syntax
sig-overrides:: $prog \Rightarrow sig \Rightarrow (qtname \times methd) \Rightarrow (qtname \times methd) \Rightarrow bool$
(-,-⊢ - *overrides* - [61,61,61,61] 60)

translations
$G,s⊢new\ overrides\ old \rightharpoonup G⊢(qmdecl\ s\ new)\ overrides\ (qmdecl\ s\ old)$

Hiding

constdefs *hides*::
$prog \Rightarrow (qtname \times mdecl) \Rightarrow (qtname \times mdecl) \Rightarrow bool$
(-⊢ - *hides* - [61,61,61] 60)
$G⊢new\ hides\ old$
$\equiv is\text{-}static\ new \wedge msig\ new = msig\ old \wedge$
$G⊢(declclass\ new) \prec_C (declclass\ old) \wedge$
$G⊢Method\ new\ declared\text{-}in\ (declclass\ new) \wedge$
$G⊢Method\ old\ declared\text{-}in\ (declclass\ old) \wedge$
$G⊢Method\ old\ inheritable\text{-}in\ pid\ (declclass\ new)$

syntax
sig-hides:: $prog \Rightarrow sig \Rightarrow (qtname \times mdecl) \Rightarrow (qtname \times mdecl) \Rightarrow bool$
(-,-⊢ - *hides* - [61,61,61,61] 60)

translations
$G,s⊢new\ hides\ old \rightharpoonup G⊢(qmdecl\ s\ new)\ hides\ (qmdecl\ s\ old)$

lemma *hidesI*:
$[\![is\text{-}static\ new;\ msig\ new = msig\ old;$
$G⊢(declclass\ new) \prec_C (declclass\ old);$
$G⊢Method\ new\ declared\text{-}in\ (declclass\ new);$
$G⊢Method\ old\ declared\text{-}in\ (declclass\ old);$
$G⊢Method\ old\ inheritable\text{-}in\ pid\ (declclass\ new)$
$]\!] \Longrightarrow G⊢new\ hides\ old$
⟨*proof*⟩

lemma *hidesD*:
$[\![G⊢new\ hides\ old]\!] \Longrightarrow$
$declclass\ new \neq Object \wedge is\text{-}static\ new \wedge msig\ new = msig\ old \wedge$
$G⊢(declclass\ new) \prec_C (declclass\ old) \wedge$
$G⊢Method\ new\ declared\text{-}in\ (declclass\ new) \wedge$
$G⊢Method\ old\ declared\text{-}in\ (declclass\ old)$
⟨*proof*⟩

lemma *overrides-commonD*:
$[\![G⊢new\ overrides\ old]\!] \Longrightarrow$
$declclass\ new \neq Object \wedge \neg\ is\text{-}static\ new \wedge \neg\ is\text{-}static\ old \wedge$
$accmodi\ new \neq Private \wedge$
$msig\ new = msig\ old\ \wedge$
$G⊢(declclass\ new) \prec_C (declclass\ old) \wedge$
$G⊢Method\ new\ declared\text{-}in\ (declclass\ new) \wedge$
$G⊢Method\ old\ declared\text{-}in\ (declclass\ old)$
⟨*proof*⟩

lemma *ws-overrides-commonD*:
$[\![G\vdash new\ overrides\ old; ws\text{-}prog\ G]\!] \Longrightarrow$
$declclass\ new \neq Object \wedge \neg\ is\text{-}static\ new \wedge \neg\ is\text{-}static\ old \wedge$
$accmodi\ new \neq Private \wedge G\vdash resTy\ new \preceq resTy\ old \wedge$
$msig\ new = msig\ old\ \wedge$
$G\vdash (declclass\ new) \prec_C (declclass\ old) \wedge$
$G\vdash Method\ new\ declared\text{-}in\ (declclass\ new) \wedge$
$G\vdash Method\ old\ declared\text{-}in\ (declclass\ old)$
⟨*proof*⟩

lemma *overrides-eq-sigD*:
$[\![G\vdash new\ overrides\ old]\!] \Longrightarrow msig\ old{=}msig\ new$
⟨*proof*⟩

lemma *hides-eq-sigD*:
$[\![G\vdash new\ hides\ old]\!] \Longrightarrow msig\ old{=}msig\ new$
⟨*proof*⟩

permits access

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

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

The subcondition of the *Protected* case: $G\vdash accclass \prec_C declclass\ membr$ could also be relaxed to: $G\vdash accclass \preceq_C declclass\ membr$ 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
$accessible\text{-}fromR :: prog \Rightarrow qtname \Rightarrow (qtname \times memberdecl) \Rightarrow qtname \Rightarrow bool$
and $accessible\text{-}from :: prog \Rightarrow (qtname \times memberdecl) \Rightarrow qtname \Rightarrow qtname \Rightarrow bool$
$(\text{-} \vdash \text{-}\ of\ \text{-}\ accessible'\text{-}from\ \text{-}\ [61,61,61,61]\ 60)$
and $method\text{-}accessible\text{-}from :: prog \Rightarrow (qtname \times mdecl) \Rightarrow qtname \Rightarrow qtname \Rightarrow bool$
$(\text{-} \vdash Method\ \text{-}\ of\ \text{-}\ accessible'\text{-}from\ \text{-}\ [61,61,61,61]\ 60)$
for $G :: prog$ **and** $accclass :: qtname$

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

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

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

| *Overriding*: $[\![G\vdash membr\ member\text{-}of\ class;$
$G\vdash (Class\ class)\ accessible\text{-}in\ (pid\ accclass);$
$membr=(C,mdecl\ new);$
$G\vdash (C,new)\ overrides_S\ old;$
$G\vdash class \prec_C supr;$
$G\vdash Method\ old\ of\ supr\ accessible\text{-}from\ accclass$
$]\!] \Longrightarrow G\vdash membr\ of\ class\ accessible\text{-}from\ accclass$

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

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

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

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

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

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

| *Immediate*: $[\![G\vdash membr\ member\text{-}in\ class;$
$G\vdash membr\ in\ class\ permits\text{-}acc\text{-}from\ accclass$
$]\!] \Longrightarrow G\vdash membr\ in\ class\ dyn\text{-}accessible\text{-}from\ accclass$

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

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

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

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

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

lemma *accessible-from-commonD*: $G\vdash m\ of\ C\ accessible\text{-}from\ S$
$\Longrightarrow G\vdash m\ member\text{-}of\ C \wedge G\vdash(Class\ C)\ accessible\text{-}in\ (pid\ S)$
⟨*proof*⟩

lemma *unique-declaration*:
$[\![G\vdash m\ declared\text{-}in\ C;\ G\vdash n\ declared\text{-}in\ C;\ memberid\ m = memberid\ n\ ]\!]$
$\Longrightarrow m = n$
⟨*proof*⟩

lemma *declared-not-undeclared*:
$G\vdash m\ declared\text{-}in\ C \Longrightarrow \neg\ G\vdash\ memberid\ m\ undeclared\text{-}in\ C$
⟨*proof*⟩

lemma *undeclared-not-declared*:
$G\vdash\ memberid\ m\ undeclared\text{-}in\ C \Longrightarrow \neg\ G\vdash\ m\ declared\text{-}in\ C$
⟨*proof*⟩

lemma *not-undeclared-declared*:
$\neg\ G\vdash\ membr\text{-}id\ undeclared\text{-}in\ C \Longrightarrow (\exists\ m.\ G\vdash m\ declared\text{-}in\ C \wedge$
$membr\text{-}id = memberid\ m)$
⟨*proof*⟩

lemma *unique-declared-in*:
$[\![G\vdash m\ declared\text{-}in\ C;\ G\vdash n\ declared\text{-}in\ C;\ memberid\ m = memberid\ n]\!]$
$\Longrightarrow m = n$
⟨*proof*⟩

lemma *unique-member-of*:
assumes n: $G\vdash n\ member\text{-}of\ C$ **and**
m: $G\vdash m\ member\text{-}of\ C$ **and**
$eqid$: $memberid\ n = memberid\ m$
shows $n{=}m$
⟨*proof*⟩

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

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

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

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

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

lemma *stat-override-declclasses-relation*:
$[\![G\vdash$(*declclass new*) $\prec_{C1}$ *superNew*; $G \vdash$*Method old member-of superNew* $]\!]$
$\Longrightarrow G\vdash$(*declclass new*) $\prec_C$ (*declclass old*)
⟨*proof*⟩

lemma *stat-overrides-commonD*:
$[\![G\vdash$*new overrides*$_S$ *old*$]\!] \Longrightarrow$
declclass new $\neq$ *Object* $\wedge$ $\neg$ *is-static new* $\wedge$ *msig new* = *msig old* $\wedge$
$G\vdash$(*declclass new*) $\prec_C$ (*declclass old*) $\wedge$
$G\vdash$*Method new declared-in* (*declclass new*) $\wedge$
$G\vdash$*Method old declared-in* (*declclass old*)
⟨*proof*⟩

lemma *member-of-Package*:
$[\![G\vdash m$ *member-of* C; *accmodi* m = *Package*$]\!]$
$\Longrightarrow$ *pid* (*declclass* m) = *pid* C
⟨*proof*⟩

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

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

lemma *no-Private-stat-override*:

$[\![G\vdash new\ overrides_S\ old]\!] \Longrightarrow accmodi\ old \neq Private$
⟨*proof*⟩

lemma *no-Private-override*: $[\![G\vdash new\ overrides\ old]\!] \Longrightarrow accmodi\ old \neq Private$
⟨*proof*⟩

lemma *permits-acc-inheritance*:
$[\![G\vdash m\ in\ statC\ permits\text{-}acc\text{-}from\ accC;\ G\vdash dynC \preceq_C statC$
$]\!] \Longrightarrow G\vdash m\ in\ dynC\ permits\text{-}acc\text{-}from\ accC$
⟨*proof*⟩

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

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

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

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

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

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

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

lemma *member-of-inheritance*:
assumes m: $G\vdash m$ *member-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-prog* G
shows $G\vdash m$ *member-of* C
⟨*proof*⟩

lemma *member-of-subcls*:
assumes *old*: $G\vdash$ *old member-of* C **and**
new: $G\vdash$ *new member-of* D **and**
eqid: *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-prog* G
shows $G\vdash D \prec_C C$
⟨*proof*⟩

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

corollary *member-of-stat-overrides-subcls*:
⟦$G\vdash$ *Methd sig old member-of* C; $G\vdash$ *Methd sig new member-of* D;$G\vdash D \preceq_C C$;
G,*sig*$\vdash$*new overrides*$_S$ *old*; *ws-prog* G⟧
$\Longrightarrow G\vdash D \prec_C C$
⟨*proof*⟩

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

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

fields and methods

types
fspec = *vname* × *qtname*

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

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

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

constdefs
accimethds:: *prog* ⇒ *pname* ⇒ *qtname* ⇒ (*sig*,*qtname* × *mhead*) *tables*
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

constdefs
methd:: *prog* ⇒ *qtname* ⇒ (*sig*,*qtname* × *methd*) *table*

methd G C
≡ *class-rec* (*G*,*C*) *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.

constdefs
accmethd:: *prog* ⇒ *qtname* ⇒ *qtname* ⇒ (*sig*,*qtname* × *methd*) *table*
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*)

constdefs
dynmethd:: *prog* ⇒ *qtname* ⇒ *qtname* ⇒ (*sig*,*qtname* × *methd*) *table*
dynmethd G statC dynC
≡ λ *sig*.
(*if G*⊢*dynC* ≼$_C$ *statC*
then (*case methd G statC sig of*
None ⇒ *None*
| *Some statM*
⇒ (*class-rec* (*G*,*dynC*) *empty*
(λ*C 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)

constdefs

```
dynimethd:: prog ⇒ qtname ⇒ qtname ⇒ (sig,qtname × methd) table
dynimethd G I dynC
  ≡ λ sig. 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.

constdefs

```
dynlookup::prog ⇒ ref-ty ⇒ qtname ⇒ (sig,qtname × methd) table
dynlookup G statT dynC
  ≡ (case statT of
       NullT       ⇒ 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

constdefs

```
fields:: prog ⇒ qtname ⇒ ((vname × qtname) × field) list
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

constdefs

```
accfield:: prog ⇒ qtname ⇒ qtname ⇒ (vname, qtname  ×  field) table
accfield G S C
  ≡ let field-tab = table-of((map (λ((n,d),f).(n,(d,f)))) (fields G C))
    in filter-tab (λn (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)

constdefs

is-methd :: *prog* ⇒ *qtname* ⇒ *sig* ⇒ *bool*
is-methd G ≡ λ*C sig*. *is-class G C* ∧ *methd G C sig* ≠ *None*

constdefs *efname*:: ((*vname* × *qtname*) × *field*) ⇒ (*vname* × *qtname*)
efname ≡ *fst*

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

19 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*)) ⊕⊕
(*Option.set* ∘ *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* [*consumes 3*, *case-names NewMethod InheritedMethod*]:
assumes *im*: *im* ∈ *imethds G I sig* **and**
ifI: *iface G I* = *Some i* **and**
ws: *ws-prog G* **and**
hyp-new: *table-of* (*map* (λ(*s*, *mh*). (*s*, *I*, *mh*)) (*imethods i*)) *sig*
= *Some im* ⟹ *P* **and**
hyp-inh: ⋀ *J*. ⟦*J* ∈ *set* (*isuperIfs i*); *im* ∈ *imethds G J sig*⟧ ⟹ *P*
shows *P*
⟨*proof*⟩

20 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* ∈ *imethds G I sig*;*G*⊢*Iface I accessible-in pack*⟧
⟹ *im* ∈ *accimethds G pack I sig*
⟨*proof*⟩

21 methd

lemma *methd-rec*: ⟦*class G C* = *Some c*; *ws-prog G*⟧ ⟹
methd G C
= (*if C* = *Object*
then empty
else filter-tab (λ*sig m*. *G*⊢*C inherits method sig m*)
(*methd G* (*super c*)))
++ *table-of* (*map* (λ(*s*,*m*). (*s*,*C*,*m*)) (*methods c*))
⟨*proof*⟩

lemma *methd-norec*:
⟦*class G declC* = *Some c*; *ws-prog G*;*table-of* (*methods c*) *sig* = *Some m*⟧
⟹ *methd G declC sig* = *Some* (*declC*, *m*)
⟨*proof*⟩

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

lemma *methd-inheritedD*:
⟦*class G C* = *Some c*; *ws-prog G*;*methd G C sig* = *Some m*⟧
⟹ (*declclass m* ≠ *C* ⟶ *G* ⊢*C inherits method sig m*)
⟨*proof*⟩

lemma *methd-diff-cls*:
⟦*ws-prog G*; *is-class G C*; *is-class G D*;
methd G C sig = *m*; *methd G D sig* = *n*; *m*≠*n*
⟧ ⟹ *C*≠*D*
⟨*proof*⟩

lemma *method-declared-inI*:
⟦*table-of* (*methods c*) *sig* = *Some m*; *class G C* = *Some c*⟧
⟹ *G*⊢*mdecl* (*sig*,*m*) *declared-in C*
⟨*proof*⟩

lemma *methd-declared-in-declclass*:
⟦*methd G C sig* = *Some m*; *ws-prog G*;*is-class G C*⟧
⟹ *G*⊢*Methd sig m declared-in* (*declclass m*)
⟨*proof*⟩

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

lemma *finite-methd*:*ws-prog G* ⟹ *finite* {*methd G C sig* |*sig C*. *is-class G C*}
⟨*proof*⟩

lemma *finite-dom-methd*:
⟦*ws-prog G*; *is-class G C*⟧ ⟹ *finite* (*dom* (*methd G C*))
⟨*proof*⟩

22 accmethd

lemma *accmethd-SomeD*:
accmethd G S C sig = *Some m*
⟹ *methd G C sig* = *Some m* ∧ $G \vdash$*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*⟧
⟹ *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*⟧ ⟹
(∃ *d*. *class G* (*declclass m*)=*Some d* ∧
table-of (*methods d*) *sig*=*Some* (*mthd m*)) ∧
$G \vdash C \preceq_C$ (*declclass m*) ∧ *methd G* (*declclass m*) *sig* = *Some m* ∧
$G \vdash$*method sig m of C accessible-from S*
⟨*proof*⟩

lemma *finite-dom-accmethd*:
⟦*ws-prog G*; *is-class G C*⟧ ⟹ *finite* (*dom* (*accmethd G S C*))
⟨*proof*⟩

23 dynmethd

lemma *dynmethd-rec*:
⟦*class G dynC* = *Some c*; *ws-prog G*⟧ ⟹
dynmethd G statC dynC sig
= (*if* $G \vdash$*dynC* $\preceq_C$ *statC*
then (*case methd G statC sig of*
None ⇒ *None*
| *Some statM*
⇒ (*case methd G dynC sig of*
None ⇒ *dynmethd G statC* (*super c*) *sig*
| *Some dynM* ⇒
(*if G*,*sig*⊢ *dynM overrides statM* ∨ *dynM* = *statM*
then Some dynM

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

lemma *dynmethd-C-C*:⟦*is-class G C*; *ws-prog G*⟧
⟹ *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*⟧
⟹ *G*⊢*dynC* ≼$_C$ *statC* ∧ (∃ *statM*. *methd G statC sig* = *Some statM*)
⟨*proof*⟩

lemma *dynmethd-Some-cases* [*consumes 3*, *case-names Static Overrides*]:
assumes *dynM*: *dynmethd G statC dynC sig* = *Some dynM* **and**
is-cls-dynC: *is-class G dynC* **and**
ws: *ws-prog G* **and**
hyp-static: *methd G statC sig* = *Some dynM* ⟹ *P* **and**
hyp-override: ⋀ *statM*. ⟦*methd G statC sig* = *Some statM*;*dynM*≠*statM*;
G,*sig*⊢*dynM overrides statM*⟧ ⟹ *P*
shows *P*
⟨*proof*⟩

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

lemma *dynmethd-Some-rec-cases* [*consumes 3*,
case-names Static Override Recursion]:
assumes *dynM*: *dynmethd G statC dynC sig* = *Some dynM* **and**
clsDynC: *class G dynC* = *Some c* **and**
ws: *ws-prog G* **and**
hyp-static: *methd G statC sig* = *Some dynM* ⟹ *P* **and**
hyp-override: ⋀ *statM*. ⟦*methd G statC sig* = *Some statM*;
methd G dynC sig = *Some dynM*; *statM*≠*dynM*;
G,*sig*⊢ *dynM overrides statM*⟧ ⟹ *P* **and**

hyp-recursion: ⟦*dynC*≠*Object*;
dynmethd G statC (*super c*) *sig* = *Some dynM*⟧ ⟹ *P*
shows *P*
⟨*proof*⟩

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

$\rrbracket \Longrightarrow$
$(\exists\, d.\ class\ G\ (declclass\ m){=}Some\ d \wedge table\text{-}of\ (methods\ d)\ sig{=}Some\ (mthd\ m)) \wedge$
$G{\vdash}dynC \preceq_C (declclass\ m) \wedge methd\ G\ (declclass\ m)\ sig = Some\ m$
⟨*proof*⟩

lemma *methd-Some-dynmethd-Some*:
assumes *statM*: *methd G statC sig = Some statM* **and**
subclseq: *G⊢dynC ≼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* [*consumes 4*, *case-names Static Overrides*]:
assumes *statM*: *methd G statC sig = Some statM* **and**
subclseq: *G⊢dynC ≼C statC* **and**
is-cls-statC: *is-class G statC* **and**
ws: *ws-prog G* **and**
hyp-static: *dynmethd G statC dynC sig = Some statM* $\Longrightarrow$ *P* **and**
hyp-override: $\bigwedge$ *dynM*. ⟦*dynmethd G statC dynC sig = Some dynM*;
dynM≠statM;
G,sig⊢dynM overrides statM⟧ $\Longrightarrow$ *P*
shows *P*
⟨*proof*⟩

lemma *ws-dynmethd*:
assumes *statM*: *methd G statC sig = Some statM* **and**
subclseq: *G⊢dynC ≼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-static dynM = is-static statM ∧ *G⊢resTy dynM≼resTy statM*
⟨*proof*⟩

24 dynlookup

lemma *dynlookup-cases* [*consumes 1*, *case-names NullT IfaceT ClassT ArrayT*]:
⟦*dynlookup G statT dynC sig = x*;
⟦*statT = NullT* ; *empty sig = x* ⟧ $\Longrightarrow$ *P*;
$\bigwedge$ *I*. ⟦*statT = IfaceT I* ; *dynimethd G I dynC sig = x*⟧ $\Longrightarrow$ *P*;
$\bigwedge$ *statC*.⟦*statT = ClassT statC*; *dynmethd G statC dynC sig = x*⟧ $\Longrightarrow$ *P*;
$\bigwedge$ *ty*. ⟦*statT = ArrayT ty* ; *dynmethd G Object dynC sig = x*⟧ $\Longrightarrow$ *P*
⟧ $\Longrightarrow$ *P*
⟨*proof*⟩

25 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*:
$[\![class\ G\ fd = Some\ c;\ ws\text{-}prog\ G;\ \ table\text{-}of\ (cfields\ c)\ fn = Some\ f]\!]$
$\Longrightarrow table\text{-}of\ (fields\ G\ fd)\ (fn,fd) = Some\ f$
⟨*proof*⟩

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

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

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

lemma *fields-mono-lemma*:
$[\![x \in set\ (fields\ G\ C);\ G{\vdash}D \preceq_C C;\ ws\text{-}prog\ G]\!]$
$\Longrightarrow x \in set\ (fields\ G\ D)$
⟨*proof*⟩

lemma *ws-unique-fields-lemma*:
$[\![(efn,fd)\ \in set\ (fields\ G\ (super\ c));\ fc \in set\ (cfields\ c);\ ws\text{-}prog\ G;$
$fname\ efn = fname\ fc;\ declclassf\ efn = C;$
$class\ G\ C = Some\ c;\ C \neq Object;\ class\ G\ (super\ c) = Some\ d]\!] \Longrightarrow R$
⟨*proof*⟩

lemma *ws-unique-fields*: $[\![is\text{-}class\ G\ C;\ ws\text{-}prog\ G;$
$\bigwedge C\ c.\ [\![class\ G\ C = Some\ c]\!] \Longrightarrow unique\ (cfields\ c)\ ]\!] \Longrightarrow$
$unique\ (fields\ G\ C)$
⟨*proof*⟩

26 accfield

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

lemma *accfield-declC-is-class*:
$[\![is\text{-}class\ G\ C;\ accfield\ G\ S\ C\ en = Some\ (fd,\ f);\ ws\text{-}prog\ G]\!] \Longrightarrow$
$is\text{-}class\ G\ fd$

⟨*proof*⟩

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

27 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* (*split* (*is-methd G*)))
⟨*proof*⟩

calculation of the superclasses of a class

constdefs
superclasses:: *prog ⇒ qtname ⇒ qtname set*
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*:
⟦*G⊢C ≺C D*;*ws-prog G*; *class G C = Some c*;
⋀ *C c*. ⟦*class G C = Some c*;*C≠Object*⟧ ⟹ ∃ *sc*. *class G* (*super c*) = *Some sc*;
x∈superclasses G D
⟧ ⟹ *x∈superclasses G C*
⟨*proof*⟩

lemma *subclsEval*:
⟦*G⊢C ≺C D*;*ws-prog G*; *class G C = Some c*;
⋀ *C c*. ⟦*class G C = Some c*;*C≠Object*⟧ ⟹ ∃ *sc*. *class G* (*super c*) = *Some sc*
⟧ ⟹ *D∈superclasses G C*
⟨*proof*⟩

end

Chapter 11

WellType

28 Well-typedness of Java programs

theory *WellType* **imports** *DeclConcepts* **begin**

improvements over Java Specification 1.0:

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

simplifications:

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

design issues:

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

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

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

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

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

Static overloading: maximally specific methods

types
emhead = *ref-ty* × *mhead*

— Some mnemotic selectors for emhead
constdefs
declrefT :: *emhead* ⇒ *ref-ty*
declrefT ≡ *fst*

mhd :: *emhead* ⇒ *mhead*
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*⟩

consts
cmheads :: *prog* ⇒ *qtname* ⇒ *qtname* ⇒ *sig* ⇒ *emhead set*
Objectmheads :: *prog* ⇒ *qtname* ⇒ *sig* ⇒ *emhead set*
accObjectmheads:: *prog* ⇒ *qtname* ⇒ *ref-ty* ⇒ *sig* ⇒ *emhead set*
mheads :: *prog* ⇒ *qtname* ⇒ *ref-ty* ⇒ *sig* ⇒ *emhead set*
defs
cmheads-def:
cmheads G S C
≡ λ*sig*. (λ(*Cls*,*mthd*). (*ClassT Cls*,(*mhead mthd*))) ‘ *Option.set* (*accmethd G S C sig*)
Objectmheads-def:
Objectmheads G S
≡ λ*sig*. (λ(*Cls*,*mthd*). (*ClassT Cls*,(*mhead mthd*)))
‘ *Option.set* (*filter-tab* (λ*sig m*. *accmodi m* ≠ *Private*) (*accmethd G S Object*) *sig*)
accObjectmheads-def:
accObjectmheads G S T
≡ *if G*⊢*RefT T accessible-in* (*pid S*)
then Objectmheads G S
else λ*sig*. {}
primrec
mheads G S NullT = (λ*sig*. {})
mheads G S (*IfaceT I*) = (λ*sig*. (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*)

constdefs
— applicable methods, cf. 15.11.2.1
appl-methds :: *prog* ⇒ *qtname* ⇒ *ref-ty* ⇒ *sig* ⇒ (*emhead* × *ty list*) *set*
appl-methds G S rt ≡ λ *sig*.
{(*mh*,*pTs'*) |*mh pTs'*. *mh* ∈ *mheads G S rt* (|*name*=*name sig*,*parTs*=*pTs'*|) ∧
G⊢(*parTs sig*)[⪯]*pTs'*}

— more specific methods, cf. 15.11.2.2
more-spec :: *prog* ⇒ *emhead* × *ty list* ⇒ *emhead* × *ty list* ⇒ *bool*
more-spec G ≡ λ(*mh*,*pTs*). λ(*mh'*,*pTs'*). *G*⊢*pTs*[⪯]*pTs'*

— maximally specific methods, cf. 15.11.2.2
max-spec :: *prog* ⇒ *qtname* ⇒ *ref-ty* ⇒ *sig* ⇒ (*emhead* × *ty list*) *set*

max-spec G S rt sig ≡{*m*. *m* ∈*appl-methds G S rt sig* ∧
(∀ *m*′∈*appl-methds G S rt sig*. *more-spec G m*′ *m* ⟶ *m*′=*m*)}

lemma *max-spec2appl-meths*:
x ∈ *max-spec G S T sig* ⟹ *x* ∈ *appl-methds G S T sig*
⟨*proof*⟩

lemma *appl-methsD*: (*mh*,*pTs*′)∈*appl-methds G S T* (|*name*=*mn*,*parTs*=*pTs*|) ⟹
mh ∈ *mheads G S T* (|*name*=*mn*,*parTs*=*pTs*′|) ∧ *G*⊢*pTs*[≼]*pTs*′
⟨*proof*⟩

lemma *max-spec2mheads*:
max-spec G S rt (|*name*=*mn*,*parTs*=*pTs*|) = *insert* (*mh*, *pTs*′) *A*
⟹ *mh* ∈ *mheads G S rt* (|*name*=*mn*,*parTs*=*pTs*′|) ∧ *G*⊢*pTs*[≼]*pTs*′
⟨*proof*⟩

constdefs
empty-dt :: *dyn-ty*
empty-dt ≡ λ*a*. *None*

invmode :: (′*a*::*type*)*member-scheme* ⇒ *expr* ⇒ *inv-mode*
invmode m e ≡ *if is-static m*
then Static
else if e=*Super then SuperM else IntVir*

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

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

lemma *invmode-IntVir-eq*: (*invmode m e* = *IntVir*) = (¬(*is-static m*) ∧ *e*≠*Super*)
⟨*proof*⟩

lemma *Null-staticD*:
a′=*Null* ⟶ (*is-static m*) ⟹ *invmode m e* = *IntVir* ⟶ *a*′ ≠ *Null*
⟨*proof*⟩

Typing for unary operations

consts *unop-type* :: *unop* ⇒ *prim-ty*
primrec
unop-type UPlus = *Integer*

unop-type UMinus = *Integer*
unop-type UBitNot = *Integer*
unop-type UNot = *Boolean*

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

Typing for binary operations

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

consts *wt-binop* :: *prog* ⇒ *binop* ⇒ *ty* ⇒ *ty* ⇒ *bool*
primrec
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*⊢*t1*≼*t2* ∨ *G*⊢*t2*≼*t1*)
wt-binop G Neq *t1 t2* = (*G*⊢*t1*≼*t2* ∨ *G*⊢*t2*≼*t1*)
wt-binop G BitAnd *t1 t2* = ((*t1* = *PrimT Integer*) ∧ (*t2* = *PrimT Integer*))
wt-binop G And *t1 t2* = ((*t1* = *PrimT Boolean*) ∧ (*t2* = *PrimT Boolean*))
wt-binop G BitXor *t1 t2* = ((*t1* = *PrimT Integer*) ∧ (*t2* = *PrimT Integer*))
wt-binop G Xor *t1 t2* = ((*t1* = *PrimT Boolean*) ∧ (*t2* = *PrimT Boolean*))
wt-binop G BitOr *t1 t2* = ((*t1* = *PrimT Integer*) ∧ (*t2* = *PrimT Integer*))

wt-binop G Or t1 t2 = ((*t1* = *PrimT Boolean*) ∧ (*t2* = *PrimT Boolean*))
wt-binop G CondAnd t1 t2 = ((*t1* = *PrimT Boolean*) ∧ (*t2* = *PrimT Boolean*))
wt-binop G CondOr t1 t2 = ((*t1* = *PrimT Boolean*) ∧ (*t2* = *PrimT Boolean*))

Typing for terms

types *tys* = *ty* + *ty list*
translations
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*
(-,-⊨-::�doteq- [*51*,*51*,*51*,*51*] *50*)
where

E,*dt*⊨*s*::√ ≡ *E*,*dt*⊨*In1r s*::*Inl* (*PrimT Void*)
| *E*,*dt*⊨*e*::−*T* ≡ *E*,*dt*⊨*In1l e*::*Inl T*
| *E*,*dt*⊨*e*::=*T* ≡ *E*,*dt*⊨*In2 e*::*Inl T*
| *E*,*dt*⊨*e*::≐*T* ≡ *E*,*dt*⊨*In3 e*::*Inr T*

— well-typed statements

| *Skip*: *E*,*dt*⊨*Skip*::√

| *Expr*: ⟦*E*,*dt*⊨*e*::−*T*⟧ ⟹
E,*dt*⊨*Expr e*::√
— cf. 14.6
| *Lab*: *E*,*dt*⊨*c*::√ ⟹
E,*dt*⊨*l*• *c*::√

| *Comp*: ⟦*E*,*dt*⊨*c1*::√;
E,*dt*⊨*c2*::√⟧ ⟹
E,*dt*⊨*c1*;; *c2*::√

— cf. 14.8
| *If*: ⟦*E*,*dt*⊨*e*::−*PrimT Boolean*;
E,*dt*⊨*c1*::√;
E,*dt*⊨*c2*::√⟧ ⟹
E,*dt*⊨*If*(*e*) *c1 Else c2*::√

— cf. 14.10
| *Loop*: ⟦*E*,*dt*⊨*e*::−*PrimT Boolean*;
E,*dt*⊨*c*::√⟧ ⟹
E,*dt*⊨*l*• *While*(*e*) *c*::√
— cf. 14.13, 14.15, 14.16
| *Jmp*: *E*,*dt*⊨*Jmp jump*::√

— cf. 14.16
| *Throw*: ⟦*E*,*dt*⊨*e*::−*Class tn*;
prg E⊢*tn*≼$_C$ *SXcpt Throwable*⟧ ⟹
E,*dt*⊨*Throw e*::√
— cf. 14.18
| *Try*: ⟦*E*,*dt*⊨*c1*::√; *prg E*⊢*tn*≼$_C$ *SXcpt Throwable*;
lcl E (*VName vn*)=*None*; *E* (|*lcl* := *lcl E*(*VName vn*↦*Class tn*)|),*dt*⊨*c2*::√⟧

$\Longrightarrow$
$E,dt\models Try\ c1\ Catch(tn\ vn)\ c2::\surd$

— 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*: $[\![E,dt\models va::=T;\ va \neq LVar\ This;$
$E,dt\models v::-T';$
$prg\ E\vdash T'\preceq T]\!] \Longrightarrow$
$E,dt\models va:=v::-T'$

— cf. 15.24
| *Cond*: ⟦*E*,*dt*⊨*e0*::−*PrimT Boolean*;
E,*dt*⊨*e1*::−*T1*; *E*,*dt*⊨*e2*::−*T2*;
prg E⊢*T1*⪯*T2* ∧ *T* = *T2* ∨ *prg E*⊢*T2*⪯*T1* ∧ *T* = *T1*⟧ ⟹
E,*dt*⊨*e0* ? *e1* : *e2*::−*T*

— cf. 15.11.1, 15.11.2, 15.11.3
| *Call*: ⟦*E*,*dt*⊨*e*::−*RefT statT*;
E,*dt*⊨*ps*::≐*pTs*;
max-spec (*prg E*) (*cls E*) *statT* ⦇*name*=*mn*,*parTs*=*pTs*⦈
= {((*statDeclT*,*m*),*pTs*′)}
⟧ ⟹
E,*dt*⊨{*cls E*,*statT*,*invmode m e*}*e*·*mn*({*pTs*′}*ps*)::−(*resTy m*)

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

— well-typed expression lists

— cf. 15.11.???
| *Nil*: *E*,*dt*⊨[]::≐[]

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

syntax
-wt :: *env* ⇒ [*term*,*tys*] ⇒ *bool* (-|−-::- [*51*,*51*,*51*] *50*)
-wt-stmt :: *env* ⇒ *stmt* ⇒ *bool* (-|−-:<> [*51*,*51*] *50*)
-ty-expr :: *env* ⇒ [*expr* ,*ty*] ⇒ *bool* (-|−-:−- [*51*,*51*,*51*] *50*)
-ty-var :: *env* ⇒ [*var* ,*ty*] ⇒ *bool* (-|−-:=- [*51*,*51*,*51*] *50*)
-ty-exprs:: *env* ⇒ [*expr list*,
ty *list*] ⇒ *bool* (-|−-:#- [*51*,*51*,*51*] *50*)

syntax (*xsymbols*)
-wt :: *env* ⇒ [*term*,*tys*] ⇒ *bool* (-⊢-::- [*51*,*51*,*51*] *50*)
-wt-stmt :: *env* ⇒ *stmt* ⇒ *bool* (-⊢-::√ [*51*,*51*] *50*)
-ty-expr :: *env* ⇒ [*expr* ,*ty*] ⇒ *bool* (-⊢-::−- [*51*,*51*,*51*] *50*)
-ty-var :: *env* ⇒ [*var* ,*ty*] ⇒ *bool* (-⊢-::=- [*51*,*51*,*51*] *50*)
-ty-exprs :: *env* ⇒ [*expr list*,
ty *list*] ⇒ *bool* (-⊢-::�doteq- [*51*,*51*,*51*] *50*)

translations
E⊢*t*:: *T* == *E*,*empty-dt*⊨*t*:: *T*
E⊢*s*::√ == *E*⊢*In1r s*::*Inl* (*PrimT Void*)
E⊢*e*::−*T* == *E*⊢*In1l e*::*Inl T*
E⊢*e*::=*T* == *E*⊢*In2 e*::*Inl T*
E⊢*e*::�doteq*T* == *E*⊢*In3 e*::*Inr T*

declare *not-None-eq* [*simp del*]
declare *split-if* [*split del*] *split-if-asm* [*split del*]
declare *split-paired-All* [*simp del*] *split-paired-Ex* [*simp del*]
⟨*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*

$E,dt\models In1r\ (c1\ Finally\ c2)\quad ::x$
$E,dt\models In1r\ (Init\ C)\quad ::x$
declare *not-None-eq* [*simp*]
declare *split-if* [*split*] *split-if-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*::�doteq*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*:
$\bigwedge E.\ E,dt\models t::U \Longrightarrow case\ t\ of$
$\qquad In1\ ec \Rightarrow (case\ ec\ of$
$\qquad\qquad Inl\ e \Rightarrow \exists\ T.\ U=Inl\ T$
$\qquad\qquad \mid Inr\ s \Rightarrow U=Inl\ (PrimT\ Void))$
$\qquad \mid In2\ e \Rightarrow (\exists\ T.\ U=Inl\ T)$
$\qquad \mid In3\ e \Rightarrow (\exists\ T.\ U=Inr\ T)$
⟨*proof*⟩

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

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

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

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

⟨*ML*⟩

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

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

⟨*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* [*rule-format* (*no-asm*)]:
typeof $(\lambda a.\ None)\ v = Some\ T \longrightarrow$ *is-type G T*
⟨*proof*⟩

lemma *typeof-is-type* [*rule-format* (*no-asm*)]:
$(\forall a.\ v \neq Addr\ a) \longrightarrow (\exists T.\ typeof\ dt\ v = Some\ T \wedge is\text{-}type\ G\ T)$
⟨*proof*⟩

end

Chapter 12

DefiniteAssignment

29 Definite Assignment

theory *DefiniteAssignment* **imports** *WellType* **begin**

Definite Assignment Analysis (cf. 16)

The definite assignment analysis approximates the sets of local variables that will be assigned at a certain point of evaluation, and ensures that we will only read variables which previously were assigned. It should conform to the following idea: If the evaluation of a term completes normally (no 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.

consts *jumpNestingOkS* :: *jump set* ⇒ *stmt* ⇒ *bool*
primrec
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*

constdefs *jumpNestingOk* :: *jump set* ⇒ *term* ⇒ *bool*
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

30 Very restricted calculation fallback calculation

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

consts *assignsE* :: *expr* ⇒ *lname set*

assignsV :: *var* ⇒ *lname set*
assignsEs:: *expr list* ⇒ *lname set*

primrec
assignsE (*NewC c*) = {}
assignsE (*NewA t e*) = *assignsE e*
assignsE (*Cast t e*) = *assignsE e*
assignsE (*e InstOf r*) = *assignsE e*
assignsE (*Lit val*) = {}
assignsE (*UnOp unop e*) = *assignsE e*
assignsE (*BinOp binop e1 e2*) = (*if binop*=*CondAnd* ∨ *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*

constdefs *assigns*:: *term* ⇒ *lname set*
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*⟩

31 Analysis of constant expressions

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

lemma *constVal-Some-induct* [*consumes 1*, *case-names Lit UnOp BinOp CondL CondR*]:
assumes *const*: *constVal e* = *Some v* **and**
hyp-Lit: ⋀ *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: $\bigwedge b\ bv\ e1\ e2.\ [\![constVal\ b = Some\ bv;\ the\text{-}Bool\ bv;\ P\ b;\ P\ e1]\!]$
$\Longrightarrow P\ (b?\ e1 : e2)$ **and**
hyp-CondR: $\bigwedge b\ bv\ e1\ e2.\ [\![constVal\ b = Some\ bv;\ \neg the\text{-}Bool\ bv;\ P\ b;\ P\ e2]\!]$
$\Longrightarrow P\ (b?\ e1 : e2)$
shows $P\ e$
⟨*proof*⟩

lemma *assignsE-const-simp*: $constVal\ e = Some\ v \Longrightarrow assignsE\ e = \{\}$
⟨*proof*⟩

32 Main analysis for boolean expressions

Assigned local variables after evaluating the expression if it evaluates to a specific boolean value. If the expression cannot evaluate to a *Boolean* value UNIV is returned. If we expect true/false the opposite constant false/true will also lead to UNIV.

consts *assigns-if*:: *bool* ⇒ *expr* ⇒ *lname set*
primrec
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*⟩

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

constdefs
union-ts:: ('*a*,'*b*) *tables* ⇒ ('*a*,'*b*) *tables* ⇒ ('*a*,'*b*) *tables*
(- ⇒∪ - [*67*,*67*] *65*)
A ⇒∪ *B* ≡ λ *k*. *A k* ∪ *B k*

constdefs
intersect-ts:: ('*a*,'*b*) *tables* ⇒ ('*a*,'*b*) *tables* ⇒ ('*a*,'*b*) *tables*
(- ⇒∩ - [*72*,*72*] *71*)
A ⇒∩ *B* ≡ λ *k*. *A k* ∩ *B k*

constdefs
all-union-ts:: ('*a*,'*b*) *tables* ⇒ '*b set* ⇒ ('*a*,'*b*) *tables*
(**infixl** ⇒∪∀ *40*)
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*]: $c \in (A \Rightarrow\cap\ B)\ k = (c \in A\ k \wedge c \in B\ k)$
⟨*proof*⟩

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

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

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

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

All-Union of tables and set

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

types *breakass* = (*label*, *lname*) *tables*
— Mapping from a break label, to the set of variables that will be assigned if the evaluation terminates with this break

record *assigned* =
nrm :: *lname set* — Definetly assigned variables for normal completion
brk :: *breakass* — Definetly assigned variables for abrupt completion with a break

constdefs *rmlab* :: $'a \Rightarrow ('a,'b)$ *tables* $\Rightarrow ('a,'b)$ *tables*
rmlab k A $\equiv \lambda$ *x. if x=k then UNIV else A x*

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

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

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

inductive
da :: *env* ⇒ *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 trival break map will map all labels to *UNIV*. In the example, if no break occurs in *c2* the break maps will trivially map to *UNIV* and if a break occurs it will map to *UNIV* too, because *assigns-if False e* = *UNIV*. So in the intersection of the break maps the path *c2* will have no contribution.

| *Loop*: ⟦*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*)⟧

$\Longrightarrow$
$Env\vdash\ B\ \gg\langle Jmp\ jump\rangle\gg\ A$

— In case of a break to label l the corresponding break set is all variables assigned before the break. The assigned variables for normal completion of the *Jmp* is *UNIV*, because the statement will never complete normally. For continue and return the break map is the trivial one. In case of a return we enshure that the result value is assigned.

| *Throw*: $[\![Env\vdash\ B\ \gg\langle e\rangle\gg\ E;\ nrm\ A = UNIV;\ brk\ A = (\lambda\ l.\ UNIV)]\!]$
$\Longrightarrow Env\vdash\ B\ \gg\langle Throw\ e\rangle\gg\ A$

| *Try*: $[\![Env\vdash\ B\ \gg\langle c1\rangle\gg\ C1;$
$Env(\!|lcl := lcl\ Env(VName\ vn\mapsto Class\ C)|\!)\vdash\ (B\cup\{VName\ vn\})\ \gg\langle c2\rangle\gg\ C2;$
$nrm\ A = nrm\ C1\cap nrm\ C2;$
$brk\ A = brk\ C1\ \Rightarrow\cap\ brk\ C2]\!]$
$\Longrightarrow Env\vdash\ B\ \gg\langle Try\ c1\ Catch(C\ vn)\ c2\rangle\gg\ A$

| *Fin*: $[\![Env\vdash\ B\ \gg\langle c1\rangle\gg\ C1;$
$Env\vdash\ B\ \gg\langle c2\rangle\gg\ C2;$
$nrm\ A = nrm\ C1\cup nrm\ C2;$
$brk\ A = ((brk\ C1)\ \Rightarrow\cup_\forall\ (nrm\ C2))\ \Rightarrow\cap\ (brk\ C2)]\!]$
$\Longrightarrow$
$Env\vdash\ B\ \gg\langle c1\ Finally\ c2\rangle\gg\ A$

— The set of assigned variables before execution *c2* are the same as before execution *c1*, because *c1* could throw an exception and so we can't guarantee that any variable will be assigned in *c1*. The *Finally* statement completes normally if both *c1* and *c2* complete normally. If *c1* completes abruptly with a break, then *c2* also will be executed and may terminate normally or with a break. The overall break map then is the intersection of the maps of both paths. If *c2* terminates normally we have to extend all break sets in *brk C1* with *nrm C2* ($\Rightarrow\cup_\forall$). 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*: $Env\vdash\ B\ \gg\langle Init\ C\rangle\gg\ (\!|nrm{=}B,brk{=}\lambda\ l.\ UNIV|\!)$
— 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*: $Env\vdash\ B\ \gg\langle NewC\ C\rangle\gg\ (\!|nrm{=}B,brk{=}\lambda\ l.\ UNIV|\!)$

| *NewA*: $Env\vdash\ B\ \gg\langle e\rangle\gg\ A$
$\Longrightarrow$
$Env\vdash\ B\ \gg\langle New\ T[e]\rangle\gg\ A$

| *Cast*: $Env\vdash\ B\ \gg\langle e\rangle\gg\ A$
$\Longrightarrow$
$Env\vdash\ B\ \gg\langle Cast\ T\ e\rangle\gg\ A$

| *Inst*: *Env*⊢ *B* »⟨*e*⟩» *A*
⟹
Env⊢ *B* »⟨*e InstOf T*⟩» *A*

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

| *UnOp*: *Env*⊢ *B* »⟨*e*⟩» *A*
⟹
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*) ⟧
⟹
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 = (λ *l*. *UNIV*) ⟧
⟹
Env⊢ *B* »⟨*BinOp CondOr e1 e2*⟩» *A*

| *BinOp*: ⟦*Env*⊢ *B* »⟨*e1*⟩» *E1*; *Env*⊢ *nrm E1* »⟨*e2*⟩» *A*;
binop ≠ *CondAnd*; *binop* ≠ *CondOr*⟧
⟹
Env⊢ *B* »⟨*BinOp binop e1 e2*⟩» *A*

| *Super*: *This* ∈ *B*
⟹
Env⊢ *B* »⟨*Super*⟩» (|*nrm*=*B*,*brk*=λ *l*. *UNIV*|)

| *AccLVar*: ⟦*vn* ∈ *B*;
nrm A = *B*; *brk A* = (λ *k*. *UNIV*)⟧
⟹
Env⊢ *B* »⟨*Acc* (*LVar vn*)⟩» *A*

— To properly access a local variable we have to test the definite assignment here. The variable must occur in the set *B*

| *Acc*: ⟦∀ *vn*. *v* ≠ *LVar vn*;
Env⊢ *B* »⟨*v*⟩» *A*⟧
⟹
Env⊢ *B* »⟨*Acc v*⟩» *A*

| *AssLVar*: ⟦*Env*⊢ *B* »⟨*e*⟩» *E*; *nrm A* = *nrm E* ∪ {*vn*}; *brk A* = *brk E*⟧
⟹
Env⊢ *B* »⟨(*LVar vn*) := *e*⟩» *A*

| *Ass*: ⟦∀ *vn*. *v* ≠ *LVar vn*; *Env*⊢ *B* »⟨*v*⟩» *V*; *Env*⊢ *nrm V* »⟨*e*⟩» *A*⟧
⟹
Env⊢ *B* »⟨*v* := *e*⟩» *A*

| *CondBool*: ⟦*Env*⊢(*c* ? *e1* : *e2*)::−(*PrimT Boolean*);
Env⊢ *B* »⟨*c*⟩» *C*;
Env⊢ (*B* ∪ *assigns-if True c*) »⟨*e1*⟩» *E1*;
Env⊢ (*B* ∪ *assigns-if False c*) »⟨*e2*⟩» *E2*;
nrm A = *B* ∪ (*assigns-if True* (*c* ? *e1* : *e2*) ∩
assigns-if False (*c* ? *e1* : *e2*));

$$brk\ A = (\lambda\ l.\ UNIV)\rrbracket$$
$$\Longrightarrow$$
$$Env\vdash B \gg\langle c\ ?\ e1 : e2\rangle\gg A$$

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

| *Call*: $\llbracket Env\vdash B \gg\langle e\rangle\gg E;\ Env\vdash nrm\ E \gg\langle args\rangle\gg A\rrbracket$
$\Longrightarrow$
$Env\vdash B \gg\langle\{accC,statT,mode\}e{\cdot}mn(\{pTs\}args)\rangle\gg A$

— 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*: $\llbracket methd\ (prg\ Env)\ D\ sig = Some\ m;$
$Env\vdash B \gg\langle Body\ (declclass\ m)\ (stmt\ (mbody\ (mthd\ m)))\rangle\gg A$
$\rrbracket$
$\Longrightarrow$
$Env\vdash B \gg\langle Methd\ D\ sig\rangle\gg A$

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

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

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

| *FVar*: $Env\vdash B \gg\langle e\rangle\gg A$
$\Longrightarrow$
$Env\vdash B \gg\langle\{accC,statDeclC,stat\}e..fn\rangle\gg A$

| *AVar*: $\llbracket Env\vdash B \gg\langle e1\rangle\gg E1;\ Env\vdash nrm\ E1 \gg\langle e2\rangle\gg A\rrbracket$
$\Longrightarrow$
$Env\vdash B \gg\langle e1.[e2]\rangle\gg A$

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

| *Cons*: $\llbracket Env\vdash B \gg\langle e::expr\rangle\gg E;\ Env\vdash nrm\ E \gg\langle es\rangle\gg A\rrbracket$
$\Longrightarrow$
$Env\vdash B \gg\langle e\#es\rangle\gg A$

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

inductive-cases *da-elim-cases* [*cases set*]:
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*=λ *l*. *UNIV*⦈) ⟹ *Env*⊢ *B* ≫⟨*Lit v*⟩≫ *A*
⟨*proof*⟩

lemma *da-Super*: ⟦*This* ∈ *B*;*A* = (⦇*nrm*=*B*,*brk*=λ *l*. *UNIV*⦈)⟧ ⟹ *Env*⊢ *B* ≫⟨*Super*⟩≫ *A*
⟨*proof*⟩

lemma *da-Init*: *A* = (⦇*nrm*=*B*,*brk*=λ *l*. *UNIV*⦈) ⟹ *Env*⊢ *B* ≫⟨*Init C*⟩≫ *A*
⟨*proof*⟩

lemma *assignsE-subseteq-assigns-ifs*:
assumes *boolEx*: *E*⊢*e*::−*PrimT Boolean* (**is** *?Boolean e*)
shows *assignsE e* ⊆ *assigns-if True e* ∩ *assigns-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*′ ⟹ (*rmlab l A*) *l*′ = *UNIV*
⟨*proof*⟩

lemma *rmlab-other-label* [*simp*]: *l*≠*l*′⟹ (*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'*:
$[\![\forall\ k.\ A\ k\ \subseteq B\ k;\ x \in \Rightarrow\bigcap A]\!] \Longrightarrow x \in \Rightarrow\bigcap B$
⟨*proof*⟩

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

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

corollary *da-weakenE* [*consumes 2*]:
assumes *da*: $Env\vdash B\ \gg t\gg A$ **and**
B': $B \subseteq B'$ **and**
ex-mono: $\bigwedge A'.\ [\![Env\vdash B'\gg t\gg A';\ nrm\ A \subseteq nrm\ A';$
$\bigwedge l.\ brk\ A\ l \subseteq brk\ A'\ l]\!] \Longrightarrow P$
shows P
⟨*proof*⟩

end

Chapter 13

WellForm

34 Well-formedness of Java programs

theory *WellForm* **imports** *DefiniteAssignment* **begin**

For static checks on expressions and statements, see WellType.thy
improvements over Java Specification 1.0 (cf. 8.4.6.3, 8.4.6.4, 9.4.1):

- a method implementing or overwriting another method may have a result type that widens to the result type of the other method (instead of identical type)
- if a method hides another method (both methods have to be static!) there are no restrictions to the result type since the methods have to be static and there is no dynamic binding of static methods
- if an interface inherits more than one method with the same signature, the methods need not have identical return types

simplifications:

- Object and standard exceptions are assumed to be declared like normal classes

well-formed field declarations

well-formed field declaration (common part for classes and interfaces), cf. 8.3 and (9.3)

constdefs
wf-fdecl :: *prog* ⇒ *pname* ⇒ *fdecl* ⇒ *bool*
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

constdefs
wf-mhead :: *prog* ⇒ *pname* ⇒ *sig* ⇒ *mhead* ⇒ *bool*
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.

constdefs *callee-lcl*:: *qtname* ⇒ *sig* ⇒ *methd* ⇒ *lenv*
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*))

constdefs *parameters* :: *methd* ⇒ *lname set*
parameters m ≡ *set* (*map* (*EName* ∘ *VNam*) (*pars m*))
∪ (*if* (*static m*) *then* {} *else* {*This*})

constdefs
wf-mdecl :: *prog* ⇒ *qtname* ⇒ *mdecl* ⇒ *bool*
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*:
$\neg$ *is-static m* $\Longrightarrow$ *callee-lcl C sig m* (*This*) = *Some* (*Class C*)
⟨*proof*⟩

lemma *wf-mheadI*:
⟦*length* (*parTs sig*) = *length* (*pars m*); $\forall$ *T*$\in$*set* (*parTs sig*). *is-acc-type G P T*;
is-acc-type G P (*resTy m*); *distinct* (*pars m*)⟧ $\Longrightarrow$
wf-mhead G P sig m
⟨*proof*⟩

lemma *wf-mdeclI*: ⟦
wf-mhead G (*pid C*) *sig* (*mhead m*); *unique* (*lcls* (*mbody m*));
($\forall$ *pn*$\in$*set* (*pars m*). *table-of* (*lcls* (*mbody m*)) *pn* = *None*);
$\forall$ (*vn*,*T*)$\in$*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*))::$\surd$;
($\exists$ *A*. (|*prg*=*G*,*cls*=*C*,*lcl*=*callee-lcl C sig m*|) ⊢ *parameters m* ≫⟨*stmt* (*mbody m*)⟩≫ *A*
$\wedge$ *Result* $\in$ *nrm A*)
⟧ $\Longrightarrow$
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*));
$\forall$ *pn*$\in$*set* (*pars m*). *table-of* (*lcls* (*mbody m*)) *pn* = *None*;
$\forall$ (*vn*,*T*)$\in$*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*))::$\surd$;
($\exists$ *A*. (|*prg*=*G*,*cls*=*C*,*lcl*=*callee-lcl C sig m*|)⊢ *parameters m* ≫⟨*stmt* (*mbody m*)⟩≫ *A*
$\wedge$ *Result* $\in$ *nrm A*)
⟧ $\Longrightarrow$ *P*
⟧ $\Longrightarrow$ *P*
⟨*proof*⟩

lemma *wf-mdeclD1*:
wf-mdecl G C (*sig*,*m*) $\Longrightarrow$
wf-mhead G (*pid C*) *sig* (*mhead m*) $\wedge$ *unique* (*lcls* (*mbody m*)) $\wedge$
($\forall$ *pn*$\in$*set* (*pars m*). *table-of* (*lcls* (*mbody m*)) *pn* = *None*) $\wedge$
($\forall$ (*vn*,*T*)$\in$*set* (*lcls* (*mbody m*)). *is-acc-type G* (*pid C*) *T*)
⟨*proof*⟩

lemma *wf-mdecl-bodyD*:
wf-mdecl G C (*sig*,*m*) $\Longrightarrow$
($\exists$ *T*. (|*prg*=*G*,*cls*=*C*,*lcl*=*callee-lcl C sig m*|)⊢*Body C* (*stmt* (*mbody m*))::−*T* $\wedge$
G⊢*T*≼(*resTy m*))
⟨*proof*⟩

lemma *rT-is-acc-type*:
wf-mhead G P sig m $\Longrightarrow$ *is-acc-type G P* (*resTy m*)
⟨*proof*⟩

well-formed interface declarations

A interface declaration is wellformed if:

- the interface hierarchy is wellstructured
- there is no class with the same name
- the method heads are wellformed and not static and have Public access
- the methods are uniquely named
- all superinterfaces are accessible
- the result type of a method overriding a method of Object widens to the result type of the overridden method. Shadowing static methods is forbidden.
- the result type of a method overriding a set of methods defined in the superinterfaces widens to each of the corresponding result types

constdefs
wf-idecl :: *prog* ⇒ *idecl* ⇒ *bool*
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*)) ∧
(*Option.set* ∘ *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*. *Option.set* (*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 $(\lambda$ *new old. accmodi old* $\neq$ *Private)*
entails $(\lambda$*new old. G*$\vdash$*resTy new*$\preceq$*resTy old* $\wedge$
is-static new = is-static old))
⟨*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)*$\hat{}+$
⟨*proof*⟩

well-formed class declarations

A class declaration is wellformed if:

- there is no interface with the same name
- all superinterfaces are accessible and for all methods implementing an interface method the result type widens to the result type of the interface method, the method is not static and offers at least as much access (this actually means that the method has Public access, since all interface methods have public access)
- all field declarations are wellformed and the field names are unique
- all method declarations are wellformed and the method names are unique
- the initialization statement is welltyped
- the classhierarchy is wellstructured
- Unless the class is Object:
  - the superclass is accessible
  - for all methods overriding another method (of a superclass)the result type widens to the result type of the overridden method, the access modifier of the new method provides at least as much access as the overwritten one.
  - for all methods hiding a method (of a superclass) the hidden method must be static and offer at least as much access rights. Remark: In contrast to the Java Language Specification we don't restrict the result types of the method (as in case of overriding), because there seems to be no reason, since there is no dynamic binding of static methods. (cf. 8.4.6.3 vs. 15.12.1). Stricly speaking the restrictions on the access rights aren't necessary to, since the static type and the access rights together determine which method is to be called statically. But if a class gains more then one static method with the same signature due to inheritance, it is confusing when the method selection depends on the access rights only: e.g. Class C declares static public method foo(). Class D is subclass of C and declares static method foo() with default package access. D.foo() ? if this call is in the same package as D then foo of class D is called, otherwise foo of class C.

constdefs *entails*:: *('a,'b) table* $\Rightarrow$ *('b* $\Rightarrow$ *bool)* $\Rightarrow$ *bool*
(- entails - 20)
t entails P $\equiv$ $\forall$ *k.* $\forall$ *x* $\in$ *t k*: *P x*

lemma *entailsD*:
$[\![t\ entails\ P;\ t\ k = Some\ x]\!] \Longrightarrow P\ x$
⟨*proof*⟩

lemma *empty-entails*[*simp*]: *empty entails P*
⟨*proof*⟩

constdefs
wf-cdecl :: *prog* ⇒ *cdecl* ⇒ *bool*
wf-cdecl G ≡
$\lambda(C,c).$
$\neg$*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*=*empty*|)⊢ {} ≫⟨*init c*⟩≫ *A*) ∧
(|*prg*=*G*,*cls*=*C*,*lcl*=*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*=*empty*|)⊢ {} ≫⟨*init c*⟩≫ *A*;
(|*prg*=*G*,*cls*=*C*,*lcl*=*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* ∧

$$
\begin{array}{l}
\qquad\qquad\qquad\qquad\quad accmodi\ old \le accmodi\ new \wedge \\
\qquad\qquad\qquad\qquad\quad \neg is\text{-}static\ old)) \wedge \\
\qquad\qquad\qquad (G,sig \vdash new\ hides\ old \\
\qquad\qquad\qquad\quad \longrightarrow (accmodi\ old \le accmodi\ new \wedge \\
\qquad\qquad\qquad\qquad is\text{-}static\ old)))) \\
\qquad\qquad))]\!] \Longrightarrow P \\
\quad]\!] \Longrightarrow P
\end{array}
$$

⟨*proof*⟩

lemma *wf-cdecl-unique*:
wf-cdecl G (C,c) ⟹ *unique (cfields c)* ∧ *unique (methods c)*
⟨*proof*⟩

lemma *wf-cdecl-fdecl*:
⟦*wf-cdecl G (C,c)*; *f*∈*set (cfields c)*⟧ ⟹ *wf-fdecl G (pid C) f*
⟨*proof*⟩

lemma *wf-cdecl-mdecl*:
⟦*wf-cdecl G (C,c)*; *m*∈*set (methods c)*⟧ ⟹ *wf-mdecl G C m*
⟨*proof*⟩

lemma *wf-cdecl-impD*:
⟦*wf-cdecl G (C,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*))
⟨*proof*⟩

lemma *wf-cdecl-supD*:
⟦*wf-cdecl G (C,c)*; *C* ≠ *Object*⟧ ⟹
is-acc-class G (pid C) (super c) ∧ *(super c,C)* ∉ *(subcls1 G)*^+ ∧
*(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))))
⟨*proof*⟩

lemma *wf-cdecl-overrides-SomeD*:
⟦*wf-cdecl G (C,c)*; *C* ≠ *Object*; *table-of (methods c) sig* = *Some newM*;
G,sig⊢*(C,newM) overrides*$_S$ *old*
⟧ ⟹ *G*⊢*resTy newM*⪯*resTy old* ∧
accmodi old ≤ *accmodi newM* ∧
¬ *is-static old*
⟨*proof*⟩

lemma *wf-cdecl-hides-SomeD*:
⟦*wf-cdecl G* (*C*,*c*); *C* ≠ *Object*; *table-of* (*methods c*) *sig* = *Some newM*;
G,*sig*⊢(*C*,*newM*) *hides old*
⟧ ⟹ *accmodi old* ≤ *access newM* ∧
is-static old
⟨*proof*⟩

lemma *wf-cdecl-wt-init*:
wf-cdecl G (*C*, *c*) ⟹ (|*prg*=*G*,*cls*=*C*,*lcl*=*empty*|)⊢*init c*::√
⟨*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

constdefs
wf-prog :: *prog* ⇒ *bool*
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* = *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*⊢*U*≼*T*; *G*⊢*S*≼*U*; *wf-prog G*⟧ ⟹ *G*⊢*S*≼*T*
⟨*proof*⟩

lemma *Xcpt-subcls-Throwable* [*simp*]:
wf-prog G ⟹ *G*⊢*SXcpt xn*≼$_C$ *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*⊢*D*≼$_C$ *C*;
is-class G D; *wf-prog G*⟧
⟹ *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*⟧ ⟹
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*⟧ ⟹
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*⟧ ⟹
G⊢*C*≼$_C$ (*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*⟧
⟹ *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*⟧
⟹ *is-type G* (*resTy m*)
⟨*proof*⟩

lemma *methd-Object-SomeD*:
⟦*wf-prog G*;*methd G Object sig* = *Some m*⟧
⟹ *declclass m* = *Object*
⟨*proof*⟩

lemma *wf-imethdsD*:
⟦*im* ∈ *imethds G I sig*;*wf-prog G*; *is-iface G I*⟧
⟹ ¬*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* ⊢*new overrides*$_S$ *old* **and** *wf*: *wf-prog G*
shows
G⊢*resTy new*≼*resTy old* ∧
accmodi old ≤ *accmodi new* ∧
¬ *is-static old*
⟨*proof*⟩

lemma *static-to-dynamic-overriding*:
assumes *stat-override*: *G*⊢*new overrides*$_S$ *old* **and** *wf* : *wf-prog G*
shows *G*⊢*new overrides old*
⟨*proof*⟩

lemma *non-Package-instance-method-inheritance*:
assumes *old-inheritable*: $G \vdash$*Method old inheritable-in* (*pid C*) **and**
accmodi-old: *accmodi old* $\neq$ *Package* **and**
instance-method: $\neg$ *is-static old* **and**
subcls: $G \vdash C \prec_C$ *declclass old* **and**
old-declared: $G \vdash$*Method old declared-in* (*declclass old*) **and**
wf: *wf-prog G*
shows $G \vdash$*Method old member-of C* $\vee$
($\exists$ *new*. $G \vdash$ *new overrides*$_S$ *old* $\wedge$ $G \vdash$*Method new member-of C*)
⟨*proof*⟩

lemma *non-Package-instance-method-inheritance-cases* [*consumes 6*, *case-names Inheritance Overriding*]:
assumes *old-inheritable*: $G \vdash$*Method old inheritable-in* (*pid C*) **and**
accmodi-old: *accmodi old* $\neq$ *Package* **and**
instance-method: $\neg$ *is-static old* **and**
subcls: $G \vdash C \prec_C$ *declclass old* **and**
old-declared: $G \vdash$*Method old declared-in* (*declclass old*) **and**
wf: *wf-prog G* **and**
inheritance: $G \vdash$*Method old member-of C* $\Longrightarrow$ *P* **and**
overriding: $\bigwedge$ *new*.
$[\![G \vdash$ *new overrides*$_S$ *old*;$G \vdash$*Method new member-of C*$]\!]$
$\Longrightarrow$ *P*
shows *P*
⟨*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*⟩

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* [*consumes 4*, *case-names NewMethod InheritedMethod*]:
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*
shows
⟦*table-of* (*map* (λ(*s*, *m*). (*s*, *C*, *m*)) (*methods c*)) *sig* = *Some m* $\Longrightarrow$ *P*;
⟦*G*$\vdash$*C inherits* (*method sig m*); *methd G* (*super c*) *sig* = *Some m*⟧ $\Longrightarrow$ *P*
⟧ $\Longrightarrow$ *P*
⟨*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 ≠ Object* **and**
old: *methd G (super c) sig = Some old* **and**
accmodi-old: *Protected ≤ accmodi old* **and**
new: *table-of (methods c) sig = Some new*
shows *is-static new = is-static old*
⟨*proof*⟩

lemma *inheritable-instance-methd*:
assumes *subclseq-C-D*: *G⊢C ≼C D* **and**
is-cls-D: *is-class G D* **and**
wf: *wf-prog G* **and**
old: *methd G D sig = Some old* **and**
accmodi-old: *Protected ≤ accmodi old* **and**
not-static-old: *¬ is-static old*
shows
∃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* [*consumes 6*
, case-names Inheritance Overriding]:
assumes *subclseq-C-D*: *G⊢C ≼C D* **and**
is-cls-D: *is-class G D* **and**
wf: *wf-prog G* **and**
old: *methd G D sig = Some old* **and**
accmodi-old: *Protected ≤ accmodi old* **and**
not-static-old: *¬ is-static old* **and**
inheritance: *methd G C sig = Some old ⟹ P* **and**
overriding: *⋀ new. ⟦methd G C sig = Some new;*
G,sig⊢new overrides_S old⟧ ⟹ P

shows *P*
⟨*proof*⟩

lemma *inheritable-instance-methd-props*:
assumes *subclseq-C-D*: *G⊢C ≼C D* **and**
is-cls-D: *is-class G D* **and**
wf: *wf-prog G* **and**
old: *methd G D sig = Some old* **and**
accmodi-old: *Protected ≤ accmodi old* **and**
not-static-old: *¬ is-static old*
shows
∃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 ∈ A ⟹ P x ⟹ ∃x∈A. P x* ⟨*proof*⟩

lemma *ballE'*: *∀x∈A. P x ⟹ (x ∉ A ⟹ Q) ⟹ (P x ⟹ Q) ⟹ Q* ⟨*proof*⟩

lemma *subint-widen-imethds*:
$[\![G\vdash I\preceq I\ J;\ wf\text{-}prog\ G;\ is\text{-}iface\ G\ J;\ jm \in imethds\ G\ J\ sig]\!] \Longrightarrow$
$\exists\ im \in imethds\ G\ I\ sig.\ is\text{-}static\ im = is\text{-}static\ jm \wedge$
$accmodi\ im = accmodi\ jm \wedge$
$G\vdash resTy\ im\preceq resTy\ jm$
⟨*proof*⟩

lemma *implmt1-methd*:
$\bigwedge sig.\ [\![G\vdash C\leadsto 1I;\ wf\text{-}prog\ G;\ im \in imethds\ G\ I\ sig]\!] \Longrightarrow$
$\exists\, cm \in methd\ G\ C\ sig:\ \neg\ is\text{-}static\ cm \wedge \neg\ is\text{-}static\ im \wedge$
$G\vdash resTy\ cm\preceq resTy\ im \wedge$
$accmodi\ im = Public \wedge accmodi\ cm = Public$
⟨*proof*⟩

lemma *implmt-methd* [*rule-format* (*no-asm*)]:
$[\![wf\text{-}prog\ G;\ G\vdash C\leadsto I]\!] \Longrightarrow is\text{-}iface\ G\ I \longrightarrow$
$(\forall\ im \in imethds\ G\ I\ sig.$
$\exists\ cm \in methd\ G\ C\ sig:\ \neg is\text{-}static\ cm \wedge \neg\ is\text{-}static\ im \wedge$
$G\vdash resTy\ cm\preceq resTy\ im \wedge$
$accmodi\ im = Public \wedge accmodi\ cm = Public)$
⟨*proof*⟩

lemma *mheadsD* [*rule-format* (*no-asm*)]:
$emh \in mheads\ G\ S\ t\ sig \longrightarrow wf\text{-}prog\ G \longrightarrow$
$(\exists\, C\ D\ m.\ t = ClassT\ C \wedge declrefT\ emh = ClassT\ D \wedge$
$accmethd\ G\ S\ C\ sig = Some\ m \wedge$
$(declclass\ m = D) \wedge mhead\ (mthd\ m) = (mhd\ emh)) \vee$
$(\exists\, I.\ t = IfaceT\ I \wedge ((\exists\, im.\ im \in accimethds\ G\ (pid\ S)\ I\ sig \wedge$
$mthd\ im = mhd\ emh) \vee$
$(\exists\, m.\ G\vdash Iface\ I\ accessible\text{-}in\ (pid\ S) \wedge accmethd\ G\ S\ Object\ sig = Some\ m \wedge$
$accmodi\ m \neq Private \wedge$
$declrefT\ emh = ClassT\ Object \wedge mhead\ (mthd\ m) = mhd\ emh))) \vee$
$(\exists\, T\ m.\ t = ArrayT\ T \wedge G\vdash Array\ T\ accessible\text{-}in\ (pid\ S) \wedge$
$accmethd\ G\ S\ Object\ sig = Some\ m \wedge accmodi\ m \neq Private \wedge$
$declrefT\ emh = ClassT\ Object \wedge mhead\ (mthd\ m) = mhd\ emh)$
⟨*proof*⟩

lemma *mheads-cases* [*consumes 2*, *case-names Class-methd*
Iface-methd Iface-Object-methd Array-Object-methd]:
$[\![emh \in mheads\ G\ S\ t\ sig;\ wf\text{-}prog\ G;$
$\bigwedge C\ D\ m.\ [\![t = ClassT\ C; declrefT\ emh = ClassT\ D;\ accmethd\ G\ S\ C\ sig = Some\ m;$
$(declclass\ m = D);\ mhead\ (mthd\ m) = (mhd\ emh)]\!] \Longrightarrow P\ emh;$
$\bigwedge I\ im.\ [\![t = IfaceT\ I;\ im \in accimethds\ G\ (pid\ S)\ I\ sig;\ mthd\ im = mhd\ emh]\!]$
$\Longrightarrow P\ emh;$
$\bigwedge I\ m.\ [\![t = IfaceT\ I;\ G\vdash Iface\ I\ accessible\text{-}in\ (pid\ S);$
$accmethd\ G\ S\ Object\ sig = Some\ m;\ accmodi\ m \neq Private;$

declrefT emh = ClassT Object; mhead (mthd m) = mhd emh⟧ ⟹ *P 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⟧ ⟹ *P emh*
⟧ ⟹ *P 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

consts *valid-lookup-cls*:: *prog* ⇒ *ref-ty* ⇒ *qtname* ⇒ *bool* ⇒ *bool*
(-,- ⊢ - *valid'-lookup'-cls'-for* - [*61*,*61*,*61*,*61*] *60*)
primrec
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* ≼$_C$ (*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*::�doteq*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*⟩

35 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*:
$[\![G{\vdash}m\ of\ C\ accessible\text{-}from\ accC;wf\text{-}prog\ G]\!]$
$\Longrightarrow G{\vdash}m\ in\ C\ dyn\text{-}accessible\text{-}from\ accC$
⟨*proof*⟩

lemma *static-to-dynamic-accessible-from*:
assumes *stat-acc*: $G{\vdash}m\ of\ statC\ accessible\text{-}from\ accC$ **and**
subclseq: $G{\vdash}dynC \preceq_C statC$ **and**
wf: *wf-prog G*
shows $G{\vdash}m\ in\ dynC\ dyn\text{-}accessible\text{-}from\ accC$
⟨*proof*⟩

lemma *static-to-dynamic-accessible-from-static*:
assumes *stat-acc*: $G{\vdash}m\ of\ statC\ accessible\text{-}from\ accC$ **and**
static: *is-static m* **and**
wf: *wf-prog G*
shows $G{\vdash}m\ in\ (declclass\ m)\ dyn\text{-}accessible\text{-}from\ accC$
⟨*proof*⟩

lemma *dynmethd-member-in*:
assumes *m*: *dynmethd G statC dynC sig* = *Some m* **and**
iscls-statC: *is-class G statC* **and**
wf: *wf-prog G*
shows $G{\vdash}Methd\ sig\ m\ member\text{-}in\ dynC$
⟨*proof*⟩

lemma *dynmethd-access-prop*:
assumes *statM*: *methd G statC sig* = *Some statM* **and**
stat-acc: $G{\vdash}Methd\ sig\ statM\ of\ statC\ accessible\text{-}from\ accC$ **and**
dynM: *dynmethd G statC dynC sig* = *Some dynM* **and**
wf: *wf-prog G*
shows $G{\vdash}Methd\ sig\ dynM\ in\ dynC\ dyn\text{-}accessible\text{-}from\ accC$
⟨*proof*⟩

lemma *implmt-methd-access*:
fixes *accC*::*qtname*
assumes *iface-methd*: *imethds G I sig* ≠ {} **and**
implements: $G{\vdash}dynC{\rightsquigarrow}I$ **and**
isif-I: *is-iface G I* **and**
wf: *wf-prog G*
shows ∃ *dynM*. *methd G dynC sig* = *Some dynM* ∧
$G{\vdash}Methd\ sig\ dynM\ in\ dynC\ dyn\text{-}accessible\text{-}from\ accC$
⟨*proof*⟩

corollary *implmt-dynimethd-access*:
fixes *accC*::*qtname*
assumes *iface-methd*: *imethds G I sig* ≠ {} **and**
implements: $G{\vdash}dynC{\rightsquigarrow}I$ **and**
isif-I: *is-iface G I* **and**
wf: *wf-prog G*
shows ∃ *dynM*. *dynimethd G I dynC sig* = *Some dynM* ∧
$G{\vdash}Methd\ sig\ dynM\ in\ dynC\ dyn\text{-}accessible\text{-}from\ accC$
⟨*proof*⟩

lemma *dynlookup-access-prop*:
assumes *emh*: *emh* ∈ *mheads G accC statT sig* **and**
dynM: *dynlookup G statT dynC sig* = *Some dynM* **and**
dynC-prop: *G*,*statT* ⊢ *dynC valid-lookup-cls-for is-static emh* **and**
isT-statT: *isrtype G statT* **and**
wf: *wf-prog G*
shows *G* ⊢*Methd sig dynM in dynC dyn-accessible-from accC*
⟨*proof*⟩

lemma *dynlookup-access*:
assumes *emh*: *emh* ∈ *mheads G accC statT sig* **and**
dynC-prop: *G*,*statT* ⊢ *dynC valid-lookup-cls-for* (*is-static emh*) **and**
isT-statT: *isrtype G statT* **and**
wf: *wf-prog G*
shows ∃ *dynM*. *dynlookup G statT dynC sig* = *Some dynM* ∧
G⊢*Methd sig dynM in dynC dyn-accessible-from accC*
⟨*proof*⟩

lemma *stat-overrides-Package-old*:
assumes *stat-override*: *G* ⊢ *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* ⊢ *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* ⊢ *m in C dyn-accessible-from accC*; *accmodi m* = *Package*;
wf-prog G⟧
⟹ *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* ⊢ *f in C dyn-accessible-from accC* **and**
pack: *accmodi f* = *Package* **and**
field: *is-field f*
shows *pid accC* = *pid* (*declclass f*)
⟨*proof*⟩

dyn-accessible-instance-field-Protected only works for fields since methods can break the package bounds due to overriding

lemma *dyn-accessible-instance-field-Protected*:
assumes *dyn-acc*: *G* ⊢ *f in C dyn-accessible-from accC* **and**
prot: *accmodi f* = *Protected* **and**
field: *is-field f* **and**

instance-field: ¬ *is-static f* **and**
outside: *pid* (*declclass f*) ≠ *pid accC*
shows *G*⊢ *C* ≼$_C$ *accC*
⟨*proof*⟩

lemma *dyn-accessible-static-field-Protected*:
assumes *dyn-acc*: *G* ⊢ *f in C dyn-accessible-from accC* **and**
prot: *accmodi f* = *Protected* **and**
field: *is-field f* **and**
static-field: *is-static f* **and**
outside: *pid* (*declclass f*) ≠ *pid accC*
shows *G*⊢ *accC* ≼$_C$ *declclass f* ∧ *G*⊢*C* ≼$_C$ *declclass f*
⟨*proof*⟩

end

Chapter 14

State

36 State for evaluation of Java expressions and statements

theory *State* **imports** *DeclConcepts* **begin**

design issues:

- all kinds of objects (class instances, arrays, and class objects) are 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)

types *vn* = *fspec* + *int* — variable name
record *obj* =
tag :: *obj-tag* — generalized object
values :: (*vn*, *val*) *table*

translations
fspec <= (*type*) *vname* × *qtname*
vn <= (*type*) *fspec* + *int*
obj <= (*type*) (|*tag*::*obj-tag*, *values*::*vn* ⇒ *val option*|)
obj <= (*type*) (|*tag*::*obj-tag*, *values*::*vn* ⇒ *val option*,...::'*a*|)

constdefs

the-Arr :: *obj option* ⇒ *ty* × *int* × (*vn*, *val*) *table*
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*⟩

constdefs

upd-obj :: *vn* ⇒ *val* ⇒ *obj* ⇒ *obj*
upd-obj n v ≡ λ *obj* . *obj* (|*values*:=(*values obj*)(*n*↦*v*)|)

lemma *upd-obj-def2* [*simp*]:
upd-obj n v obj = *obj* (|*values*:=(*values obj*)(*n*↦*v*)|)
⟨*proof*⟩

constdefs
obj-ty :: *obj* ⇒ *ty*
obj-ty obj ≡ *case tag obj of*

$$\begin{array}{l} \quad CInst\ C \Rightarrow Class\ C \\ \mid Arr\ T\ k \Rightarrow T.[] \end{array}$$

lemma *obj-ty-eq* [*intro*!]: $obj\text{-}ty\ (\!|tag{=}oi,values{=}x|\!) = obj\text{-}ty\ (\!|tag{=}oi,values{=}y|\!)$
⟨*proof*⟩

lemma *obj-ty-eq1* [*intro*!,*dest*]:
$tag\ obj = tag\ obj' \Longrightarrow obj\text{-}ty\ obj = obj\text{-}ty\ obj'$
⟨*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*:
$G{\vdash}obj\text{-}ty\ obj{\preceq}RefT\ t \Longrightarrow (\exists\, C.\ tag\ obj = CInst\ C) \vee (\exists\, T\ k.\ tag\ obj = Arr\ T\ k)$
⟨*proof*⟩

constdefs

$$\begin{array}{l} obj\text{-}class :: obj \Rightarrow qtname \\ obj\text{-}class\ obj \equiv case\ tag\ obj\ of \\ \quad\quad CInst\ C \Rightarrow C \\ \quad \mid Arr\ T\ k \Rightarrow Object \end{array}$$

lemma *obj-class-CInst* [*simp*]: $obj\text{-}class\ (\!|tag{=}CInst\ C,values{=}vs|\!) = C$
⟨*proof*⟩

lemma *obj-class-CInst1* [*simp*,*intro*!,*dest*]:
$tag\ obj = CInst\ C \Longrightarrow obj\text{-}class\ obj = C$
⟨*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*⊢*obj-ty obj*≼ *Class statC* = *G*⊢*obj-class obj* ≼$_C$ *statC*
⟨*proof*⟩

object references

types *oref* = *loc* + *qtname* — generalized object reference
syntax
Heap :: *loc* ⇒ *oref*
Stat :: *qtname* ⇒ *oref*

translations
Heap => *Inl*
Stat => *Inr*
oref <= (*type*) *loc* + *qtname*

constdefs
fields-table::
prog ⇒ *qtname* ⇒ (*fspec* ⇒ *field* ⇒ *bool*) ⇒ (*fspec*, *ty*) *table*
fields-table G C P
≡ *Option.map type* ∘ *table-of* (*filter* (*split 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*⟩

constdefs
in-bounds :: *int* ⇒ *int* ⇒ *bool* ((-/ *in'-bounds* -) [*50*, *51*] *50*)
i in-bounds k ≡ *0* ≤ *i* ∧ *i* < *k*

arr-comps :: *'a* ⇒ *int* ⇒ *int* ⇒ *'a option*
arr-comps T k ≡ λ*i*. *if i in-bounds k then Some T else None*

var-tys :: *prog* ⇒ *obj-tag* ⇒ *oref* ⇒ (*vn*, *ty*) *table*
var-tys G oi r

$\equiv$ *case r of*
Heap a $\Rightarrow$ (*case oi of*
CInst C $\Rightarrow$ *fields-table G C* (λ*n f*. $\neg$*static f*) (+) *empty*
| *Arr T k* $\Rightarrow$ *empty* (+) *arr-comps T k*)
| *Stat C* $\Rightarrow$ *fields-table G C* (λ*fn f*. *declclassf fn* = *C* $\wedge$ *static f*)
(+) *empty*

lemma *var-tys-Some-eq*:
var-tys G oi r n = *Some T*
= (*case r of*
Inl a $\Rightarrow$ (*case oi of*
CInst C $\Rightarrow$ ($\exists$ *nt*. *n* = *Inl nt* $\wedge$ *fields-table G C* (λ*n f*.
$\neg$*static f*) *nt* = *Some T*)
| *Arr t k* $\Rightarrow$ ($\exists$ *i*. *n* = *Inr i* $\wedge$ *i in-bounds k* $\wedge$ *t* = *T*))
| *Inr C* $\Rightarrow$ ($\exists$ *nt*. *n* = *Inl nt* $\wedge$
fields-table G C (λ*fn f*. *declclassf fn* = *C* $\wedge$ *static f*) *nt*
= *Some T*))
⟨*proof*⟩

stores

types *globs* — global variables: heap and static variables
= (*oref* , *obj*) *table*
heap
= (*loc* , *obj*) *table*

translations
globs <= (*type*) (*oref* , *obj*) *table*
heap <= (*type*) (*loc* , *obj*) *table*

datatype *st* =
st globs locals

37 access

constdefs

globs :: *st* $\Rightarrow$ *globs*
globs $\equiv$ *st-case* (λ*g l*. *g*)

locals :: *st* $\Rightarrow$ *locals*
locals $\equiv$ *st-case* (λ*g l*. *l*)

heap :: *st* $\Rightarrow$ *heap*
heap s $\equiv$ *globs s* $\circ$ *Heap*

lemma *globs-def2* [*simp*]: *globs* (*st g l*) = *g*
⟨*proof*⟩

lemma *locals-def2* [*simp*]: *locals* (*st g l*) = *l*
⟨*proof*⟩

lemma *heap-def2* [*simp*]: *heap s a=globs s* (*Heap a*)
⟨*proof*⟩

syntax
val-this :: *st* ⇒ *val*
lookup-obj :: *st* ⇒ *val* ⇒ *obj*

translations
val-this s == *CONST the* (*locals s This*)
lookup-obj s a′ == *CONST the* (*heap s* (*the-Addr a′*))

38 memory allocation

constdefs
new-Addr :: *heap* ⇒ *loc option*
new-Addr h ≡ *if* (∀ *a*. *h a* ≠ *None*) *then None else Some* (*SOME a*. *h a* = *None*)

lemma *new-AddrD*: *new-Addr h* = *Some a* ⟹ *h a* = *None*
⟨*proof*⟩

lemma *new-AddrD2*: *new-Addr h* = *Some a* ⟹ ∀ *b*. *h b* ≠ *None* ⟶ *b* ≠ *a*
⟨*proof*⟩

lemma *new-Addr-SomeI*: *h a* = *None* ⟹ ∃ *b*. *new-Addr h* = *Some b* ∧ *h b* = *None*
⟨*proof*⟩

39 initialization

syntax

init-vals :: (*′a*, *ty*) *table* ⇒ (*′a*, *val*) *table*

translations
init-vals vs == *CONST Option.map default-val* ∘ *vs*

lemma *init-arr-comps-base* [*simp*]: *init-vals* (*arr-comps T 0*) = *empty*
⟨*proof*⟩

lemma *init-arr-comps-step* [*simp*]:
0 < *j* ⟹ *init-vals* (*arr-comps T j*) =
init-vals (*arr-comps T* (*j* − *1*))(*j* − *1*↦*default-val T*)
⟨*proof*⟩

40 update

constdefs
gupd :: *oref* ⇒ *obj* ⇒ *st* ⇒ *st* (*gupd′*(-↦-′)[*10*,*10*]*1000*)
gupd r obj ≡ *st-case* (λ*g l*. *st* (*g*(*r*↦*obj*)) *l*)

lupd :: *lname* ⇒ *val* ⇒ *st* ⇒ *st* (*lupd′*(-↦-′)[*10*,*10*]*1000*)
lupd vn v ≡ *st-case* (λ*g l*. *st g* (*l*(*vn*↦*v*)))

upd-gobj :: *oref* ⇒ *vn* ⇒ *val* ⇒ *st* ⇒ *st*

upd-gobj r n v ≡ *st-case* (λ*g l*. *st* (*chg-map* (*upd-obj n v*) *r g*) *l*)

set-locals :: *locals* ⇒ *st* ⇒ *st*
set-locals l ≡ *st-case* (λ*g l'*. *st g l*)

init-obj :: *prog* ⇒ *obj-tag* ⇒ *oref* ⇒ *st* ⇒ *st*
init-obj G oi r ≡ *gupd*(*r*↦(|*tag*=*oi*, *values*=*init-vals* (*var-tys G oi r*)|))

syntax
init-class-obj :: *prog* ⇒ *qtname* ⇒ *st* ⇒ *st*

translations
init-class-obj G C == *init-obj G CONST undefined* (*Inr C*)

lemma *gupd-def2* [*simp*]: *gupd*(*r*↦*obj*) (*st g l*) = *st* (*g*(*r*↦*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

consts

the-Xcpt :: *abrupt* ⇒ *xcpt*
the-Jump :: *abrupt* => *jump*
the-Loc :: *xcpt* ⇒ *loc*
the-Std :: *xcpt* ⇒ *xname*

primrec *the-Xcpt* (*Xcpt x*) = *x*
primrec *the-Jump* (*Jump j*) = *j*
primrec *the-Loc* (*Loc a*) = *a*
primrec *the-Std* (*Std x*) = *x*

constdefs
abrupt-if :: *bool* ⇒ *abopt* ⇒ *abopt* ⇒ *abopt*
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*⟩

syntax

raise-if :: *bool* ⇒ *xname* ⇒ *abopt* ⇒ *abopt*
np :: *val* ⇒ *abopt* ⇒ *abopt*
check-neg:: *val* ⇒ *abopt* ⇒ *abopt*
error-if :: *bool* ⇒ *error* ⇒ *abopt* ⇒ *abopt*

translations

raise-if c xn == abrupt-if c (Some (Xcpt (Std xn)))
np v == raise-if (v = Null) NullPointer
check-neg i′ == raise-if (the-Intg i′<0) NegArrSize
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*⟩

constdefs
absorb :: jump ⇒ abopt ⇒ abopt
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

types
state = *abopt* × *st* — state including abruption information

syntax
Norm :: *st* ⇒ *state*
abrupt :: *state* ⇒ *abopt*
store :: *state* ⇒ *st*

translations

Norm s == (*None*,*s*)
abrupt => *fst*
store => *snd*
abopt <= (*type*) *State.abrupt option*
abopt <= (*type*) *abrupt option*
state <= (*type*) *abopt* × *State.st*
state <= (*type*) *abopt* × *st*

lemma *single-stateE*: ∀ *Z*. *Z* = (*s*::*state*) ⟹ *False*
⟨*proof*⟩

lemma *state-not-single*: *All* (*op* = (*x*::*state*)) ⟹ *R*
⟨*proof*⟩

constdefs

normal :: *state* ⇒ *bool*
normal ≡ λ*s*. *abrupt s* = *None*

lemma *normal-def2* [*simp*]: *normal s* = (*abrupt s* = *None*)
⟨*proof*⟩

constdefs
heap-free :: *nat* ⇒ *state* ⇒ *bool*
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*⟩

41 update

constdefs

abupd :: (*abopt* ⇒ *abopt*) ⇒ *state* ⇒ *state*
abupd f ≡ *prod-fun f id*

supd :: (*st* ⇒ *st*) ⇒ *state* ⇒ *state*
supd ≡ *prod-fun 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*]:
⋀ *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*⟩

syntax

set-lvars :: *locals* ⇒ *state* ⇒ *state*
restore-lvars :: *state* ⇒ *state* ⇒ *state*

translations

set-lvars l == *supd* (*set-locals l*)
restore-lvars s′ s == *set-lvars* (*locals* (*store s′*)) *s*

lemma *set-set-lvars* [*simp*]: ⋀ *s*. *set-lvars l* (*set-lvars l′ s*) = *set-lvars l s*
⟨*proof*⟩

lemma *set-lvars-id* [*simp*]: ⋀ *s*. *set-lvars* (*locals* (*store s*)) *s* = *s*
⟨*proof*⟩

initialisation test

constdefs

inited :: *qtname* ⇒ *globs* ⇒ *bool*
inited C g ≡ *g* (*Stat C*) ≠ *None*

initd :: *qtname* ⇒ *state* ⇒ *bool*
initd C ≡ *inited C* ∘ *globs* ∘ *store*

lemma *not-inited-empty* [*simp*]: ¬*inited C 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

constdefs *error-free*:: *state* ⇒ *bool*
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 $\Longrightarrow$ *error-free* (*abupd* (*absorb j*) *s*)
⟨*proof*⟩

lemma *error-free-absorb* [*simp*,*intro*]:
error-free (*a*,*s*) $\Longrightarrow$ *error-free* (*absorb j a*, *s*)
⟨*proof*⟩

lemma *error-free-abrupt-if* [*simp*,*intro*]:
⟦*error-free s*; ¬ (∃ *err*. *x*=*Error err*)⟧
$\Longrightarrow$ *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*)⟧
$\Longrightarrow$ *error-free* (*abrupt-if p* (*Some x*) *a*, *s*)
⟨*proof*⟩

lemma *error-free-abrupt-if-Xcpt* [*simp*,*intro*]:
error-free s
$\Longrightarrow$ *error-free* (*abupd* (*abrupt-if p* (*Some* (*Xcpt x*))) *s*)
⟨*proof*⟩

lemma *error-free-abrupt-if-Xcpt1* [*simp*,*intro*]:
error-free (*a*,*s*)
$\Longrightarrow$ *error-free* (*abrupt-if p* (*Some* (*Xcpt x*)) *a*, *s*)
⟨*proof*⟩

lemma *error-free-abrupt-if-Jump* [*simp*,*intro*]:
error-free s
$\Longrightarrow$ *error-free* (*abupd* (*abrupt-if p* (*Some* (*Jump j*))) *s*)
⟨*proof*⟩

lemma *error-free-abrupt-if-Jump1* [*simp*,*intro*]:
error-free (*a*,*s*)
$\Longrightarrow$ *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

42 Operational evaluation (big-step) semantics of Java expressions and statements

theory *Eval* **imports** *State DeclConcepts* **begin**

improvements over Java Specification 1.0:

- dynamic method lookup does not need to consider the return type (cf.15.11.4.4)
- throw raises a 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 occurence.
- 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

types *vvar* = *val* × (*val* ⇒ *state* ⇒ *state*)
vals = (*val*, *vvar*, *val list*) *sum3*
translations
vvar <= (*type*) *val* × (*val* ⇒ *state* ⇒ *state*)
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.

syntax (*xsymbols*)
dummy-res :: *vals* (◇)
translations
◇ == *In1 Unit*

syntax
val-inj-vals:: *expr* ⇒ *term* (⌊-⌋$_e$ *1000*)
var-inj-vals:: *var* ⇒ *term* (⌊-⌋$_v$ *1000*)
lst-inj-vals:: *expr list* ⇒ *term* (⌊-⌋$_l$ *1000*)

translations

$\lfloor e \rfloor_e \rightharpoonup In1\ e$
$\lfloor v \rfloor_v \rightharpoonup In2\ v$
$\lfloor es \rfloor_l \rightharpoonup In3\ es$

constdefs
undefined3 :: $('al + 'ar, 'b, 'c)\ sum3 \Rightarrow vals$
undefined3 $\equiv$ *sum3-case* $(In1 \circ sum\text{-}case\ (\lambda x.\ undefined)\ (\lambda x.\ Unit))$
$(\lambda x.\ In2\ undefined)\ (\lambda x.\ In3\ undefined)$

lemma [*simp*]: *undefined3* $(In1l\ x) = In1\ undefined$
⟨*proof*⟩

lemma [*simp*]: *undefined3* $(In1r\ x) = \diamondsuit$
⟨*proof*⟩

lemma [*simp*]: *undefined3* $(In2\ \ x) = In2\ undefined$
⟨*proof*⟩

lemma [*simp*]: *undefined3* $(In3\ \ x) = In3\ undefined$
⟨*proof*⟩

exception throwing and catching

constdefs
throw :: *val* $\Rightarrow$ *abopt* $\Rightarrow$ *abopt*
throw $a'\ x \equiv$ *abrupt-if True* $(Some\ (Xcpt\ (Loc\ (the\text{-}Addr\ a'))))\ (np\ a'\ x)$

lemma *throw-def2*:
throw $a'\ x =$ *abrupt-if True* $(Some\ (Xcpt\ (Loc\ (the\text{-}Addr\ a'))))\ (np\ a'\ x)$
⟨*proof*⟩

constdefs
fits :: *prog* $\Rightarrow$ *st* $\Rightarrow$ *val* $\Rightarrow$ *ty* $\Rightarrow$ *bool* (-,-⊢- *fits* -[61,61,61,61]60)
$G,s \vdash a'\ fits\ T \equiv (\exists rt.\ T = RefT\ rt) \longrightarrow a' = Null \vee G \vdash obj\text{-}ty(lookup\text{-}obj\ s\ a') \preceq T$

lemma *fits-Null* [*simp*]: $G,s \vdash Null\ fits\ T$
⟨*proof*⟩

lemma *fits-Addr-RefT* [*simp*]:
$G,s \vdash Addr\ a\ fits\ RefT\ t = G \vdash obj\text{-}ty\ (the\ (heap\ s\ a)) \preceq RefT\ t$
⟨*proof*⟩

lemma *fitsD*: $\bigwedge X.\ G,s \vdash a'\ fits\ T \Longrightarrow (\exists pt.\ T = PrimT\ pt) \vee$
$(\exists t.\ T = RefT\ t) \wedge a' = Null \vee$
$(\exists t.\ T = RefT\ t) \wedge a' \neq Null \wedge\ \ G \vdash obj\text{-}ty\ (lookup\text{-}obj\ s\ a') \preceq T$
⟨*proof*⟩

constdefs
catch ::*prog* $\Rightarrow$ *state* $\Rightarrow$ *qtname* $\Rightarrow$ *bool* (-,-⊢*catch* -[61,61,61]60)
$G,s \vdash catch\ C \equiv \exists xc.\ abrupt\ s = Some\ (Xcpt\ xc) \wedge$

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*⟩

constdefs
new-xcpt-var :: *vname* ⇒ *state* ⇒ *state*
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

constdefs

assign :: ('*a* ⇒ *state* ⇒ *state*) ⇒ '*a* ⇒ *state* ⇒ *state*
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*]: $\neg$ *normal s* $\Longrightarrow$ *assign f v s* = *s*
⟨*proof*⟩

lemma *assign-supd* [*simp*]:
assign (λ*v*. *supd* (*f v*)) *v* (*x*,*s*)
= (*x*, *if x* = *None then f v s else s*)
⟨*proof*⟩

lemma *assign-raise-if* [*simp*]:
assign (λ*v* (*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* ∧ ¬*b s v then f v s else s*)
⟨*proof*⟩

constdefs

init-comp-ty :: *ty* ⇒ *stmt*
init-comp-ty T ≡ *if* (∃ *C*. *T* = *Class C*) *then Init* (*the-Class T*) *else Skip*

lemma *init-comp-ty-PrimT* [*simp*]: *init-comp-ty* (*PrimT pt*) = *Skip*
⟨*proof*⟩

constdefs

invocation-class :: *inv-mode* ⇒ *st* ⇒ *val* ⇒ *ref-ty* ⇒ *qtname*
invocation-class m s a′ statT
≡ (*case m of*
Static ⇒ *if* (∃ *statC*. *statT* = *ClassT statC*)
then the-Class (*RefT statT*)
else Object
| *SuperM* ⇒ *the-Class* (*RefT statT*)
| *IntVir* ⇒ *obj-class* (*lookup-obj s a′*))

invocation-declclass::*prog* ⇒ *inv-mode* ⇒ *st* ⇒ *val* ⇒ *ref-ty* ⇒ *sig* ⇒ *qtname*
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* (∃ *statC*. *statT* = *ClassT statC*)
then the-Class (*RefT statT*)

else Object)
⟨*proof*⟩

constdefs
init-lvars :: *prog* ⇒ *qtname* ⇒ *sig* ⇒ *inv-mode* ⇒ *val* ⇒ *val list* ⇒
state ⇒ *state*
init-lvars G C sig mode a′ pvs
≡ λ (*x*,*s*).
let m = *mthd* (*the* (*methd G C sig*));
l = λ *k*.
(*case k of*
EName e
⇒ (*case e of*
VNam v ⇒ (*empty* ((*pars m*)[↦]*pvs*)) *v*
| *Res* ⇒ *None*)
| *This*
⇒ (*if mode*=*Static then None else Some a′*))
in set-lvars l (*if mode* = *Static then x else np a′ x*,*s*)

lemma *init-lvars-def2*: — better suited for simplification
init-lvars G C sig mode a′ pvs (*x*,*s*) =
set-lvars
(λ *k*.
(*case k of*
EName e
⇒ (*case e of*
VNam v
⇒ (*empty* ((*pars* (*mthd* (*the* (*methd G C sig*))))[↦]*pvs*)) *v*
| *Res* ⇒ *None*)
| *This*
⇒ (*if mode*=*Static then None else Some a′*)))
(*if mode* = *Static then x else np a′ x*,*s*)
⟨*proof*⟩

constdefs
body :: *prog* ⇒ *qtname* ⇒ *sig* ⇒ *expr*
body G C sig ≡ *let m* = *the* (*methd G C sig*)
in Body (*declclass m*) (*stmt* (*mbody* (*mthd m*)))

lemma *body-def2*: — better suited for simplification
body G C sig = *Body* (*declclass* (*the* (*methd G C sig*)))
(*stmt* (*mbody* (*mthd* (*the* (*methd G C sig*)))))
⟨*proof*⟩

variables

constdefs

lvar :: *lname* ⇒ *st* ⇒ *vvar*
lvar vn s ≡ (*the* (*locals s vn*), λ*v*. *supd* (*lupd*(*vn*↦*v*)))

fvar :: *qtname* ⇒ *bool* ⇒ *vname* ⇒ *val* ⇒ *state* ⇒ *vvar* × *state*
fvar C stat fn a′ s
≡ *let* (*oref*,*xf*) = *if stat then* (*Stat C*,*id*)
else (*Heap* (*the-Addr a′*),*np a′*);

```
                n = Inl (fn,C);
                f = (λv. supd (upd-gobj oref n v))
      in ((the (values (the (globs (store s) oref)) n),f),abupd xf s)

  avar :: prog ⇒ val ⇒ val ⇒ state ⇒ vvar × state
 avar G i' a' s
    ≡ let   oref = Heap (the-Addr a');
              i = the-Intg i';
              n = Inr i;
        (T,k,cs) = the-Arr (globs (store s) oref);
              f = (λv (x,s). (raise-if (¬G,s⊢v fits T)
                                       ArrStore x
                               ,upd-gobj oref n v s))
      in ((the (cs n),f)
         ,abupd (raise-if (¬i in-bounds k) 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*⟩

constdefs

```
check-field-access::
prog ⇒ qtname ⇒ qtname ⇒ vname ⇒ bool ⇒ val ⇒ state ⇒ state
check-field-access G accC statDeclC fn stat a' s
 ≡ let oref = if stat then Stat statDeclC
                      else Heap (the-Addr a');
       dynC = case oref of
                Heap a ⇒ obj-class (the (globs (store s) oref))
              | Stat C ⇒ C;
       f    = (the (table-of (DeclConcepts.fields G dynC) (fn,statDeclC)))
   in abupd
       (error-if (¬ G⊢Field fn (statDeclC,f) in dynC dyn-accessible-from accC)
                 AccessViolation)
       s
```

constdefs
check-method-access::
prog ⇒ *qtname* ⇒ *ref-ty* ⇒ *inv-mode* ⇒ *sig* ⇒ *val* ⇒ *state* ⇒ *state*
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*:
$G\vdash(Some\ xc,s)\ -t\succ\!\!\rightarrow\ (undefined3\ t,\ (Some\ xc,\ s))$

— execution of statements

— cf. 14.5
| *Skip*: $G\vdash Norm\ s\ -Skip\rightarrow\ Norm\ s$

— cf. 14.7
| *Expr*: $[\![G\vdash Norm\ s0\ -e-\succ v\rightarrow\ s1]\!] \Longrightarrow$
$G\vdash Norm\ s0\ -Expr\ e\rightarrow\ s1$

| *Lab*: $[\![G\vdash Norm\ s0\ -c\ \rightarrow\ s1]\!] \Longrightarrow$
$G\vdash Norm\ s0\ -l\bullet\ c\rightarrow\ abupd\ (absorb\ l)\ s1$
— cf. 14.2
| *Comp*: $[\![G\vdash Norm\ s0\ -c1\ \rightarrow\ s1;$
$G\vdash\ \ s1\ -c2\ \rightarrow\ s2]\!] \Longrightarrow$
$G\vdash Norm\ s0\ -c1;;\ c2\rightarrow\ s2$

— cf. 14.8.2
| *If*: $[\![G\vdash Norm\ s0\ -e-\succ b\rightarrow\ s1;$
$G\vdash\ \ s1-(if\ the\text{-}Bool\ b\ then\ c1\ else\ c2)\rightarrow\ s2]\!] \Longrightarrow$
$G\vdash Norm\ s0\ -If(e)\ c1\ Else\ c2\ \rightarrow\ s2$

— cf. 14.10, 14.10.1

— A continue jump from the while body c is handled by this rule. If a continue jump with the proper label was invoked inside c this label (Cont l) is deleted out of the abrupt component of the state before the iterative evaluation of the while statement. A break jump is handled by the Lab Statement *Lab l* (*while*...).
| *Loop*: $[\![G\vdash Norm\ s0\ -e-\succ b\rightarrow\ s1;$
$if\ the\text{-}Bool\ b$
$then\ (G\vdash s1\ -c\rightarrow\ s2\ \wedge$
$G\vdash(abupd\ (absorb\ (Cont\ l))\ s2)\ -l\bullet\ While(e)\ c\rightarrow\ s3)$
$else\ s3\ =\ s1]\!] \Longrightarrow$
$G\vdash Norm\ s0\ -l\bullet\ While(e)\ c\rightarrow\ s3$

| *Jmp*: $G\vdash Norm\ s\ -Jmp\ j\rightarrow\ (Some\ (Jump\ j),\ s)$

— cf. 14.16
| *Throw*: $[\![G\vdash Norm\ s0\ -e-\succ a'\rightarrow\ s1]\!] \Longrightarrow$
$G\vdash Norm\ s0\ -Throw\ e\rightarrow\ abupd\ (throw\ a')\ s1$

— cf. 14.18.1
| *Try*: $[\![G\vdash Norm\ s0\ -c1\rightarrow\ s1;\ G\vdash s1\ -sxalloc\rightarrow\ s2;$
$if\ G,s2\vdash catch\ C\ then\ G\vdash new\text{-}xcpt\text{-}var\ vn\ s2\ -c2\rightarrow\ s3\ else\ s3\ =\ s2]\!] \Longrightarrow$
$G\vdash Norm\ s0\ -Try\ c1\ Catch(C\ vn)\ c2\rightarrow\ s3$

— cf. 14.18.2
| *Fin*: $[\![G\vdash Norm\ s0\ -c1\rightarrow\ (x1,s1);$
$G\vdash Norm\ s1\ -c2\rightarrow\ s2;$
$s3=(if\ (\exists\ err.\ x1=Some\ (Error\ err))$
$then\ (x1,s1)$
$else\ abupd\ (abrupt\text{-}if\ (x1\neq None)\ x1)\ s2)\ ]\!]$
$\Longrightarrow$
$G\vdash Norm\ s0\ -c1\ Finally\ c2\rightarrow\ s3$
— 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 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;*

$$G\vdash \quad s1\ -e-\succ v\ \rightarrow s2\rrbracket \Longrightarrow$$
$$G\vdash Norm\ s0\ -va:=e-\succ v\rightarrow\ assign\ f\ v\ s2$$

— cf. 15.24

| *Cond*: $\llbracket G\vdash Norm\ s0\ -e0-\succ b\rightarrow\ s1;$
$G\vdash \quad s1\ -(if\ the\text{-}Bool\ b\ then\ e1\ else\ e2)-\succ v\rightarrow\ s2\rrbracket \Longrightarrow$
$G\vdash Norm\ s0\ -e0\ ?\ e1\ :\ e2-\succ v\rightarrow\ s2$

— The interplay of *Call*, *Methd* and *Body*: Method invocation is split up into these three rules:

Call Calculates the target address and evaluates the arguments of the method, and then performs dynamic or static lookup of the method, corresponding to the call mode. Then the *Methd* rule is evaluated on the calculated declaration class of the method invocation.

Methd A syntactic bridge for the folded method body. It is used by the axiomatic semantics to add the proper hypothesis for recursive calls of the method.

Body An extra syntactic entity for the unfolded method body was introduced to properly trigger class initialisation. Without class initialisation we could just evaluate the body statement.

— cf. 15.11.4.1, 15.11.4.2, 15.11.4.4, 15.11.4.5

| *Call*:
$\llbracket G\vdash Norm\ s0\ -e-\succ a'\rightarrow\ s1;\ G\vdash s1\ -args\doteq\succ vs\rightarrow\ s2;$
$D = invocation\text{-}declclass\ G\ mode\ (store\ s2)\ a'\ statT\ (\!|name=mn,parTs=pTs|\!);$
$s3=init\text{-}lvars\ G\ D\ (\!|name=mn,parTs=pTs|\!)\ mode\ a'\ vs\ s2;$
$s3' = check\text{-}method\text{-}access\ G\ accC\ statT\ mode\ (\!|name=mn,parTs=pTs|\!)\ a'\ s3;$
$G\vdash s3'\ -Methd\ D\ (\!|name=mn,parTs=pTs|\!)-\succ v\rightarrow\ s4\rrbracket$
$\Longrightarrow$
$G\vdash Norm\ s0\ -\{accC,statT,mode\}e\cdot mn(\{pTs\}args)-\succ v\rightarrow\ (restore\text{-}lvars\ s2\ s4)$

— 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*: $\llbracket G\vdash Norm\ s0\ -body\ G\ D\ sig-\succ v\rightarrow\ s1\rrbracket \Longrightarrow$
$G\vdash Norm\ s0\ -Methd\ D\ sig-\succ v\rightarrow\ s1$

| *Body*: $\llbracket G\vdash Norm\ s0\ -Init\ D\rightarrow\ s1;\ G\vdash s1\ -c\rightarrow\ s2;$
$s3 = (if\ (\exists\ l.\ abrupt\ s2 = Some\ (Jump\ (Break\ l))\ \vee$
$abrupt\ s2 = Some\ (Jump\ (Cont\ l)))$
$then\ abupd\ (\lambda\ x.\ Some\ (Error\ CrossMethodJump))\ s2$
$else\ s2)\rrbracket \Longrightarrow$
$G\vdash Norm\ s0\ -Body\ D\ c-\succ the\ (locals\ (store\ s2)\ Result)$
$\rightarrow abupd\ (absorb\ Ret)\ s3$

— 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*: $\llbracket G\vdash Norm\ s0\ -Init\ statDeclC\rightarrow\ s1;\ G\vdash s1\ -e-\succ a\rightarrow\ s2;$
$(v,s2') = fvar\ statDeclC\ stat\ fn\ a\ s2;$
$s3 = check\text{-}field\text{-}access\ G\ accC\ statDeclC\ fn\ stat\ a\ s2'\rrbracket \Longrightarrow$
$G\vdash Norm\ s0\ -\{accC,statDeclC,stat\}e..fn=\succ v\rightarrow\ s3$

— 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*⊢ *Norm s0* −*e1*−≻*a*→ *s1*; *G*⊢*s1* −*e2*−≻*i*→ *s2*;
(*v*,*s2*′) = *avar G i a s2*⟧ ⟹
G⊢*Norm s0* −*e1*.[*e2*]=≻*v*→ *s2*′

— evaluation of expression lists

— cf. 15.11.4.2
| *Nil*:
G⊢*Norm s0* −[]≐≻[]→ *Norm s0*

— cf. 15.6.4
| *Cons*: ⟦*G*⊢*Norm s0* −*e* −≻ *v* → *s1*;
G⊢ *s1* −*es*≐≻*vs*→ *s2*⟧ ⟹
G⊢*Norm s0* −*e*#*es*≐≻*v*#*vs*→ *s2*

⟨*ML*⟩

lemmas *eval-induct* = *eval-induct-* [*split-format* **and and and and and and and and and and and and and and** *s1* **and and** *s2* **and and and and and and and** *s2* **and and** *s2*]

declare *split-if* [*split del*] *split-if-asm* [*split del*]
option.split [*split del*] *option.split-asm* [*split del*]
inductive-cases *halloc-elim-cases*:
G⊢(*Some xc*,*s*) −*halloc oi*≻*a*→ *s*′
G⊢(*Norm* *s*) −*halloc oi*≻*a*→ *s*′

inductive-cases *sxalloc-elim-cases*:
G⊢ *Norm* *s* −*sxalloc*→ *s*′
G⊢(*Some* (*Jump j*),*s*) −*sxalloc*→ *s*′
G⊢(*Some* (*Error e*),*s*) −*sxalloc*→ *s*′
G⊢(*Some* (*Xcpt* (*Loc a*)),*s*) −*sxalloc*→ *s*′
G⊢(*Some* (*Xcpt* (*Std xn*)),*s*) −*sxalloc*→ *s*′
inductive-cases *sxalloc-cases*: *G*⊢*s* −*sxalloc*→ *s*′

lemma *sxalloc-elim-cases2*: ⟦*G*⊢*s* −*sxalloc*→ *s*′;
⋀*s*. ⟦*s*′ = *Norm s*⟧ ⟹ *P*;
⋀*j s*. ⟦*s*′ = (*Some* (*Jump j*),*s*)⟧ ⟹ *P*;
⋀*e s*. ⟦*s*′ = (*Some* (*Error e*),*s*)⟧ ⟹ *P*;
⋀*a s*. ⟦*s*′ = (*Some* (*Xcpt* (*Loc a*)),*s*)⟧ ⟹ *P*
⟧ ⟹ *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*⊢*s* −*t*≻→ (*v*, *s*′)

inductive-cases *eval-elim-cases* [*cases set*]:
G⊢(*Some xc*,*s*) −*t* ≻→ (*v*, *s*′)
G⊢*Norm s* −*In1r Skip* ≻→ (*x*, *s*′)

$G\vdash Norm\ s\ -In1r\ (Jmp\ j)\ \succ\!\!\rightarrow\ (x,\ s')$
$G\vdash Norm\ s\ -In1r\ (l\cdot\ c)\ \succ\!\!\rightarrow\ (x,\ s')$
$G\vdash Norm\ s\ -In3\ ([])\ \succ\!\!\rightarrow\ (v,\ s')$
$G\vdash Norm\ s\ -In3\ (e\#es)\ \succ\!\!\rightarrow\ (v,\ s')$
$G\vdash Norm\ s\ -In1l\ (Lit\ w)\ \succ\!\!\rightarrow\ (v,\ s')$
$G\vdash Norm\ s\ -In1l\ (UnOp\ unop\ e)\ \succ\!\!\rightarrow\ (v,\ s')$
$G\vdash Norm\ s\ -In1l\ (BinOp\ binop\ e1\ e2)\ \succ\!\!\rightarrow\ (v,\ s')$
$G\vdash Norm\ s\ -In2\ (LVar\ vn)\ \succ\!\!\rightarrow\ (v,\ s')$
$G\vdash Norm\ s\ -In1l\ (Cast\ T\ e)\ \succ\!\!\rightarrow\ (v,\ s')$
$G\vdash Norm\ s\ -In1l\ (e\ InstOf\ T)\ \succ\!\!\rightarrow\ (v,\ s')$
$G\vdash Norm\ s\ -In1l\ (Super)\ \succ\!\!\rightarrow\ (v,\ s')$
$G\vdash Norm\ s\ -In1l\ (Acc\ va)\ \succ\!\!\rightarrow\ (v,\ s')$
$G\vdash Norm\ s\ -In1r\ (Expr\ e)\ \succ\!\!\rightarrow\ (x,\ s')$
$G\vdash Norm\ s\ -In1r\ (c1;;\ c2)\ \succ\!\!\rightarrow\ (x,\ s')$
$G\vdash Norm\ s\ -In1l\ (Methd\ C\ sig)\ \succ\!\!\rightarrow\ (x,\ s')$
$G\vdash Norm\ s\ -In1l\ (Body\ D\ c)\ \succ\!\!\rightarrow\ (x,\ s')$
$G\vdash Norm\ s\ -In1l\ (e0\ ?\ e1\ :\ e2)\ \succ\!\!\rightarrow\ (v,\ s')$
$G\vdash Norm\ s\ -In1r\ (If(e)\ c1\ Else\ c2)\ \succ\!\!\rightarrow\ (x,\ s')$
$G\vdash Norm\ s\ -In1r\ (l\cdot\ While(e)\ c)\ \succ\!\!\rightarrow\ (x,\ s')$
$G\vdash Norm\ s\ -In1r\ (c1\ Finally\ c2)\ \succ\!\!\rightarrow\ (x,\ s')$
$G\vdash Norm\ s\ -In1r\ (Throw\ e)\ \succ\!\!\rightarrow\ (x,\ s')$
$G\vdash Norm\ s\ -In1l\ (NewC\ C)\ \succ\!\!\rightarrow\ (v,\ s')$
$G\vdash Norm\ s\ -In1l\ (New\ T[e])\ \succ\!\!\rightarrow\ (v,\ s')$
$G\vdash Norm\ s\ -In1l\ (Ass\ va\ e)\ \succ\!\!\rightarrow\ (v,\ s')$
$G\vdash Norm\ s\ -In1r\ (Try\ c1\ Catch(tn\ vn)\ c2)\ \succ\!\!\rightarrow\ (x,\ s')$
$G\vdash Norm\ s\ -In2\ (\{accC,statDeclC,stat\}e..fn)\ \succ\!\!\rightarrow\ (v,\ s')$
$G\vdash Norm\ s\ -In2\ (e1.[e2])\ \succ\!\!\rightarrow\ (v,\ s')$
$G\vdash Norm\ s\ -In1l\ (\{accC,statT,mode\}e\cdot mn(\{pT\}p))\ \succ\!\!\rightarrow\ (v,\ s')$
$G\vdash Norm\ s\ -In1r\ (Init\ C)\ \succ\!\!\rightarrow\ (x,\ s')$

declare *not-None-eq* [*simp*]
declare *split-paired-All* [*simp*] *split-paired-Ex* [*simp*]
⟨*ML*⟩
declare *split-if* [*split*] *split-if-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 ⇒ (*case ec of*
Inl e ⇒ (∃ *v*. *w* = *In1 v*)
| *Inr c* ⇒ *w* = ◇)
| *In2 e* ⇒ (∃ *v*. *w* = *In2 v*)
| *In3 e* ⇒ (∃ *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*:
$\bigwedge s\ s'.\ G\vdash s\ -t\succ\rightarrow\ (w,s')\Longrightarrow\ normal\ s'\longrightarrow\ normal\ s$
⟨*proof*⟩

lemma *eval-no-abrupt*:
$G\vdash(x,s)\ -t\succ\rightarrow\ (w,Norm\ s')\ =$
$(x\ =\ None\ \wedge\ G\vdash Norm\ s\ -t\succ\rightarrow\ (w,Norm\ s'))$
⟨*proof*⟩

⟨*ML*⟩

lemma *eval-abrupt-lemma*:
$G\vdash s\ -t\succ\rightarrow\ (v,s')\Longrightarrow\ abrupt\ s=Some\ xc\longrightarrow\ s'=\ s\ \wedge\ v\ =\ undefined3\ t$
⟨*proof*⟩

lemma *eval-abrupt*:
$G\vdash(Some\ xc,s)\ -t\succ\rightarrow\ (w,s')\ =$
$(s'=(Some\ xc,s)\ \wedge\ w=undefined3\ t\ \wedge$
$G\vdash(Some\ xc,s)\ -t\succ\rightarrow\ (undefined3\ t,(Some\ xc,s)))$
⟨*proof*⟩

⟨*ML*⟩

lemma *LitI*: $G\vdash s\ -Lit\ v-\succ(if\ normal\ s\ then\ v\ else\ undefined)\rightarrow\ s$
⟨*proof*⟩

lemma *SkipI* [*intro*!]: $G\vdash s\ -Skip\rightarrow\ s$
⟨*proof*⟩

lemma *ExprI*: $G\vdash s\ -e-\succ v\rightarrow\ s' \Longrightarrow G\vdash s\ -Expr\ e\rightarrow\ s'$
⟨*proof*⟩

lemma *CompI*: $[\![G\vdash s\ -c1\rightarrow\ s1;\ G\vdash s1\ -c2\rightarrow\ s2]\!] \Longrightarrow G\vdash s\ -c1;;\ c2\rightarrow\ s2$
⟨*proof*⟩

lemma *CondI*:
$\bigwedge s1.\ [\![G\vdash s\ -e-\succ b\rightarrow\ s1;\ G\vdash s1\ -(if\ the\text{-}Bool\ b\ then\ e1\ else\ e2)-\succ v\rightarrow\ s2]\!] \Longrightarrow$
$G\vdash s\ -e\ ?\ e1\ :\ e2-\succ(if\ normal\ s1\ then\ v\ else\ undefined)\rightarrow\ s2$
⟨*proof*⟩

lemma *IfI*: $[\![G\vdash s\ -e-\succ v\rightarrow\ s1;\ G\vdash s1\ -(if\ the\text{-}Bool\ v\ then\ c1\ else\ c2)\rightarrow\ s2]\!]$
$\Longrightarrow G\vdash s\ -If(e)\ c1\ Else\ c2\rightarrow\ s2$
⟨*proof*⟩

lemma *MethdI*: $G\vdash s\ -body\ G\ C\ sig-\succ v\rightarrow\ s'$
$\Longrightarrow G\vdash s\ -Methd\ C\ sig-\succ v\rightarrow\ s'$
⟨*proof*⟩

lemma *eval-Call*:
$[\![G\vdash Norm\ s0\ -e-\succ a'\rightarrow\ s1;\ G\vdash s1\ -ps\doteq\succ pvs\rightarrow\ s2;$
$D = invocation\text{-}declclass\ G\ mode\ (store\ s2)\ a'\ statT\ (\!|name=mn,parTs=pTs|\!);$
$s3 = init\text{-}lvars\ G\ D\ (\!|name=mn,parTs=pTs|\!)\ mode\ a'\ pvs\ s2;$
$s3' = check\text{-}method\text{-}access\ G\ accC\ statT\ mode\ (\!|name=mn,parTs=pTs|\!)\ a'\ s3;$
$G\vdash s3'-Methd\ D\ (\!|name=mn,parTs=pTs|\!)-\succ\ v\rightarrow\ s4;$
$s4' = restore\text{-}lvars\ s2\ s4]\!] \Longrightarrow$
$G\vdash Norm\ s0\ -\{accC,statT,mode\}e\cdot mn(\{pTs\}ps)-\succ v\rightarrow\ s4'$
⟨*proof*⟩

lemma *eval-Init*:
$[\![if\ inited\ C\ (globs\ s0)\ then\ s3 = Norm\ s0$
$else\ G\vdash Norm\ (init\text{-}class\text{-}obj\ G\ C\ s0)$
$-(if\ C = Object\ then\ Skip\ else\ Init\ (super\ (the\ (class\ G\ C))))\rightarrow\ s1\ \wedge$
$G\vdash set\text{-}lvars\ empty\ s1\ -(init\ (the\ (class\ G\ C)))\rightarrow\ s2\ \wedge$
$s3 = restore\text{-}lvars\ s1\ s2]\!] \Longrightarrow$
$G\vdash Norm\ s0\ -Init\ C\rightarrow\ s3$
⟨*proof*⟩

lemma *init-done*: $initd\ C\ s \Longrightarrow G\vdash s\ -Init\ C\rightarrow\ s$
⟨*proof*⟩

lemma *eval-StatRef*:
$G\vdash s\ -StatRef\ rt-\succ(if\ abrupt\ s=None\ then\ Null\ else\ undefined)\rightarrow\ s$
⟨*proof*⟩

lemma *SkipD* [*dest*!]: $G\vdash s\ -Skip\rightarrow\ s' \Longrightarrow s' = s$
⟨*proof*⟩

lemma *Skip-eq* [*simp*]: $G\vdash s\ -Skip\rightarrow\ s' = (s = s')$
⟨*proof*⟩

lemma *init-retains-locals* [*rule-format* (*no-asm*)]: $G\vdash s\ -t\succ\rightarrow\ (w,s') \Longrightarrow$
$(\forall\, C.\ t=In1r\ (Init\ C) \longrightarrow locals\ (store\ s) = locals\ (store\ s'))$
⟨*proof*⟩

lemma *halloc-xcpt* [*dest*!]:
$\bigwedge s'.\ G\vdash(Some\ xc,s)\ -halloc\ oi\succ a\rightarrow\ s' \Longrightarrow s'=(Some\ xc,s)$
⟨*proof*⟩

lemma *eval-Methd*:
$G\vdash s\ -In1l(body\ G\ C\ sig)\succ\rightarrow\ (w,s')$
$\Longrightarrow G\vdash s\ -In1l(Methd\ C\ sig)\succ\rightarrow\ (w,s')$
⟨*proof*⟩

lemma *eval-Body*: $[\![G\vdash Norm\ s0\ -Init\ D\rightarrow\ s1;\ G\vdash s1\ -c\rightarrow\ s2;$
$res=the\ (locals\ (store\ s2)\ Result);$
$s3 = (if\ (\exists\ l.\ abrupt\ s2 = Some\ (Jump\ (Break\ l)) \vee$
$abrupt\ s2 = Some\ (Jump\ (Cont\ l)))$
$then\ abupd\ (\lambda\ x.\ Some\ (Error\ CrossMethodJump))\ s2$
$else\ s2);$
$s4=abupd\ (absorb\ Ret)\ s3]\!] \Longrightarrow$
$G\vdash Norm\ s0\ -Body\ D\ c-\succ res\rightarrow s4$
⟨*proof*⟩

lemma *eval-binop-arg2-indep*:
$\neg\ need\text{-}second\text{-}arg\ binop\ v1 \Longrightarrow eval\text{-}binop\ binop\ v1\ x = eval\text{-}binop\ binop\ v1\ y$
⟨*proof*⟩

lemma *eval-BinOp-arg2-indepI*:
assumes *eval-e1*: $G\vdash Norm\ s0\ -e1-\succ v1\rightarrow\ s1$ **and**
no-need: $\neg\ need\text{-}second\text{-}arg\ binop\ v1$
shows $G\vdash Norm\ s0\ -BinOp\ binop\ e1\ e2-\succ(eval\text{-}binop\ binop\ v1\ v2)\rightarrow\ s1$
(**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-valued {((*s*,*oi*),(*a*,*s'*)). *G*⊢*s* −*halloc oi*≻*a* → *s'*}
⟨*proof*⟩

lemma *unique-sxalloc* [*rule-format* (*no-asm*)]:
G⊢*s* −*sxalloc*→ *s'* ⟹ *G*⊢*s* −*sxalloc*→ *s''* ⟶ *s''* = *s'*
⟨*proof*⟩

lemma *single-valued-sxalloc*: *single-valued* {(*s*,*s'*). *G*⊢*s* −*sxalloc*→ *s'*}
⟨*proof*⟩

lemma *split-pairD*: (*x*,*y*) = *p* ⟹ *x* = *fst p* & *y* = *snd p*
⟨*proof*⟩

lemma *unique-eval* [*rule-format* (*no-asm*)]:
G⊢*s* −*t*≻→ (*w*, *s'*) ⟹ (∀ *w' s''*. *G*⊢*s* −*t*≻→ (*w'*, *s''*) ⟶ *w'* = *w* ∧ *s''* = *s'*)
⟨*proof*⟩

lemma *single-valued-eval*:
single-valued {((*s*, *t*), (*v*, *s'*)). *G*⊢*s* −*t*≻→ (*v*, *s'*)}
⟨*proof*⟩

end

Chapter 16

Example

43 Example Bali program

theory *Example* **imports** *Eval WellForm* **begin**

The following example Bali program includes:

- class and interface declarations with inheritance, hiding of fields, overriding of methods (with refined result type), array type,
- method call (with dynamic binding), parameter access, return expressions,
- expression statements, sequential composition, literal values, local assignment, local access, field assignment, type cast,
- exception generation and propagation, try and catch statement, throw statement
- instance creation and (default) static initialization

```
package java_lang

public interface HasFoo {
  public Base foo(Base z);
}

public class Base implements HasFoo {
  static boolean arr[] = new boolean[2];
  public HasFoo vee;
  public Base foo(Base z) {
    return z;
  }
}

public class Ext extends Base {
  public int vee;
  public Ext foo(Base z) {
    ((Ext)z).vee = 1;
    return null;
  }
}

public class Main {
  public static void main(String args[]) throws Throwable {
    Base e = new Ext();
    try {e.foo(null); }
    catch(NullPointerException z) {
      while(Ext.arr[2]) ;
    }
  }
}
```

declare *widen.null* [*intro*]

lemma *wf-fdecl-def2*: $\bigwedge fd.$ *wf-fdecl G P fd* = *is-acc-type G P* (*type* (*snd fd*))
⟨*proof*⟩

declare *wf-fdecl-def2* [*iff*]

type and expression names

datatype *tnam′* = *HasFoo′* | *Base′* | *Ext′* | *Main′*
datatype *vnam′* = *arr′* | *vee′* | *z′* | *e′*
datatype *label′* = *lab1′*

consts

tnam′ :: *tnam′* ⇒ *tnam*
vnam′ :: *vnam′* ⇒ *vname*
label′:: *label′* ⇒ *label*
axioms

inj-tnam′ [*simp*]: (*tnam′ x* = *tnam′ y*) = (*x* = *y*)
inj-vnam′ [*simp*]: (*vnam′ x* = *vnam′ y*) = (*x* = *y*)
inj-label′ [*simp*]: (*label′ x* = *label′ y*) = (*x* = *y*)

surj-tnam′: ∃ *m*. *n* = *tnam′ m*
surj-vnam′: ∃ *m*. *n* = *vnam′ m*
surj-label′: ∃ *m*. *n* = *label′ m*

abbreviation
HasFoo :: *qtname* **where**
HasFoo == (|*pid*=*java-lang*,*tid*=*TName* (*tnam′ HasFoo′*)|)

abbreviation
Base :: *qtname* **where**
Base == (|*pid*=*java-lang*,*tid*=*TName* (*tnam′ Base′*)|)

abbreviation
Ext :: *qtname* **where**
Ext == (|*pid*=*java-lang*,*tid*=*TName* (*tnam′ Ext′*)|)

abbreviation
Main :: *qtname* **where**
Main == (|*pid*=*java-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

defs

Object-mdecls-def: *Object-mdecls* ≡ []
SXcpt-mdecls-def: *SXcpt-mdecls* ≡ []

consts

foo :: *mname*

constdefs

foo-sig :: *sig*
foo-sig ≡ (|*name*=*foo*,*parTs*=[*Class Base*]|)

foo-mhead :: *mhead*
foo-mhead ≡ (|*access*=*Public*,*static*=*False*,*pars*=[*z*],*resT*=*Class Base*|)

constdefs

Base-foo :: *mdecl*
Base-foo ≡ (*foo-sig*, (|*access*=*Public*,*static*=*False*,*pars*=[*z*],*resT*=*Class Base*,
mbody=(|*lcls*=[],*stmt*=*Return* (!!*z*)|)|))

constdefs

Ext-foo :: *mdecl*
Ext-foo ≡ (*foo-sig*,
(|*access*=*Public*,*static*=*False*,*pars*=[*z*],*resT*=*Class Ext*,
mbody=(|*lcls*=[]
,*stmt*=*Expr*({*Ext*,*Ext*,*False*}*Cast* (*Class Ext*) (!!*z*)..*vee* :=
Lit (*Intg 1*)) ;;
Return (*Lit Null*)|)
|))

constdefs

arr-viewed-from :: *qtname* ⇒ *qtname* ⇒ *var*
arr-viewed-from accC C ≡ {*accC*,*Base*,*True*}*StatRef* (*ClassT C*)..*arr*

BaseCl :: *class*
BaseCl ≡ (|*access*=*Public*,
cfields=[(*arr*, (|*access*=*Public*,*static*=*True* ,*type*=*PrimT Boolean*.[]|)),
(*vee*, (|*access*=*Public*,*static*=*False*,*type*=*Iface HasFoo* |))],
methods=[*Base-foo*],
init=*Expr*(*arr-viewed-from Base Base*
:=*New* (*PrimT Boolean*)[*Lit* (*Intg 2*)]),
super=*Object*,
superIfs=[*HasFoo*]|)

ExtCl :: *class*
ExtCl ≡ (|*access*=*Public*,
cfields=[(*vee*, (|*access*=*Public*,*static*=*False*,*type*= *PrimT Integer*|))],
methods=[*Ext-foo*],
init=*Skip*,
super=*Base*,
superIfs=[]|)

MainCl :: *class*
MainCl ≡ (|*access*=*Public*,
cfields=[],
methods=[],
init=*Skip*,
super=*Object*,
superIfs=[]|)

constdefs

HasFooInt :: *iface*
HasFooInt ≡ (|*access*=*Public*,*imethods*=[(*foo-sig*, *foo-mhead*)],*isuperIfs*=[]|)

Ifaces ::*idecl list*
Ifaces ≡ [(*HasFoo*,*HasFooInt*)]

Classes ::*cdecl list*
Classes ≡ [(*Base*,*BaseCl*),(*Ext*,*ExtCl*),(*Main*,*MainCl*)]@*standard-classes*

lemmas *table-classes-defs* =
Classes-def standard-classes-def ObjectC-def SXcptC-def

lemma *table-ifaces* [*simp*]: *table-of Ifaces* = *empty*(*HasFoo*↦*HasFooInt*)
⟨*proof*⟩

lemma *table-classes-Object* [*simp*]:
table-of Classes Object = *Some* (|*access*=*Public*,*cfields*=[]
,*methods*=*Object-mdecls*
,*init*=*Skip*,*super*=*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*⦈)

constdefs
test :: (*ty*)*list* ⇒ *stmt*
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*⦈)

$\in (subcls1\ tprg)\hat{}+ \longrightarrow R$
⟨*proof*⟩

lemma *ws-idecl-HasFoo*: *ws-idecl tprg HasFoo* []
⟨*proof*⟩

lemma *ws-cdecl-Object*: *ws-cdecl tprg Object any*
⟨*proof*⟩

lemma *ws-cdecl-Throwable*: *ws-cdecl tprg* (*SXcpt Throwable*) *Object*
⟨*proof*⟩

lemma *ws-cdecl-SXcpt*: *ws-cdecl tprg* (*SXcpt xn*) (*SXcpt Throwable*)
⟨*proof*⟩

lemma *ws-cdecl-Base*: *ws-cdecl tprg Base Object*
⟨*proof*⟩

lemma *ws-cdecl-Ext*: *ws-cdecl tprg Ext Base*
⟨*proof*⟩

lemma *ws-cdecl-Main*: *ws-cdecl tprg Main Object*
⟨*proof*⟩

lemmas *ws-cdecls* = *ws-cdecl-SXcpt ws-cdecl-Object ws-cdecl-Throwable*
ws-cdecl-Base ws-cdecl-Ext ws-cdecl-Main

declare *not-Object-subcls-any* [*rule del*]
not-Throwable-subcls-SXcpt [*rule del*]
not-SXcpt-n-subcls-SXcpt-n [*rule del*]
not-Base-subcls-Ext [*rule del*] *not-TName-n-subcls-TName-n* [*rule del*]

lemma *ws-idecl-all*:
G=*tprg* ⟹ (∀ (*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*⊢*SXcpt xn*≼$_C$ *SXcpt Throwable*
⟨*proof*⟩

lemma *Ext-subclseq-Base* [*simp*]: *tprg*⊢*Ext* ≼$_C$ *Base*
⟨*proof*⟩

lemma *Ext-subcls-Base* [*simp*]: *tprg*⊢*Ext* ≺$_C$ *Base*
⟨*proof*⟩

fields and method lookup

lemma *fields-tprg-Object* [*simp*]: *DeclConcepts.fields tprg Object* = []
⟨*proof*⟩

lemma *fields-tprg-Throwable* [*simp*]:
DeclConcepts.fields tprg (*SXcpt Throwable*) = []
⟨*proof*⟩

lemma *fields-tprg-SXcpt* [*simp*]: *DeclConcepts.fields tprg* (*SXcpt xn*) = []
⟨*proof*⟩

lemmas *fields-rec′* = *fields-rec* [*OF - ws-tprg*]

lemma *fields-Base* [*simp*]:
DeclConcepts.fields tprg Base
= [((*arr*,*Base*), (|*access*=*Public*,*static*=*True* ,*type*=*PrimT Boolean*.[]|)),
((*vee*,*Base*), (|*access*=*Public*,*static*=*False*,*type*=*Iface HasFoo* |))]
⟨*proof*⟩

lemma *fields-Ext* [*simp*]:
DeclConcepts.fields tprg Ext
= [((*vee*,*Ext*), (|*access*=*Public*,*static*=*False*,*type*= *PrimT Integer*|))]
@ *DeclConcepts.fields tprg Base*
⟨*proof*⟩

lemmas *imethds-rec′* = *imethds-rec* [*OF - ws-tprg*]
lemmas *methd-rec′* = *methd-rec* [*OF - ws-tprg*]

lemma *imethds-HasFoo* [*simp*]:
imethds tprg HasFoo = *Option.set* ∘ *empty*(*foo-sig*↦(*HasFoo*, *foo-mhead*))

⟨*proof*⟩

lemma *methd-tprg-Object* [*simp*]: *methd tprg Object* = *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*)
(*empty*(*fst* ((λ(*s*, *m*). (*s*, *Base*, *m*)) *Base-foo*)↦
snd ((λ(*s*, *m*). (*s*, *Base*, *m*)) *Base-foo*)))
= *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 = 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*
⟹ *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-foo)
⟨*proof*⟩

lemma *Base-fields-accessible*[*simp*]:
accfield tprg S Base
= table-of((*map* (λ((*n*,*d*),*f*).(*n*,(*d*,*f*)))) (*DeclConcepts.fields tprg Base*))
⟨*proof*⟩

lemma *arr-member-of-Base*:
tprg⊢(*Base*, *fdecl* (*arr*,
(⦇*access* = *Public*, *static* = *True*, *type* = *PrimT Boolean.*[]⦈)))
member-of Base
⟨*proof*⟩

lemma *arr-member-in-Base*:
tprg⊢(*Base*, *fdecl* (*arr*,
(⦇*access* = *Public*, *static* = *True*, *type* = *PrimT Boolean.*[]⦈)))
member-in Base
⟨*proof*⟩

lemma *arr-member-of-Ext*:
tprg⊢(*Base*, *fdecl* (*arr*,
(⦇*access* = *Public*, *static* = *True*, *type* = *PrimT Boolean.*[]⦈)))
member-of Ext
⟨*proof*⟩

lemma *arr-member-in-Ext*:
tprg⊢(*Base*, *fdecl* (*arr*,
(⦇*access* = *Public*, *static* = *True*, *type* = *PrimT Boolean.*[]⦈)))
member-in Ext
⟨*proof*⟩

lemma *Ext-fields-accessible*[*simp*]:
accfield tprg S Ext
= table-of((*map* (λ((*n*,*d*),*f*).(*n*,(*d*,*f*)))) (*DeclConcepts.fields tprg Ext*))
⟨*proof*⟩

lemma *arr-Base-dyn-accessible* [*simp*]:
tprg⊢(*Base*, *fdecl* (*arr*, (⦇*access*=*Public*,*static*=*True* ,*type*=*PrimT Boolean.*[]⦈)))
in Base dyn-accessible-from S
⟨*proof*⟩

lemma *arr-Ext-dyn-accessible*[*simp*]:
tprg⊢(*Base*, *fdecl* (*arr*, (|*access*=*Public*,*static*=*True* ,*type*=*PrimT Boolean*.[]|)))
in Ext dyn-accessible-from S
⟨*proof*⟩

lemma *array-of-PrimT-acc* [*simp*]:
is-acc-type tprg java-lang (*PrimT t*.[])
⟨*proof*⟩

lemma *PrimT-acc* [*simp*]:
is-acc-type tprg java-lang (*PrimT t*)
⟨*proof*⟩

lemma *Object-acc* [*simp*]:
is-acc-class tprg java-lang Object
⟨*proof*⟩

well-formedness

lemma *wf-HasFoo*: *wf-idecl tprg* (*HasFoo*, *HasFooInt*)
⟨*proof*⟩

declare *member-is-static-simp* [*simp*]
declare *wt.Skip* [*rule del*] *wt.Init* [*rule del*]
⟨*ML*⟩
lemmas *wtIs* = *wt-Call wt-Super wt-FVar wt-StatRef wt-intros*
lemmas *daIs* = *assigned.select-convs da-Skip da-NewC da-Lit da-Super da.intros*

lemmas *Base-foo-defs* = *Base-foo-def foo-sig-def foo-mhead-def*
lemmas *Ext-foo-defs* = *Ext-foo-def foo-sig-def*

lemma *wf-Base-foo*: *wf-mdecl tprg Base Base-foo*
⟨*proof*⟩

lemma *wf-Ext-foo*: *wf-mdecl tprg Ext Ext-foo*
⟨*proof*⟩

declare *mhead-resTy-simp* [*simp add*]
declare *member-is-static-simp* [*simp add*]

lemma *wf-BaseC*: *wf-cdecl tprg* (*Base*,*BaseCl*)
⟨*proof*⟩

lemma *wf-ExtC*: *wf-cdecl tprg* (*Ext*,*ExtCl*)
⟨*proof*⟩

lemma *wf-MainC*: *wf-cdecl tprg* (*Main*,*MainCl*)
⟨*proof*⟩

lemma *wf-idecl-all*: *p*=*tprg* ⟹ *Ball* (*set Ifaces*) (*wf-idecl p*)
⟨*proof*⟩

lemma *wf-cdecl-all-standard-classes*:
Ball (*set standard-classes*) (*wf-cdecl tprg*)
⟨*proof*⟩

lemma *wf-cdecl-all*: *p*=*tprg* ⟹ *Ball* (*set Classes*) (*wf-cdecl p*)
⟨*proof*⟩

theorem *wf-tprg*: *wf-prog tprg*
⟨*proof*⟩

max spec

lemma *appl-methds-Base-foo*:
appl-methds tprg S (*ClassT Base*) (|*name*=*foo*, *parTs*=[*NT*]|) =
{((*ClassT Base*, (|*access*=*Public*,*static*=*False*,*pars*=[*z*],*resT*=*Class Base*|))
,[*Class Base*])}
⟨*proof*⟩

lemma *max-spec-Base-foo*: *max-spec tprg S* (*ClassT Base*) (|*name*=*foo*,*parTs*=[*NT*]|) =
{((*ClassT Base*, (|*access*=*Public*,*static*=*False*,*pars*=[*z*],*resT*=*Class Base*|))
, [*Class Base*])}
⟨*proof*⟩

well-typedness

lemma *wt-test*: (|*prg*=*tprg*,*cls*=*Main*,*lcl*=*empty*(*VName e*↦*Class Base*)|)⊢*test ?pTs*::√
⟨*proof*⟩

definite assignment

lemma *da-test*: (|*prg*=*tprg*,*cls*=*Main*,*lcl*=*empty*(*VName e*↦*Class Base*)|)
⊢{} »⟨*test ?pTs*⟩» (|*nrm*={*VName e*},*brk*=λ *l*. *UNIV*|)
⟨*proof*⟩

execution

lemma *alloc-one*: ⋀*a obj*. ⟦*the* (*new-Addr h*) = *a*; *atleast-free h* (*Suc n*)⟧ ⟹
new-Addr h = *Some a* ∧ *atleast-free* (*h*(*a*↦*obj*)) *n*
⟨*proof*⟩

declare *fvar-def2* [*simp*] *avar-def2* [*simp*] *init-lvars-def2* [*simp*]
declare *init-obj-def* [*simp*] *var-tys-def* [*simp*] *fields-table-def* [*simp*]
declare *BaseCl-def* [*simp*] *ExtCl-def* [*simp*] *Ext-foo-def* [*simp*]
Base-foo-defs [*simp*]

⟨*ML*⟩
lemmas *eval-Is* = *eval-Init eval-StatRef AbruptIs eval-intros*

consts

a :: *loc*
b :: *loc*
c :: *loc*

abbreviation *one* == *Suc 0*
abbreviation *two* == *Suc one*
abbreviation *tree* == *Suc two*
abbreviation *four* == *Suc tree*

syntax
obj-a :: *obj*
obj-b :: *obj*
obj-c :: *obj*
arr-N :: (*vn*, *val*) *table*
arr-a :: (*vn*, *val*) *table*
globs1 :: *globs*
globs2 :: *globs*
globs3 :: *globs*
globs8 :: *globs*
locs3 :: *locals*
locs4 :: *locals*
locs8 :: *locals*
s0 :: *state*
s0' :: *state*
s9' :: *state*
s1 :: *state*
s1' :: *state*
s2 :: *state*
s2' :: *state*
s3 :: *state*
s3' :: *state*
s4 :: *state*
s4' :: *state*
s6' :: *state*
s7' :: *state*
s8 :: *state*
s8' :: *state*

translations
obj-a <= (|*tag*=*Arr* (*PrimT Boolean*) (*CONST two*)
,*values*=*CONST empty*(*Inr 0*↦*Bool False*)(*Inr* (*CONST one*)↦*Bool False*)|)
obj-b <= (|*tag*=*CInst* (*CONST Ext*)
,*values*=(*CONST empty*(*Inl* (*CONST vee*, *CONST Base*)↦*Null*)
(*Inl* (*CONST vee*, *CONST Ext*)↦*Intg 0*))|)
obj-c == (|*tag*=*CInst* (*SXcpt NullPointer*),*values*=*CONST empty*|)
arr-N == *CONST empty*(*Inl* (*CONST arr*, *CONST Base*)↦*Null*)
arr-a == *CONST empty*(*Inl* (*CONST arr*, *CONST Base*)↦*Addr a*)
globs1 == *CONST empty*(*Inr* (*CONST Ext*) ↦(|*tag*=*CONST undefined*, *values*=*CONST empty*|))
(*Inr* (*CONST Base*) ↦(|*tag*=*CONST undefined*, *values*=*arr-N*|))
(*Inr Object*↦(|*tag*=*CONST undefined*, *values*=*CONST empty*|))
globs2 == *CONST empty*(*Inr* (*CONST Ext*) ↦(|*tag*=*CONST undefined*, *values*=*CONST empty*|))
(*Inr Object*↦(|*tag*=*CONST undefined*, *values*=*CONST empty*|))
(*Inl a*↦*obj-a*)
(*Inr* (*CONST Base*) ↦(|*tag*=*CONST undefined*, *values*=*arr-a*|))
globs3 == *globs2*(*Inl b*↦*obj-b*)
globs8 == *globs3*(*Inl c*↦*obj-c*)
locs3 == *CONST empty*(*VName* (*CONST e*)↦*Addr b*)
locs4 == *CONST empty*(*VName* (*CONST z*)↦*Null*)(*Inr*()↦*Addr b*)
locs8 == *locs3*(*VName* (*CONST z*)↦*Addr c*)

s0 == *st* (*CONST empty*) (*CONST empty*)
s0' == *Norm s0*
s1 == *st globs1* (*CONST empty*)
s1' == *Norm s1*
s2 == *st globs2* (*CONST empty*)
s2' == *Norm s2*
s3 == *st globs3 locs3*
s3' == *Norm s3*
s4 == *st globs3 locs4*
s4' == *Norm s4*
s6' == (*Some* (*Xcpt* (*Std NullPointer*)), *s4*)
s7' == (*Some* (*Xcpt* (*Std NullPointer*)), *s3*)
s8 == *st globs8 locs8*
s8' == *Norm s8*
s9' == (*Some* (*Xcpt* (*Std IndOutBound*)), *s8*)

declare *Pair-eq* [*simp del*]

lemma *exec-test*:
⟦*the* (*new-Addr* (*heap s1*)) = *a*;
the (*new-Addr* (*heap ?s2*)) = *b*;
the (*new-Addr* (*heap ?s3*)) = *c*⟧ ⟹
atleast-free (*heap s0*) *four* ⟹
tprg⊢*s0'* −*test* [*Class Base*]→ *?s9'*
⟨*proof*⟩
declare *Pair-eq* [*simp*]

end

Chapter 17

Conform

44 Conformance notions for the type soundness proof for Java

theory *Conform* **imports** *State* **begin**

design issues:

- lconf allows for (arbitrary) inaccessible values
- "conforms" does not directly imply that the dynamic types of all objects on the heap are indeed existing classes. Yet this can be inferred for all referenced objs.

types $env' = prog \times (lname, ty)\ table$

extension of global store

constdefs

$gext :: st \Rightarrow st \Rightarrow bool \quad (\text{-}\leq|\text{-} \quad [71,71] \quad 70)$
$s\leq|s' \equiv \forall r.\ \forall obj{\in}globs\ s\ r{:}\ \exists obj'{\in}globs\ s'\ r{:}\ tag\ obj' = tag\ obj$

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*:
$[\![s\leq|s';\ globs\ s\ r = Some\ obj]\!]$
$\Longrightarrow \exists obj'.\ globs\ s'\ r = Some\ obj' \wedge tag\ obj' = tag\ obj$
⟨*proof*⟩

lemma *rev-gext-objD*:
$[\![globs\ s\ r = Some\ obj;\ s\leq|s']\!]$
$\Longrightarrow \exists obj'.\ globs\ s'\ r = Some\ obj' \wedge tag\ obj' = tag\ obj$
⟨*proof*⟩

lemma *init-class-obj-inited*:
$init\text{-}class\text{-}obj\ G\ C\ s1\leq|s2 \Longrightarrow inited\ C\ (globs\ s2)$
⟨*proof*⟩

lemma *gext-refl* [*intro*!, *simp*]: $s\leq|s$
⟨*proof*⟩

lemma *gext-gupd* [*simp*, *elim*!]: $\bigwedge s.\ globs\ s\ r = None \Longrightarrow s\leq|gupd(r{\mapsto}x)s$
⟨*proof*⟩

lemma *gext-new* [*simp*, *elim*!]: $\bigwedge s.\ globs\ s\ r = None \Longrightarrow s\leq|init\text{-}obj\ G\ oi\ r\ s$
⟨*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-locals l s1≤|s2 = s1≤|s2*
⟨*proof*⟩

lemma *gext-cong2* [*simp*]: *s1≤|set-locals l s2 = s1≤|s2*
⟨*proof*⟩

lemma *gext-lupd1* [*simp*]: *lupd(vn↦v)s1≤|s2 = s1≤|s2*
⟨*proof*⟩

lemma *gext-lupd2* [*simp*]: *s1≤|lupd(vn↦v)s2 = s1≤|s2*
⟨*proof*⟩

lemma *inited-gext*: ⟦*inited C (globs s)*; *s≤|s'*⟧ ⟹ *inited C (globs s')*
⟨*proof*⟩

value conformance

constdefs

conf :: *prog* ⇒ *st* ⇒ *val* ⇒ *ty* ⇒ *bool* (-,-⊢-::≼- [71,71,71,71] 70)
G,s⊢v::≼T ≡ ∃ *T'*∈*typeof* (λ*a*. *Option.map obj-ty* (*heap s a*)) *v*:*G⊢T'≼T*

lemma *conf-cong* [*simp*]: *G,set-locals l s⊢v::≼T = G,s⊢v::≼T*
⟨*proof*⟩

lemma *conf-lupd* [*simp*]: *G,lupd(vn↦va)s⊢v::≼T = G,s⊢v::≼T*
⟨*proof*⟩

lemma *conf-PrimT* [*simp*]: ∀ *dt*. *typeof dt v = Some (PrimT t)* ⟹ *G,s⊢v::≼PrimT t*
⟨*proof*⟩

lemma *conf-Boolean*: *G,s⊢v::≼PrimT Boolean* ⟹ ∃ *b*. *v=Bool b*
⟨*proof*⟩

lemma *conf-litval* [*rule-format* (*no-asm*)]:
typeof (λ*a*. *None*) *v = Some T* ⟶ *G,s⊢v::≼T*
⟨*proof*⟩

lemma *conf-Null* [*simp*]: *G,s⊢Null::≼T = G⊢NT≼T*
⟨*proof*⟩

lemma *conf-Addr*:
G,s⊢Addr a::≼T = (∃ *obj*. *heap s a = Some obj* ∧ *G⊢obj-ty obj≼T*)
⟨*proof*⟩

lemma *conf-AddrI*:⟦*heap s a = Some obj*; *G⊢obj-ty obj≼T*⟧ ⟹ *G,s⊢Addr a::≼T*
⟨*proof*⟩

lemma *defval-conf* [*rule-format* (*no-asm*), *elim*]:
is-type G T ⟶ *G,s⊢default-val T::≼T*
⟨*proof*⟩

lemma *conf-widen* [*rule-format* (*no-asm*), *elim*]:
G⊢T≼T′ ⟹ *G,s⊢x::≼T* ⟶ *ws-prog G* ⟶ *G,s⊢x::≼T′*
⟨*proof*⟩

lemma *conf-gext* [*rule-format* (*no-asm*), *elim*]:
G,s⊢v::≼T ⟶ *s≤|s′* ⟶ *G,s′⊢v::≼T*
⟨*proof*⟩

lemma *conf-list-widen* [*rule-format* (*no-asm*)]:
ws-prog G ⟹
∀ *Ts Ts′*. *list-all2* (*conf G s*) *vs Ts*
⟶ *G⊢Ts*[≼] *Ts′* ⟶ *list-all2* (*conf G s*) *vs Ts′*
⟨*proof*⟩

lemma *conf-RefTD* [*rule-format* (*no-asm*)]:
G,s⊢a′::≼RefT T
⟶ *a′ = Null* ∨ (∃ *a obj T′*. *a′ = Addr a* ∧ *heap s a = Some obj* ∧
obj-ty obj = T′ ∧ *G⊢T′≼RefT T*)
⟨*proof*⟩

value list conformance

constdefs

lconf :: *prog* ⇒ *st* ⇒ (′*a*, *val*) *table* ⇒ (′*a*, *ty*) *table* ⇒ *bool*
(-,-⊢-[::≼]- [*71*,*71*,*71*,*71*] *70*)
G,s⊢vs[::≼]*Ts* ≡ ∀ *n*. ∀ *T*∈*Ts n*: ∃ *v*∈*vs n*: *G,s⊢v::≼T*

lemma *lconfD*: ⟦*G,s⊢vs*[::≼]*Ts*; *Ts n = Some T*⟧ ⟹ *G,s⊢*(*the* (*vs n*))*::≼T*
⟨*proof*⟩

lemma *lconf-cong* [*simp*]: ⋀*s*. *G,set-locals x s⊢l*[::≼]*L = G,s⊢l*[::≼]*L*
⟨*proof*⟩

lemma *lconf-lupd* [*simp*]: *G,lupd*(*vn↦v*)*s⊢l*[::≼]*L = G,s⊢l*[::≼]*L*
⟨*proof*⟩

lemma *lconf-new*: ⟦*L vn = None*; *G,s⊢l*[::≼]*L*⟧ ⟹ *G,s⊢l*(*vn↦v*)[::≼]*L*

⟨*proof*⟩

lemma *lconf-upd*: ⟦*G*,*s*⊢*l*[::≼]*L*; *G*,*s*⊢*v*::≼*T*; *L vn* = *Some T*⟧ ⟹
G,*s*⊢*l*(*vn*↦*v*)[::≼]*L*
⟨*proof*⟩

lemma *lconf-ext*: ⟦*G*,*s*⊢*l*[::≼]*L*; *G*,*s*⊢*v*::≼*T*⟧ ⟹ *G*,*s*⊢*l*(*vn*↦*v*)[::≼]*L*(*vn*↦*T*)
⟨*proof*⟩

lemma *lconf-map-sum* [*simp*]:
G,*s*⊢*l1* (+) *l2*[::≼]*L1* (+) *L2* = (*G*,*s*⊢*l1*[::≼]*L1* ∧ *G*,*s*⊢*l2*[::≼]*L2*)
⟨*proof*⟩

lemma *lconf-ext-list* [*rule-format* (*no-asm*)]:
⋀*X*. ⟦*G*,*s*⊢*l*[::≼]*L*⟧ ⟹
∀ *vs Ts*. *distinct vns* ⟶ *length Ts* = *length vns*
⟶ *list-all2* (*conf G s*) *vs Ts* ⟶ *G*,*s*⊢*l*(*vns*[↦]*vs*)[::≼]*L*(*vns*[↦]*Ts*)
⟨*proof*⟩

lemma *lconf-deallocL*: ⟦*G*,*s*⊢*l*[::≼]*L*(*vn*↦*T*); *L vn* = *None*⟧ ⟹ *G*,*s*⊢*l*[::≼]*L*
⟨*proof*⟩

lemma *lconf-gext* [*elim*]: ⟦*G*,*s*⊢*l*[::≼]*L*; *s*≤|*s*′⟧ ⟹ *G*,*s*′⊢*l*[::≼]*L*
⟨*proof*⟩

lemma *lconf-empty* [*simp*, *intro*!]: *G*,*s*⊢*vs*[::≼]*empty*
⟨*proof*⟩

lemma *lconf-init-vals* [*intro*!]:
∀ *n*. ∀ *T*∈*fs n*:*is-type G T* ⟹ *G*,*s*⊢*init-vals fs*[::≼]*fs*
⟨*proof*⟩

weak value list conformance

Only if the value is defined it has to conform to its type. This is the contribution of the definite assignment analysis to the notion of conformance. The definite assignment analysis ensures that the program only attempts to access local variables that actually have a defined value in the state. So conformance must only ensure that the defined values are of the right type, and not also that the value is defined.

constdefs

wlconf :: *prog* ⇒ *st* ⇒ (′*a*, *val*) *table* ⇒ (′*a*, *ty*) *table* ⇒ *bool*
(-,-⊢-[∼::≼]- [*71*,*71*,*71*,*71*] *70*)
G,*s*⊢*vs*[∼::≼]*Ts* ≡ ∀ *n*. ∀ *T*∈*Ts n*: ∀ *v*∈*vs n*: *G*,*s*⊢*v*::≼*T*

lemma *wlconfD*: ⟦*G*,*s*⊢*vs*[∼::≼]*Ts*; *Ts n* = *Some T*; *vs n* = *Some v*⟧ ⟹ *G*,*s*⊢*v*::≼*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*]: $[\![G,s\vdash l[\sim::\preceq]L;\ s\preceq|s']\!] \Longrightarrow G,s'\vdash l[\sim::\preceq]L$
⟨*proof*⟩

lemma *wlconf-empty* [*simp*, *intro*!]: $G,s\vdash vs[\sim::\preceq]empty$
⟨*proof*⟩

lemma *wlconf-empty-vals*: $G,s\vdash empty[\sim::\preceq]ts$
⟨*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

constdefs

oconf :: *prog* ⇒ *st* ⇒ *obj* ⇒ *oref* ⇒ *bool* (-,-⊢-::≼√- [71,71,71,71] 70)
G,*s*⊢*obj*::≼√*r* ≡ *G*,*s*⊢*values obj*[::≼]*var-tys G* (*tag obj*) *r* ∧
(*case r of*
Heap a ⇒ *is-type G* (*obj-ty obj*)
| *Stat C* ⇒ *True*)

lemma *oconf-is-type*: *G*,*s*⊢*obj*::≼√*Heap a* ⟹ *is-type G* (*obj-ty obj*)
⟨*proof*⟩

lemma *oconf-lconf*: *G*,*s*⊢*obj*::≼√*r* ⟹ *G*,*s*⊢*values obj*[::≼]*var-tys G* (*tag obj*) *r*
⟨*proof*⟩

lemma *oconf-cong* [*simp*]: *G*,*set-locals l s*⊢*obj*::≼√*r* = *G*,*s*⊢*obj*::≼√*r*
⟨*proof*⟩

lemma *oconf-init-obj-lemma*:
⟦⋀*C c*. *class G C* = *Some c* ⟹ *unique* (*DeclConcepts.fields G C*);
⋀*C c f fld*. ⟦*class G C* = *Some c*;
table-of (*DeclConcepts.fields G C*) *f* = *Some fld* ⟧
⟹ *is-type G* (*type fld*);
(*case r of*
Heap a ⇒ *is-type G* (*obj-ty obj*)
| *Stat C* ⇒ *is-class G C*)
⟧ ⟹ *G*,*s*⊢*obj* (|*values*:=*init-vals* (*var-tys G* (*tag obj*) *r*)|)::≼√*r*
⟨*proof*⟩

state conformance

constdefs

conforms :: *state* ⇒ *env'* ⇒ *bool* (-::≼- [71,71] 70)
xs::≼*E* ≡ *let* (*G*, *L*) = *E*; *s* = *snd xs*; *l* = *locals s in*
(∀ *r*. ∀ *obj*∈*globs s r*: *G*,*s*⊢*obj* ::≼√*r*) ∧
G,*s*⊢*l* [∼::≼]*L* ∧
(∀ *a*. *fst xs*=*Some*(*Xcpt* (*Loc a*)) ⟶ *G*,*s*⊢*Addr a*::≼ *Class* (*SXcpt Throwable*)) ∧
(*fst xs*=*Some*(*Jump Ret*) ⟶ *l Result* ≠ *None*)

conforms

lemma *conforms-globsD*:
⟦(*x*, *s*)::≼(*G*, *L*); *globs s r* = *Some obj*⟧ ⟹ *G*,*s*⊢*obj*::≼√*r*
⟨*proof*⟩

lemma *conforms-localD*: (*x*, *s*)::≼(*G*, *L*) ⟹ *G*,*s*⊢*locals s*[∼::≼]*L*
⟨*proof*⟩

lemma *conforms-XcptLocD*: ⟦(*x*, *s*)::≼(*G*, *L*); *x* = *Some* (*Xcpt* (*Loc a*))⟧ ⟹
G,*s*⊢*Addr a*::≼ *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*: $[\![\forall\, r.\ \forall\, obj{\in}globs\ s\ r\colon\ G,s\vdash obj::\preceq\surd r;$
$G,s\vdash locals\ s[\sim::\preceq]L;$
$\forall\, a.\ x = Some\ (Xcpt\ (Loc\ a)) \longrightarrow G,s\vdash Addr\ a::\preceq Class\ (SXcpt\ Throwable);$
$x = Some\ (Jump\ Ret) \longrightarrow locals\ s\ Result \neq None]\!] \Longrightarrow$
$(x,\ s)::\preceq(G,\ L)$
⟨*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

45 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'*:
$is\text{-}acc\text{-}type\ (prg\ Env)\ (pid\ (cls\ Env))\ T \Longrightarrow Env{\vdash}init\text{-}comp\text{-}ty\ T{::}\surd$
⟨*proof*⟩

lemma *fvar-upd-no-jump*:
assumes *upd*: $upd = snd\ (fst\ (fvar\ statDeclC\ stat\ fn\ a\ s'))$
and *noJmp*: $abrupt\ s \neq Some\ (Jump\ j)$
shows $abrupt\ (upd\ val\ s) \neq Some\ (Jump\ j)$
⟨*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*: $abrupt\ s \neq Some\ (Jump\ j)$
shows $abrupt\ (upd\ val\ s) \neq Some\ (Jump\ j)$
⟨*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*: $G{\vdash}\ s0\ -t{\succ}{\rightarrow}\ (v,s1)$
and *jmpOk*: $jumpNestingOk\ jmps\ t$
and *wt*: $Env{\vdash}t{::}T$
and *wf*: $wf\text{-}prog\ G$

and G: $prg\ Env = G$
and $no\text{-}jmp$: $\forall j.\ abrupt\ s0 = Some\ (Jump\ j) \longrightarrow j \in jmps$
(**is** $?Jmp\ jmps\ s0$)
shows $(\forall j.\ fst\ s1 = Some\ (Jump\ j) \longrightarrow j \in jmps) \wedge$
$(normal\ s1 \longrightarrow$
$(\forall\ w\ upd.\ v{=}In2\ (w,upd)$
$\longrightarrow\ (\forall\ s\ j\ val.$
$abrupt\ s \neq Some\ (Jump\ j) \longrightarrow$
$abrupt\ (upd\ val\ s) \neq Some\ (Jump\ j))))$
(**is** $?Jmp\ jmps\ s1 \wedge\ ?Upd\ v\ s1$)
⟨*proof*⟩

lemmas $jumpNestingOk\text{-}evalE = jumpNestingOk\text{-}eval\ [THEN\ conjE, rule\text{-}format]$

lemma $jumpNestingOk\text{-}eval\text{-}no\text{-}jump$:
assumes $eval$: $prg\ Env{\vdash}\ s0\ -t{\succ}{\rightarrow}\ (v,s1)$ **and**
$jmpOk$: $jumpNestingOk\ \{\}\ t$ **and**
$no\text{-}jmp$: $abrupt\ s0 \neq Some\ (Jump\ j)$ **and**
wt: $Env{\vdash}t::T$ **and**
wf: $wf\text{-}prog\ (prg\ Env)$
shows $abrupt\ s1 \neq Some\ (Jump\ j) \wedge$
$(normal\ s1 \longrightarrow v{=}In2\ (w,upd)$
$\longrightarrow abrupt\ s \neq Some\ (Jump\ j')$
$\longrightarrow abrupt\ (upd\ val\ s) \neq Some\ (Jump\ j'))$
⟨*proof*⟩

lemmas $jumpNestingOk\text{-}eval\text{-}no\text{-}jumpE$
$= jumpNestingOk\text{-}eval\text{-}no\text{-}jump\ [THEN\ conjE, rule\text{-}format]$

corollary $eval\text{-}expression\text{-}no\text{-}jump$:
assumes $eval$: $prg\ Env{\vdash}s0\ -e{-}{\succ}v{\rightarrow}\ s1$ **and**
$no\text{-}jmp$: $abrupt\ s0 \neq Some\ (Jump\ j)$ **and**
wt: $Env{\vdash}e::{-}T$ **and**
wf: $wf\text{-}prog\ (prg\ Env)$
shows $abrupt\ s1 \neq Some\ (Jump\ j)$
⟨*proof*⟩

corollary $eval\text{-}var\text{-}no\text{-}jump$:
assumes $eval$: $prg\ Env{\vdash}s0\ -var{=}{\succ}(w,upd){\rightarrow}\ s1$ **and**
$no\text{-}jmp$: $abrupt\ s0 \neq Some\ (Jump\ j)$ **and**
wt: $Env{\vdash}var::{=}T$ **and**
wf: $wf\text{-}prog\ (prg\ Env)$
shows $abrupt\ s1 \neq Some\ (Jump\ j) \wedge$
$(normal\ s1 \longrightarrow$
$(abrupt\ s \neq Some\ (Jump\ j')$
$\longrightarrow abrupt\ (upd\ val\ s) \neq Some\ (Jump\ j')))$
⟨*proof*⟩

lemmas $eval\text{-}var\text{-}no\text{-}jumpE = eval\text{-}var\text{-}no\text{-}jump\ [THEN\ conjE, rule\text{-}format]$

corollary $eval\text{-}statement\text{-}no\text{-}jump$:
assumes $eval$: $prg\ Env{\vdash}s0\ -c{\rightarrow}\ s1$ **and**
$jmpOk$: $jumpNestingOkS\ \{\}\ c$ **and**
$no\text{-}jmp$: $abrupt\ s0 \neq Some\ (Jump\ j)$ **and**
wt: $Env{\vdash}c::\surd$ **and**
wf: $wf\text{-}prog\ (prg\ Env)$
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* ≠ *Some* (*Jump j*) **and**
wt: *Env*⊢*es*::≐*T* **and**
wf: *wf-prog* (*prg Env*)
shows *abrupt s1* ≠ *Some* (*Jump j*)
⟨*proof*⟩

lemma *union-subseteq-elim* [*elim*]: ⟦*A* ∪ *B* ⊆ *C*; ⟦*A* ⊆ *C*; *B* ⊆ *C*⟧ ⟹ *P*⟧ ⟹ *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*⊢ *s0* −*t*≻→ (*v*,*s1*)
shows *dom* (*locals* (*store s0*)) ⊆ *dom* (*locals* (*store s1*)) ∧
(∀ *vv*. *v*=*In2 vv* ∧ *normal s1*
⟶ (∀ *s val*. *dom* (*locals* (*store s*))
⊆ *dom* (*locals* (*store* ((*snd vv*) *val s*)))))
⟨*proof*⟩

lemma *dom-locals-eval-mono-elim*:
assumes *eval*: *G*⊢ *s0* −*t*≻→ (*v*,*s1*)
obtains *dom* (*locals* (*store s0*)) ⊆ *dom* (*locals* (*store s1*)) **and**
⋀ *vv s val*. ⟦*v*=*In2 vv*; *normal s1*⟧
⟹ *dom* (*locals* (*store s*))
⊆ *dom* (*locals* (*store* ((*snd vv*) *val s*)))
⟨*proof*⟩

lemma *halloc-no-abrupt*:
assumes *halloc*: *G*⊢*s0*−*halloc oi*≻*a*→*s1* **and**
normal: *normal s1*
shows *normal s0*
⟨*proof*⟩

lemma *sxalloc-mono-no-abrupt*:
assumes *sxalloc*: *G*⊢*s0*−*sxalloc*→*s1* **and**
normal: *normal s1*
shows *normal s0*
⟨*proof*⟩

lemma *union-subseteqI*: ⟦*A* ∪ *B* ⊆ *C*; *A'* ⊆ *A*; *B'* ⊆ *B*⟧ ⟹ *A'* ∪ *B'* ⊆ *C*
⟨*proof*⟩

lemma *union-subseteqIl*: ⟦*A* ∪ *B* ⊆ *C*; *A'* ⊆ *A*⟧ ⟹ *A'* ∪ *B* ⊆ *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* ∧ *normal s*
⟨*proof*⟩

lemmas *constVal-eval-elim* = *constVal-eval* [*THEN conjE*]

lemma *eval-unop-type*:
typeof dt (*eval-unop unop v*) = *Some* (*PrimT* (*unop-type unop*))
⟨*proof*⟩

lemma *eval-binop-type*:
typeof dt (*eval-binop binop v1 v2*) = *Some* (*PrimT* (*binop-type binop*))
⟨*proof*⟩

lemma *constVal-Boolean*:
assumes *const*: *constVal e* = *Some c* **and**
wt: *Env*⊢*e*::−*PrimT Boolean*
shows *typeof empty-dt c* = *Some* (*PrimT Boolean*)
⟨*proof*⟩

lemma *assigns-if-good-approx*:
assumes
eval: *prg Env*⊢ *s0* −*e*−≻*b*→ *s1* **and**
normal: *normal s1* **and**
bool: *Env*⊢ *e*::−*PrimT Boolean*
shows *assigns-if* (*the-Bool b*) *e* ⊆ *dom* (*locals* (*store s1*))
⟨*proof*⟩

lemma *assigns-if-good-approx′*:
assumes *eval*: *G*⊢*s0* −*e*−≻*b*→ *s1*
and *normal*: *normal s1*
and *bool*: (|*prg*=*G*,*cls*=*C*,*lcl*=*L*|)⊢*e*::− (*PrimT Boolean*)
shows *assigns-if* (*the-Bool b*) *e* ⊆ *dom* (*locals* (*store s1*))
⟨*proof*⟩

lemma *subset-Intl*: *A* ⊆ *C* ⟹ *A* ∩ *B* ⊆ *C*
⟨*proof*⟩

lemma *subset-Intr*: *B* ⊆ *C* ⟹ *A* ∩ *B* ⊆ *C*
⟨*proof*⟩

lemma *da-good-approx*:
assumes *eval*: *prg Env*⊢*s0* −*t*≻→ (*v*,*s1*) **and**
wt: *Env*⊢*t*::*T* (**is** *?Wt Env t T*) **and**
da: *Env*⊢ *dom* (*locals* (*store s0*)) »*t*» *A* (**is** *?Da Env s0 t A*) **and**
wf: *wf-prog* (*prg Env*)
shows (*normal s1* ⟶ (*nrm A* ⊆ *dom* (*locals* (*store s1*)))) ∧
(∀ *l*. *abrupt s1* = *Some* (*Jump* (*Break l*)) ∧ *normal s0*
⟶ (*brk A l* ⊆ *dom* (*locals* (*store s1*)))) ∧
(*abrupt s1* = *Some* (*Jump Ret*) ∧ *normal s0*
⟶ *Result* ∈ *dom* (*locals* (*store s1*)))
(**is** *?NormalAssigned s1 A* ∧ *?BreakAssigned s0 s1 A* ∧ *?ResAssigned s0 s1*)
⟨*proof*⟩

lemma *da-good-approxE*:
assumes
prg Env⊢*s0* −*t*≻→ (*v*, *s1*) **and** *Env*⊢*t*::*T* **and**
Env⊢ *dom* (*locals* (*store s0*)) »*t*» *A* **and** *wf-prog* (*prg Env*)
obtains
normal s1 ⟹ *nrm A* ⊆ *dom* (*locals* (*store s1*)) **and**
⋀ *l*. ⟦*abrupt s1* = *Some* (*Jump* (*Break l*)); *normal s0*⟧

$\Longrightarrow$ *brk A l* $\subseteq$ *dom* (*locals* (*store s1*)) **and**
⟦*abrupt s1* = *Some* (*Jump Ret*);*normal s0*⟧$\Longrightarrow$*Result* $\in$ *dom* (*locals* (*store s1*))
⟨*proof*⟩

lemma *da-good-approxE′*:
assumes *eval*: *G*⊢*s0* $-t\succ\rightarrow$ (*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* $\Longrightarrow$ *nrm A* $\subseteq$ *dom* (*locals* (*store s1*)) **and**
$\bigwedge$ *l*. ⟦*abrupt s1* = *Some* (*Jump* (*Break l*)); *normal s0*⟧
$\Longrightarrow$ *brk A l* $\subseteq$ *dom* (*locals* (*store s1*)) **and**
⟦*abrupt s1* = *Some* (*Jump Ret*);*normal s0*⟧
$\Longrightarrow$ *Result* $\in$ *dom* (*locals* (*store s1*))
⟨*proof*⟩

declare [[*simproc add*: *wt-expr wt-var wt-exprs wt-stmt*]]

end

Chapter 19

TypeSafe

46 The type soundness proof for Java

theory *TypeSafe*
imports *DefiniteAssignmentCorrect Conform*
begin

error free

hide *const field*

lemma *error-free-halloc*:
assumes *halloc*: $G\vdash s0\ -halloc\ oi\succ a\rightarrow\ s1$ **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* (λv. *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

constdefs

assign-conforms :: *st* ⇒ (*val* ⇒ *state* ⇒ *state*) ⇒ *ty* ⇒ *env'* ⇒ *bool*
(-≤|-⪯-::⪯- [71,71,71,71] 70)
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'*)))

constdefs
rconf :: *prog* ⇒ *lenv* ⇒ *st* ⇒ *term* ⇒ *vals* ⇒ *tys* ⇒ *bool*
(-,-,-⊢-≻-::⪯- [71,71,71,71,71,71] 70)
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* ⟹
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* ⟹
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*⊢*In3 es*≻*In3 vs*::⪯*Inr Ts* = *list-all2* (λ*v T*. *G*,*s*⊢*v*::⪯*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*: $\bigwedge s1\ s2.\ G\vdash s1\ -halloc\ oi\succ a\rightarrow\ s2 \Longrightarrow snd\ s1 \leq| snd\ s2$
⟨*proof*⟩

lemma *sxalloc-gext*: $\bigwedge s1\ s2.\ G\vdash s1\ -sxalloc\rightarrow\ s2 \Longrightarrow snd\ s1 \leq| snd\ s2$
⟨*proof*⟩

lemma *eval-gext-lemma* [*rule-format* (*no-asm*)]:
$G\vdash s\ -t\succ\rightarrow\ (w,s') \Longrightarrow snd\ s \leq| snd\ s' \wedge (case\ w\ of$
$\quad In1\ v \Rightarrow True$
$\mid In2\ vf \Rightarrow normal\ s \longrightarrow (\forall v\ x\ s.\ s \leq| snd\ (assign\ (snd\ vf)\ v\ (x,s)))$
$\mid In3\ vs \Rightarrow True)$
⟨*proof*⟩

lemma *evar-gext-f*:
$G\vdash Norm\ s1\ -e=\succ vf\ \rightarrow\ s2 \Longrightarrow s \leq| snd\ (assign\ (snd\ vf)\ v\ (x,s))$
⟨*proof*⟩

lemmas *eval-gext* = *eval-gext-lemma* [*THEN conjunct1*]

lemma *eval-gext'*: $G\vdash(x1,s1)\ -t\succ\rightarrow\ (w,(x2,s2)) \Longrightarrow s1 \leq| s2$
⟨*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*: $[\![globs\ s\ oref = None;\ (x,\ s)::\preceq(G,L);$
$wf\text{-}prog\ G;\ case\ oref\ of\ Heap\ a \Rightarrow is\text{-}type\ G\ (obj\text{-}ty\ (\!|tag=oi,values=vs|\!))$
$\quad | \ Stat\ C \Rightarrow is\text{-}class\ G\ C]\!] \Longrightarrow$
$(x,\ init\text{-}obj\ G\ oi\ oref\ s)::\preceq(G,\ L)$
⟨*proof*⟩

lemma *conforms-init-class-obj*:
$[\![(x,s)::\preceq(G,\ L);\ wf\text{-}prog\ G;\ class\ G\ C=Some\ y;\ \neg\ inited\ C\ (globs\ s)]\!] \Longrightarrow$
$(x,init\text{-}class\text{-}obj\ G\ C\ s)::\preceq(G,\ L)$
⟨*proof*⟩

lemma *fst-init-lvars*[*simp*]:
$fst\ (init\text{-}lvars\ G\ C\ sig\ (invmode\ m\ e)\ a'\ pvs\ (x,s)) =$
$(if\ is\text{-}static\ m\ then\ x\ else\ (np\ a')\ x)$
⟨*proof*⟩

lemma *halloc-conforms*: $\bigwedge s1.\ [\![G\vdash s1\ -halloc\ oi\succ a\rightarrow\ s2;\ wf\text{-}prog\ G;\ s1::\preceq(G,\ L);$
$is\text{-}type\ G\ (obj\text{-}ty\ (\!|tag=oi,values=fs|\!))]\!] \Longrightarrow s2::\preceq(G,\ L)$
⟨*proof*⟩

lemma *halloc-type-sound*:
$\bigwedge s1.\ [\![G\vdash s1\ -halloc\ oi\succ a\rightarrow\ (x,s);\ wf\text{-}prog\ G;\ s1::\preceq(G,\ L);$
$T = obj\text{-}ty\ (\!|tag=oi,values=fs|\!);\ is\text{-}type\ G\ T]\!] \Longrightarrow$
$(x,s)::\preceq(G,\ L) \wedge (x = None \longrightarrow G,s\vdash Addr\ a::\preceq T)$
⟨*proof*⟩

lemma *sxalloc-type-sound*:
$\bigwedge s1\ s2.\ [\![G\vdash s1\ -sxalloc\rightarrow\ s2;\ wf\text{-}prog\ G]\!] \Longrightarrow$
$case\ fst\ s1\ of$
$\quad None \Rightarrow s2 = s1$
$\ | \ Some\ abr \Rightarrow (case\ abr\ of$
$\qquad Xcpt\ x \Rightarrow (\exists a.\ fst\ s2 = Some(Xcpt\ (Loc\ a)) \wedge$
$\qquad\qquad (\forall L.\ s1::\preceq(G,L) \longrightarrow s2::\preceq(G,L)))$
$\qquad | \ Jump\ j \Rightarrow s2 = s1$
$\qquad | \ Error\ e \Rightarrow s2 = s1)$
⟨*proof*⟩

lemma *wt-init-comp-ty*:
$is\text{-}acc\text{-}type\ G\ (pid\ C)\ T \Longrightarrow (\!|prg=G,cls=C,lcl=L|\!)\vdash init\text{-}comp\text{-}ty\ T::\surd$
⟨*proof*⟩

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

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

constdefs
$DynT\text{-}prop::[prog,inv\text{-}mode,qtname,ref\text{-}ty] \Rightarrow bool$
$\qquad (\text{-}\vdash\text{-}\rightarrow\text{-}\preceq\text{-}[71,71,71,71]\ 70)$

$G\vdash mode\rightarrow D\preceq t \equiv mode = IntVir \longrightarrow is\text{-}class\ G\ D \wedge$
$(if\ (\exists\ T.\ t=ArrayT\ T)\ then\ D=Object\ else\ G\vdash Class\ D\preceq RefT\ t)$

lemma *DynT-propI*:
$[\![(x,s)::\preceq(G,\ L);\ G,s\vdash a'::\preceq RefT\ statT;\ wf\text{-}prog\ G;\ mode = IntVir \longrightarrow a' \neq Null]\!]$
$\Longrightarrow\ G\vdash mode\rightarrow invocation\text{-}class\ mode\ s\ a'\ statT\preceq statT$
⟨*proof*⟩

lemma *invocation-methd*:
$[\![wf\text{-}prog\ G;\ statT \neq NullT;$
$(\forall\ statC.\ statT = ClassT\ statC \longrightarrow is\text{-}class\ G\ statC);$
$(\forall\ \ \ I.\ statT = IfaceT\ I \longrightarrow is\text{-}iface\ G\ I \wedge mode \neq SuperM);$
$(\forall\ \ \ T.\ statT = ArrayT\ T \longrightarrow mode \neq SuperM);$
$G\vdash mode\rightarrow invocation\text{-}class\ mode\ s\ a'\ statT\preceq statT;$
$dynlookup\ G\ statT\ (invocation\text{-}class\ mode\ s\ a'\ statT)\ sig = Some\ m\ ]\!]$
$\Longrightarrow methd\ G\ (invocation\text{-}declclass\ G\ mode\ s\ a'\ statT\ sig)\ sig = Some\ m$
⟨*proof*⟩

lemma *DynT-mheadsD*:
$[\![G\vdash invmode\ sm\ e\rightarrow invC\preceq statT;$
$wf\text{-}prog\ G;\ (\!|prg=G,cls=C,lcl=L|\!)\vdash e::-RefT\ statT;$
$(statDeclT,sm) \in mheads\ G\ C\ statT\ sig;$
$invC = invocation\text{-}class\ (invmode\ sm\ e)\ s\ a'\ statT;$
$declC = invocation\text{-}declclass\ G\ (invmode\ sm\ e)\ s\ a'\ statT\ sig$
$]\!] \Longrightarrow$
$\exists\ dm.$
$methd\ G\ declC\ sig = Some\ dm \wedge dynlookup\ G\ statT\ invC\ sig = Some\ dm\ \wedge$
$G\vdash resTy\ (mthd\ dm)\preceq resTy\ sm \wedge$
$wf\text{-}mdecl\ G\ declC\ (sig,\ mthd\ dm) \wedge$
$declC = declclass\ dm \wedge$
$is\text{-}static\ dm = is\text{-}static\ sm \wedge$
$is\text{-}class\ G\ invC \wedge is\text{-}class\ G\ declC\ \wedge G\vdash invC\preceq_C declC \wedge$
$(if\ invmode\ sm\ e = IntVir$
$\quad then\ (\forall\ statC.\ statT=ClassT\ statC \longrightarrow G\vdash invC\preceq_C statC)$
$\quad else\ (\ (\exists\ statC.\ statT=ClassT\ statC \wedge G\vdash statC\preceq_C declC)$
$\quad\quad \vee\ (\forall\ statC.\ statT\neq ClassT\ statC \wedge declC=Object)) \wedge$
$\quad\quad statDeclT = ClassT\ (declclass\ dm))$
⟨*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-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*: $\bigwedge dm.\ [\![methd\ G\ declC\ sig = Some\ dm;$
$dynlookup\ G\ statT\ invC\ sig = Some\ dm;$
$G\vdash resTy\ (mthd\ dm)\preceq resTy\ sm;$
$wf\text{-}mdecl\ G\ declC\ (sig,\ mthd\ dm);$
$declC = declclass\ dm;$
$is\text{-}static\ dm = is\text{-}static\ sm;$
$is\text{-}class\ G\ invC;\ is\text{-}class\ G\ declC;\ G\vdash invC\preceq_C declC;$
$(if\ invmode\ sm\ e = IntVir$

then (∀ *statC*. *statT*=*ClassT statC* ⟶ *G*⊢*invC* ≼$_C$ *statC*)
else ((∃ *statC*. *statT*=*ClassT statC* ∧ *G*⊢*statC*≼$_C$ *declC*)
∨ (∀ *statC*. *statT*≠*ClassT statC* ∧ *declC*=*Object*)) ∧
statDeclT = *ClassT* (*declclass dm*))⟧ ⟹ *P*

shows *P*
⟨*proof*⟩

lemma *DynT-conf*: ⟦*G*⊢*invocation-class mode s a′ statT* ≼$_C$ *declC*; *wf-prog G*;
isrtype G (*statT*);
G,*s*⊢*a′*::≼*RefT statT*; *mode* = *IntVir* ⟶ *a′* ≠ *Null*;
mode ≠ *IntVir* ⟶ (∃ *statC*. *statT*=*ClassT statC* ∧ *G*⊢*statC*≼$_C$ *declC*)
∨ (∀ *statC*. *statT*≠*ClassT statC* ∧ *declC*=*Object*)⟧
⟹*G*,*s*⊢*a′*::≼ *Class declC*
⟨*proof*⟩

lemma *Ass-lemma*:
⟦ *G*⊢*Norm s0* −*var*=≻(*w*, *f*)→ *Norm s1*; *G*⊢*Norm s1* −*e*−≻*v*→ *Norm s2*;
G,*s2*⊢*v*::≼*eT*;*s1*≤|*s2* ⟶ *assign f v* (*Norm s2*)::≼(*G*, *L*)⟧
⟹ *assign f v* (*Norm s2*)::≼(*G*, *L*) ∧
(*normal* (*assign f v* (*Norm s2*)) ⟶ *G*,*store* (*assign f v* (*Norm s2*))⊢*v*::≼*eT*)
⟨*proof*⟩

lemma *Throw-lemma*: ⟦*G*⊢*tn*≼$_C$ *SXcpt Throwable*; *wf-prog G*; (*x1*,*s1*)::≼(*G*, *L*);
x1 = *None* ⟶ *G*,*s1*⊢*a′*::≼ *Class tn*⟧ ⟹ (*throw a′ x1*, *s1*)::≼(*G*, *L*)
⟨*proof*⟩

lemma *Try-lemma*: ⟦*G*⊢*obj-ty* (*the* (*globs s1′* (*Heap a*)))≼ *Class tn*;
(*Some* (*Xcpt* (*Loc a*)), *s1′*)::≼(*G*, *L*); *wf-prog G*⟧
⟹ *Norm* (*lupd*(*vn*↦*Addr a*) *s1′*)::≼(*G*, *L*(*vn*↦*Class tn*))
⟨*proof*⟩

lemma *Fin-lemma*:
⟦*G*⊢*Norm s1* −*c2*→ (*x2*,*s2*); *wf-prog G*; (*Some a*, *s1*)::≼(*G*, *L*); (*x2*,*s2*)::≼(*G*, *L*);
dom (*locals s1*) ⊆ *dom* (*locals s2*)⟧
⟹ (*abrupt-if True* (*Some a*) *x2*, *s2*)::≼(*G*, *L*)
⟨*proof*⟩

lemma *FVar-lemma1*:
⟦*table-of* (*DeclConcepts.fields G statC*) (*fn*, *statDeclC*) = *Some f* ;
x2 = *None* ⟶ *G*,*s2*⊢*a*::≼ *Class statC*; *wf-prog G*; *G*⊢*statC*≼$_C$ *statDeclC*;
statDeclC ≠ *Object*;
class G statDeclC = *Some y*; (*x2*,*s2*)::≼(*G*, *L*); *s1*≤|*s2*;
inited statDeclC (*globs s1*);
(*if static f then id else np a*) *x2* = *None*⟧
⟹
∃ *obj*. *globs s2* (*if static f then Inr statDeclC else Inl* (*the-Addr a*))
= *Some obj* ∧
var-tys G (*tag obj*) (*if static f then Inr statDeclC else Inl*(*the-Addr a*))
(*Inl*(*fn*,*statDeclC*)) = *Some* (*type f*)
⟨*proof*⟩

lemma *FVar-lemma2*: *error-free state*
⟹ *error-free*
(*assign*
(λ*v*. *supd*
(*upd-gobj*
(*if static field then Inr statDeclC*
else Inl (*the-Addr a*))
(*Inl* (*fn*, *statDeclC*)) *v*))
w state)
⟨*proof*⟩

declare *split-paired-All* [*simp del*] *split-paired-Ex* [*simp del*]
declare *split-if* [*split del*] *split-if-asm* [*split del*]
option.split [*split del*] *option.split-asm* [*split del*]
⟨*ML*⟩

lemma *FVar-lemma*:
⟦(*v*, *f*), *Norm s2'*) = *fvar statDeclC* (*static field*) *fn a* (*x2*, *s2*);
G⊢*statC*≼$_C$ *statDeclC*;
table-of (*DeclConcepts.fields G statC*) (*fn*, *statDeclC*) = *Some field*;
wf-prog G;
x2 = *None* ⟶ *G*,*s2*⊢*a*::≼*Class statC*;
statDeclC ≠ *Object*; *class G statDeclC* = *Some y*;
(*x2*, *s2*)::≼(*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 *split-if* [*split*] *split-if-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*⊢*empty* (*pars mh*[↦]*pvs*)
[∼::≼]*table-of lvars*(*pars mh*[↦]*parTs sig*)
⟨*proof*⟩

lemma *lconf-map-lname* [*simp*]:
G,*s*⊢(*lname-case l1 l2*)[::≼](*lname-case L1 L2*)
=
(*G*,*s*⊢*l1*[::≼]*L1* ∧ *G*,*s*⊢(λ*x*::*unit* . *l2*)[::≼](λ*x*::*unit*. *L2*))
⟨*proof*⟩

lemma *wlconf-map-lname* [*simp*]:
G,*s*⊢(*lname-case l1 l2*)[∼::≼](*lname-case L1 L2*)
=
(*G*,*s*⊢*l1*[∼::≼]*L1* ∧ *G*,*s*⊢(λ*x*::*unit* . *l2*)[∼::≼](λ*x*::*unit*. *L2*))
⟨*proof*⟩

lemma *lconf-map-ename* [*simp*]:
G,*s*⊢(*ename-case l1 l2*)[::≼](*ename-case L1 L2*)
=
(*G*,*s*⊢*l1*[::≼]*L1* ∧ *G*,*s*⊢(λ*x*::*unit*. *l2*)[::≼](λ*x*::*unit*. *L2*))
⟨*proof*⟩

lemma *wlconf-map-ename* [*simp*]:
G,*s*⊢(*ename-case l1 l2*)[∼::≼](*ename-case L1 L2*)
=
(*G*,*s*⊢*l1*[∼::≼]*L1* ∧ *G*,*s*⊢(λ*x*::*unit*. *l2*)[∼::≼](λ*x*::*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 *split-if* [*split del*] *split-if-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 *split-if* [*split*] *split-if-asm* [*split*]
option.split [*split*] *option.split-asm* [*split*]

⟨*ML*⟩

47 accessibility

theorem *dynamic-field-access-ok*:
assumes *wf*: *wf-prog G* **and**
not-Null: ¬ *stat* ⟶ *a≠Null* **and**
conform-a: *G,(store s)⊢a::≼ Class statC* **and**
conform-s: *s::≼(G, L)* **and**
normal-s: *normal s* **and**
wt-e: (|*prg=G,cls=accC,lcl=L*|)*⊢e::−Class statC* **and**
f: *accfield G accC statC fn = Some f* **and**
dynC: *if stat then dynC=declclass f*
else dynC=obj-class (lookup-obj (store s) a) **and**
stat: *if stat then (is-static f) else (¬ is-static f)*
shows *table-of (DeclConcepts.fields G dynC) (fn,declclass f) = Some (fld f)*∧
G⊢Field fn f in dynC dyn-accessible-from accC

⟨*proof*⟩

lemma *error-free-field-access*:
assumes *accfield*: *accfield G accC statC fn = Some (statDeclC, f)* **and**
wt-e: (|*prg = G, cls = accC, lcl = L*|)*⊢e::−Class statC* **and**
eval-init: *G⊢Norm s0 −Init statDeclC→ s1* **and**
eval-e: *G⊢s1 −e−≻a→ s2* **and**
conf-s2: *s2::≼(G, L)* **and**
conf-a: *normal s2 ⟹ G, store s2⊢a::≼Class statC* **and**
fvar: *(v,s2′)=fvar statDeclC (is-static f) fn a s2* **and**
wf: *wf-prog G*
shows *check-field-access G accC statDeclC fn (is-static f) a s2′ = s2′*

⟨*proof*⟩

lemma *call-access-ok*:
assumes *invC-prop*: $G\vdash invmode\ statM\ e\rightarrow invC\preceq statT$
and *wf*: *wf-prog G*
and *wt-e*: $(\!|prg=G,cls=C,lcl=L|\!)\vdash e::-RefT\ statT$
and *statM*: $(statDeclT,statM)\in mheads\ G\ accC\ statT\ sig$
and *invC*: $invC = invocation\text{-}class\ (invmode\ statM\ e)\ s\ a\ statT$
shows $\exists\ dynM.\ dynlookup\ G\ statT\ invC\ sig = Some\ dynM\ \wedge$
$G\vdash Methd\ sig\ dynM\ in\ invC\ dyn\text{-}accessible\text{-}from\ accC$
⟨*proof*⟩

lemma *error-free-call-access*:
assumes
eval-args: $G\vdash s1\ -args\doteq\succ vs\rightarrow\ s2$ **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: $normal\ s1 \Longrightarrow G,\ store\ s1\vdash a::\preceq RefT\ statT$ **and**
invProp: $normal\ s3 \Longrightarrow$
$G\vdash invmode\ statM\ e\rightarrow invC\preceq statT$ **and**
s3: $s3=init\text{-}lvars\ G\ invDeclC\ (\!|name = mn,\ parTs = pTs'|\!)$
$(invmode\ statM\ e)\ a\ vs\ s2$ **and**
invC: $invC = invocation\text{-}class\ (invmode\ statM\ e)\ (store\ s2)\ a\ statT$**and**
invDeclC: $invDeclC = invocation\text{-}declclass\ G\ (invmode\ statM\ e)\ (store\ s2)$
$a\ statT\ (\!|name = mn,\ parTs = pTs'|\!)$ **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*:

$\bigwedge tab\ tab'\ ys\ z.\ [\![(tab(xs[\mapsto]ys))\ vn = Some\ z; tab\ vn \neq Some\ z]\!]$
$\quad \Longrightarrow\ (tab'(xs[\mapsto]ys))\ vn = Some\ z$
⟨*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*:
$\bigwedge\ tab\ tab'\ ys.\ [\![(tab(xs[\mapsto]ys))\ vn = Some\ el;\ (tab'(xs[\mapsto]ys))\ vn = None]\!]$
$\quad \Longrightarrow tab\ vn = Some\ el$
⟨*proof*⟩

lemma *dom-vname-split*:
$dom\ (lname\text{-}case\ (ename\text{-}case\ (tab(x\mapsto y)(xs[\mapsto]ys))\ a)\ b)$
$\quad = dom\ (lname\text{-}case\ (ename\text{-}case\ (tab(x\mapsto y))\ a)\ b) \cup$
$\quad\ dom\ (lname\text{-}case\ (ename\text{-}case\ (tab(xs[\mapsto]ys))\ a)\ b)$
(**is** *?List x xs y ys* = *?Hd x y* ∪ *?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-ename-case-None-simp*:
$dom\ (ename\text{-}case\ vname\text{-}tab\ None) = VNam\ `\ (dom\ vname\text{-}tab)$
⟨*proof*⟩

lemma *dom-ename-case-Some-simp*:
$dom\ (ename\text{-}case\ vname\text{-}tab\ (Some\ a)) = VNam\ `\ (dom\ vname\text{-}tab) \cup \{Res\}$
⟨*proof*⟩

lemma *dom-lname-case-None-simp*:

dom (lname-case ename-tab None) = EName ‘ (dom ename-tab)
⟨*proof*⟩

lemma *dom-lname-case-Some-simp*:
dom (lname-case ename-tab (Some a)) = EName ‘ (dom ename-tab) ∪ {This}
⟨*proof*⟩

lemmas *dom-lname-ename-case-simps =*
dom-ename-case-None-simp dom-ename-case-Some-simp
dom-lname-case-None-simp dom-lname-case-Some-simp

lemma *image-comp*:
f ‘ g ‘ A = (f ∘ 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*: $(\!|prg = G,\ cls = accC,\ lcl = L|\!)\vdash dom\ (locals\ (snd\ s0))\gg t\ll A$
and *wf*: *wf-prog G*
and *elim*: $[\![s1::\preceq(G,\ L);\ normal\ s1 \Longrightarrow G,L,snd\ s1\vdash t\succ v::\preceq T;$
$error\text{-}free\ s0 = error\text{-}free\ s1]\!] \Longrightarrow P$
shows *P*
⟨*proof*⟩

corollary *eval-ts*:
$[\![G\vdash s\ -e-\succ v\ \rightarrow\ s';\ wf\text{-}prog\ G;\ s::\preceq(G,L);\ (\!|prg=G,cls=C,lcl=L|\!)\vdash e::-T;$
$(\!|prg=G,cls=C,lcl=L|\!)\vdash dom\ (locals\ (store\ s))\gg In1l\ e\ll A]\!]$
$\Longrightarrow\ s'::\preceq(G,L)\ \wedge\ (normal\ s'\longrightarrow G,store\ s'\vdash v::\preceq T)\ \wedge$
$(error\text{-}free\ s = error\text{-}free\ s')$
⟨*proof*⟩

corollary *evals-ts*:
$[\![G\vdash s\ -es\doteq\succ vs\rightarrow\ s';\ wf\text{-}prog\ G;\ s::\preceq(G,L);\ (\!|prg=G,cls=C,lcl=L|\!)\vdash es::\doteq Ts;$
$(\!|prg=G,cls=C,lcl=L|\!)\vdash dom\ (locals\ (store\ s))\gg In3\ es\ll A]\!]$
$\Longrightarrow\ s'::\preceq(G,L)\ \wedge\ (normal\ s'\longrightarrow list\text{-}all2\ (conf\ G\ (store\ s'))\ vs\ Ts)\ \wedge$
$(error\text{-}free\ s = error\text{-}free\ s')$
⟨*proof*⟩

corollary *evar-ts*:
$[\![G\vdash s\ -v=\succ vf\rightarrow\ s';\ wf\text{-}prog\ G;\ s::\preceq(G,L);\ (\!|prg=G,cls=C,lcl=L|\!)\vdash v::=T;$
$(\!|prg=G,cls=C,lcl=L|\!)\vdash dom\ (locals\ (store\ s))\gg In2\ v\ll A]\!] \Longrightarrow$
$s'::\preceq(G,L)\ \wedge\ (normal\ s'\longrightarrow G,L,(store\ s')\vdash In2\ v\succ In2\ vf::\preceq Inl\ T)\ \wedge$
$(error\text{-}free\ s = error\text{-}free\ s')$
⟨*proof*⟩

theorem *exec-ts*:
$[\![G\vdash s\ -c\rightarrow\ s';\ wf\text{-}prog\ G;\ s::\preceq(G,L);\ (\!|prg=G,cls=C,lcl=L|\!)\vdash c::\surd;$
$(\!|prg=G,cls=C,lcl=L|\!)\vdash dom\ (locals\ (store\ s))\gg In1r\ c\ll A]\!]$
$\Longrightarrow s'::\preceq(G,L)\ \wedge\ (error\text{-}free\ s\longrightarrow error\text{-}free\ s')$
⟨*proof*⟩

lemma *wf-eval-Fin*:
assumes *wf*: *wf-prog G*
and *wt-c1*: $(\!|prg = G,\ cls = C,\ lcl = L|\!)\vdash In1r\ c1::Inl\ (PrimT\ Void)$
and *da-c1*: $(\!|prg=G,cls=C,lcl=L|\!)\vdash dom\ (locals\ (store\ (Norm\ s0)))\gg In1r\ c1\ll A$
and *conf-s0*: $Norm\ s0::\preceq(G,\ L)$
and *eval-c1*: $G\vdash Norm\ s0\ -c1\rightarrow\ (x1,s1)$
and *eval-c2*: $G\vdash Norm\ s1\ -c2\rightarrow\ s2$
and *s3*: $s3=abupd\ (abrupt\text{-}if\ (x1\neq None)\ x1)\ s2$
shows $G\vdash Norm\ s0\ -c1\ Finally\ c2\rightarrow\ s3$
⟨*proof*⟩

48 Ideas for the future

In the type soundness proof and the correctness proof of definite assignment we perform induction on the evaluation relation with the further preconditions that the term is welltyped and definitely assigned. During the proofs we have to establish the welltypedness and definite assignment of the subterms to be able to apply the induction hypothesis. So large parts of both proofs are the same work in propagating welltypedness and definite assignment. So we can derive a new induction rule for induction on the evaluation of a wellformed term, were these propagations is already done, once and forever. Then we can do the proofs with this rule and can enjoy the time we have saved. Here is a first and incomplete sketch of such a rule.

theorem *wellformed-eval-induct* [*consumes 4*, *case-names Abrupt Skip Expr Lab Comp If*]:
assumes *eval*: $G\vdash s0\ -t\succ\!\!\rightarrow\ (v,s1)$
and *wt*: $(\!|prg=G,cls=accC,lcl=L|\!)\vdash t::T$
and *da*: $(\!|prg=G,cls=accC,lcl=L|\!)\vdash dom\ (locals\ (store\ s0))\gg\!t\gg\!A$
and *wf*: *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*: $\bigwedge s\ L\ accC.\ P\ L\ accC\ (Norm\ s)\ \langle Skip\rangle_s\ \diamondsuit\ (Norm\ s)$
and *expr*: $\bigwedge e\ s0\ s1\ v\ L\ accC\ eT\ E.$
$[\![(\!|prg=G,cls=accC,lcl=L|\!)\vdash e::-eT;$
$(\!|prg=G,cls=accC,lcl=L|\!)$
$\vdash dom\ (locals\ (store\ ((Norm\ s0)::state)))\gg\langle e\rangle_e\gg E;$
$P\ L\ accC\ (Norm\ s0)\ \langle e\rangle_e\ \lfloor v\rfloor_e\ s1]\!]$
$\Longrightarrow\ P\ L\ accC\ (Norm\ s0)\ \langle Expr\ e\rangle_s\ \diamondsuit\ s1$
and *lab*: $\bigwedge c\ l\ s0\ s1\ L\ accC\ C.$
$[\![(\!|prg=G,cls=accC,lcl=L|\!)\vdash c::\surd;$
$(\!|prg=G,cls=accC,\ lcl=L|\!)$
$\vdash dom\ (locals\ (store\ ((Norm\ s0)::state)))\gg\langle c\rangle_s\gg C;$
$P\ L\ accC\ (Norm\ s0)\ \langle c\rangle_s\ \diamondsuit\ s1]\!]$
$\Longrightarrow P\ L\ accC\ (Norm\ s0)\ \langle l\bullet\ c\rangle_s\ \diamondsuit\ (abupd\ (absorb\ l)\ s1)$
and *comp*: $\bigwedge c1\ c2\ s0\ s1\ s2\ L\ accC\ C1.$
$[\![G\vdash Norm\ s0\ -c1\ \rightarrow\ s1;G\vdash s1\ -c2\ \rightarrow\ s2;$
$(\!|prg=G,cls=accC,lcl=L|\!)\vdash c1::\surd;$
$(\!|prg=G,cls=accC,lcl=L|\!)\vdash c2::\surd;$
$(\!|prg=G,cls=accC,lcl=L|\!)\vdash$
$dom\ (locals\ (store\ ((Norm\ s0)::state)))\gg\langle c1\rangle_s\gg\ C1;$
$P\ L\ accC\ (Norm\ s0)\ \langle c1\rangle_s\ \diamondsuit\ s1;$
$\bigwedge Q.\ [\![normal\ s1;$
$\bigwedge C2.[\![(\!|prg=G,cls=accC,lcl=L|\!)$
$\vdash dom\ (locals\ (store\ s1))\gg\langle c2\rangle_s\gg\ C2;$
$P\ L\ accC\ s1\ \langle c2\rangle_s\ \diamondsuit\ s2]\!]\Longrightarrow Q$
$]\!]\Longrightarrow Q$
$]\!]\Longrightarrow P\ L\ accC\ (Norm\ s0)\ \langle c1;;\ c2\rangle_s\ \diamondsuit\ s2$
and *if*: $\bigwedge b\ c1\ c2\ e\ s0\ s1\ s2\ L\ accC\ E.$
$[\![G\vdash Norm\ s0\ -e-\succ b\rightarrow\ s1;$
$G\vdash s1\ -(if\ the\text{-}Bool\ b\ then\ c1\ else\ c2)\rightarrow\ s2;$
$(\!|prg=G,cls=accC,lcl=L|\!)\vdash e::-PrimT\ Boolean;$
$(\!|prg=G,\ cls=accC,\ lcl=L|\!)\vdash(if\ the\text{-}Bool\ b\ then\ c1\ else\ c2)::\surd;$
$(\!|prg=G,cls=accC,lcl=L|\!)\vdash$
$dom\ (locals\ (store\ ((Norm\ s0)::state)))\gg\langle e\rangle_e\gg\ E;$
$P\ L\ accC\ (Norm\ s0)\ \langle e\rangle_e\ \lfloor b\rfloor_e\ s1;$
$\bigwedge Q.\ [\![normal\ s1;$
$\bigwedge C.\ [\![(\!|prg=G,cls=accC,lcl=L|\!)\vdash\ (dom\ (locals\ (store\ s1)))$
$\gg\langle if\ the\text{-}Bool\ b\ then\ c1\ else\ c2\rangle_s\gg\ C;$
$P\ L\ accC\ s1\ \langle if\ the\text{-}Bool\ b\ then\ c1\ else\ c2\rangle_s\ \diamondsuit\ s2$
$]\!]\Longrightarrow Q$
$]\!]\Longrightarrow Q$
$]\!]\Longrightarrow P\ L\ accC\ (Norm\ s0)\ \langle If(e)\ c1\ Else\ c2\rangle_s\ \diamondsuit\ s2$
shows $P\ L\ accC\ s0\ t\ v\ s1$
⟨*proof*⟩

end

Chapter 20

Evaln

49 Operational evaluation (big-step) semantics of Java expressions and statements

theory *Evaln* **imports** *TypeSafe* **begin**

Variant of *eval* relation with counter for bounded recursive depth. In principal *evaln* could replace *eval*.

Validity of the axiomatic semantics builds on *evaln*. For recursive method calls the axiomatic semantics rule assumes the method ok to derive a proof for the body. To prove the method rule sound we need to perform induction on the recursion depth. For the completeness proof of the axiomatic semantics the notion of the most general formula is used. The most general formula right now builds on the ordinary evaluation relation *eval*. So sometimes we have to switch between *evaln* and *eval* and vice versa. To make this switch easy *evaln* also does all the technical accessibility tests *check-field-access* and *check-method-access* like *eval*. If it would omit them *evaln* and *eval* would only be equivalent for welltyped, and definitely assigned terms.

inductive
  evaln :: [*prog*, *state*, *term*, *nat*, *vals*, *state*] ⇒ *bool*
    (⊢- −-≻−-→ ′(-, -′) [61,61,80,61,0,0] 60)
  and *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*
    (⊢- −-�doteq≻-−-→ - [61,61,61,61,61,61] 60)
  and *execn* :: [*prog*, *state*, *stmt*, *nat*, *state*] ⇒ *bool*
    (⊢- −-−-→ - [61,61,65, 61,61] 60)
  for *G* :: *prog*
where

  G⊢*s* −*c* −*n*→ *s′* ≡ *G*⊢*s* −*In1r* *c*≻−*n*→ (◇ , *s′*)
| *G*⊢*s* −*e*−≻*v* −*n*→ *s′* ≡ *G*⊢*s* −*In1l* *e*≻−*n*→ (*In1 v* , *s′*)
| *G*⊢*s* −*e*=≻*vf* −*n*→ *s′* ≡ *G*⊢*s* −*In2* *e*≻−*n*→ (*In2 vf*, *s′*)
| *G*⊢*s* −*e*≐≻*v* −*n*→ *s′* ≡ *G*⊢*s* −*In3* *e*≻−*n*→ (*In3 v* , *s′*)

— propagation of abrupt completion

| *Abrupt*: *G*⊢(*Some xc*,*s*) −*t*≻−*n*→ (*undefined3 t*,(*Some xc*,*s*))

— evaluation of variables

| *LVar*: *G*⊢*Norm s* −*LVar vn*=≻*lvar vn s*−*n*→ *Norm s*

| *FVar*: ⟦*G*⊢*Norm s0* −*Init statDeclC*−*n*→ *s1*; *G*⊢*s1* −*e*−≻*a*−*n*→ *s2*;
        (*v*,*s2′*) = *fvar statDeclC stat fn a s2*;
        s3 = *check-field-access G accC statDeclC fn stat a s2′*⟧ ⟹
        G⊢*Norm s0* −{*accC*,*statDeclC*,*stat*}*e*..*fn*=≻*v*−*n*→ *s3*

| *AVar*: ⟦*G*⊢ *Norm s0* −*e1*−≻*a*−*n*→ *s1* ; *G*⊢*s1* −*e2*−≻*i*−*n*→ *s2*;
        (*v*,*s2′*) = *avar G i a s2*⟧ ⟹
                G⊢*Norm s0* −*e1*.[*e2*]=≻*v*−*n*→ *s2′*

— evaluation of expressions

| *NewC*: ⟦*G*⊢*Norm s0* −*Init C*−*n*→ *s1*;

$$G\vdash \quad s1\ -halloc\ (CInst\ C)\succ a\rightarrow\ s2\rrbracket \Longrightarrow G\vdash Norm\ s0\ -NewC\ C\succ Addr\ a-n\rightarrow\ s2$$

| *NewA*: $\llbracket G\vdash Norm\ s0\ -init\text{-}comp\text{-}ty\ T-n\rightarrow\ s1;\ G\vdash s1\ -e\succ i'-n\rightarrow\ s2;$
$G\vdash abupd\ (check\text{-}neg\ i')\ s2\ -halloc\ (Arr\ T\ (the\text{-}Intg\ i'))\succ a\rightarrow\ s3\rrbracket \Longrightarrow$
$G\vdash Norm\ s0\ -New\ T[e]\succ Addr\ a-n\rightarrow\ s3$

| *Cast*: $\llbracket G\vdash Norm\ s0\ -e\succ v-n\rightarrow\ s1;$
$s2 = abupd\ (raise\text{-}if\ (\neg G,snd\ s1\vdash v\ fits\ T)\ ClassCast)\ s1\rrbracket \Longrightarrow$
$G\vdash Norm\ s0\ -Cast\ T\ e\succ v-n\rightarrow\ s2$

| *Inst*: $\llbracket G\vdash Norm\ s0\ -e\succ v-n\rightarrow\ s1;$
$b = (v\neq Null\ \wedge\ G,store\ s1\vdash v\ fits\ RefT\ T)\rrbracket \Longrightarrow$
$G\vdash Norm\ s0\ -e\ InstOf\ T\succ Bool\ b-n\rightarrow\ s1$

| *Lit*: $G\vdash Norm\ s\ -Lit\ v\succ v-n\rightarrow\ Norm\ s$

| *UnOp*: $\llbracket G\vdash Norm\ s0\ -e\succ v-n\rightarrow\ s1\rrbracket$
$\Longrightarrow G\vdash Norm\ s0\ -UnOp\ unop\ e\succ(eval\text{-}unop\ unop\ v)-n\rightarrow\ s1$

| *BinOp*: $\llbracket G\vdash Norm\ s0\ -e1\succ v1-n\rightarrow\ s1;$
$G\vdash s1\ -(if\ need\text{-}second\text{-}arg\ binop\ v1\ then\ (In1l\ e2)\ else\ (In1r\ Skip))$
$\succ -n\rightarrow\ (In1\ v2,s2)\rrbracket$
$\Longrightarrow G\vdash Norm\ s0\ -BinOp\ binop\ e1\ e2\succ(eval\text{-}binop\ binop\ v1\ v2)-n\rightarrow\ s2$

| *Super*: $G\vdash Norm\ s\ -Super\succ val\text{-}this\ s-n\rightarrow\ Norm\ s$

| *Acc*: $\llbracket G\vdash Norm\ s0\ -va=\succ(v,f)-n\rightarrow\ s1\rrbracket \Longrightarrow$
$G\vdash Norm\ s0\ -Acc\ va\succ v-n\rightarrow\ s1$

| *Ass*: $\llbracket G\vdash Norm\ s0\ -va=\succ(w,f)-n\rightarrow\ s1;$
$G\vdash \quad s1\ -e\succ v\quad -n\rightarrow\ s2\rrbracket \Longrightarrow$
$G\vdash Norm\ s0\ -va:=e\succ v-n\rightarrow\ assign\ f\ v\ s2$

| *Cond*: $\llbracket G\vdash Norm\ s0\ -e0\succ b-n\rightarrow\ s1;$
$G\vdash \quad s1\ -(if\ the\text{-}Bool\ b\ then\ e1\ else\ e2)\succ v-n\rightarrow\ s2\rrbracket \Longrightarrow$
$G\vdash Norm\ s0\ -e0\ ?\ e1\ :\ e2\succ v-n\rightarrow\ s2$

| *Call*:
$\llbracket G\vdash Norm\ s0\ -e\succ a'-n\rightarrow\ s1;\ G\vdash s1\ -args\doteq\succ vs-n\rightarrow\ s2;$
$D = invocation\text{-}declclass\ G\ mode\ (store\ s2)\ a'\ statT\ (\!|name=mn,parTs=pTs|\!);$
$s3=init\text{-}lvars\ G\ D\ (\!|name=mn,parTs=pTs|\!)\ mode\ a'\ vs\ s2;$
$s3' = check\text{-}method\text{-}access\ G\ accC\ statT\ mode\ (\!|name=mn,parTs=pTs|\!)\ a'\ s3;$
$G\vdash s3'-Methd\ D\ (\!|name=mn,parTs=pTs|\!)\succ v-n\rightarrow\ s4$
$\rrbracket$
$\Longrightarrow$
$G\vdash Norm\ s0\ -\{accC,statT,mode\}e\cdot mn(\{pTs\}args)\succ v-n\rightarrow\ (restore\text{-}lvars\ s2\ s4)$

| *Methd*: $\llbracket G\vdash Norm\ s0\ -body\ G\ D\ sig\succ v-n\rightarrow\ s1\rrbracket \Longrightarrow$
$G\vdash Norm\ s0\ -Methd\ D\ sig\succ v-Suc\ n\rightarrow\ s1$

| *Body*: $\llbracket G\vdash Norm\ s0-Init\ D-n\rightarrow\ s1;\ G\vdash s1\ -c-n\rightarrow\ s2;$
$s3 = (if\ (\exists\ l.\ abrupt\ s2 = Some\ (Jump\ (Break\ l))\ \vee$
$abrupt\ s2 = Some\ (Jump\ (Cont\ l)))$
$then\ abupd\ (\lambda\ x.\ Some\ (Error\ CrossMethodJump))\ s2$
$else\ s2)\rrbracket\Longrightarrow$
$G\vdash Norm\ s0\ -Body\ D\ c$
$\succ the\ (locals\ (store\ s2)\ Result)-n\rightarrow abupd\ (absorb\ Ret)\ s3$

— evaluation of expression lists

| *Nil*:
$$G\vdash Norm\ s0\ -[]\doteq\succ[]-n\rightarrow\ Norm\ s0$$

| *Cons*: $[\![G\vdash Norm\ s0\ -e\ -\succ\ v\ -n\rightarrow\ s1;$
$G\vdash\quad s1\ -es\doteq\succ vs-n\rightarrow\ s2]\!] \Longrightarrow$
$$G\vdash Norm\ s0\ -e\#es\doteq\succ v\#vs-n\rightarrow\ s2$$

— execution of statements

| *Skip*: $G\vdash Norm\ s\ -Skip-n\rightarrow\ Norm\ s$

| *Expr*: $[\![G\vdash Norm\ s0\ -e-\succ v-n\rightarrow\ s1]\!] \Longrightarrow$
$G\vdash Norm\ s0\ -Expr\ e-n\rightarrow\ s1$

| *Lab*: $[\![G\vdash Norm\ s0\ -c\ -n\rightarrow\ s1]\!] \Longrightarrow$
$G\vdash Norm\ s0\ -l\bullet\ c-n\rightarrow\ abupd\ (absorb\ l)\ s1$

| *Comp*: $[\![G\vdash Norm\ s0\ -c1\ -n\rightarrow\ s1;$
$G\vdash\quad s1\ -c2\ -n\rightarrow\ s2]\!] \Longrightarrow$
$G\vdash Norm\ s0\ -c1;;\ c2-n\rightarrow\ s2$

| *If*: $[\![G\vdash Norm\ s0\ -e-\succ b-n\rightarrow\ s1;$
$G\vdash\quad s1-(if\ the\text{-}Bool\ b\ then\ c1\ else\ c2)-n\rightarrow\ s2]\!] \Longrightarrow$
$G\vdash Norm\ s0\ -If(e)\ c1\ Else\ c2\ -n\rightarrow\ s2$

| *Loop*: $[\![G\vdash Norm\ s0\ -e-\succ b-n\rightarrow\ s1;$
$if\ the\text{-}Bool\ b$
$then\ (G\vdash s1\ -c-n\rightarrow\ s2\ \wedge$
$G\vdash(abupd\ (absorb\ (Cont\ l))\ s2)\ -l\bullet\ While(e)\ c-n\rightarrow\ s3)$
$else\ s3\ =\ s1]\!] \Longrightarrow$
$G\vdash Norm\ s0\ -l\bullet\ While(e)\ c-n\rightarrow\ s3$

| *Jmp*: $G\vdash Norm\ s\ -Jmp\ j-n\rightarrow\ (Some\ (Jump\ j),\ s)$

| *Throw*:$[\![G\vdash Norm\ s0\ -e-\succ a'-n\rightarrow\ s1]\!] \Longrightarrow$
$G\vdash Norm\ s0\ -Throw\ e-n\rightarrow\ abupd\ (throw\ a')\ s1$

| *Try*: $[\![G\vdash Norm\ s0\ -c1-n\rightarrow\ s1;\ G\vdash s1\ -sxalloc\rightarrow\ s2;$
$if\ G,s2\vdash catch\ tn\ then\ G\vdash new\text{-}xcpt\text{-}var\ vn\ s2\ -c2-n\rightarrow\ s3\ else\ s3\ =\ s2]\!]$
$\Longrightarrow$
$G\vdash Norm\ s0\ -Try\ c1\ Catch(tn\ vn)\ c2-n\rightarrow\ s3$

| *Fin*: $[\![G\vdash Norm\ s0\ -c1-n\rightarrow\ (x1,s1);$
$G\vdash Norm\ s1\ -c2-n\rightarrow\ s2;$
$s3=(if\ (\exists\ err.\ x1=Some\ (Error\ err))$
$then\ (x1,s1)$
$else\ abupd\ (abrupt\text{-}if\ (x1\neq None)\ x1)\ s2)]\!] \Longrightarrow$
$G\vdash Norm\ s0\ -c1\ Finally\ c2-n\rightarrow\ s3$

| *Init*: $[\![the\ (class\ G\ C)\ =\ c;$
$if\ inited\ C\ (globs\ s0)\ then\ s3\ =\ Norm\ s0$
$else\ (G\vdash Norm\ (init\text{-}class\text{-}obj\ G\ C\ s0)$
$-(if\ C\ =\ Object\ then\ Skip\ else\ Init\ (super\ c))-n\rightarrow\ s1\ \wedge$
$G\vdash set\text{-}lvars\ empty\ s1\ -init\ c-n\rightarrow\ s2\ \wedge$
$s3\ =\ restore\text{-}lvars\ s1\ s2)]\!]$
$\Longrightarrow$

$G\vdash Norm\ s0\ -Init\ C-n\rightarrow\ s3$

monos
if-bool-eq-conj

declare *split-if* [*split del*] *split-if-asm* [*split del*]
option.split [*split del*] *option.split-asm* [*split del*]
not-None-eq [*simp del*]
split-paired-All [*simp del*] *split-paired-Ex* [*simp del*]
⟨*ML*⟩

inductive-cases *evaln-cases*: $G\vdash s\ -t\succ-n\rightarrow\ (v,\ s')$

inductive-cases *evaln-elim-cases*:
$G\vdash(Some\ xc,\ s)\ -t\succ-n\rightarrow\ (v,\ s')$
$G\vdash Norm\ s\ -In1r\ Skip\succ-n\rightarrow\ (x,\ s')$
$G\vdash Norm\ s\ -In1r\ (Jmp\ j)\succ-n\rightarrow\ (x,\ s')$
$G\vdash Norm\ s\ -In1r\ (l\cdot\ c)\succ-n\rightarrow\ (x,\ s')$
$G\vdash Norm\ s\ -In3\ ([])\succ-n\rightarrow\ (v,\ s')$
$G\vdash Norm\ s\ -In3\ (e\#es)\succ-n\rightarrow\ (v,\ s')$
$G\vdash Norm\ s\ -In1l\ (Lit\ w)\succ-n\rightarrow\ (v,\ s')$
$G\vdash Norm\ s\ -In1l\ (UnOp\ unop\ e)\succ-n\rightarrow\ (v,\ s')$
$G\vdash Norm\ s\ -In1l\ (BinOp\ binop\ e1\ e2)\succ-n\rightarrow\ (v,\ s')$
$G\vdash Norm\ s\ -In2\ (LVar\ vn)\succ-n\rightarrow\ (v,\ s')$
$G\vdash Norm\ s\ -In1l\ (Cast\ T\ e)\succ-n\rightarrow\ (v,\ s')$
$G\vdash Norm\ s\ -In1l\ (e\ InstOf\ T)\succ-n\rightarrow\ (v,\ s')$
$G\vdash Norm\ s\ -In1l\ (Super)\succ-n\rightarrow\ (v,\ s')$
$G\vdash Norm\ s\ -In1l\ (Acc\ va)\succ-n\rightarrow\ (v,\ s')$
$G\vdash Norm\ s\ -In1r\ (Expr\ e)\succ-n\rightarrow\ (x,\ s')$
$G\vdash Norm\ s\ -In1r\ (c1;;\ c2)\succ-n\rightarrow\ (x,\ s')$
$G\vdash Norm\ s\ -In1l\ (Methd\ C\ sig)\succ-n\rightarrow\ (x,\ s')$
$G\vdash Norm\ s\ -In1l\ (Body\ D\ c)\succ-n\rightarrow\ (x,\ s')$
$G\vdash Norm\ s\ -In1l\ (e0\ ?\ e1\ :\ e2)\succ-n\rightarrow\ (v,\ s')$
$G\vdash Norm\ s\ -In1r\ (If(e)\ c1\ Else\ c2)\succ-n\rightarrow\ (x,\ s')$
$G\vdash Norm\ s\ -In1r\ (l\cdot\ While(e)\ c)\succ-n\rightarrow\ (x,\ s')$
$G\vdash Norm\ s\ -In1r\ (c1\ Finally\ c2)\succ-n\rightarrow\ (x,\ s')$
$G\vdash Norm\ s\ -In1r\ (Throw\ e)\succ-n\rightarrow\ (x,\ s')$
$G\vdash Norm\ s\ -In1l\ (NewC\ C)\succ-n\rightarrow\ (v,\ s')$
$G\vdash Norm\ s\ -In1l\ (New\ T[e])\succ-n\rightarrow\ (v,\ s')$
$G\vdash Norm\ s\ -In1l\ (Ass\ va\ e)\succ-n\rightarrow\ (v,\ s')$
$G\vdash Norm\ s\ -In1r\ (Try\ c1\ Catch(tn\ vn)\ c2)\succ-n\rightarrow\ (x,\ s')$
$G\vdash Norm\ s\ -In2\ (\{accC,statDeclC,stat\}e..fn)\succ-n\rightarrow\ (v,\ s')$
$G\vdash Norm\ s\ -In2\ (e1.[e2])\succ-n\rightarrow\ (v,\ s')$
$G\vdash Norm\ s\ -In1l\ (\{accC,statT,mode\}e\cdot mn(\{pT\}p))\succ-n\rightarrow\ (v,\ s')$
$G\vdash Norm\ s\ -In1r\ (Init\ C)\succ-n\rightarrow\ (x,\ s')$
$G\vdash Norm\ s\ -In1r\ (Init\ C)\succ-n\rightarrow\ (x,\ s')$

declare *split-if* [*split*] *split-if-asm* [*split*]
option.split [*split*] *option.split-asm* [*split*]
not-None-eq [*simp*]
split-paired-All [*simp*] *split-paired-Ex* [*simp*]
⟨*ML*⟩

lemma *evaln-Inj-elim*: $G\vdash s\ -t\succ-n\rightarrow\ (w,s')\Longrightarrow case\ t\ of\ In1\ ec\Rightarrow$
(*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 *evaln-expr-eq*: $G\vdash s\ -In1l\ t\succ -n\rightarrow\ (w,\ s')\ =\ (\exists\, v.\ w=In1\ v\ \wedge\ G\vdash s\ -t-\succ v\ -n\rightarrow\ s')$
⟨*proof*⟩

lemma *evaln-var-eq*: $G\vdash s\ -In2\ t\succ -n\rightarrow\ (w,\ s')\ =\ (\exists\, vf.\ w=In2\ vf\ \wedge\ G\vdash s\ -t=\succ vf-n\rightarrow\ s')$
⟨*proof*⟩

lemma *evaln-exprs-eq*: $G\vdash s\ -In3\ t\succ -n\rightarrow\ (w,\ s')\ =\ (\exists\, vs.\ w=In3\ vs\ \wedge\ G\vdash s\ -t\doteq\succ vs-n\rightarrow\ s')$
⟨*proof*⟩

lemma *evaln-stmt-eq*: $G\vdash s\ -In1r\ t\succ -n\rightarrow\ (w,\ s')\ =\ (w=\diamondsuit\ \wedge\ G\vdash s\ -t\ -n\rightarrow\ s')$
⟨*proof*⟩

⟨*ML*⟩
declare *evaln-AbruptIs* [*intro*!]

lemma *evaln-Callee*: $G\vdash Norm\ s-In1l\ (Callee\ l\ e)\succ -n\rightarrow\ (v,s')\ =\ False$
⟨*proof*⟩

lemma *evaln-InsInitE*: $G\vdash Norm\ s-In1l\ (InsInitE\ c\ e)\succ -n\rightarrow\ (v,s')\ =\ False$
⟨*proof*⟩

lemma *evaln-InsInitV*: $G\vdash Norm\ s-In2\ (InsInitV\ c\ w)\succ -n\rightarrow\ (v,s')\ =\ False$
⟨*proof*⟩

lemma *evaln-FinA*: $G\vdash Norm\ s-In1r\ (FinA\ a\ c)\succ -n\rightarrow\ (v,s')\ =\ False$
⟨*proof*⟩

lemma *evaln-abrupt-lemma*: $G\vdash s\ -e\succ -n\rightarrow\ (v,s')\ \Longrightarrow$
$fst\ s\ =\ Some\ xc\ \longrightarrow\ s'=s\ \wedge\ v\ =\ undefined3\ e$
⟨*proof*⟩

lemma *evaln-abrupt*:
$\bigwedge s'.\ G\vdash(Some\ xc,s)\ -e\succ -n\rightarrow\ (w,s')\ =\ (s'\ =\ (Some\ xc,s)\ \wedge$
$w=undefined3\ e\ \wedge\ G\vdash(Some\ xc,s)\ -e\succ -n\rightarrow\ (undefined3\ e,(Some\ xc,s)))$
⟨*proof*⟩

⟨*ML*⟩

lemma *evaln-LitI*: $G\vdash s\ -Lit\ v-\succ(if\ normal\ s\ then\ v\ else\ undefined)-n\rightarrow\ s$
⟨*proof*⟩

lemma *CondI*:
$\bigwedge s1.\ [\![G\vdash s\ -e-\succ b-n\rightarrow\ s1;\ G\vdash s1\ -(if\ the\text{-}Bool\ b\ then\ e1\ else\ e2)-\succ v-n\rightarrow\ s2]\!]\ \Longrightarrow$
$G\vdash s\ -e\ ?\ e1\ :\ e2-\succ(if\ normal\ s1\ then\ v\ else\ undefined)-n\rightarrow\ s2$

⟨*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*:
$⟦G⊢s\ -e-≻v-n→\ s1;\ G⊢s1\ -(if\ the\text{-}Bool\ v\ then\ c1\ else\ c2)-n→\ s2⟧ ⟹$
$G⊢s\ -If(e)\ c1\ Else\ c2-n→\ s2$
⟨*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*: $⟦Suc\ n <= m'; (⋀m.\ n <= m ⟹ P\ (Suc\ m))\ ⟧ ⟹ P\ m'$
⟨*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*:

$[\![G\vdash s1\ -t1\succ -n1\rightarrow\ (w1,\ s1')\ ;\ G\vdash s2\ -t2\succ -n2\rightarrow\ (w2,\ s2')\ ;\ G\vdash s3\ -t3\succ -n3\rightarrow\ (w3,\ s3')]\!] \Longrightarrow$
$G\vdash s1\ -t1\succ -max\ (max\ n1\ n2)\ n3\rightarrow\ (w1,\ s1')\ \wedge$
$G\vdash s2\ -t2\succ -max\ (max\ n1\ n2)\ n3\rightarrow\ (w2,\ s2')\ \wedge$
$G\vdash s3\ -t3\succ -max\ (max\ n1\ n2)\ n3\rightarrow\ (w3,\ s3')$
⟨*proof*⟩

corollary *evaln-max3E*:
$[\![G\vdash s1\ -t1\succ -n1\rightarrow\ (w1,\ s1')\ ;\ G\vdash s2\ -t2\succ -n2\rightarrow\ (w2,\ s2')\ ;\ G\vdash s3\ -t3\succ -n3\rightarrow\ (w3,\ s3')\ ;$
$[\![G\vdash s1\ -t1\succ -max\ (max\ n1\ n2)\ n3\rightarrow\ (w1,\ s1')\ ;$
$G\vdash s2\ -t2\succ -max\ (max\ n1\ n2)\ n3\rightarrow\ (w2,\ s2')\ ;$
$G\vdash s3\ -t3\succ -max\ (max\ n1\ n2)\ n3\rightarrow\ (w3,\ s3')$
$]\!] \Longrightarrow P$
$]\!] \Longrightarrow P$
⟨*proof*⟩

lemma *le-max3I1*: $(n2::nat) \leq max\ n1\ (max\ n2\ n3)$
⟨*proof*⟩

lemma *le-max3I2*: $(n3::nat) \leq max\ n1\ (max\ n2\ n3)$
⟨*proof*⟩

declare [[*simproc del*: *wt-expr wt-var wt-exprs wt-stmt*]]

eval implies evaln

lemma *eval-evaln*:
assumes *eval*: $G\vdash s0\ -t\succ\rightarrow\ (v,s1)$
shows $\exists\, n.\ G\vdash s0\ -t\succ -n\rightarrow\ (v,s1)$
⟨*proof*⟩

end

Chapter 21

Trans

theory *Trans* **imports** *Evaln* **begin**

constdefs *groundVar*:: *var* ⇒ *bool*
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* [*consumes 1*, *case-names LVar FVar AVar*]:
assumes *ground*: *groundVar v* **and**
LVar: ⋀ *ln*. ⟦*v*=*LVar ln*⟧ ⟹ *P* **and**
FVar: ⋀ *accC statDeclC stat a fn*.
⟦*v*={*accC*,*statDeclC*,*stat*}(*Lit a*)..*fn*⟧ ⟹ *P* **and**
AVar: ⋀ *a i*. ⟦*v*=(*Lit a*).[*Lit i*]⟧ ⟹ *P*
shows *P*
⟨*proof*⟩

constdefs *groundExprs*:: *expr list* ⇒ *bool*
groundExprs es ≡ *list-all* (λ *e*. ∃ *v*. *e*=*Lit v*) *es*

consts *the-val*:: *expr* ⇒ *val*
primrec
the-val (*Lit v*) = *v*

consts *the-var*:: *prog* ⇒ *state* ⇒ *var* ⇒ (*vvar* × *state*)
primrec
the-var G s (*LVar ln*) =(*lvar ln* (*store s*),*s*)
the-var-FVar-def:
the-var G s ({*accC*,*statDeclC*,*stat*}*a*..*fn*) =*fvar statDeclC stat fn* (*the-val a*) *s*
the-var-AVar-def:
the-var G s(*a*.[*i*]) =*avar G* (*the-val i*) (*the-val a*) *s*

lemma *the-var-FVar-simp*[*simp*]:
the-var G s ({*accC*,*statDeclC*,*stat*}(*Lit a*)..*fn*) = *fvar statDeclC stat fn a s*
⟨*proof*⟩
declare *the-var-FVar-def* [*simp del*]

lemma *the-var-AVar-simp*:

the-var G s ((Lit a).[Lit i]) = avar G i a s
⟨*proof*⟩
declare *the-var-AVar-def* [*simp del*]

syntax (*xsymbols*)
Ref :: *loc* ⇒ *expr*
SKIP :: *expr*

translations
Ref a == *Lit* (*Addr a*)
SKIP == *Lit Unit*

inductive
step :: [*prog*,*term* × *state*,*term* × *state*] ⇒ *bool* (-⊢- ↦*1* -[*61*,*82*,*82*] *81*)
for *G* :: *prog*
where

Abrupt: ⟦∀ *v*. *t* ≠ ⟨*Lit v*⟩;
∀ *t*. *t* ≠ ⟨*l*• *Skip*⟩;
∀ *C vn c*. *t* ≠ ⟨*Try Skip Catch*(*C vn*) *c*⟩;
∀ *x c*. *t* ≠ ⟨*Skip Finally c*⟩ ∧ *xc* ≠ *Xcpt x*;
∀ *a c*. *t* ≠ ⟨*FinA a c*⟩⟧
⟹
G⊢(*t*,*Some xc*,*s*) ↦*1* (⟨*Lit undefined*⟩,*Some xc*,*s*)

| *InsInitE*: ⟦*G*⊢(⟨*c*⟩,*Norm s*) ↦*1* (⟨*c*′⟩, *s*′)⟧
⟹
G⊢(⟨*InsInitE c e*⟩,*Norm s*) ↦*1* (⟨*InsInitE c*′ *e*⟩, *s*′)

| *NewC*: *G*⊢(⟨*NewC C*⟩,*Norm s*) ↦*1* (⟨*InsInitE* (*Init C*) (*NewC C*)⟩, *Norm s*)
| *NewCInited*: ⟦*G*⊢ *Norm s* −*halloc* (*CInst C*)≻*a*→ *s*′⟧
⟹
G⊢(⟨*InsInitE Skip* (*NewC C*)⟩,*Norm s*) ↦*1* (⟨*Ref a*⟩, *s*′)

| *NewA*:
G⊢(⟨*New T*[*e*]⟩,*Norm s*) ↦*1* (⟨*InsInitE* (*init-comp-ty T*) (*New T*[*e*])⟩,*Norm s*)
| *InsInitNewAIdx*:
⟦*G*⊢(⟨*e*⟩,*Norm s*) ↦*1* (⟨*e*′⟩, *s*′)⟧
⟹
G⊢(⟨*InsInitE Skip* (*New T*[*e*])⟩,*Norm s*) ↦*1* (⟨*InsInitE Skip* (*New T*[*e*′])⟩,*s*′)
| *InsInitNewA*:
⟦*G*⊢*abupd* (*check-neg i*) (*Norm s*) −*halloc* (*Arr T* (*the-Intg i*))≻*a*→ *s*′ ⟧
⟹
G⊢(⟨*InsInitE Skip* (*New T*[*Lit i*])⟩,*Norm s*) ↦*1* (⟨*Ref a*⟩,*s*′)

| *CastE*:
⟦*G*⊢(⟨*e*⟩,*Norm s*) ↦*1* (⟨*e*′⟩,*s*′)⟧
⟹

$G\vdash(\langle Cast\ T\ e\rangle,None,s) \mapsto 1\ (\langle Cast\ T\ e'\rangle,s')$

| *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*: $[\![G\vdash(\langle e\rangle,Norm\ s) \mapsto 1\ (\langle e'\rangle,s')]\!]$
$$\Longrightarrow$$
$$G\vdash(\langle\{accC,statT,mode\}e{\cdot}mn(\{pTs\}args)\rangle,Norm\ s)$$
$$\mapsto 1\ (\langle\{accC,statT,mode\}e'{\cdot}mn(\{pTs\}args)\rangle,s')$$

| *CallArgs*: $[\![G\vdash(\langle args\rangle,Norm\ s) \mapsto 1\ (\langle args'\rangle,s')]\!]$
$$\Longrightarrow$$
$$G\vdash(\langle\{accC,statT,mode\}Lit\ a{\cdot}mn(\{pTs\}args)\rangle,Norm\ s)$$
$$\mapsto 1\ (\langle\{accC,statT,mode\}Lit\ a{\cdot}mn(\{pTs\}args')\rangle,s')$$

| *Call*: $[\![groundExprs\ args;\ vs = map\ the\text{-}val\ args;$
$D = invocation\text{-}declclass\ G\ mode\ s\ a\ statT\ (\!|name{=}mn,parTs{=}pTs|\!);$
$s'{=}init\text{-}lvars\ G\ D\ (\!|name{=}mn,parTs{=}pTs|\!)\ mode\ a'\ vs\ (Norm\ s)]\!]$
$$\Longrightarrow$$
$$G\vdash(\langle\{accC,statT,mode\}Lit\ a{\cdot}mn(\{pTs\}args)\rangle,Norm\ s)$$
$$\mapsto 1\ (\langle Callee\ (locals\ s)\ (Methd\ D\ (\!|name{=}mn,parTs{=}pTs|\!))\rangle,s')$$

| *Callee*: $[\![G\vdash(\langle e\rangle,Norm\ s) \mapsto 1\ (\langle e'::expr\rangle,s')]\!]$
$$\Longrightarrow$$
$$G\vdash(\langle Callee\ lcls\text{-}caller\ e\rangle,Norm\ s) \mapsto 1\ (\langle e'\rangle,s')$$

| *CalleeRet*: $G\vdash(\langle Callee\ lcls\text{-}caller\ (Lit\ v)\rangle,Norm\ s)$
$$\mapsto 1\ (\langle Lit\ v\rangle,(set\text{-}lvars\ lcls\text{-}caller\ (Norm\ s)))$$

| *Methd*: $G\vdash(\langle Methd\ D\ sig\rangle,Norm\ s) \mapsto 1\ (\langle body\ G\ D\ sig\rangle,Norm\ s)$

| *Body*: $G\vdash(\langle Body\ D\ c\rangle,Norm\ s) \mapsto 1\ (\langle InsInitE\ (Init\ D)\ (Body\ D\ c)\rangle,Norm\ s)$

| *InsInitBody*:
$[\![G\vdash(\langle c\rangle,Norm\ s) \mapsto 1\ (\langle c'\rangle,s')]\!]$
$$\Longrightarrow$$
$$G\vdash(\langle InsInitE\ Skip\ (Body\ D\ c)\rangle,Norm\ s) \mapsto 1(\langle InsInitE\ Skip\ (Body\ D\ c')\rangle,s')$$

| *InsInitBodyRet*:
$G\vdash(\langle InsInitE\ Skip\ (Body\ D\ Skip)\rangle,Norm\ s)$
$$\mapsto 1\ (\langle Lit\ (the\ ((locals\ s)\ Result))\rangle,abupd\ (absorb\ Ret)\ (Norm\ s))$$

| *FVar*: $[\![\neg\ inited\ statDeclC\ (globs\ s)]\!]$
$$\Longrightarrow$$
$$G\vdash(\langle\{accC,statDeclC,stat\}e..fn\rangle,Norm\ s)$$
$$\mapsto 1\ (\langle InsInitV\ (Init\ statDeclC)\ (\{accC,statDeclC,stat\}e..fn)\rangle,Norm\ s)$$

| *InsInitFVarE*:
$[\![G\vdash(\langle e\rangle,Norm\ s) \mapsto 1\ (\langle e'\rangle,s')]\!]$
$$\Longrightarrow$$
$$G\vdash(\langle InsInitV\ Skip\ (\{accC,statDeclC,stat\}e..fn)\rangle,Norm\ s)$$
$$\mapsto 1\ (\langle InsInitV\ Skip\ (\{accC,statDeclC,stat\}e'..fn)\rangle,s')$$

| *InsInitFVar*:
$G\vdash(\langle InsInitV\ Skip\ (\{accC,statDeclC,stat\}Lit\ a..fn)\rangle,Norm\ s)$
$$\mapsto 1\ (\langle\{accC,statDeclC,stat\}Lit\ a..fn\rangle,Norm\ s)$$

— Notice, that we do not have literal values for *vars*. The rules for accessing variables (*Acc*) and assigning to variables (*Ass*), test this with the predicate *groundVar*. After initialisation is done and the *FVar* is evaluated, we can't just throw away the *InsInitFVar* term and return a literal value, as in the cases of *New* or *NewC*. Instead we just return the evaluated *FVar* and test for initialisation in the rule *FVar*.

| *AVarE1*: $[\![G\vdash(\langle e1\rangle,Norm\ s) \mapsto 1\ (\langle e1'\rangle,s')]\!]$
$\Longrightarrow$
$G\vdash(\langle e1.[e2]\rangle,Norm\ s) \mapsto 1\ (\langle e1'.[e2]\rangle,s')$

| *AVarE2*: $G\vdash(\langle e2\rangle,Norm\ s) \mapsto 1\ (\langle e2'\rangle,s')$
$\Longrightarrow$
$G\vdash(\langle Lit\ a.[e2]\rangle,Norm\ s) \mapsto 1\ (\langle Lit\ a.[e2']\rangle,s')$

— *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* × *state*,*nat*,*term* × *state*] ⇒ *bool* (-⊢- ↦- -[*61*,*82*,*82*] *81*)
where $G\vdash p \mapsto n\ p' \equiv (p,p') \in \{(x,\ y).\ step\ G\ x\ y\}\hat{}n$

abbreviation
steptr:: [*prog*,*term* × *state*,*term* × *state*] ⇒ *bool* (-⊢- ↦* -[*61*,*82*,*82*] *81*)
where $G\vdash p \mapsto * p' \equiv (p,p') \in \{(x,\ y).\ step\ G\ x\ y\}^*$

lemma *rtrancl-imp-rel-pow*: $p \in R\hat{}* \Longrightarrow \exists n.\ p \in R\hat{}n$
⟨*proof*⟩

end

Chapter 22

AxSem

50 Axiomatic semantics of Java expressions and statements (see also Eval.thy)

theory *AxSem* **imports** *Evaln TypeSafe* **begin**

design issues:

- a strong version of validity for triples with premises, namely one that takes the recursive depth needed to complete execution, enables correctness proof
- auxiliary variables are handled first-class (-¿ 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)

types *res* = *vals* — result entry
syntax
Val :: *val* ⇒ *res*
Var :: *var* ⇒ *res*
Vals :: *val list* ⇒ *res*
translations
Val x => (*In1 x*)
Var x => (*In2 x*)
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* == (λ*v*. *b*) ∘ *the-In1*
λ*Var*:*v* . *b* == (λ*v*. *b*) ∘ *the-In2*
λ*Vals*:*v*. *b* == (λ*v*. *b*) ∘ *the-In3*

— relation on result values, state and auxiliary variables
types 'a *assn* = *res* ⇒ *state* ⇒ 'a ⇒ *bool*
translations
res <= (*type*) *AxSem.res*
a assn <= (*type*) *vals* ⇒ *state* ⇒ *a* ⇒ *bool*

constdefs
assn-imp :: 'a *assn* ⇒ 'a *assn* ⇒ *bool* (**infixr** ⇒ *25*)
P ⇒ *Q* ≡ ∀ *Y s Z*. *P Y s Z* ⟶ *Q Y s Z*

lemma *assn-imp-def2* [*iff*]: $(P \Rightarrow Q) = (\forall\, Y\ s\ Z.\ P\ Y\ s\ Z \longrightarrow Q\ Y\ s\ Z)$
⟨*proof*⟩

assertion transformers

51 peek-and

constdefs
peek-and :: $'a\ assn \Rightarrow (state \Rightarrow bool) \Rightarrow 'a\ assn$ (**infixl** $\wedge.$ *13*)
$P \wedge.\ p \equiv \lambda Y\ s\ Z.\ P\ Y\ s\ Z \wedge p\ s$

lemma *peek-and-def2* [*simp*]: *peek-and P p Y s* $= (\lambda Z.\ (P\ Y\ s\ Z \wedge p\ s))$
⟨*proof*⟩

lemma *peek-and-Not* [*simp*]: $(P \wedge.\ (\lambda s.\ \neg\, f\ s)) = (P \wedge.\ Not \circ f)$
⟨*proof*⟩

lemma *peek-and-and* [*simp*]: *peek-and* (*peek-and P p*) *p* = *peek-and P p*
⟨*proof*⟩

lemma *peek-and-commut*: $(P \wedge.\ p \wedge.\ q) = (P \wedge.\ q \wedge.\ p)$
⟨*proof*⟩

syntax
Normal :: $'a\ assn \Rightarrow 'a\ assn$
translations
Normal P == $P \wedge.$ *normal*

lemma *peek-and-Normal* [*simp*]: *peek-and* (*Normal P*) *p* = *Normal* (*peek-and P p*)
⟨*proof*⟩

52 assn-supd

constdefs
assn-supd :: $'a\ assn \Rightarrow (state \Rightarrow state) \Rightarrow 'a\ assn$ (**infixl** ;. *13*)
$P\ ;.\ f \equiv \lambda Y\ s'\ Z.\ \exists\, s.\ P\ Y\ s\ Z \wedge s' = f\ s$

lemma *assn-supd-def2* [*simp*]: *assn-supd P f Y s' Z* $= (\exists\, s.\ P\ Y\ s\ Z \wedge s' = f\ s)$
⟨*proof*⟩

53 supd-assn

constdefs
supd-assn :: $(state \Rightarrow state) \Rightarrow 'a\ assn \Rightarrow 'a\ assn$ (**infixr** .; *13*)
$f\ .;\ P \equiv \lambda Y\ s.\ P\ Y\ (f\ s)$

lemma *supd-assn-def2* [*simp*]: $(f\ .;\ P)\ Y\ s = P\ Y\ (f\ s)$
⟨*proof*⟩

lemma *supd-assn-supdD* [*elim*]: $((f\ .;\ Q)\ ;.\ f)\ Y\ s\ Z \Longrightarrow Q\ Y\ s\ Z$

⟨*proof*⟩

lemma *supd-assn-supdI* [*elim*]: $Q\ Y\ s\ Z \Longrightarrow (f\ .;\ (Q\ ;.\ f))\ Y\ s\ Z$
⟨*proof*⟩

54 subst-res

constdefs
subst-res :: $'a\ assn \Rightarrow res \Rightarrow 'a\ assn$ (-←- [60,61] 60)
$P{\leftarrow}w \equiv \lambda Y.\ P\ w$

lemma *subst-res-def2* [*simp*]: $(P{\leftarrow}w)\ Y = P\ w$
⟨*proof*⟩

lemma *subst-subst-res* [*simp*]: $P{\leftarrow}w{\leftarrow}v = P{\leftarrow}w$
⟨*proof*⟩

lemma *peek-and-subst-res* [*simp*]: $(P\ \wedge.\ p){\leftarrow}w = (P{\leftarrow}w\ \wedge.\ p)$
⟨*proof*⟩

55 subst-Bool

constdefs
subst-Bool :: $'a\ assn \Rightarrow bool \Rightarrow 'a\ assn$ (-←=- [60,61] 60)
$P{\leftarrow}{=}b \equiv \lambda Y\ s\ Z.\ \exists v.\ P\ (Val\ v)\ s\ Z \wedge (normal\ s \longrightarrow the\text{-}Bool\ v{=}b)$

lemma *subst-Bool-def2* [*simp*]:
$(P{\leftarrow}{=}b)\ Y\ s\ Z = (\exists v.\ P\ (Val\ v)\ s\ Z \wedge (normal\ s \longrightarrow the\text{-}Bool\ v{=}b))$
⟨*proof*⟩

lemma *subst-Bool-the-BoolI*: $P\ (Val\ b)\ s\ Z \Longrightarrow (P{\leftarrow}{=}the\text{-}Bool\ b)\ Y\ s\ Z$
⟨*proof*⟩

56 peek-res

constdefs
peek-res :: $(res \Rightarrow 'a\ assn) \Rightarrow 'a\ assn$
$peek\text{-}res\ Pf \equiv \lambda Y.\ Pf\ Y\ Y$

syntax
@*peek-res* :: $pttrn \Rightarrow 'a\ assn \Rightarrow 'a\ assn$ (λ-:. - [0,3] 3)
translations
$\lambda w{:}.\ P$ == $peek\text{-}res\ (\lambda w.\ P)$

lemma *peek-res-def2* [*simp*]: $peek\text{-}res\ P\ Y = P\ Y\ Y$
⟨*proof*⟩

lemma *peek-res-subst-res* [*simp*]: $peek\text{-}res\ P{\leftarrow}w = P\ w{\leftarrow}w$
⟨*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*⟩

57 ign-res

constdefs
ign-res :: $'a\ \mathit{assn} \Rightarrow 'a\ \mathit{assn}$ (-↓ [*1000*] *1000*)
$P\!\downarrow \equiv \lambda Y\ s\ Z.\ \exists\, Y.\ P\ Y\ s\ Z$

lemma *ign-res-def2* [*simp*]: $P\!\downarrow\ Y\ s\ Z = (\exists\, Y.\ P\ Y\ s\ Z)$
⟨*proof*⟩

lemma *ign-ign-res* [*simp*]: $P\!\downarrow\!\downarrow = P\!\downarrow$
⟨*proof*⟩

lemma *ign-subst-res* [*simp*]: $P\!\downarrow\!\leftarrow w = P\!\downarrow$
⟨*proof*⟩

lemma *peek-and-ign-res* [*simp*]: $(P \wedge.\ p)\!\downarrow = (P\!\downarrow \wedge.\ p)$
⟨*proof*⟩

58 peek-st

constdefs
peek-st :: $(\mathit{st} \Rightarrow 'a\ \mathit{assn}) \Rightarrow 'a\ \mathit{assn}$
$\mathit{peek\text{-}st}\ P \equiv \lambda Y\ s.\ P\ (\mathit{store}\ s)\ Y\ s$

syntax
@*peek-st* :: *pttrn* ⇒ $'a\ \mathit{assn} \Rightarrow 'a\ \mathit{assn}$ (λ-.. - [*0*,*3*] *3*)
translations
$\lambda s..\ P$ == $\mathit{peek\text{-}st}\ (\lambda s.\ P)$

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*]: $(\lambda s..(Normal\ (P\ s))) = Normal\ (\lambda s..\ P\ s)$
⟨*proof*⟩

59 ign-res-eq

constdefs
ign-res-eq :: *'a assn ⇒ res ⇒ 'a assn* (-↓=- [*60*,*61*] *60*)
$P{\downarrow}{=}w \equiv \lambda Y{:}.\ P{\downarrow} \wedge .\ (\lambda s.\ Y{=}w)$

lemma *ign-res-eq-def2* [*simp*]: $(P{\downarrow}{=}w)\ Y\ s\ Z = ((\exists\ Y.\ P\ Y\ s\ Z) \wedge Y{=}w)$
⟨*proof*⟩

lemma *ign-ign-res-eq* [*simp*]: $(P{\downarrow}{=}w){\downarrow} = P{\downarrow}$
⟨*proof*⟩

lemma *ign-res-eq-subst-res*: $P{\downarrow}{=}w{\leftarrow}w = P{\downarrow}$
⟨*proof*⟩

lemma *subst-Bool-ign-res-eq*: $((P{\leftarrow}{=}b){\downarrow}{=}x)\ Y\ s\ Z = ((P{\leftarrow}{=}b)\ Y\ s\ Z\ \wedge\ Y{=}x)$
⟨*proof*⟩

60 RefVar

constdefs
RefVar :: *(state ⇒ vvar × state) ⇒ 'a assn ⇒ 'a assn*(**infixr** ..; *13*)
vf ..; P $\equiv \lambda Y\ s.$ *let* $(v,s') =$ *vf s in P* (*Var v*) s'

lemma *RefVar-def2* [*simp*]: (*vf ..; P*) *Y s* =
P (*Var* (*fst* (*vf s*))) (*snd* (*vf s*))
⟨*proof*⟩

61 allocation

constdefs
Alloc :: *prog ⇒ obj-tag ⇒ 'a assn ⇒ 'a assn*
Alloc G otag P $\equiv \lambda Y\ s\ Z.$
$\forall\ s'\ a.\ G{\vdash}s\ {-}halloc\ otag{\succ}a{\rightarrow}\ s'{\longrightarrow}\ P\ (Val\ (Addr\ a))\ s'\ Z$

SXAlloc :: *prog ⇒ 'a assn ⇒ 'a assn*
SXAlloc G P $\equiv \lambda Y\ s\ Z.\ \forall\ s'.\ G{\vdash}s\ {-}sxalloc{\rightarrow}\ s' \longrightarrow P\ Y\ s'\ Z$

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

constdefs
type-ok :: *prog* ⇒ *term* ⇒ *state* ⇒ *bool*
type-ok G t s ≡
∃ *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*)
∧ *s*::≼(*G*,*L*)

datatype *'a triple* = *triple* (*'a assn*) *term* (*'a assn*)
({(*1-*)}/ ->/ {(*1-*)} [*3*,*65*,*3*]*75*)
types *'a triples* = *'a triple set*

syntax

var-triple :: [*'a assn*, *var* ,*'a assn*] ⇒ *'a triple*
({(*1-*)}/ -=>/ {(*1-*)} [*3*,*80*,*3*] *75*)
expr-triple :: [*'a assn*, *expr* ,*'a assn*] ⇒ *'a triple*
({(*1-*)}/ -->/ {(*1-*)} [*3*,*80*,*3*] *75*)
exprs-triple :: [*'a assn*, *expr list* ,*'a assn*] ⇒ *'a triple*
({(*1-*)}/ -#>/ {(*1-*)} [*3*,*65*,*3*] *75*)
stmt-triple :: [*'a assn*, *stmt*, *'a assn*] ⇒ *'a triple*
({(*1-*)}/ .-./ {(*1-*)} [*3*,*65*,*3*] *75*)

syntax (*xsymbols*)

triple :: [*'a assn*, *term* ,*'a assn*] ⇒ *'a triple*
({(*1-*)}/ -≻/ {(*1-*)} [*3*,*65*,*3*] *75*)
var-triple :: [*'a assn*, *var* ,*'a assn*] ⇒ *'a triple*
({(*1-*)}/ -=≻/ {(*1-*)} [*3*,*80*,*3*] *75*)
expr-triple :: [*'a assn*, *expr* ,*'a assn*] ⇒ *'a triple*
({(*1-*)}/ -−≻/ {(*1-*)} [*3*,*80*,*3*] *75*)
exprs-triple :: [*'a assn*, *expr list* ,*'a assn*] ⇒ *'a triple*
({(*1-*)}/ -�doteq≻/ {(*1-*)} [*3*,*65*,*3*] *75*)

translations
{*P*} *e*−≻ {*Q*} == {*P*} *In1l e*≻ {*Q*}
{*P*} *e*=≻ {*Q*} == {*P*} *In2 e*≻ {*Q*}
{*P*} *e*≐≻ {*Q*} == {*P*} *In3 e*≻ {*Q*}
{*P*} *.c.* {*Q*} == {*P*} *In1r c*≻ {*Q*}

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*⟩

constdefs
mtriples :: (*'c* ⇒ *'sig* ⇒ *'a assn*) ⇒ (*'c* ⇒ *'sig* ⇒ *expr*) ⇒
(*'c* ⇒ *'sig* ⇒ *'a assn*) ⇒ (*'c* × *'sig*) *set* ⇒ *'a triples*
({{(*1-*)}/ -−≻/ {(*1-*)} | -}[*3*,*65*,*3*,*65*]*75*)
{{*P*} *tf*−≻ {*Q*} | *ms*} ≡ (λ(*C*,*sig*). {*Normal*(*P C sig*)} *tf C sig*−≻ {*Q C sig*})‘*ms*

consts

triple-valid :: *prog* ⇒ *nat* ⇒ *'a triple* ⇒ *bool*
(-⊨-:- [*61*,*0*, *58*] *57*)

ax-valids :: *prog* ⇒ *'b triples* ⇒ *'a triples* ⇒ *bool*
(-,-|⊨- [*61*,*58*,*58*] *57*)

syntax

triples-valid:: *prog* ⇒ *nat* ⇒ *'a triples* ⇒ *bool*
(-||=-:- [*61*,*0*, *58*] *57*)
ax-valid :: *prog* ⇒ *'b triples* ⇒ *'a triple* ⇒ *bool*
(-,-|=- [*61*,*58*,*58*] *57*)

syntax (*xsymbols*)

triples-valid:: *prog* ⇒ *nat* ⇒ *'a triples* ⇒ *bool*
(-|⊨-:- [*61*,*0*, *58*] *57*)
ax-valid :: *prog* ⇒ *'b triples* ⇒ *'a triple* ⇒ *bool*
(-,-⊨- [*61*,*58*,*58*] *57*)

defs *triple-valid-def*: $G⊨n:t \equiv$ *case t of* $\{P\}\ t\succ \{Q\} \Rightarrow$
$\forall Y\ s\ Z.\ P\ Y\ s\ Z \longrightarrow$ *type-ok G t s* $\longrightarrow$
$(\forall Y'\ s'.\ G⊢s\ -t\succ-n\rightarrow (Y',s') \longrightarrow Q\ Y'\ s'\ Z)$
translations $G|⊨n:ts == Ball\ ts\ (triple\text{-}valid\ G\ n)$
defs *ax-valids-def*: $G,A|⊨ts \equiv \forall n.\ G|⊨n:A \longrightarrow G|⊨n:ts$
translations $G,A ⊨t == G,A|⊨\{t\}$

lemma *triple-valid-def2*: $G⊨n:\{P\}\ t\succ \{Q\} =$
$(\forall Y\ s\ Z.\ P\ Y\ s\ Z$
$\longrightarrow (\exists L.\ (normal\ s \longrightarrow (\exists\ C\ T\ A.\ (\!|prg=G,cls=C,lcl=L|\!)⊢t::T \wedge$
$(\!|prg=G,cls=C,lcl=L|\!)⊢dom\ (locals\ (store\ s))\gg t\gg A)) \wedge$
$s::\preceq(G,L))$
$\longrightarrow (\forall Y'\ s'.\ G⊢s\ -t\succ-n\rightarrow (Y',s') \longrightarrow Q\ Y'\ s'\ Z))$
⟨*proof*⟩

declare *split-paired-All* [*simp del*] *split-paired-Ex* [*simp del*]
declare *split-if* [*split del*] *split-if-asm* [*split del*]
option.split [*split del*] *option.split-asm* [*split del*]
⟨*ML*⟩

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

$G,A ⊢t \equiv G,A|⊢\{t\}$

| *empty*: $G,A|⊢\{\}$
| *insert*: $⟦G,A⊢t;\ G,A|⊢ts⟧ \Longrightarrow$
$G,A|⊢insert\ t\ ts$

| *asm*: $ts⊆A \Longrightarrow G,A|⊢ts$

| *weaken*: $⟦G,A|⊢ts';\ ts \subseteq ts'⟧ \Longrightarrow G,A|⊢ts$

| *conseq*: $\forall Y\ s\ Z\ .\ P\ \ Y\ s\ Z\ \longrightarrow (\exists P'\ Q'.\ G,A⊢\{P'\}\ t\succ \{Q'\} \wedge (\forall Y'\ s'.$
$(\forall Y\ \ Z'.\ P'\ Y\ s\ Z' \longrightarrow Q'\ Y'\ s'\ Z') \longrightarrow$
$Q\ \ Y'\ s'\ Z\ ))$

$$\Longrightarrow G,A\vdash\{P\ \}\ t\succ\ \{Q\ \}$$

| *hazard*:$G,A\vdash\{P \wedge.\ Not \circ type\text{-}ok\ G\ t\}\ t\succ\ \{Q\}$

| *Abrupt*: $G,A\vdash\{P\leftarrow(undefined3\ t) \wedge.\ Not \circ normal\}\ t\succ\ \{P\}$

— variables

| *LVar*: $G,A\vdash\{Normal\ (\lambda s..\ P\leftarrow Var\ (lvar\ vn\ s))\}\ LVar\ vn=\succ\ \{P\}$

| *FVar*: $[\![G,A\vdash\{Normal\ P\}\ .Init\ C.\ \{Q\};$
$G,A\vdash\{Q\}\ e-\succ\ \{\lambda Val{:}a{:}.\ fvar\ C\ stat\ fn\ a\ ..;\ R\}]\!]\Longrightarrow$
$G,A\vdash\{Normal\ P\}\ \{accC,C,stat\}e..fn=\succ\ \{R\}$

| *AVar*: $[\![G,A\vdash\{Normal\ P\}\ e1-\succ\ \{Q\};$
$\forall a.\ G,A\vdash\{Q\leftarrow Val\ a\}\ e2-\succ\ \{\lambda Val{:}i{:}.\ avar\ G\ i\ a\ ..;\ R\}]\!]\Longrightarrow$
$G,A\vdash\{Normal\ P\}\ e1.[e2]=\succ\ \{R\}$

— expressions

| *NewC*: $[\![G,A\vdash\{Normal\ P\}\ .Init\ C.\ \{Alloc\ G\ (CInst\ C)\ Q\}]\!]\Longrightarrow$
$G,A\vdash\{Normal\ P\}\ NewC\ C-\succ\ \{Q\}$

| *NewA*: $[\![G,A\vdash\{Normal\ P\}\ .init\text{-}comp\text{-}ty\ T.\ \{Q\};\ \ G,A\vdash\{Q\}\ e-\succ$
$\{\lambda Val{:}i{:}.\ abupd\ (check\text{-}neg\ i)\ .;\ Alloc\ G\ (Arr\ T\ (the\text{-}Intg\ i))\ R\}]\!]\Longrightarrow$
$G,A\vdash\{Normal\ P\}\ New\ T[e]-\succ\ \{R\}$

| *Cast*: $[\![G,A\vdash\{Normal\ P\}\ e-\succ\ \{\lambda Val{:}v{:}.\ \lambda s..$
$abupd\ (raise\text{-}if\ (\neg G,s\vdash v\ fits\ T)\ ClassCast)\ .;\ Q\leftarrow Val\ v\}]\!]\Longrightarrow$
$G,A\vdash\{Normal\ P\}\ Cast\ T\ e-\succ\ \{Q\}$

| *Inst*: $[\![G,A\vdash\{Normal\ P\}\ e-\succ\ \{\lambda Val{:}v{:}.\ \lambda s..$
$Q\leftarrow Val\ (Bool\ (v\neq Null \wedge G,s\vdash v\ fits\ RefT\ T))\}]\!]\Longrightarrow$
$G,A\vdash\{Normal\ P\}\ e\ InstOf\ T-\succ\ \{Q\}$

| *Lit*: $G,A\vdash\{Normal\ (P\leftarrow Val\ v)\}\ Lit\ v-\succ\ \{P\}$

| *UnOp*: $[\![G,A\vdash\{Normal\ P\}\ e-\succ\ \{\lambda Val{:}v{:}.\ Q\leftarrow Val\ (eval\text{-}unop\ unop\ v)\}]\!]$
$\Longrightarrow$
$G,A\vdash\{Normal\ P\}\ UnOp\ unop\ e-\succ\ \{Q\}$

| *BinOp*:
$[\![G,A\vdash\{Normal\ P\}\ e1-\succ\ \{Q\};$
$\forall v1.\ G,A\vdash\{Q\leftarrow Val\ v1\}$
$(if\ need\text{-}second\text{-}arg\ binop\ v1\ then\ (In1l\ e2)\ else\ (In1r\ Skip))\succ$
$\{\lambda Val{:}v2{:}.\ R\leftarrow Val\ (eval\text{-}binop\ binop\ v1\ v2)\}]\!]$
$\Longrightarrow$
$G,A\vdash\{Normal\ P\}\ BinOp\ binop\ e1\ e2-\succ\ \{R\}$

| *Super*: $G,A\vdash\{Normal\ (\lambda s..\ P\leftarrow Val\ (val\text{-}this\ s))\}\ Super-\succ\ \{P\}$

| *Acc*: $[\![G,A\vdash\{Normal\ P\}\ va=\succ\ \{\lambda Var{:}(v,f){:}.\ Q\leftarrow Val\ v\}]\!]\Longrightarrow$
$G,A\vdash\{Normal\ P\}\ Acc\ va-\succ\ \{Q\}$

| *Ass*: $[\![G,A\vdash\{Normal\ P\}\ va=\succ\ \{Q\};$
$\forall vf.\ G,A\vdash\{Q\leftarrow Var\ vf\}\ e-\succ\ \{\lambda Val{:}v{:}.\ assign\ (snd\ vf)\ v\ .;\ R\}]\!]\Longrightarrow$
$G,A\vdash\{Normal\ P\}\ va:=e-\succ\ \{R\}$

| *Cond*: $[\![G,A\vdash\{Normal\ P\}\ e0-\succ\ \{P'\};$
$\forall b.\ G,A\vdash\{P'\leftarrow=b\}\ (if\ b\ then\ e1\ else\ e2)-\succ\ \{Q\}]\!]\Longrightarrow$
$G,A\vdash\{Normal\ P\}\ e0\ ?\ e1\ :\ e2-\succ\ \{Q\}$

| *Call*:
$[\![G,A\vdash\{Normal\ P\}\ e{-}\succ\ \{Q\};\ \forall a.\ G,A\vdash\{Q\leftarrow Val\ a\}\ args\doteq\succ\ \{R\ a\};$
$\forall a\ vs\ invC\ declC\ l.\ G,A\vdash\{(R\ a\leftarrow Vals\ vs\ \wedge.$
$(\lambda s.\ declC{=}invocation\text{-}declclass\ G\ mode\ (store\ s)\ a\ statT\ (\!|name{=}mn,parTs{=}pTs|\!)\ \wedge$
$invC = invocation\text{-}class\ mode\ (store\ s)\ a\ statT\ \wedge$
$l = locals\ (store\ s))\ ;.$
$init\text{-}lvars\ G\ declC\ (\!|name{=}mn,parTs{=}pTs|\!)\ mode\ a\ vs)\ \wedge.$
$(\lambda s.\ normal\ s\longrightarrow G\vdash mode\rightarrow invC\preceq statT)\}$
$Methd\ declC\ (\!|name{=}mn,parTs{=}pTs|\!){-}\succ\ \{set\text{-}lvars\ l\ .;\ S\}]\!]\Longrightarrow$
$G,A\vdash\{Normal\ P\}\ \{accC,statT,mode\}e{\cdot}mn(\{pTs\}args){-}\succ\ \{S\}$

| *Methd*:$[\![G,A\cup\{\{P\}\ Methd{-}\succ\ \{Q\}\mid ms\}\mid\!\vdash\{\{P\}\ body\ G{-}\succ\ \{Q\}\mid ms\}]\!]\Longrightarrow$
$G,A|\!\vdash\{\{P\}\ Methd{-}\succ\ \{Q\}\mid ms\}$

| *Body*: $[\![G,A\vdash\{Normal\ P\}\ .Init\ D.\ \{Q\};$
$G,A\vdash\{Q\}\ .c.\ \{\lambda s..\ abupd\ (absorb\ Ret)\ .;\ R\leftarrow(In1\ (the\ (locals\ s\ Result)))\}]\!]$
$\Longrightarrow$
$G,A\vdash\{Normal\ P\}\ Body\ D\ c{-}\succ\ \{R\}$

— expression lists

| *Nil*: $G,A\vdash\{Normal\ (P\leftarrow Vals\ [])\}\ []\doteq\succ\ \{P\}$

| *Cons*: $[\![G,A\vdash\{Normal\ P\}\ e{-}\succ\ \{Q\};$
$\forall v.\ G,A\vdash\{Q\leftarrow Val\ v\}\ es\doteq\succ\ \{\lambda Vals{:}vs{:}.\ R\leftarrow Vals\ (v\#vs)\}]\!]\Longrightarrow$
$G,A\vdash\{Normal\ P\}\ e\#es\doteq\succ\ \{R\}$

— 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*⊢{*Normal P*} .*c1*. {*Q*};
∀ *x*. *G*,*A*⊢{*Q* ∧. (λ*s*. *x* = *fst s*) ;. *abupd* (λ*x*. *None*)}
.*c2*. {*abupd* (*abrupt-if* (*x*≠*None*) *x*) .; *R*}⟧ ⟹
G,*A*⊢{*Normal P*} .*c1 Finally c2*. {*R*}

| *Done*: *G*,*A*⊢{*Normal* (*P*←◇ ∧. *initd C*)} .*Init C*. {*P*}

| *Init*: ⟦*the* (*class G C*) = *c*;
G,*A*⊢{*Normal* ((*P* ∧. *Not* ∘ *initd C*) ;. *supd* (*init-class-obj G C*))}
.(*if C* = *Object then Skip else Init* (*super c*)). {*Q*};
∀ *l*. *G*,*A*⊢{*Q* ∧. (λ*s*. *l* = *locals* (*store s*)) ;. *set-lvars empty*}
.*init c*. {*set-lvars l* .; *R*}⟧ ⟹
G,*A*⊢{*Normal* (*P* ∧. *Not* ∘ *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*⊢{*Normal P*} *InsInitV c v*=≻ {*Q*}
| *InsInitE*: *G*,*A*⊢{*Normal P*} *InsInitE c e*–≻ {*Q*}
| *Callee*: *G*,*A*⊢{*Normal P*} *Callee l e*–≻ {*Q*}
| *FinA*: *G*,*A*⊢{*Normal P*} .*FinA a c*. {*Q*}

constdefs
adapt-pre :: *'a assn* ⇒ *'a assn* ⇒ *'a assn* ⇒ *'a assn*
adapt-pre P Q Q'≡λ*Y s Z*. ∀ *Y' s'*. ∃ *Z'*. *P Y s Z'* ∧ (*Q Y' s' Z'* ⟶ *Q' Y' s' Z*)

rules derived by induction

lemma *cut-valid*: ⟦*G*,*A'*|⊨*ts*; *G*,*A*|⊨*A'*⟧ ⟹ *G*,*A*|⊨*ts*
⟨*proof*⟩

lemma *ax-thin* [*rule-format* (*no-asm*)]:
G,(*A'*::*'a triple set*)|⊢(*ts*::*'a triple set*) ⟹ ∀ *A*. *A'* ⊆ *A* ⟶ *G*,*A*|⊢*ts*
⟨*proof*⟩

lemma *ax-thin-insert*: *G*,(*A*::*'a triple set*)⊢(*t*::*'a triple*) ⟹ *G*,*insert x A*⊢*t*
⟨*proof*⟩

lemma *subset-mtriples-iff*:
ts ⊆ {{*P*} *mb*–≻ {*Q*} | *ms*} = (∃ *ms'*. *ms'*⊆*ms* ∧ *ts* = {{*P*} *mb*–≻ {*Q*} | *ms'*})
⟨*proof*⟩

lemma *weaken*:
G,(*A*::*'a triple set*)|⊢(*ts'*::*'a triple set*) ⟹ !*ts*. *ts* ⊆ *ts'* ⟶ *G*,*A*|⊢*ts*
⟨*proof*⟩

rules derived from conseq

In the following rules we often have to give some type annotations like: *G*,*A*⊢{*P*} *t*≻ {*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*:
$[\![\forall\ Y\ s\ Z.\ P\ Y\ s\ Z$
$\longrightarrow G,(A::'a\ triple\ set)\vdash\{\lambda Y'\ s'\ (Z'::'a).\ (Y',s')=(Y,s)\}$
$t\succ$
$\{\lambda Y\ s\ Z'.\ Q\ Y\ s\ Z\}$
$]\!]\Longrightarrow\ G,A\vdash\{P::'a\ assn\}\ t\succ\ \{Q::'a\ assn\}$
⟨*proof*⟩

lemma *ax-constant*: $[\![\ C\Longrightarrow G,(A::'a\ triple\ set)\vdash\{P::'a\ assn\}\ t\succ\ \{Q\}]\!]$
$\Longrightarrow G,A\vdash\{\lambda Y\ s\ Z.\ C\wedge P\ Y\ s\ Z\}\ t\succ\ \{Q\}$
⟨*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*: $[\![P\ Y\ s;\ All\ (op=w)]\!]\Longrightarrow P\ w\ s$
⟨*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*:
$[\![G,(A::'a\ triple\ set)\vdash\{P1::'a\ assn\}\ t\succ\ \{Q1\};\ G,A\vdash\{P2::'a\ assn\}\ t\succ\ \{Q2\}]\!]$
$\Longrightarrow\ G,A\vdash\{\lambda Y\ s\ Z.\ P1\ Y\ s\ Z \vee P2\ Y\ s\ Z\}\ t\succ\ \{\lambda Y\ s\ Z.\ Q1\ Y\ s\ Z \vee Q2\ Y\ s\ Z\}$
⟨*proof*⟩

lemma *ax-supd-shuffle*:
$(\exists\, Q.\ G,(A::'a\ triple\ set)\vdash\{P::'a\ assn\}\ .c1.\ \{Q\} \wedge G,A\vdash\{Q\ ;.\ f\}\ .c2.\ \{R\}) =$
$\quad(\exists\, Q'.\ G,A\vdash\{P\}\ .c1.\ \{f\ .;\ Q'\} \wedge G,A\vdash\{Q'\}\ .c2.\ \{R\})$
⟨*proof*⟩

lemma *ax-cases*:
$[\![G,(A::'a\ triple\ set)\vdash\{P \wedge.\ \ \ C\}\ t\succ\ \{Q::'a\ assn\};$
$\qquad G,A\vdash\{P \wedge.\ Not \circ C\}\ t\succ\ \{Q\}]\!] \Longrightarrow G,A\vdash\{P\}\ t\succ\ \{Q\}$
⟨*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*:
$\forall\, G\ (A::'a\ triple\ set)\ t.\ G,A\models\{P\}\ t\succ\ \{Q\} \longrightarrow G,A\models\{P'\}\ t\succ\ \{Q'\} \Longrightarrow$
$\quad P' \Rightarrow adapt\text{-}pre\ P\ Q\ (Q'::'a\ assn)$
⟨*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*:
$\forall\, v.\ G,(A::'a\ triple\ set)\vdash\{(P'\quad\quad\quad\quad\quad v\ )\leftarrow Val\ v\}\ t\succ\ \{(Q\ v)::'a\ assn\}$
$\Longrightarrow\ \forall\, v.\ G,A\vdash\{(\lambda w::\ P'\ (the\text{-}In1\ w))\leftarrow Val\ v\}\ t\succ\ \{Q\ v\}$
⟨*proof*⟩

lemma *ax-subst-Var-allI*:
$\forall\, v.\ G,(A::'a\ triple\ set)\vdash\{(P'\quad\quad\quad\quad\quad v\ )\leftarrow Var\ v\}\ t\succ\ \{(Q\ v)::'a\ assn\}$
$\Longrightarrow\ \forall\, v.\ G,A\vdash\{(\lambda w::\ P'\ (the\text{-}In2\ w))\leftarrow Var\ v\}\ t\succ\ \{Q\ v\}$
⟨*proof*⟩

lemma *ax-subst-Vals-allI*:
$(\forall\, v.\ G,(A::'a\ triple\ set)\vdash\{(\quad P'\quad\quad\quad v\ )\leftarrow Vals\ v\}\ t\succ\ \{(Q\ v)::'a\ assn\})$
$\Longrightarrow\ \forall\, v.\ G,A\vdash\{(\lambda w::\ P'\ (the\text{-}In3\ w))\leftarrow Vals\ v\}\ t\succ\ \{Q\ v\}$
⟨*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*: $[\![F\subseteq ms;\ finite\ ms;\ \forall\,(C,sig){\in}ms.$
$G,(A::'a\ triple\ set)\vdash\{Normal\ (P\ C\ sig)::'a\ assn\}\ mb\ C\ sig{-}\succ\ \{Q\ C\ sig\}]\!]\Longrightarrow$
$G,A|\vdash\{\{P\}\ mb{-}\succ\ \{Q\}\mid F\}$
⟨*proof*⟩
lemmas *ax-finite-mtriples = ax-finite-mtriples-lemma [OF subset-refl]*

lemma *ax-derivs-insertD*:
$G,(A::'a\ triple\ set)|\vdash insert\ (t::'a\ triple)\ ts \Longrightarrow G,A\vdash t \wedge G,A|\vdash ts$
⟨*proof*⟩

lemma *ax-methods-spec*:
$[\![G,(A::'a\ triple\ set)|\vdash split\ f\ `\ ms;\ (C,sig) \in ms]\!] \Longrightarrow G,A\vdash((f\ C\ sig)::'a\ triple)$
⟨*proof*⟩

lemma *ax-finite-pointwise-lemma* [*rule-format*]: $[\![F \subseteq ms;\ finite\ ms]\!] \Longrightarrow$
$((\forall (C,sig)\in F.\ G,(A::'a\ triple\ set)\vdash(f\ C\ sig::'a\ triple)) \longrightarrow (\forall (C,sig)\in ms.\ G,A\vdash(g\ C\ sig::'a\ triple))) \longrightarrow$
$G,A|\vdash split\ f\ `\ F \longrightarrow G,A|\vdash split\ g\ `\ F$
⟨*proof*⟩
lemmas *ax-finite-pointwise* = *ax-finite-pointwise-lemma* [*OF subset-refl*]

lemma *ax-no-hazard*:
$G,(A::'a\ triple\ set)\vdash\{P \wedge.\ type\text{-}ok\ G\ t\}\ t\succ\ \{Q::'a\ assn\} \Longrightarrow G,A\vdash\{P\}\ t\succ\ \{Q\}$
⟨*proof*⟩

lemma *ax-free-wt*:
$(\exists\ T\ L\ C.\ (\!|prg=G,cls=C,lcl=L|\!)\vdash t::T)$
$\longrightarrow G,(A::'a\ triple\ set)\vdash\{Normal\ P\}\ t\succ\ \{Q::'a\ assn\} \Longrightarrow$
$G,A\vdash\{Normal\ P\}\ t\succ\ \{Q\}$
⟨*proof*⟩

⟨*ML*⟩
declare *ax-Abrupts* [*intro*!]

lemmas *ax-Normal-cases* = *ax-cases* [*of - - - normal*]

lemma *ax-Skip* [*intro*!]: $G,(A::'a\ triple\ set)\vdash\{P\leftarrow\diamondsuit\}\ .Skip.\ \{P::'a\ assn\}$
⟨*proof*⟩
lemmas *ax-SkipI* = *ax-Skip* [*THEN conseq1*, *standard*]

derived rules for methd call

lemma *ax-Call-known-DynT*:
$[\![G\vdash IntVir\rightarrow C\preceq statT;$
$\forall a\ vs\ l.\ G,A\vdash\{(R\ a\leftarrow Vals\ vs \wedge.\ (\lambda s.\ l = locals\ (store\ s))\ ;.$
$init\text{-}lvars\ G\ C\ (\!|name=mn,parTs=pTs|\!)\ IntVir\ a\ vs)\}$
$Methd\ C\ (\!|name=mn,parTs=pTs|\!)\text{-}\succ\ \{set\text{-}lvars\ l\ .;\ S\};$
$\forall a.\ G,A\vdash\{Q\leftarrow Val\ a\}\ args\doteq\succ$
$\{R\ a \wedge.\ (\lambda s.\ C = obj\text{-}class\ (the\ (heap\ (store\ s))\ (the\text{-}Addr\ a))) \wedge$
$C = invocation\text{-}declclass$
$G\ IntVir\ (store\ s)\ a\ statT\ (\!|name=mn,parTs=pTs|\!)\ )\};$
$G,(A::'a\ triple\ set)\vdash\{Normal\ P\}\ e\text{-}\succ\ \{Q::'a\ assn\}]\!]$
$\Longrightarrow G,A\vdash\{Normal\ P\}\ \{accC,statT,IntVir\}e\cdot mn(\{pTs\}args)\text{-}\succ\ \{S\}$
⟨*proof*⟩

lemma *ax-Call-Static*:

$[\![\forall\, a\ vs\ l.\ G,A\vdash\{R\ a\leftarrow Vals\ vs\ \wedge.\ (\lambda s.\ l = locals\ (store\ s))\ ;.$
$\quad init\text{-}lvars\ G\ C\ (\!|name{=}mn,parTs{=}pTs|\!)\ Static\ any\text{-}Addr\ vs\}$
$\quad Methd\ C\ (\!|name{=}mn,parTs{=}pTs|\!)\text{-}\succ\ \{set\text{-}lvars\ l\ .;\ S\};$
$G,A\vdash\{Normal\ P\}\ e\text{-}\succ\ \{Q\};$
$\forall\ a.\ G,(A::'a\ triple\ set)\vdash\{Q\leftarrow Val\ a\}\ args\doteq\succ\ \{(R::val\Rightarrow 'a\ assn)\ \ a$
$\wedge.\ (\lambda\ s.\ C{=}invocation\text{-}declclass$
$\quad G\ Static\ (store\ s)\ a\ statT\ (\!|name{=}mn,parTs{=}pTs|\!)\}$
$]\!] \Longrightarrow\ G,A\vdash\{Normal\ P\}\ \{accC,statT,Static\}e{\cdot}mn(\{pTs\}args)\text{-}\succ\ \{S\}$
⟨*proof*⟩

lemma *ax-Methd1*:
$[\![G,A\cup\{\{P\}\ Methd\text{-}\succ\ \{Q\}\mid ms\}|\vdash\ \{\{P\}\ body\ G\text{-}\succ\ \{Q\}\mid ms\};\ (C,sig)\in ms]\!] \Longrightarrow$
$\quad G,A\vdash\{Normal\ (P\ C\ sig)\}\ Methd\ C\ sig\text{-}\succ\ \{Q\ C\ sig\}$
⟨*proof*⟩

lemma *ax-MethdN*:
$G,insert(\{Normal\ P\}\ Methd\ \ C\ sig\text{-}\succ\ \{Q\})\ A\vdash$
$\quad \{Normal\ P\}\ body\ G\ C\ sig\text{-}\succ\ \{Q\} \Longrightarrow$
$G,A\vdash\{Normal\ P\}\ Methd\ \ C\ sig\text{-}\succ\ \{Q\}$
⟨*proof*⟩

lemma *ax-StatRef*:
$G,(A::'a\ triple\ set)\vdash\{Normal\ (P\leftarrow Val\ Null)\}\ StatRef\ rt\text{-}\succ\ \{P::'a\ assn\}$
⟨*proof*⟩

rules derived from Init and Done

lemma *ax-InitS*: $[\![the\ (class\ G\ C) = c;\ C \neq Object;$
$\forall\, l.\ G,A\vdash\{Q\ \wedge.\ (\lambda s.\ l = locals\ (store\ s))\ ;.\ set\text{-}lvars\ empty\}$
$.init\ c.\ \{set\text{-}lvars\ l\ .;\ R\};$
$G,A\vdash\{Normal\ ((P\ \wedge.\ Not\circ initd\ C)\ ;.\ supd\ (init\text{-}class\text{-}obj\ G\ C))\}$
$.Init\ (super\ c).\ \{Q\}]\!] \Longrightarrow$
$G,(A::'a\ triple\ set)\vdash\{Normal\ (P\ \wedge.\ Not\circ initd\ C)\}\ .Init\ C.\ \{R::'a\ assn\}$
⟨*proof*⟩

lemma *ax-Init-Skip-lemma*:
$\forall\, l.\ G,(A::'a\ triple\ set)\vdash\{P\leftarrow\diamondsuit\ \wedge.\ (\lambda s.\ l = locals\ (store\ s))\ ;.\ set\text{-}lvars\ l'\}$
$.Skip.\ \{(set\text{-}lvars\ l\ .;\ P)::'a\ assn\}$
⟨*proof*⟩

lemma *ax-triv-InitS*: $[\![the\ (class\ G\ C) = c;init\ c = Skip;\ C \neq Object;$
$P\leftarrow\diamondsuit \Rightarrow (supd\ (init\text{-}class\text{-}obj\ G\ C)\ .;\ P);$
$G,A\vdash\{Normal\ (P\ \wedge.\ initd\ C)\}\ .Init\ (super\ c).\ \{(P\ \wedge.\ initd\ C)\leftarrow\diamondsuit\}]\!] \Longrightarrow$
$G,(A::'a\ triple\ set)\vdash\{Normal\ P\leftarrow\diamondsuit\}\ .Init\ C.\ \{(P\ \wedge.\ initd\ C)::'a\ assn\}$
⟨*proof*⟩

lemma *ax-Init-Object*: *wf-prog* $G \Longrightarrow G,(A::'a\ triple\ set)\vdash$
$\{Normal\ ((supd\ (init\text{-}class\text{-}obj\ G\ Object)\ .;\ P\leftarrow\diamondsuit)\ \wedge.\ Not\circ initd\ Object)\}$
$.Init\ Object.\ \{(P\ \wedge.\ initd\ Object)::'a\ assn\}$
⟨*proof*⟩

lemma *ax-triv-Init-Object*: ⟦*wf-prog G*;
(*P*::'*a assn*) ⇒ (*supd* (*init-class-obj G Object*) .; *P*)⟧ ⟹
G,(*A*::'*a triple set*)⊢{*Normal P*←◇} .*Init Object*. {*P* ∧. *initd Object*}
⟨*proof*⟩

introduction rules for Alloc and SXAlloc

lemma *ax-SXAlloc-Normal*:
G,(*A*::'*a triple set*)⊢{*P*::'*a assn*} .*c*. {*Normal Q*}
⟹ *G*,*A*⊢{*P*} .*c*. {*SXAlloc G Q*}
⟨*proof*⟩

lemma *ax-Alloc*:
G,(*A*::'*a triple set*)⊢{*P*::'*a assn*} *t*≻
{*Normal* (λ*Y* (*x*,*s*) *Z*. (∀ *a*. *new-Addr* (*heap s*) = *Some a* ⟶
Q (*Val* (*Addr a*)) (*Norm*(*init-obj G* (*CInst C*) (*Heap a*) *s*)) *Z*)) ∧.
heap-free (*Suc* (*Suc 0*))}
⟹ *G*,*A*⊢{*P*} *t*≻ {*Alloc G* (*CInst C*) *Q*}
⟨*proof*⟩

lemma *ax-Alloc-Arr*:
G,(*A*::'*a triple set*)⊢{*P*::'*a assn*} *t*≻
{λ*Val*:*i*:. *Normal* (λ*Y* (*x*,*s*) *Z*. ¬*the-Intg i*<*0* ∧
(∀ *a*. *new-Addr* (*heap s*) = *Some a* ⟶
Q (*Val* (*Addr a*)) (*Norm* (*init-obj G* (*Arr T* (*the-Intg i*)) (*Heap a*) *s*)) *Z*)) ∧.
heap-free (*Suc* (*Suc 0*))}
⟹
G,*A*⊢{*P*} *t*≻ {λ*Val*:*i*:. *abupd* (*check-neg i*) .; *Alloc G* (*Arr T*(*the-Intg i*)) *Q*}
⟨*proof*⟩

lemma *ax-SXAlloc-catch-SXcpt*:
⟦*G*,(*A*::'*a triple set*)⊢{*P*::'*a assn*} *t*≻
{(λ*Y* (*x*,*s*) *Z*. *x*=*Some* (*Xcpt* (*Std xn*)) ∧
(∀ *a*. *new-Addr* (*heap s*) = *Some a* ⟶
Q Y (*Some* (*Xcpt* (*Loc a*)),*init-obj G* (*CInst* (*SXcpt xn*)) (*Heap a*) *s*) *Z*))
∧. *heap-free* (*Suc* (*Suc 0*))}⟧
⟹
G,*A*⊢{*P*} *t*≻ {*SXAlloc G* (λ*Y s Z*. *Q Y s Z* ∧ *G*,*s*⊢*catch SXcpt xn*)}
⟨*proof*⟩

end

Chapter 23

AxSound

62 Soundness proof for Axiomatic semantics of Java expressions and statements

theory *AxSound* **imports** *AxSem* **begin**

validity

consts

$$\begin{array}{l} \textit{triple-valid2}:: \textit{prog} \Rightarrow \textit{nat} \Rightarrow \quad 'a\ \textit{triple} \Rightarrow \textit{bool} \\ \qquad\qquad\qquad\qquad\qquad\qquad (\quad \text{-}\models\text{-::-}[61,0,\ 58]\ 57) \\ \quad \textit{ax-valids2}:: \textit{prog} \Rightarrow 'a\ \textit{triples} \Rightarrow 'a\ \textit{triples} \Rightarrow \textit{bool} \\ \qquad\qquad\qquad\qquad\qquad\qquad (\text{-,-}|\models\text{::-}\ [61,58,58]\ 57) \end{array}$$

defs *triple-valid2-def*: $G\models n::t \equiv \textit{case}\ t\ \textit{of}\ \{P\}\ t\succ\ \{Q\} \Rightarrow$
$\forall\ Y\ s\ Z.\ P\ Y\ s\ Z \longrightarrow (\forall\ L.\ s::\preceq(G,L)$
$\longrightarrow (\forall\ T\ C\ A.\ (\textit{normal}\ s \longrightarrow ((\!|prg=G,cls=C,lcl=L|\!)\vdash t::T\ \wedge$
$\qquad (\!|prg=G,cls=C,lcl=L|\!)\vdash \textit{dom}\ (\textit{locals}\ (\textit{store}\ s))\gg t\gg A)) \longrightarrow$
$(\forall\ Y'\ s'.\ G\vdash s\ -t\succ -n\rightarrow\ (Y',s') \longrightarrow Q\ Y'\ s'\ Z\ \wedge\ s'::\preceq(G,L))))$

This definition differs from the ordinary *triple-valid-def* manly in the conclusion: We also ensures conformance of the result state. So we don't have to apply the type soundness lemma all the time during induction. This definition is only introduced for the soundness proof of the axiomatic semantics, in the end we will conclude to the ordinary definition.

defs *ax-valids2-def*: $G,A|\models::ts \equiv\ \forall\ n.\ (\forall\ t\in A.\ G\models n::t) \longrightarrow (\forall\ t\in ts.\ G\models n::t)$

lemma *triple-valid2-def2*: $G\models n::\{P\}\ t\succ\ \{Q\} =$
$(\forall\ Y\ s\ Z.\ P\ Y\ s\ Z \longrightarrow (\forall\ Y'\ s'.\ G\vdash s\ -t\succ -n\rightarrow\ (Y',s')\longrightarrow$
$\ (\forall\ L.\ s::\preceq(G,L) \longrightarrow (\forall\ T\ C\ A.\ (\textit{normal}\ s \longrightarrow ((\!|prg=G,cls=C,lcl=L|\!)\vdash t::T\ \wedge$
$\qquad (\!|prg=G,cls=C,lcl=L|\!)\vdash \textit{dom}\ (\textit{locals}\ (\textit{store}\ s))\gg t\gg A)) \longrightarrow$
$\ Q\ Y'\ s'\ Z\ \wedge\ s'::\preceq(G,L)))))$
⟨*proof*⟩

lemma *triple-valid2-eq* [*rule-format* (*no-asm*)]:
$\ \textit{wf-prog}\ G ==> \textit{triple-valid2}\ G = \textit{triple-valid}\ G$
⟨*proof*⟩

lemma *ax-valids2-eq*: $\textit{wf-prog}\ G \Longrightarrow G,A|\models::ts = G,A|\models ts$
⟨*proof*⟩

lemma *triple-valid2-Suc* [*rule-format* (*no-asm*)]: $G\models \textit{Suc}\ n::t \longrightarrow G\models n::t$
⟨*proof*⟩

lemma *Methd-triple-valid2-0*: $G\models 0::\{\textit{Normal}\ P\}\ \textit{Methd}\ C\ \textit{sig}-\succ\ \{Q\}$
⟨*proof*⟩

lemma *Methd-triple-valid2-SucI*:
$[\![G\models n::\{\textit{Normal}\ P\}\ \textit{body}\ G\ C\ \textit{sig}-\succ\{Q\}]\!]$
$\ \Longrightarrow G\models \textit{Suc}\ n::\{\textit{Normal}\ P\}\ \textit{Methd}\ C\ \textit{sig}-\succ\ \{Q\}$
⟨*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))»⟨c⟩s»C;

$G⊢s0\ -c-n→\ s1;\ P\ Y\ s0\ Z⟧ ⟹ Q\ ◇\ s1\ Z\ ∧\ s1::≼(G,L)$
shows $G,A|⊨::\{\ \{P\}\ ⟨c⟩_s≻\ \{Q\}\ \}$
⟨*proof*⟩

lemma *valid-stmt-NormalI*:
assumes $I: ⋀\ n\ s0\ L\ accC\ C\ s1\ Y\ Z.$
$⟦∀\ t∈A.\ G⊨n::t;\ s0::≼(G,L);\ normal\ s0;\ (|prg=G,cls=accC,lcl=L|)⊢c::√;$
$(|prg=G,cls=accC,lcl=L|)⊢dom\ (locals\ (store\ s0))≫⟨c⟩_s≫C;$
$G⊢s0\ -c-n→\ s1;\ (Normal\ P)\ Y\ s0\ Z⟧ ⟹ Q\ ◇\ s1\ Z\ ∧\ s1::≼(G,L)$
shows $G,A|⊨::\{\ \{Normal\ P\}\ ⟨c⟩_s≻\ \{Q\}\ \}$
⟨*proof*⟩

lemma *valid-var-NormalI*:
assumes $I: ⋀\ n\ s0\ L\ accC\ T\ C\ vf\ s1\ Y\ Z.$
$⟦∀\ t∈A.\ G⊨n::t;\ s0::≼(G,L);\ normal\ s0;$
$(|prg=G,cls=accC,lcl=L|)⊢t::=T;$
$(|prg=G,cls=accC,lcl=L|)⊢dom\ (locals\ (store\ s0))≫⟨t⟩_v≫C;$
$G⊢s0\ -t=≻vf-n→\ s1;\ (Normal\ P)\ Y\ s0\ Z⟧$
$⟹ Q\ (In2\ vf)\ s1\ Z\ ∧\ s1::≼(G,L)$
shows $G,A|⊨::\{\ \{Normal\ P\}\ ⟨t⟩_v≻\ \{Q\}\ \}$
⟨*proof*⟩

lemma *valid-expr-NormalI*:
assumes $I: ⋀\ n\ s0\ L\ accC\ T\ C\ v\ s1\ Y\ Z.$
$⟦∀\ t∈A.\ G⊨n::t;\ s0::≼(G,L);\ normal\ s0;$
$(|prg=G,cls=accC,lcl=L|)⊢t::-T;$
$(|prg=G,cls=accC,lcl=L|)⊢dom\ (locals\ (store\ s0))≫⟨t⟩_e≫C;$
$G⊢s0\ -t-≻v-n→\ s1;\ (Normal\ P)\ Y\ s0\ Z⟧$
$⟹ Q\ (In1\ v)\ s1\ Z\ ∧\ s1::≼(G,L)$
shows $G,A|⊨::\{\ \{Normal\ P\}\ ⟨t⟩_e≻\ \{Q\}\ \}$
⟨*proof*⟩

lemma *valid-expr-list-NormalI*:
assumes $I: ⋀\ n\ s0\ L\ accC\ T\ C\ vs\ s1\ Y\ Z.$
$⟦∀\ t∈A.\ G⊨n::t;\ s0::≼(G,L);\ normal\ s0;$
$(|prg=G,cls=accC,lcl=L|)⊢t::�doteq T;$
$(|prg=G,cls=accC,lcl=L|)⊢dom\ (locals\ (store\ s0))≫⟨t⟩_l≫C;$
$G⊢s0\ -t�doteq≻vs-n→\ s1;\ (Normal\ P)\ Y\ s0\ Z⟧$
$⟹ Q\ (In3\ vs)\ s1\ Z\ ∧\ s1::≼(G,L)$
shows $G,A|⊨::\{\ \{Normal\ P\}\ ⟨t⟩_l≻\ \{Q\}\ \}$
⟨*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: $(\!|prg=G,cls=accC,lcl=L|\!)\vdash dom\ (locals\ (store\ s0))\ \gg t\gg\ A$ **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-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

63 Completeness proof for Axiomatic semantics of Java expressions and statements

theory *AxCompl* **imports** *AxSem* **begin**

design issues:

- proof structured by Most General Formulas (-¿ Thomas Kleymann)

set of not yet initialzed classes

constdefs

nyinitcls :: *prog* ⇒ *state* ⇒ *qtname set*
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*], *standard*]

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*]:!!*s*. *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*:
$[\![card\ (insert\ a\ A) \leq Suc\ n;\ a \notin A;\ finite\ A]\!] \Longrightarrow card\ A \leq n$
⟨*proof*⟩

lemma *nyinitcls-le-SucD*:
$[\![card\ (nyinitcls\ G\ (x,s)) \leq Suc\ n;\ \neg inited\ C\ (globs\ s);\ class\ G\ C = Some\ y]\!] \Longrightarrow$
$card\ (nyinitcls\ G\ (x,init\text{-}class\text{-}obj\ G\ C\ s)) \leq n$
⟨*proof*⟩

lemma *inited-gext'*: $[\![s \leq| s';inited\ C\ (globs\ s)]\!] \Longrightarrow inited\ C\ (globs\ s')$
⟨*proof*⟩

lemma *nyinitcls-gext*: $snd\ s \leq| snd\ s' \Longrightarrow nyinitcls\ G\ s' \subseteq nyinitcls\ G\ s$
⟨*proof*⟩

lemma *card-nyinitcls-gext*:
$[\![snd\ s \leq| snd\ s';\ card\ (nyinitcls\ G\ s) \leq n]\!] \Longrightarrow card\ (nyinitcls\ G\ s') \leq n$
⟨*proof*⟩

init-le

constdefs
init-le :: *prog* ⇒ *nat* ⇒ *state* ⇒ *bool* (-⊢*init*≤- [*51*,*51*] *50*)
$G \vdash init \leq n \equiv \lambda s.\ card\ (nyinitcls\ G\ s) \leq n$

lemma *init-le-def2* [*simp*]: $(G \vdash init \leq n)\ s = (card\ (nyinitcls\ G\ s) \leq n)$
⟨*proof*⟩

lemma *All-init-leD*:
$\forall n::nat.\ G,(A::'a\ triple\ set) \vdash \{P \wedge.\ G \vdash init \leq n\}\ t \succ \{Q::'a\ assn\}$
$\Longrightarrow G,A \vdash \{P\}\ t \succ \{Q\}$
⟨*proof*⟩

Most General Triples and Formulas

constdefs

remember-init-state :: *state assn* (�doteq)
$\doteq \equiv \lambda Y\ s\ Z.\ s = Z$

lemma *remember-init-state-def2* [*simp*]: $\doteq Y = op =$
⟨*proof*⟩

consts

MGF ::[*state assn*, *term*, *prog*] ⇒ *state triple* ({-} -≻ {-→}[*3*,*65*,*3*]*62*)
MGFn::[*nat* , *term*, *prog*] ⇒ *state triple* ({=:-} -≻ {-→}[*3*,*65*,*3*]*62*)

defs

MGF-def:
$\{P\}\ t\succ\ \{G\rightarrow\} \equiv \{P\}\ t\succ\ \{\lambda Y\ s'\ s.\ G\vdash s\ -t\succ\rightarrow\ (Y,s')\}$

MGFn-def:
$\{=:n\}\ t\succ\ \{G\rightarrow\} \equiv \{\doteq \wedge.\ G\vdash init\leq n\}\ t\succ\ \{G\rightarrow\}$

lemma *MGF-valid*: *wf-prog* $G \Longrightarrow G,\{\}\models\{\doteq\}\ t\succ\ \{G\rightarrow\}$
⟨*proof*⟩

lemma *MGF-res-eq-lemma* [*simp*]:
$(\forall\ Y'\ Y\ s.\ Y = Y' \wedge\ P\ s \longrightarrow\ Q\ s) = (\forall\ s.\ P\ s \longrightarrow\ Q\ s)$
⟨*proof*⟩

lemma *MGFn-def2*:
$G,A\vdash\{=:n\}\ t\succ\ \{G\rightarrow\} = G,A\vdash\{\doteq \wedge.\ G\vdash init\leq n\}$
$t\succ\ \{\lambda Y\ s'\ s.\ G\vdash s\ -t\succ\rightarrow\ (Y,s')\}$
⟨*proof*⟩

lemma *MGF-MGFn-iff*:
$G,(A::state\ triple\ set)\vdash\{\doteq\}\ t\succ\ \{G\rightarrow\} = (\forall\ n.\ G,A\vdash\{=:n\}\ t\succ\ \{G\rightarrow\})$
⟨*proof*⟩

lemma *MGFnD*:
$G,(A::state\ triple\ set)\vdash\{=:n\}\ t\succ\ \{G\rightarrow\} \Longrightarrow$
$G,A\vdash\{(\lambda Y'\ s'\ s.\ s' = s\ \ \wedge\ P\ s) \wedge.\ G\vdash init\leq n\}$
$t\succ\ \ \{(\lambda Y'\ s'\ s.\ G\vdash s-t\succ\rightarrow(Y',s') \wedge\ P\ s) \wedge.\ G\vdash init\leq n\}$
⟨*proof*⟩
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*: $G,A\vdash\{Normal\ \doteq\}\ t\succ\ \{G\rightarrow\} \Longrightarrow$
$G,(A::state\ triple\ set)\vdash\{\doteq::state\ assn\}\ t\succ\ \{G\rightarrow\}$
⟨*proof*⟩

lemma *MGFNormalD*:
$G,(A::state\ triple\ set)\vdash\{\doteq\}\ t\succ\ \{G\rightarrow\} \Longrightarrow G,A\vdash\{Normal\ \doteq\}\ t\succ\ \{G\rightarrow\}$
⟨*proof*⟩

Additionally to *MGFNormalI*, we also expand the definition of the most general formula here

lemma *MGFn-NormalI*:
$G,(A::state\ triple\ set)\vdash\{Normal((\lambda Y'\ s'\ s.\ s'=s \wedge\ normal\ s) \wedge.\ G\vdash init\leq n)\}t\succ$
$\{\lambda Y\ s'\ s.\ G\vdash s\ -t\succ\rightarrow\ (Y,s')\} \Longrightarrow G,A\vdash\{=:n\}t\succ\{G\rightarrow\}$
⟨*proof*⟩

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*:
$(\exists\ T\ L\ C.\ (\!|prg=G,cls=C,lcl=L|\!)\vdash t::T)$
$\longrightarrow\ G,(A::state\ triple\ set)\vdash\{=:n\}\ t\succ\ \{G\rightarrow\}$

$\Longrightarrow G,A\vdash\{=:n\}\ t\succ\ \{G\rightarrow\}$
⟨*proof*⟩

To derive the most general formula, we can restrict ourselves to welltyped terms and assume that the state in the precondition conforms to the environment. All type violations can be uniformally handled by the hazard rule.

lemma *MGFn-free-wt-NormalConformI*:
$(\forall\ T\ L\ C\ .\ (\!|prg=G,cls=C,lcl=L|\!)\vdash t::T$
$\longrightarrow G,(A::state\ triple\ set)$
$\vdash\{Normal((\lambda Y'\ s'\ s.\ s'=s\ \wedge\ normal\ s)\ \wedge.\ G\vdash init\leq n)\ \wedge.\ (\lambda\ s.\ s::\preceq(G,\ L))\}$
$t\succ$
$\{\lambda Y\ s'\ s.\ G\vdash s\ -t\succ\rightarrow\ (Y,s')\}$)
$\Longrightarrow G,A\vdash\{=:n\}t\succ\{G\rightarrow\}$
⟨*proof*⟩

To derive the most general formula, we can restrict ourselves to welltyped terms and assume that the state in the precondition conforms to the environment and that the term is definetly assigned with respect to this state. All type violations can be uniformally handled by the hazard rule.

lemma *MGFn-free-wt-da-NormalConformI*:
$(\forall\ T\ L\ C\ B.\ (\!|prg=G,cls=C,lcl=L|\!)\vdash t::T$
$\longrightarrow G,(A::state\ triple\ set)$
$\vdash\{Normal((\lambda Y'\ s'\ s.\ s'=s\ \wedge\ normal\ s)\ \wedge.\ G\vdash init\leq n)\ \wedge.\ (\lambda\ s.\ s::\preceq(G,\ L))$
$\wedge.\ (\lambda\ s.\ (\!|prg=G,cls=C,lcl=L|\!)\vdash dom\ (locals\ (store\ s))\gg t\gg B)\}$
$t\succ$
$\{\lambda Y\ s'\ s.\ G\vdash s\ -t\succ\rightarrow\ (Y,s')\}$)
$\Longrightarrow G,A\vdash\{=:n\}t\succ\{G\rightarrow\}$
⟨*proof*⟩

main lemmas

lemma *MGFn-Init*:
assumes *mgf-hyp*: $\forall\, m.\ Suc\ m\leq n\ \longrightarrow\ (\forall\, t.\ G,A\vdash\{=:m\}\ t\succ\ \{G\rightarrow\})$
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.

constdefs
unroll:: *prog* ⇒ *label* ⇒ *expr* ⇒ *stmt* ⇒ (*state* × *state*) *set*

unroll G l e c ≡ {(*s*,*t*). ∃ *v s1 s2*.
G⊢*s* −*e*−≻*v*→ *s1* ∧ *the-Bool v* ∧ *normal s1* ∧
G⊢*s1* −*c*→ *s2* ∧ *t*=(*abupd* (*absorb* (*Cont l*)) *s2*)}

lemma *unroll-while*:
assumes *unroll*: (*s*, *t*) ∈ (*unroll G l e c*)*
and *eval-e*: *G*⊢*t* −*e*−≻*v*→ *s'*
and *normal-termination*: *normal s'* ⟶ ¬ *the-Bool v*
and *wt*: (|*prg*=*G*,*cls*=*C*,*lcl*=*L*|)⊢*e*::−*T*
and *wf*: *wf-prog G*
shows *G*⊢*s* −*l*• *While*(*e*) *c*→ *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*: $G,A\vdash\{=:n\}\ \langle Init\ D\rangle_s\succ\ \{G\rightarrow\}$
and *mgf-c*: $G,A\vdash\{=:n\}\ \langle c\rangle_s\succ\ \{G\rightarrow\}$
shows $G,(A::state\ triple\ set)\vdash\{=:n\}\ \langle Body\ D\ c\rangle_e\succ\ \{G\rightarrow\}$
⟨*proof*⟩

lemma *MGFn-lemma*:
assumes *mgf-methds*:
$\bigwedge n.\ \forall\ C\ sig.\ G,(A::state\ triple\ set)\vdash\{=:n\}\ \langle Methd\ C\ sig\rangle_e\succ\ \{G\rightarrow\}$
and *wf*: *wf-prog G*
shows $\bigwedge t.\ G,A\vdash\{=:n\}\ t\succ\ \{G\rightarrow\}$
⟨*proof*⟩

lemma *MGF-asm*:
$[\![\forall\ C\ sig.\ is\text{-}methd\ G\ C\ sig\longrightarrow G,A\vdash\{\doteq\}\ In1l\ (Methd\ C\ sig)\succ\ \{G\rightarrow\};\ wf\text{-}prog\ G]\!]$
$\Longrightarrow G,(A::state\ triple\ set)\vdash\{\doteq\}\ t\succ\ \{G\rightarrow\}$
⟨*proof*⟩

nested version

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*: $\bigwedge A\ pn.\ \forall\, b{\in}bdy\ pn.\ P\ (insert\ (mgf\text{-}call\ pn)\ A)\ \{mgf\ b\}$
$\Longrightarrow P\ A\ \{mgf\text{-}call\ pn\}$
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*: $uA = mgf\text{-}call`U$
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-prog G* $\Longrightarrow G,(\{\}::state\ triple\ set)\vdash\{\doteq\}\ t\succ\ \{G\rightarrow\}$
⟨*proof*⟩

simultaneous version

lemma *MGF-simult-Methd-lemma*: *finite ms* $\Longrightarrow$
$G,A \cup (\lambda(C,sig).\ \{Normal\ \doteq\}\ \langle Methd\ \ C\ sig\rangle_e\succ\ \{G\rightarrow\})\ `\ ms$
$\quad\vdash(\lambda(C,sig).\ \{Normal\ \doteq\}\ \langle body\ G\ C\ sig\rangle_e\succ\ \{G\rightarrow\})\ `\ ms \Longrightarrow$
$G,A\vdash(\lambda(C,sig).\ \{Normal\ \doteq\}\ \langle Methd\ \ C\ sig\rangle_e\succ\ \{G\rightarrow\})\ `\ ms$
⟨*proof*⟩

lemma *MGF-simult-Methd*: *wf-prog G* $\Longrightarrow$
$G,(\{\}::state\ triple\ set)\vdash(\lambda(C,sig).\ \{Normal\ \doteq\}\ \langle Methd\ C\ sig\rangle_e\succ\ \{G\rightarrow\})$
$`\ Collect\ (split\ (is\text{-}methd\ G))$
⟨*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

64 Example of a proof based on the Bali axiomatic semantics

theory *AxExample* **imports** *AxSem Example* **begin**

constdefs
  arr-inv :: *st* ⇒ *bool*
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 *split-if-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*}
⟨*proof*⟩

end