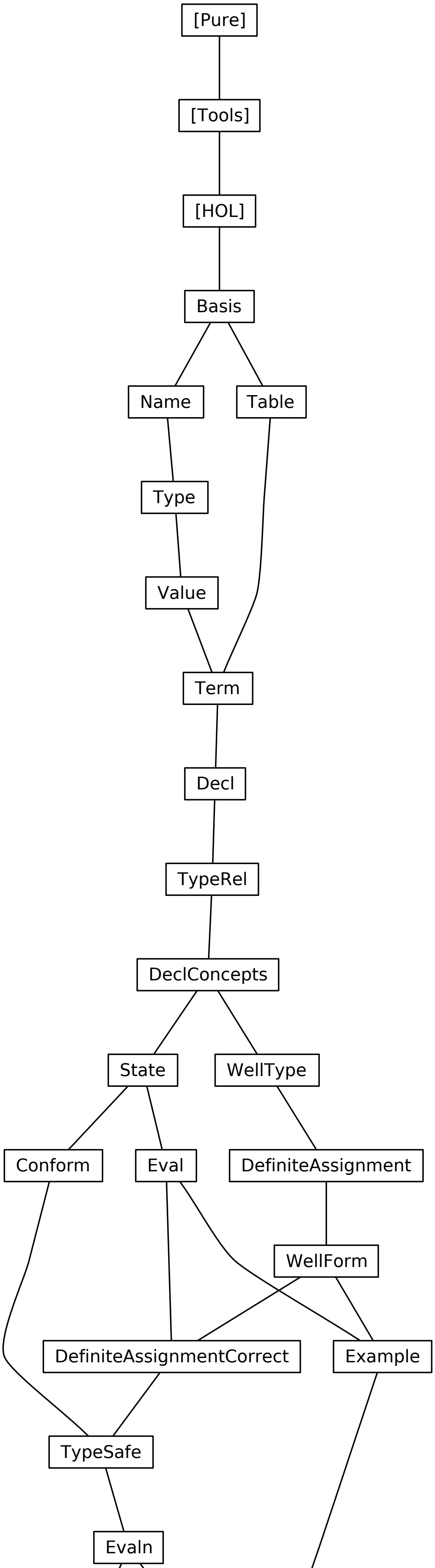
Java Source and Bytecode Formalizations in Isabelle: Bali

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

September 11, 2023

Contents


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

Chapter 1

Overview

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

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

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

The following features are missing in Bali wrt. JavaCard:

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

- Inner Classes

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

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

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

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

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

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

Term JavaCard terms. Variables, expressions and statements.

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

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

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

WellType Typesystem on the JavaCard term level.

DefiniteAssignment The definite assignment analysis on the JavaCard term level.

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

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

Eval Operational (big step) semantics for JavaCard.

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

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

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

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

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

Trans A smallstep operational semantics for JavaCard.

AxSem An axiomatic semantics (Hoare logic) for JavaCard.

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

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

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

Chapter 2

Basis

1 Definitions extending HOL as logical basis of Bali

theory *Basis*
imports *Main*
begin

misc

ML ‹*fun strip-tac ctxt i = REPEAT (resolve-tac ctxt [impI, allI] i)*›

declare *if-split-asm* [*split*] *option.split* [*split*] *option.split-asm* [*split*]
setup ‹*map-theory-simpset (fn ctxt => ctxt addloop (split-all-tac, split-all-tac))*›
declare *if-weak-cong* [*cong del*] *option.case-cong-weak* [*cong del*]
declare *length-Suc-conv* [*iff*]

lemma *Collect-split-eq*: $\{p.\ P\ (case\text{-}prod\ f\ p)\} = \{(a,b).\ P\ (f\ a\ b)\}$
  by *auto*

lemma *subset-insertD*: $A \subseteq insert\ x\ B \Longrightarrow A \subseteq B \wedge x \notin A \vee (\exists B'.\ A = insert\ x\ B' \wedge B' \subseteq B)$
  apply (*case-tac* $x \in A$)
   apply (*rule disjI2*)
   apply (*rule-tac* $x = A - \{x\}$ **in** *exI*)
   apply *fast+*
  done

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

lemma *irrefl-tranclI'*: $r^{-1} \cap r^{+} = \{\} \Longrightarrow \forall x.\ (x, x) \notin r^{+}$
  by (*blast elim*: *tranclE dest*: *trancl-into-rtrancl*)

lemma *trancl-rtrancl-trancl*: $[\![(x, y) \in r^{+}; (y, z) \in r^{*}]\!] \Longrightarrow (x, z) \in r^{+}$
  by (*auto dest*: *tranclD rtrancl-trans rtrancl-into-trancl2*)

lemma *rtrancl-into-trancl3*: $[\![(a, b) \in r^{*}; a \neq b]\!] \Longrightarrow (a, b) \in r^{+}$
  apply (*drule rtranclD*)
  apply *auto*

done

lemma *rtrancl-into-rtrancl2*: $[\![(a,\ b) \in\ r;\ (b,\ c) \in r^*]\!] \Longrightarrow (a,\ c) \in r^*$
by (*auto intro*: *rtrancl-trans*)

lemma *triangle-lemma*:
assumes *unique*: $\bigwedge a\ b\ c.\ [\![(a,b) \in r;\ (a,c) \in r]\!] \Longrightarrow b = c$
and *ax*: $(a,x) \in r^*$ **and** *ay*: $(a,y) \in r^*$
shows $(x,y) \in r^* \vee (y,x) \in r^*$
using *ax ay*
proof (*induct rule*: *converse-rtrancl-induct*)
assume $(x,y) \in r^*$
then show *?thesis* **by** *blast*
next
fix *a v*
assume *a-v-r*: $(a,\ v) \in r$
and *v-x-rt*: $(v,\ x) \in r^*$
and *a-y-rt*: $(a,\ y) \in r^*$
and *hyp*: $(v,\ y) \in r^* \Longrightarrow (x,\ y) \in r^* \vee (y,\ x) \in r^*$
from *a-y-rt* **show** $(x,\ y) \in r^* \vee (y,\ x) \in r^*$
proof (*cases rule*: *converse-rtranclE*)
assume $a = y$
with *a-v-r v-x-rt* **have** $(y,x) \in r^*$
by (*auto intro*: *rtrancl-trans*)
then show *?thesis* **by** *blast*
next
fix *w*
assume *a-w-r*: $(a,\ w) \in r$
and *w-y-rt*: $(w,\ y) \in r^*$
from *a-v-r a-w-r unique* **have** $v=w$ **by** *auto*
with *w-y-rt hyp* **show** *?thesis* **by** *blast*
qed
qed

lemma *rtrancl-cases*:
assumes $(a,b) \in r^*$
obtains (*Refl*) $a = b$
| (*Trancl*) $(a,b) \in r^+$
apply (*rule rtranclE* [*OF assms*])
apply (*auto dest*: *rtrancl-into-trancl1*)
done

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

lemma *finite-SetCompr2*:
finite $\{f\ y\ x\ |x\ y.\ P\ y\}$ **if** *finite* (*Collect P*)
$\forall y.\ P\ y \longrightarrow$ *finite* (*range* $(f\ y)$)
proof −
have $\{f\ y\ x\ |x\ y.\ P\ y\} = (\bigcup y \in Collect\ P.\ range\ (f\ y))$
by *auto*
with *that* **show** *?thesis* **by** *simp*
qed

lemma *list-all2-trans*: $\forall\, a\ b\ c.\ P1\ a\ b \longrightarrow P2\ b\ c \longrightarrow P3\ a\ c \Longrightarrow$
$\forall\, xs2\ xs3.\ list\text{-}all2\ P1\ xs1\ xs2 \longrightarrow list\text{-}all2\ P2\ xs2\ xs3 \longrightarrow list\text{-}all2\ P3\ xs1\ xs3$
apply (*induct-tac xs1*)
apply *simp*
apply (*rule allI*)
apply (*induct-tac xs2*)
apply *simp*
apply (*rule allI*)
apply (*induct-tac xs3*)
apply *auto*
done

pairs

lemma *surjective-pairing5*:
$p = (fst\ p,\ fst\ (snd\ p),\ fst\ (snd\ (snd\ p)),\ fst\ (snd\ (snd\ (snd\ p))),$
$snd\ (snd\ (snd\ (snd\ p))))$
by *auto*

lemma *fst-splitE* [*elim*!]:
assumes $fst\ s' = x'$
obtains $x\ s$ **where** $s' = (x,s)$ **and** $x = x'$
using *assms* **by** (*cases* s') *auto*

lemma *fst-in-set-lemma*: $(x,\ y) \in set\ l \Longrightarrow x \in fst\ `\ set\ l$
by (*induct l*) *auto*

quantifiers

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

lemma *ex-ex-miniscope1* [*simp*]: $(\exists\, w\ v.\ P\ w\ v \wedge Q\ v) = (\exists\, v.\ (\exists\, w.\ P\ w\ v) \wedge Q\ v)$
by *auto*

lemma *ex-miniscope2* [*simp*]: $(\exists\, v.\ P\ v \wedge Q \wedge R\ v) = (Q \wedge (\exists\, v.\ P\ v \wedge R\ v))$
by *auto*

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

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

sums

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

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

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

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

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

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

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

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

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

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

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

ML ‹
```
fun sum3-instantiate ctxt thm =
  map (fn s =>
    simplify (ctxt delsimps @{thms not-None-eq})
      (Rule-Insts.read-instantiate ctxt [((("t", 0), Position.none), "In" ^ s ^ " x")] ["x"] thm))
    ["1l","2","3","1r"]
```
›

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 set-option A*. *P*
  ∃ *x*∈*A*: *P* ⇌ ∃ *x*∈*CONST set-option A*. *P*

Special map update

Deemed too special for theory Map.

definition *chg-map* :: ('b ⇒ 'b) ⇒ 'a ⇒ ('a ⇀ 'b) ⇒ ('a ⇀ 'b)
  where *chg-map f a m* = (*case m a of None* ⇒ *m* | *Some b* ⇒ *m*(*a*↦*f b*))

lemma *chg-map-new*[*simp*]: *m a* = *None* ⟹ *chg-map f a m* = *m*
  unfolding *chg-map-def* **by** *auto*

lemma *chg-map-upd*[*simp*]: $m\ a = Some\ b \Longrightarrow chg\text{-}map\ f\ a\ m = m(a \mapsto f\ b)$
unfolding *chg-map-def* **by** *auto*

lemma *chg-map-other* [*simp*]: $a \neq b \Longrightarrow chg\text{-}map\ f\ a\ m\ b = m\ b$
by (*auto simp*: *chg-map-def*)

unique association lists

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

lemma *uniqueD*: $unique\ l \Longrightarrow (x,\ y) \in set\ l \Longrightarrow (x',\ y') \in set\ l \Longrightarrow x = x' \Longrightarrow y = y'$
unfolding *unique-def o-def*
by (*induct l*) (*auto dest*: *fst-in-set-lemma*)

lemma *unique-Nil* [*simp*]: *unique* []
by (*simp add*: *unique-def*)

lemma *unique-Cons* [*simp*]: $unique\ ((x,y)\#l) = (unique\ l \wedge (\forall\, y.\ (x,y) \notin set\ l))$
by (*auto simp*: *unique-def dest*: *fst-in-set-lemma*)

lemma *unique-ConsD*: $unique\ (x\#xs) \Longrightarrow unique\ xs$
by (*simp add*: *unique-def*)

lemma *unique-single* [*simp*]: $\bigwedge p.\ unique\ [p]$
by *simp*

lemma *unique-append* [*rule-format* (*no-asm*)]: $unique\ l' \Longrightarrow unique\ l \Longrightarrow$
$(\forall\, (x,y){\in}set\ l.\ \forall\, (x',y'){\in}set\ l'.\ x' \neq x) \longrightarrow unique\ (l\ @\ l')$
by (*induct l*) (*auto dest*: *fst-in-set-lemma*)

lemma *unique-map-inj*: $unique\ l \Longrightarrow inj\ f \Longrightarrow unique\ (map\ (\lambda(k,x).\ (f\ k,\ g\ k\ x))\ l)$
by (*induct l*) (*auto dest*: *fst-in-set-lemma simp add*: *inj-eq*)

lemma *map-of-SomeI*: $unique\ l \Longrightarrow (k,\ x) \in set\ l \Longrightarrow map\text{-}of\ l\ k = Some\ x$
by (*induct l*) *auto*

list patterns

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

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

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

lemma *lsplit* [*simp*]: *lsplit c* (*x*#*xs*) = *c x xs*
by (*simp add*: *lsplit-def*)

lemma *lsplit2* [*simp*]: *lsplit P* (*x*#*xs*) *y z* = *P x xs y z*
by (*simp add*: *lsplit-def*)

end

Chapter 3

Table

1 Abstract tables and their implementation as lists

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

design issues:

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

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

map of / table of

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

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

lemma *map-add-find-left*[*simp*]: $n\ k = None \Longrightarrow (m\ {+}{+}\ n)\ k = m\ k$
  by (*simp add*: *map-add-def*)

Conditional Override

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

— when merging tables old and new, only override an entry of table old when the condition cond holds
cond-override cond old new =
 (λ*k*.
  (*case new k of*
     None ⇒ *old k*
   | *Some new-val* ⇒ (*case old k of*
        None ⇒ *Some new-val*
      | *Some old-val* ⇒ (*if cond new-val old-val*
           then Some new-val
           else Some old-val))))

lemma *cond-override-empty1*[*simp*]: *cond-override c Map.empty t* = *t*
 by (*simp add*: *cond-override-def fun-eq-iff*)

lemma *cond-override-empty2*[*simp*]: *cond-override c t Map.empty* = *t*
 by (*simp add*: *cond-override-def fun-eq-iff*)

lemma *cond-override-None*[*simp*]:
 old k = *None* ⟹ (*cond-override c old new*) *k* = *new k*
 by (*simp add*: *cond-override-def*)

lemma *cond-override-override*:
 ⟦*old k* = *Some ov*;*new k* = *Some nv*; *C nv ov*⟧
  ⟹ (*cond-override C old new*) *k* = *Some nv*
 by (*auto simp add*: *cond-override-def*)

lemma *cond-override-noOverride*:
 ⟦*old k* = *Some ov*;*new k* = *Some nv*; ¬ (*C nv ov*)⟧
  ⟹ (*cond-override C old new*) *k* = *Some ov*
 by (*auto simp add*: *cond-override-def*)

lemma *dom-cond-override*: *dom* (*cond-override C s t*) ⊆ *dom s* ∪ *dom t*
 by (*auto simp add*: *cond-override-def dom-def*)

lemma *finite-dom-cond-override*:
 ⟦ *finite* (*dom s*); *finite* (*dom t*) ⟧ ⟹ *finite* (*dom* (*cond-override C s t*))
apply (*rule-tac B*=*dom s* ∪ *dom t* **in** *finite-subset*)
apply (*rule dom-cond-override*)
by (*rule finite-UnI*)

Filter on Tables

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

lemma *filter-tab-empty*[*simp*]: *filter-tab c Map.empty* = *Map.empty*
by (*simp add*: *filter-tab-def empty-def*)

lemma *filter-tab-True*[*simp*]: *filter-tab* ($\lambda x\ y.\ True$) $t = t$
by (*simp add*: *fun-eq-iff filter-tab-def*)

lemma *filter-tab-False*[*simp*]: *filter-tab* ($\lambda x\ y.\ False$) $t = Map.empty$
by (*simp add*: *fun-eq-iff filter-tab-def empty-def*)

lemma *filter-tab-ran-subset*: *ran* (*filter-tab c t*) $\subseteq$ *ran t*
by (*auto simp add*: *filter-tab-def ran-def*)

lemma *filter-tab-range-subset*: *range* (*filter-tab c t*) $\subseteq$ *range t* $\cup$ {*None*}
apply (*auto simp add*: *filter-tab-def*)
apply (*drule sym*, *blast*)
done

lemma *finite-range-filter-tab*:
finite (*range t*) $\Longrightarrow$ *finite* (*range* (*filter-tab c t*))
apply (*rule-tac B=range t* $\cup$ {*None*} **in** *finite-subset*)
apply (*rule filter-tab-range-subset*)
apply (*auto intro*: *finite-UnI*)
done

lemma *filter-tab-SomeD*[*dest*!]:
filter-tab c t k = *Some x* $\Longrightarrow$ (*t k* = *Some x*) $\wedge$ *c k x*
by (*auto simp add*: *filter-tab-def*)

lemma *filter-tab-SomeI*: $[\![$*t k* = *Some x*;*C k x*$]\!]$ $\Longrightarrow$*filter-tab C t k* = *Some x*
by (*simp add*: *filter-tab-def*)

lemma *filter-tab-all-True*:
$\forall$ *k y*. *t k* = *Some y* $\longrightarrow$ *p k y* $\Longrightarrow$*filter-tab p t* = *t*
apply (*auto simp add*: *filter-tab-def fun-eq-iff*)
done

lemma *filter-tab-all-True-Some*:
$[\![\forall$ *k y*. *t k* = *Some y* $\longrightarrow$ *p k y*; *t k* = *Some v*$]\!]$ $\Longrightarrow$ *filter-tab p t k* = *Some v*
by (*auto simp add*: *filter-tab-def fun-eq-iff*)

lemma *filter-tab-all-False*:
$\forall$ *k y*. *t k* = *Some y* $\longrightarrow$ $\neg$ *p k y* $\Longrightarrow$*filter-tab p t* = *Map.empty*
by (*auto simp add*: *filter-tab-def fun-eq-iff*)

lemma *filter-tab-None*: *t k* = *None* $\Longrightarrow$ *filter-tab p t k* = *None*
apply (*simp add*: *filter-tab-def fun-eq-iff*)
done

lemma *filter-tab-dom-subset*: *dom* (*filter-tab C t*) $\subseteq$ *dom t*
by (*auto simp add*: *filter-tab-def dom-def*)

lemma *filter-tab-eq*: ⟦*a*=*b*⟧ ⟹ *filter-tab C a* = *filter-tab C b*
by (*auto simp add*: *fun-eq-iff filter-tab-def*)

lemma *finite-dom-filter-tab*:
finite (*dom t*) ⟹ *finite* (*dom* (*filter-tab C t*))
apply (*rule-tac B*=*dom t* **in** *finite-subset*)
by (*rule filter-tab-dom-subset*)

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*
by (*force simp add*: *filter-tab-def*)

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*)
by (*auto simp add*: *fun-eq-iff cond-override-def filter-tab-def*)

Misc

lemma *Ball-set-table*: (∀ (*x*,*y*)∈ *set l*. *P x y*) ⟹ ∀ *x*. ∀ *y*∈ *map-of l x*: *P x y*
apply (*erule rev-mp*)
apply (*induct l*)
apply *simp*
apply (*simp* (*no-asm*))
apply *auto*
done

lemma *Ball-set-tableD*:
⟦(∀ (*x*,*y*)∈ *set l*. *P x y*); *x* ∈ *set-option* (*table-of l xa*)⟧ ⟹ *P xa x*
apply (*frule Ball-set-table*)
by *auto*

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

lemma *table-of-Some-in-set*:
table-of l k = *Some x* ⟹ (*k*,*x*) ∈ *set l*
by *auto*

lemma *set-get-eq*:
unique l ⟹ (*k*, *the* (*table-of l k*)) ∈ *set l* = (*table-of l k* ≠ *None*)
by (*auto dest*!: *weak-map-of-SomeI*)

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

apply (*rule inj-onI*)
apply *auto*
done

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*
  by (*induct t*) *auto*

lemma *table-of-mapconst-NoneI*:
  ⟦*table-of t k* = *None*⟧ ⟹
    table-of (*map* (λ(*k*,*x*). (*k*,*c*,*x*)) *t*) *k* = *None*
  by (*induct t*) *auto*

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

lemma *table-of-map-SomeI*: *table-of t k* = *Some x* ⟹
   table-of (*map* (λ(*k*,*x*). (*k*, *f x*)) *t*) *k* = *Some* (*f x*)
  by (*induct t*) *auto*

lemma *table-of-remap-SomeD*:
  table-of (*map* (λ((*k*,*k*′),*x*). (*k*,(*k*′,*x*))) *t*) *k* = *Some* (*k*′,*x*) ⟹
    table-of t (*k*, *k*′) = *Some x*
  by (*induct t*) *auto*

lemma *table-of-mapf-Some*:
  ∀ *x y*. *f x* = *f y* ⟶ *x* = *y* ⟹
    table-of (*map* (λ(*k*,*x*). (*k*,*f x*)) *t*) *k* = *Some* (*f x*) ⟹ *table-of t k* = *Some x*
  by (*induct t*) *auto*

lemma *table-of-mapf-SomeD* [*dest*!]:
  table-of (*map* (λ(*k*,*x*). (*k*, *f x*)) *t*) *k* = *Some z* ⟹ (∃ *y*∈*table-of t k*: *z*=*f y*)
  by (*induct t*) *auto*

lemma *table-of-mapf-NoneD* [*dest*!]:
  table-of (*map* (λ(*k*,*x*). (*k*, *f x*)) *t*) *k* = *None* ⟹ (*table-of t k* = *None*)
  by (*induct t*) *auto*

lemma *table-of-mapkey-SomeD* [*dest*!]:
  table-of (*map* (λ(*k*,*x*). ((*k*,*C*),*x*)) *t*) (*k*,*D*) = *Some x* ⟹ *C* = *D* ∧ *table-of t k* = *Some x*
  by (*induct t*) *auto*

lemma *table-of-mapkey-SomeD2* [*dest*!]:
  table-of (*map* (λ(*k*,*x*). ((*k*,*C*),*x*)) *t*) *ek* = *Some x* ⟹
    C = *snd ek* ∧ *table-of t* (*fst ek*) = *Some x*
  by (*induct t*) *auto*

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

apply (*simp*)
apply (*rule map-add-Some-iff*)
done

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*
by (*induct xs*) (*auto del*: *map-of-SomeD intro*!: *map-of-SomeD*)

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

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

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

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

definition
— variant for a unique table and conditional overriding:
cond-hiding-entails :: ('*a*, '*b*) *table* $\Rightarrow$ ('*a*, '*c*) *table*
$\Rightarrow$ ('*b* $\Rightarrow$ '*c* $\Rightarrow$ *bool*) $\Rightarrow$ ('*b* $\Rightarrow$ '*c* $\Rightarrow$ *bool*) $\Rightarrow$ *bool*
(- *hiding* - *under* - *entails* - *20*)
where (*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*]: $t \in ts \Longrightarrow x \in t\ k \Longrightarrow x \in$ *Un-tables ts k*
by (*auto simp add*: *Un-tables-def*)

lemma *Un-tablesD* [*dest*!]: $x \in$ *Un-tables ts k* $\Longrightarrow \exists t.\ t \in ts \wedge x \in t\ k$
by (*auto simp add*: *Un-tables-def*)

lemma *Un-tables-empty* [*simp*]: *Un-tables* {} = (λk. {})
by (*simp add*: *Un-tables-def*)

overrides

lemma *empty-overrides-t* [*simp*]: (λk. {}) $\oplus\oplus$ *m* = *m*
by (*simp add*: *overrides-t-def*)

lemma *overrides-empty-t* [*simp*]: *m* $\oplus\oplus$ (λk. {}) = *m*
by (*simp add*: *overrides-t-def*)

lemma *overrides-t-Some-iff*:

$(x \in (s \oplus\oplus\ t)\ k) = (x \in t\ k \vee t\ k = \{\} \wedge x \in s\ k)$
by (*simp add*: *overrides-t-def*)

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

lemma *overrides-t-right-empty* [*simp*]: $n\ k = \{\} \Longrightarrow (m \oplus\oplus\ n)\ k = m\ k$
by (*simp add*: *overrides-t-def*)

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

hiding entails

lemma *hiding-entailsD*:
t hiding s entails R $\Longrightarrow$ *t k* = *Some x* $\Longrightarrow$ *s k* = *Some y* $\Longrightarrow$ *R x y*
by (*simp add*: *hiding-entails-def*)

lemma *empty-hiding-entails* [*simp*]: *Map.empty hiding s entails R*
by (*simp add*: *hiding-entails-def*)

lemma *hiding-empty-entails* [*simp*]: *t hiding Map.empty entails R*
by (*simp add*: *hiding-entails-def*)

cond hiding entails

lemma *cond-hiding-entailsD*:
⟦*t hiding s under C entails R*; *t k* = *Some x*; *s k* = *Some y*; *C x y*⟧ $\Longrightarrow$ *R x y*
by (*simp add*: *cond-hiding-entails-def*)

lemma *empty-cond-hiding-entails*[*simp*]: *Map.empty hiding s under C entails R*
by (*simp add*: *cond-hiding-entails-def*)

lemma *cond-hiding-empty-entails*[*simp*]: *t hiding Map.empty under C entails R*
by (*simp add*: *cond-hiding-entails-def*)

lemma *hidings-entailsD*: ⟦*t hidings s entails R*; $x \in t\ k$; $y \in s\ k$⟧ $\Longrightarrow$ *R x y*
by (*simp add*: *hidings-entails-def*)

lemma *hidings-empty-entails* [*intro*!]: *t hidings* ($\lambda k.\ \{\}$) *entails R*
apply (*unfold hidings-entails-def*)
apply (*simp* (*no-asm*))
done

lemma *empty-hidings-entails* [*intro*!]:
($\lambda k.\ \{\}$) *hidings s entails R***apply** (*unfold hidings-entails-def*)
by (*simp* (*no-asm*))

primrec *atleast-free* :: ('*a* ⇀ '*b*) => *nat* => *bool*

where
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*
apply (*induct-tac n*)
apply (*simp* (*no-asm*))
apply *clarify*
apply (*simp* (*no-asm*))
apply (*drule atleast-free-Suc* [*THEN iffD1*])
apply *fast*
done

lemma *atleast-free-SucI*:
[| *h a = None*; ∀ *obj. atleast-free (h(a|−>obj)) n* |] ==> *atleast-free h (Suc n)*
by *force*

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*)
apply (*induct-tac n*)
apply *auto*
apply (*rule-tac x = a* **in** *exI*)
apply (*rule conjI*)
apply (*force simp add*: *fun-upd-apply*)
apply (*erule-tac V = m a = None* **in** *thin-rl*)
apply *clarify*
apply (*subst fun-upd-twist*)
apply (*erule not-sym*)
apply (*rename-tac ba*)
apply (*drule-tac x = ba* **in** *spec*)
apply *clarify*
apply (*tactic smp-tac* ***context*** *2 1*)
apply (*erule* (*1*) *notE impE*)
apply (*case-tac aa = b*)
apply *fast+*
done
declare *fun-upd-apply* [*simp*]

lemma *atleast-free-SucD*: *atleast-free h (Suc n)* ==> *atleast-free (h(a|−>b)) n*
apply *auto*
apply (*case-tac aa = a*)
apply *auto*
apply (*erule atleast-free-SucD-lemma*)
apply *auto*
done

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

end

Chapter 4

Name

1 Java names

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

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

datatype *ename* — expression name
  = *VNam vname*
  | *Res* — special name to model the return value of methods

datatype *lname* — names for local variables and the This pointer
  = *EName ename*
  | *This*
abbreviation *VName* :: *vname* ⇒ *lname*
  where *VName n* == *EName* (*VNam n*)

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

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

lemma *xn-cases*:
 $xn = Throwable \lor xn = NullPointer \lor$
 $xn = OutOfMemory \lor xn = ClassCast \lor$
 $xn = NegArrSize \lor xn = IndOutBound \lor xn = ArrStore$
apply (*induct xn*)
apply *auto*
done

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

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

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

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

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

instance ..

end

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

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

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

instance ..

end

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

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

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

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

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

lemma *Object-neq-SXcpt* [*simp*]: *Object* ≠ *SXcpt xn*
by (*simp add*: *Object-def SXcpt-def*)

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

end

Chapter 5

Value

1 Java values

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

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

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

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

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

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

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

primrec *typeof* :: *dyn-ty* ⇒ *val* ⇒ *ty option*
where
  typeof dt Unit = *Some* (*PrimT Void*)
| *typeof dt* (*Bool b*) = *Some* (*PrimT Boolean*)
| *typeof dt* (*Intg i*) = *Some* (*PrimT Integer*)
| *typeof dt Null* = *Some NT*
| *typeof dt* (*Addr a*) = *dt a*

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

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

end

Chapter 6

Type

1 Java types

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

simplifications:

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

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

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

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

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

definition
the-Class :: *ty* ⇒ *qtname*
where *the-Class T* = (*SOME C*. *T* = *Class C*)

lemma *the-Class-eq* [*simp*]: *the-Class* (*Class C*)= *C*
by (*auto simp add*: *the-Class-def*)

end

Chapter 7

Term

1 Java expressions and statements

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

design issues:

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

simplifications:

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

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

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

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

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

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

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

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

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

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

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

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

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

— function codes for binary operations

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

The boolean operators `&` and `|` strictly evaluate both of their arguments. The conditional operators `&&` and `||` only evaluate the second argument if the value of the whole expression isn't allready determined by the first argument. e.g.: `false && e` e is not evaluated; `true || e` e is not evaluated;

datatype *var*
= *LVar lname* — local variable (incl. parameters)
| *FVar qtname qtname bool expr vname* ({-,-,-}-..-[*10*,*10*,*10*,*85*,*99*]*90*)
— class field
— {*accC*,*statDeclC*,*stat*}*e*..*fn*
— *accC*: accessing class (static class were
— the code is declared. Annotation only needed for
— evaluation to check accessibility)
— *statDeclC*: static declaration class of field
— *stat*: static or instance field?
— *e*: reference to object
— *fn*: field name
| *AVar expr expr* (-.[-][*90*,*10*]*90*)
— array component
— *e1*.[*e2*]: e1 array reference; e2 index
| *InsInitV stmt var*
— insertion of initialization before evaluation
— of var (technical term for smallstep semantics.)

and *expr*
= *NewC qtname* — class instance creation
| *NewA ty expr* (*New* -[-][*99*,*10*]*85*)
— array creation
| *Cast ty expr* — type cast
| *Inst expr ref-ty* (- *InstOf* -[*85*,*99*] *85*)
— instanceof
| *Lit val* — literal value, references not allowed
| *UnOp unop expr* — unary operation
| *BinOp binop expr expr* — binary operation

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

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

and *stmt*
= *Skip* — empty statement
| *Expr expr* — expression statement
| *Lab jump stmt* (-• - [*99*,*66*]*66*)
— labeled statement; handles break
| *Comp stmt stmt* (-;; - [*66*,*65*]*65*)
| *If′ expr stmt stmt* (*If′*(-′) - *Else* - [*80*,*79*,*79*]*70*)
| *Loop label expr stmt* (-• *While′*(-′) - [*99*,*80*,*79*]*70*)
| *Jmp jump* — break, continue, return
| *Throw expr*
| *TryC stmt qtname vname stmt* (*Try* - *Catch′*(- -′) - [*79*,*99*,*80*,*79*]*70*)
— *Try c1 Catch*(*C vn*) *c2*
— *c1*: block were exception may be thrown
— *C*: execption class to catch
— *vn*: local name for exception used in *c2*
— *c2*: block to execute when exception is 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

datatype-compat *var expr stmt*

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

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

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

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

abbreviation *LAcc* :: *vname* ⇒ *expr* (!!)
where !!*v* == *Acc* (*LVar* (*EName* (*VNam v*)))

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

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

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

definition
is-stmt :: *term* ⇒ *bool*
where *is-stmt t* = (∃ *c. t=In1r c*)

ML ‹*ML-Thms.bind-thms* (*is-stmt-rews*, *sum3-instantiate* ***context*** @{*thm is-stmt-def*})›

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

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

abbreviation (*input*)
expr-inj-term :: *expr* ⇒ *term* ($\langle$-$\rangle_e$ *1000*)
where $\langle e\rangle_e$ == *In1l e*

abbreviation (*input*)
stmt-inj-term :: *stmt* ⇒ *term* ($\langle$-$\rangle_s$ *1000*)
where $\langle c\rangle_s$ == *In1r c*

abbreviation (*input*)
var-inj-term :: *var* ⇒ *term* ($\langle$-$\rangle_v$ *1000*)
where $\langle v\rangle_v$ == *In2 v*

abbreviation (*input*)
lst-inj-term :: *expr list* ⇒ *term* ($\langle$-$\rangle_l$ *1000*)
where $\langle es\rangle_l$ == *In3 es*

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

class *inj-term* =
fixes *inj-term*:: *'a* ⇒ *term* (⟨-⟩ *1000*)

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

instantiation *stmt* :: *inj-term*

begin

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

instance ..

end

lemma *stmt-inj-term-simp*: ⟨*c*::*stmt*⟩ = *In1r c*
by (*simp add*: *stmt-inj-term-def*)

lemma *stmt-inj-term* [*iff*]: ⟨*x*::*stmt*⟩ = ⟨*y*⟩ ≡ *x* = *y*
by (*simp add*: *stmt-inj-term-simp*)

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

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

instance ..

end

lemma *expr-inj-term-simp*: ⟨*e*::*expr*⟩ = *In1l e*
by (*simp add*: *expr-inj-term-def*)

lemma *expr-inj-term* [*iff*]: ⟨*x*::*expr*⟩ = ⟨*y*⟩ ≡ *x* = *y*
by (*simp add*: *expr-inj-term-simp*)

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

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

instance ..

end

lemma *var-inj-term-simp*: ⟨*v*::*var*⟩ = *In2 v*
by (*simp add*: *var-inj-term-def*)

lemma *var-inj-term* [*iff*]: ⟨*x*::*var*⟩ = ⟨*y*⟩ ≡ *x* = *y*
by (*simp add*: *var-inj-term-simp*)

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

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

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

instance ..

end

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

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

instance ..

end

lemma *expr-list-inj-term-def*:
⟨*es*::*expr list*⟩ ≡ *In3 es*
by (*simp add*: *inj-term-list-def expr-of-expr-def*)

lemma *expr-list-inj-term-simp*: ⟨*es*::*expr list*⟩ = *In3 es*
by (*simp add*: *expr-list-inj-term-def*)

lemma *expr-list-inj-term* [*iff*]: ⟨*x*::*expr list*⟩ = ⟨*y*⟩ ≡ *x* = *y*
by (*simp add*: *expr-list-inj-term-simp*)

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*⟩
by (*simp add*: *inj-term-simps*)

lemma *expr-stmt-inj-term* [*iff*]: ⟨*t*::*expr*⟩ ≠ ⟨*w*::*stmt*⟩
by (*simp add*: *inj-term-simps*)

lemma *stmt-var-inj-term* [*iff*]: ⟨*t*::*stmt*⟩ ≠ ⟨*w*::*var*⟩
by (*simp add*: *inj-term-simps*)

lemma *var-stmt-inj-term* [*iff*]: ⟨*t*::*var*⟩ ≠ ⟨*w*::*stmt*⟩
by (*simp add*: *inj-term-simps*)

lemma *stmt-elist-inj-term* [*iff*]: ⟨*t*::*stmt*⟩ ≠ ⟨*w*::*expr list*⟩
by (*simp add*: *inj-term-simps*)

lemma *elist-stmt-inj-term* [*iff*]: ⟨*t*::*expr list*⟩ ≠ ⟨*w*::*stmt*⟩
by (*simp add*: *inj-term-simps*)

lemma *expr-var-inj-term* [*iff*]: ⟨*t*::*expr*⟩ ≠ ⟨*w*::*var*⟩
by (*simp add*: *inj-term-simps*)

lemma *var-expr-inj-term* [*iff*]: $\langle t{::}var\rangle \neq \langle w{::}expr\rangle$
by (*simp add*: *inj-term-simps*)

lemma *expr-elist-inj-term* [*iff*]: $\langle t{::}expr\rangle \neq \langle w{::}expr\ list\rangle$
by (*simp add*: *inj-term-simps*)

lemma *elist-expr-inj-term* [*iff*]: $\langle t{::}expr\ list\rangle \neq \langle w{::}expr\rangle$
by (*simp add*: *inj-term-simps*)

lemma *var-elist-inj-term* [*iff*]: $\langle t{::}var\rangle \neq \langle w{::}expr\ list\rangle$
by (*simp add*: *inj-term-simps*)

lemma *elist-var-inj-term* [*iff*]: $\langle t{::}expr\ list\rangle \neq \langle w{::}var\rangle$
by (*simp add*: *inj-term-simps*)

lemma *term-cases*:
$[\![\bigwedge v.\ P\ \langle v\rangle_v;\ \bigwedge e.\ P\ \langle e\rangle_e;\bigwedge c.\ P\ \langle c\rangle_s;\bigwedge l.\ P\ \langle l\rangle_l]\!]$
$\Longrightarrow P\ t$
apply (*cases t*)
apply (*rename-tac a*, *case-tac a*)
apply *auto*
done

Evaluation of unary operations

primrec *eval-unop* :: *unop* ⇒ *val* ⇒ *val*
where
eval-unop UPlus v = *Intg* (*the-Intg v*)
| *eval-unop UMinus v* = *Intg* (− (*the-Intg v*))
| *eval-unop UBitNot v* = *Intg 42* — FIXME: Not yet implemented
| *eval-unop UNot v* = *Bool* (¬ *the-Bool v*)

Evaluation of binary operations

primrec *eval-binop* :: *binop* ⇒ *val* ⇒ *val* ⇒ *val*
where
eval-binop Mul v1 v2 = *Intg* ((*the-Intg v1*) ∗ (*the-Intg v2*))
| *eval-binop Div v1 v2* = *Intg* ((*the-Intg v1*) *div* (*the-Intg v2*))
| *eval-binop Mod v1 v2* = *Intg* ((*the-Intg v1*) *mod* (*the-Intg v2*))
| *eval-binop Plus v1 v2* = *Intg* ((*the-Intg v1*) + (*the-Intg v2*))
| *eval-binop Minus v1 v2* = *Intg* ((*the-Intg v1*) − (*the-Intg v2*))

— Be aware of the explicit coercion of the shift distance to nat
| *eval-binop LShift v1 v2* = *Intg* ((*the-Intg v1*) ∗ (*2*^(*nat* (*the-Intg v2*))))
| *eval-binop RShift v1 v2* = *Intg* ((*the-Intg v1*) *div* (*2*^(*nat* (*the-Intg v2*))))
| *eval-binop RShiftU v1 v2* = *Intg 42* — FIXME: Not yet implemented

| *eval-binop Less v1 v2* = *Bool* ((*the-Intg v1*) < (*the-Intg v2*))
| *eval-binop Le v1 v2* = *Bool* ((*the-Intg v1*) ≤ (*the-Intg v2*))
| *eval-binop Greater v1 v2* = *Bool* ((*the-Intg v2*) < (*the-Intg v1*))
| *eval-binop Ge v1 v2* = *Bool* ((*the-Intg v2*) ≤ (*the-Intg v1*))

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

| *eval-binop BitOr v1 v2 = Intg 42* — FIXME: Not yet implemented
| *eval-binop Or v1 v2 = Bool ((the-Bool v1) ∨ (the-Bool v2))*
| *eval-binop CondAnd v1 v2 = Bool ((the-Bool v1) ∧ (the-Bool v2))*
| *eval-binop CondOr v1 v2 = Bool ((the-Bool v1) ∨ (the-Bool v2))*

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

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

lemma *need-second-arg-CondAnd* [*simp*]: *need-second-arg CondAnd* (*Bool b*) = *b*
by (*simp add*: *need-second-arg-def*)

lemma *need-second-arg-CondOr* [*simp*]: *need-second-arg CondOr* (*Bool b*) = (¬ *b*)
by (*simp add*: *need-second-arg-def*)

lemma *need-second-arg-strict*[*simp*]:
⟦*binop*≠*CondAnd*; *binop*≠*CondOr*⟧ ⟹ *need-second-arg binop b*
by (*cases binop*)
(*simp-all add*: *need-second-arg-def*)
end

Chapter 8

Decl

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

theory *Decl*
imports *Term Table*

begin

improvements:

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

simplifications:

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

2 Modifier

Access modifier

datatype *acc-modi*
  = *Private* | *Package* | *Protected* | *Public*

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

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

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

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

instance
proof
fix *x y z*::*acc-modi*
show $(x < y) = (x \leq y \wedge \neg\, y \leq x)$
by (*auto simp add*: *le-acc-def less-acc-def split*: *acc-modi.split*)
show $x \leq x$ — reflexivity
by (*auto simp add*: *le-acc-def*)
{
assume $x \leq y$ $y \leq z$ — transitivity
then show $x \leq z$
by (*auto simp add*: *le-acc-def less-acc-def split*: *acc-modi.split*)
next
assume $x \leq y$ $y \leq x$ — antisymmetry
moreover have $\forall$ *x y*. *x* < (*y*::*acc-modi*) $\wedge$ $y < x \longrightarrow$ *False*
by (*auto simp add*: *less-acc-def split*: *acc-modi.split*)
ultimately show $x = y$ **by** (*unfold le-acc-def*) *iprover*
next
fix *x y*:: *acc-modi*
show $x \leq y \vee y \leq x$
by (*auto simp add*: *less-acc-def le-acc-def split*: *acc-modi.split*)
}
qed

end

lemma *acc-modi-top* [*simp*]: *Public* $\leq$ *a* $\Longrightarrow$ *a* = *Public*
by (*auto simp add*: *le-acc-def less-acc-def split*: *acc-modi.splits*)

lemma *acc-modi-top1* [*simp*, *intro*!]: *a* $\leq$ *Public*
by (*auto simp add*: *le-acc-def less-acc-def split*: *acc-modi.splits*)

lemma *acc-modi-le-Public*:
a $\leq$ *Public* $\Longrightarrow$ *a*=*Private* $\vee$ *a* = *Package* $\vee$ *a*=*Protected* $\vee$ *a*=*Public*
by (*auto simp add*: *le-acc-def less-acc-def split*: *acc-modi.splits*)

lemma *acc-modi-bottom*: *a* $\leq$ *Private* $\Longrightarrow$ *a* = *Private*
by (*auto simp add*: *le-acc-def less-acc-def split*: *acc-modi.splits*)

lemma *acc-modi-Private-le*:
Private $\leq$ *a* $\Longrightarrow$ *a*=*Private* $\vee$ *a* = *Package* $\vee$ *a*=*Protected* $\vee$ *a*=*Public*
by (*auto simp add*: *le-acc-def less-acc-def split*: *acc-modi.splits*)

lemma *acc-modi-Package-le*:
  Package ≤ *a* ⟹ *a* = *Package* ∨ *a*=*Protected* ∨ *a*=*Public*
by (*auto simp add*: *le-acc-def less-acc-def split*: *acc-modi.split*)

lemma *acc-modi-le-Package*:
  a ≤ *Package* ⟹ *a*=*Private* ∨ *a* = *Package*
by (*auto simp add*: *le-acc-def less-acc-def split*: *acc-modi.splits*)

lemma *acc-modi-Protected-le*:
  Protected ≤ *a* ⟹ *a*=*Protected* ∨ *a*=*Public*
by (*auto simp add*: *le-acc-def less-acc-def split*: *acc-modi.splits*)

lemma *acc-modi-le-Protected*:
  a ≤ *Protected* ⟹ *a*=*Private* ∨ *a* = *Package* ∨ *a* = *Protected*
by (*auto simp add*: *le-acc-def less-acc-def split*: *acc-modi.splits*)

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

lemma *acc-modi-Package-le-cases*:
  assumes *Package* ≤ *m*
  obtains (*Package*) *m* = *Package*
    | (*Protected*) *m* = *Protected*
    | (*Public*) *m* = *Public*
using *assms* **by** (*auto dest*: *acc-modi-Package-le*)

Static Modifier

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

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

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

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

4 Member (field or method)

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

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

5 Field

record *field* = *member* +

type :: *ty*

translations

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

type-synonym *fdecl*
= *vname* × *field*

translations

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

6 Method

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

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

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

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

translations

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

definition

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

lemma *access-mhead* [*simp*]:*access* (*mhead m*) = *access m*
by (*simp add*: *mhead-def*)

lemma *static-mhead* [*simp*]:*static* (*mhead m*) = *static m*
by (*simp add*: *mhead-def*)

lemma *pars-mhead* [*simp*]:*pars* (*mhead m*) = *pars m*
by (*simp add*: *mhead-def*)

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

To be able to talk uniformaly about field and method declarations we introduce the notion of a member declaration (e.g. useful to define accessiblity)

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

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

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

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

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

instance ..

end

lemma *memberid-fdecl-simp*[*simp*]: *memberid* (*fdecl* (*vn*,*f*)) = *fid vn*
by (*simp add*: *memberdecl-memberid-def*)

lemma *memberid-fdecl-simp1*: *memberid* (*fdecl f*) = *fid* (*fst f*)
by (*cases f*) (*simp add*: *memberdecl-memberid-def*)

lemma *memberid-mdecl-simp*[*simp*]: *memberid* (*mdecl* (*sig*,*m*)) = *mid sig*
by (*simp add*: *memberdecl-memberid-def*)

lemma *memberid-mdecl-simp1*: *memberid* (*mdecl m*) = *mid* (*fst m*)
by (*cases m*) (*simp add*: *memberdecl-memberid-def*)

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

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

instance ..

end

lemma *memberid-pair-simp*[*simp*]: *memberid* (*c*,*m*) = *memberid m*
by (*simp add*: *pair-memberid-def*)

lemma *memberid-pair-simp1*: *memberid p* = *memberid* (*snd p*)
by (*simp add*: *pair-memberid-def*)

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

lemma *is-fieldD*: *is-field m* ⟹ ∃ *declC f*. *m*=(*declC*,*fdecl f*)
by (*simp add*: *is-field-def*)

lemma *is-fieldI*: *is-field* (*C*,*fdecl f*)
by (*simp add*: *is-field-def*)

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

lemma *is-methodD*: *is-method membr* ⟹ ∃ *declC m*. *membr*=(*declC*,*mdecl m*)
by (*simp add*: *is-method-def*)

lemma *is-methodI*: *is-method* (*C*,*mdecl m*)
by (*simp add*: *is-method-def*)

7 Interface

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

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

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

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

lemma *access-ibody* [*simp*]: (*access* (*ibody i*)) = *access i*
by (*simp add*: *ibody-def*)

lemma *imethods-ibody* [*simp*]: (*imethods* (*ibody i*)) = *imethods i*
by (*simp add*: *ibody-def*)

8 Class

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

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

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

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

lemma *access-cbody* [*simp*]:*access* (*cbody c*) = *access c*
by (*simp add*: *cbody-def*)

lemma *cfields-cbody* [*simp*]:*cfields* (*cbody c*) = *cfields c*
by (*simp add*: *cbody-def*)

lemma *methods-cbody* [*simp*]:*methods* (*cbody c*) = *methods c*
by (*simp add*: *cbody-def*)

lemma *init-cbody* [*simp*]:*init* (*cbody c*) = *init c*
by (*simp add*: *cbody-def*)

standard classes

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

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

definition

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

lemma *ObjectC-neq-SXcptC* [*simp*]: *ObjectC* ≠ *SXcptC xn*
by (*simp add*: *ObjectC-def SXcptC-def Object-def SXcpt-def*)

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

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

programs

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

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

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

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

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

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

is type

primrec *is-type* :: *prog* ⇒ *ty* ⇒ *bool*
and *isrtype* :: *prog* ⇒ *ref-ty* ⇒ *bool*
where
is-type G (*PrimT pt*) = *True*
| *is-type G* (*RefT rt*) = *isrtype G rt*
| *isrtype G* (*NullT*) = *True*
| *isrtype G* (*IfaceT tn*) = *is-iface G tn*
| *isrtype G* (*ClassT tn*) = *is-class G tn*
| *isrtype G* (*ArrayT T*) = *is-type G T*

lemma *type-is-iface*: *is-type G* (*Iface I*) $\Longrightarrow$ *is-iface G I*
by *auto*

lemma *type-is-class*: *is-type G* (*Class C*) $\Longrightarrow$ *is-class G C*
by *auto*

subinterface and subclass relation, in anticipation of TypeRel.thy

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

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

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

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

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

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

lemma *subint1I*: ⟦*iface G I* = *Some i*; *J* ∈ *set* (*isuperIfs i*)⟧
$\Longrightarrow$ (*I*,*J*) ∈ *subint1 G*
apply (*simp add*: *subint1-def*)
done

lemma *subcls1I*:⟦*class G C* = *Some c*; *C* ≠ *Object*⟧ $\Longrightarrow$ (*C*,(*super c*)) ∈ *subcls1 G*
apply (*simp add*: *subcls1-def*)
done

lemma *subint1D*: (*I*,*J*)∈*subint1 G*$\Longrightarrow$ ∃ *i*∈*iface G I*: *J*∈*set* (*isuperIfs i*)
by (*simp add*: *subint1-def*)

lemma *subcls1D*:
(*C*,*D*)∈*subcls1 G* $\Longrightarrow$ *C*≠*Object* ∧ (∃ *c*. *class G C* = *Some c* ∧ (*super c* = *D*))
apply (*simp add*: *subcls1-def*)
apply *auto*
done

lemma *subint1-def2*:
subint1 G = (*SIGMA* I: {I. *is-iface* G I}. *set* (*isuperIfs* (*the* (*iface* G I))))
apply (*unfold subint1-def*)
apply *auto*
done

lemma *subcls1-def2*:
subcls1 G =
(*SIGMA* C: {C. *is-class* G C}. {D. $C \neq$*Object* $\wedge$ *super* (*the*(*class* G C))=D})
apply (*unfold subcls1-def*)
apply *auto*
done

lemma *subcls-is-class*:
$[\![G \vdash C \prec_C D]\!] \Longrightarrow \exists\ c.\ class\ G\ C = Some\ c$
by (*auto simp add*: *subcls1-def dest*: *tranclD*)

lemma *no-subcls1-Object*:$G \vdash Object \prec_C 1\ D \Longrightarrow P$
by (*auto simp add*: *subcls1-def*)

lemma *no-subcls-Object*: $G \vdash Object \prec_C D \Longrightarrow P$
apply (*erule trancl-induct*)
apply (*auto intro*: *no-subcls1-Object*)
done

well-structured programs

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

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

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

lemma *ws-progI*:
$[\![\forall (I,i) \in set\ (ifaces\ G).\ \forall J \in set\ (isuperIfs\ i).\ is\text{-}iface\ G\ J\ \wedge$
$(J,I) \notin (subint1\ G)^+;$
$\forall (C,c) \in set\ (classes\ G).\ C \neq Object \longrightarrow is\text{-}class\ G\ (super\ c)\ \wedge$
$((super\ c),C) \notin (subcls1\ G)^+$
$]\!] \Longrightarrow ws\text{-}prog\ G$
apply (*unfold ws-prog-def ws-idecl-def ws-cdecl-def*)
apply (*erule-tac conjI*)
apply *blast*
done

lemma *ws-prog-ideclD*:

$[\![$*iface G I* = *Some i*; *J*∈*set* (*isuperIfs i*); *ws-prog G*$]\!] \Longrightarrow$
is-iface G J ∧ (*J*,*I*)∉(*subint1 G*)$^+$
apply (*unfold ws-prog-def ws-idecl-def*)
apply *clarify*
apply (*drule-tac map-of-SomeD*)
apply *auto*
done

lemma *ws-prog-cdeclD*:
$[\![$*class G C* = *Some c*; *C*≠*Object*; *ws-prog G*$]\!] \Longrightarrow$
is-class G (*super c*) ∧ (*super c*,*C*)∉(*subcls1 G*)$^+$
apply (*unfold ws-prog-def ws-cdecl-def*)
apply *clarify*
apply (*drule-tac map-of-SomeD*)
apply *auto*
done

well-foundedness

lemma *finite-is-iface*: *finite* {*I*. *is-iface G I*}
apply (*fold dom-def*)
apply (*rule-tac finite-dom-map-of*)
done

lemma *finite-is-class*: *finite* {*C*. *is-class G C*}
apply (*fold dom-def*)
apply (*rule-tac finite-dom-map-of*)
done

lemma *finite-subint1*: *finite* (*subint1 G*)
apply (*subst subint1-def2*)
apply (*rule finite-SigmaI*)
apply (*rule finite-is-iface*)
apply (*simp* (*no-asm*))
done

lemma *finite-subcls1*: *finite* (*subcls1 G*)
apply (*subst subcls1-def2*)
apply (*rule finite-SigmaI*)
apply (*rule finite-is-class*)
apply (*rule-tac B* = {*super* (*the* (*class G C*))} **in** *finite-subset*)
apply *auto*
done

lemma *subint1-irrefl-lemma1*:
ws-prog G ⟹ (*subint1 G*)$^{-1}$ ∩ (*subint1 G*)$^+$ = {}
apply (*force dest*: *subint1D ws-prog-ideclD conjunct2*)
done

lemma *subcls1-irrefl-lemma1*:
ws-prog G ⟹ (*subcls1 G*)$^{-1}$ ∩ (*subcls1 G*)$^+$ = {}
apply (*force dest*: *subcls1D ws-prog-cdeclD conjunct2*)
done

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*) ∈ *subint1 G*; *ws-prog G*⟧ ⟹ *x* ≠ *y*
apply (*rule irrefl-trancl-rD*)
apply (*rule subint1-irrefl-lemma2*)
apply *auto*
done

lemma *subcls1-irrefl*: ⟦(*x*, *y*) ∈ *subcls1 G*; *ws-prog G*⟧ ⟹ *x* ≠ *y*
apply (*rule irrefl-trancl-rD*)
apply (*rule subcls1-irrefl-lemma2*)
apply *auto*
done

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

lemma *wf-subint1*: *ws-prog G* ⟹ *wf* ((*subint1 G*)$^{-1}$)
by (*auto intro*: *finite-acyclic-wf-converse finite-subint1 subint1-acyclic*)

lemma *wf-subcls1*: *ws-prog G* ⟹ *wf* ((*subcls1 G*)$^{-1}$)
by (*auto intro*: *finite-acyclic-wf-converse finite-subcls1 subcls1-acyclic*)

lemma *subint1-induct*:
⟦*ws-prog G*; ⋀*x*. ∀ *y*. (*x*, *y*) ∈ *subint1 G* ⟶ *P y* ⟹ *P x*⟧ ⟹ *P a*
apply (*frule wf-subint1*)
apply (*erule wf-induct*)
apply (*simp* (*no-asm-use*) *only*: *converse-iff*)
apply *blast*
done

lemma *subcls1-induct* [*consumes 1*]:
⟦*ws-prog G*; ⋀*x*. ∀ *y*. (*x*, *y*) ∈ *subcls1 G* ⟶ *P y* ⟹ *P x*⟧ ⟹ *P a*
apply (*frule wf-subcls1*)
apply (*erule wf-induct*)
apply (*simp* (*no-asm-use*) *only*: *converse-iff*)
apply *blast*
done

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*
apply (*erule rev-mp*)
apply (*rule subint1-induct*)
apply *assumption*
apply (*simp* (*no-asm*))
apply *safe*

```
apply (blast dest: subint1I ws-prog-ideclD)
done


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
apply (erule rev-mp)
apply (rule subcls1-induct)
apply  assumption
apply (simp (no-asm))
apply safe
apply (fast dest: subcls1I ws-prog-cdeclD)
done


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 −
  assume clsC: class G C = Some c
  and    init: ⋀ co. class G Object = Some co ⟹ P Object
  and    step: ⋀   C c. ⟦class G C = Some c; C ≠ Object; P (super c)⟧ ⟹ P C
  assume ws: ws-prog G
  then have is-class G C ⟹ P C
  proof (induct rule: subcls1-induct)
    fix C
    assume  hyp:∀ S. G⊢C ≺C1 S ⟶ is-class G S ⟶ P S
      and iscls:is-class G C
    show P C
    proof (cases C=Object)
      case True with iscls init show P C by auto
    next
      case False with ws step hyp iscls
      show P C by (auto dest: subcls1I ws-prog-cdeclD)
    qed
  qed
  with clsC show ?thesis by simp
qed


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
by (auto intro: ws-class-induct)


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 −
  assume clsC: class G C = Some c
  and    init: ⋀ co. class G Object = Some co ⟹ P Object co
  and    step: ⋀ C c sc . ⟦class G C = Some c; class G (super c) = Some sc;
                          C ≠ Object; P (super c) sc⟧ ⟹ P C c
  assume ws: ws-prog G
  then have ⋀ c. class G C = Some c⟹ P C c
  proof (induct rule: subcls1-induct)
    fix C c
    assume  hyp:∀ S. G⊢C ≺C1 S ⟶ (∀ s. class G S = Some s ⟶ P S s)
       and iscls:class G C = Some c
    show P C c
    proof (cases C=Object)
      case True with iscls init show P C c by auto
    next
      case False
      with ws iscls obtain sc where
        sc: class G (super c) = Some sc
        by (auto dest: ws-prog-cdeclD)
      from iscls False have G⊢C ≺C1 (super c) by (rule subcls1I)
      with False ws step hyp iscls sc
      show P C c
        by (auto)
    qed
  qed
  with clsC show P C c by auto
qed


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 −
  from is-if-I ws
  show P I
  proof (rule ws-subint1-induct)
    fix I i
    assume hyp: iface G I = Some i ∧
              (∀ J∈set (isuperIfs i). (I,J) ∈subint1 G ∧ P J ∧ is-iface G J)
    then have if-I: iface G I = Some i
      by blast
    show P I
    proof (cases isuperIfs i)
      case Nil
      with if-I hyp-sub
      show P I
        by auto
    next
      case (Cons hd tl)
      with hyp if-I hyp-sub
      show P I
        by auto
    qed
  qed
```

qed

general recursion operators for the interface and class hiearchies

function *iface-rec* :: *prog* ⇒ *qtname* ⇒ (*qtname* ⇒ *iface* ⇒ ′*a set* ⇒ ′*a*) ⇒ ′*a*
where
[*simp del*]: *iface-rec G I f* =
 (*case iface G I of*
  None ⇒ *undefined*
  | *Some i* ⇒ *if ws-prog G*
   then f I i
    ((λ*J*. *iface-rec G J f*)‘*set* (*isuperIfs i*))
   else undefined)
by *auto*
termination
by (*relation inv-image* (*same-fst ws-prog* (λ*G*. (*subint1 G*)$^{-1}$)) (%(*x*,*y*,*z*). (*x*,*y*)))
 (*auto simp*: *wf-subint1 subint1I wf-same-fst*)

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*))
apply (*subst iface-rec.simps*)
apply *simp*
done

function
 class-rec :: *prog* ⇒ *qtname* ⇒ ′*a* ⇒ (*qtname* ⇒ *class* ⇒ ′*a* ⇒ ′*a*) ⇒ ′*a*
where
[*simp del*]: *class-rec G C t f* =
 (*case class G C of*
  None ⇒ *undefined*
  | *Some c* ⇒ *if ws-prog G*
   then f C c
    (*if C* = *Object then t*
     else class-rec G (*super c*) *t f*)
   else undefined)

by *auto*
termination
by (*relation inv-image* (*same-fst ws-prog* (λ*G*. (*subcls1 G*)$^{-1}$)) (%(*x*,*y*,*z*,*w*). (*x*,*y*)))
 (*auto simp*: *wf-subcls1 subcls1I wf-same-fst*)

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*)
apply (*subst class-rec.simps*)
apply *simp*
done

definition
 imethds :: *prog* ⇒ *qtname* ⇒ (*sig*,*qtname* × *mhead*) *tables* **where**
 — methods of an interface, with overriding and inheritance, cf. 9.2
 imethds G I = *iface-rec G I*
  (λ*I i ts*. (*Un-tables ts*) ⊕⊕
   (*set-option* ∘ *table-of* (*map* (λ(*s*,*m*). (*s*,*I*,*m*)) (*imethods i*))))

end

Chapter 9

TypeRel

1 The relations between Java types

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

simplifications:

- subinterface, subclass and widening relation includes identity

improvements over Java Specification 1.0:

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

design issues:

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

definition
implmt1 :: *prog* ⇒ (*qtname* × *qtname*) *set* — direct implementation
— direct implementation, cf. 8.1.3
where *implmt1 G* = {(*C*,*I*). *C*≠*Object* ∧ (∃ *c*∈*class G C*: *I*∈*set* (*superIfs c*))}

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

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

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

notation (*ASCII*)
subint1-syntax (-|--<:I1- [71,71,71] 70) **and**
subint-syntax (-|--<=:I -[71,71,71] 70) **and**
implmt1-syntax (-|--~>1- [71,71,71] 70)

subclass and subinterface relations

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

lemma *subcls-direct1*:
$[\![$*class G C* = *Some c*; *C* ≠ *Object*;*D*=*super c*$]\!]$ ⟹ $G\vdash C\preceq_C D$
apply (*auto dest*: *subcls-direct*)
done

lemma *subcls1I1*:
$[\![$*class G C* = *Some c*; *C* ≠ *Object*;*D*=*super c*$]\!]$ ⟹ $G\vdash C\prec_C 1\ D$
apply (*auto dest*: *subcls1I*)
done

lemma *subcls-direct2*:
$[\![$*class G C* = *Some c*; *C* ≠ *Object*;*D*=*super c*$]\!]$ ⟹ $G\vdash C\prec_C D$
apply (*auto dest*: *subcls1I1*)
done

lemma *subclseq-trans*: $[\![G\vdash A \preceq_C B;\ G\vdash B \preceq_C C]\!] \Longrightarrow G\vdash A \preceq_C C$
by (*blast intro*: *rtrancl-trans*)

lemma *subcls-trans*: $[\![G\vdash A \prec_C B;\ G\vdash B \prec_C C]\!] \Longrightarrow G\vdash A \prec_C C$
by (*blast intro*: *trancl-trans*)

lemma *SXcpt-subcls-Throwable-lemma*:
$[\![$*class G* (*SXcpt xn*) = *Some xc*;
super xc = (*if xn* = *Throwable then Object else SXcpt Throwable*)$]\!]$
⟹ $G\vdash$*SXcpt xn*$\preceq_C$ *SXcpt Throwable*
apply (*case-tac xn* = *Throwable*)
apply *simp-all*
apply (*drule subcls-direct*)
apply (*auto dest*: *sym*)
done

lemma *subcls-ObjectI*: $[\![$*is-class G C*; *ws-prog G*$]\!]$ ⟹ $G\vdash C\preceq_C$ *Object*
apply (*erule ws-subcls1-induct*)
apply *clarsimp*
apply (*case-tac C* = *Object*)
apply (*fast intro*: *r-into-rtrancl* [*THEN rtrancl-trans*])+
done

lemma *subclseq-ObjectD* [*dest*!]: $G\vdash$*Object*$\preceq_C$ *C* ⟹ *C* = *Object*
apply (*erule rtrancl-induct*)
apply (*auto dest*: *subcls1D*)
done

lemma *subcls-ObjectD* [*dest*!]: $G\vdash$*Object*$\prec_C$ *C* ⟹ *False*
apply (*erule trancl-induct*)
apply (*auto dest*: *subcls1D*)

done

lemma *subcls-ObjectI1* [*intro*!]:
$[\![C \neq Object;is\text{-}class\ G\ C;ws\text{-}prog\ G]\!] \Longrightarrow G\vdash C \prec_C Object$
apply (*drule* (*1*) *subcls-ObjectI*)
apply (*auto intro*: *rtrancl-into-trancl3*)
done

lemma *subcls-is-class*: $(C,D) \in (subcls1\ G)^+ \Longrightarrow is\text{-}class\ G\ C$
apply (*erule trancl-trans-induct*)
apply (*auto dest*!: *subcls1D*)
done

lemma *subcls-is-class2* [*rule-format* (*no-asm*)]:
$G\vdash C\preceq_C D \Longrightarrow is\text{-}class\ G\ D \longrightarrow is\text{-}class\ G\ C$
apply (*erule rtrancl-induct*)
apply (*drule-tac* [*2*] *subcls1D*)
apply *auto*
done

lemma *single-inheritance*:
$[\![G\vdash A \prec_C 1\ B;\ G\vdash A \prec_C 1\ C]\!] \Longrightarrow B = C$
by (*auto simp add*: *subcls1-def*)

lemma *subcls-compareable*:
$[\![G\vdash A \preceq_C X;\ G\vdash A \preceq_C Y$
$]\!] \Longrightarrow G\vdash X \preceq_C Y \vee G\vdash Y \preceq_C X$
by (*rule triangle-lemma*) (*auto intro*: *single-inheritance*)

lemma *subcls1-irrefl*: $[\![G\vdash C \prec_C 1\ D;\ ws\text{-}prog\ G\ ]\!]$
$\Longrightarrow C \neq D$
proof
  assume *ws*: *ws-prog G* **and**
    subcls1: $G\vdash C \prec_C 1\ D$ **and**
     eq-C-D: $C=D$
  from *subcls1* **obtain** *c*
    where
      neq-C-Object: $C\neq Object$ **and**
              clsC: *class G C = Some c* **and**
          super-c: *super c = D*
    by (*auto simp add*: *subcls1-def*)
  with *super-c subcls1 eq-C-D*
  have *subcls-super-c-C*: $G\vdash super\ c \prec_C C$
    by *auto*
  from *ws clsC neq-C-Object*
  have $\neg\ G\vdash super\ c \prec_C C$
    by (*auto dest*: *ws-prog-cdeclD*)
  from *this subcls-super-c-C*
  show *False*
    by (*rule notE*)
qed

lemma *no-subcls-Object*: $G \vdash C \prec_C D \Longrightarrow C \neq Object$
by (*erule converse-trancl-induct*) (*auto dest*: *subcls1D*)

```
lemma subcls-acyclic: ⟦G⊢C ≺C D; ws-prog G⟧ ⟹ ¬ G⊢D ≺C C
proof –
  assume          ws: ws-prog G
  assume subcls-C-D: G⊢C ≺C D
  then show ?thesis
  proof (induct rule: converse-trancl-induct)
    fix C
    assume subcls1-C-D: G⊢C ≺C1 D
    then obtain c  where
        C≠Object and
        class G C = Some c and
        super c = D
      by (auto simp add: subcls1-def)
    with ws
    show ¬ G⊢D ≺C C
      by (auto dest: ws-prog-cdeclD)
  next
    fix C Z
    assume subcls1-C-Z: G⊢C ≺C1 Z and
             subcls-Z-D: G⊢Z ≺C D and
            nsubcls-D-Z: ¬ G⊢D ≺C Z
    show ¬ G⊢D ≺C C
    proof
      assume subcls-D-C: G⊢D ≺C C
      show False
      proof –
        from subcls-D-C subcls1-C-Z
        have G⊢D ≺C Z
          by (auto dest: r-into-trancl trancl-trans)
        with nsubcls-D-Z
        show ?thesis
          by (rule notE)
      qed
    qed
  qed
qed
```

```
lemma subclseq-cases:
  assumes G⊢C ≼C D
  obtains (Eq) C = D | (Subcls) G⊢C ≺C D
using assms by (blast intro: rtrancl-cases)
```

```
lemma subclseq-acyclic:
 ⟦G⊢C ≼C D; G⊢D ≼C C; ws-prog G⟧ ⟹ C=D
by (auto elim: subclseq-cases dest: subcls-acyclic)
```

```
lemma subcls-irrefl: ⟦G⊢C ≺C D; ws-prog G⟧
 ⟹ C ≠ D
proof –
  assume      ws: ws-prog G
  assume subcls: G⊢C ≺C D
  then show ?thesis
```

```
  proof (induct rule: converse-trancl-induct)
    fix C
    assume G⊢C ≺C1 D
    with ws
    show C≠D
      by (blast dest: subcls1-irrefl)
  next
    fix C Z
    assume subcls1-C-Z: G⊢C ≺C1 Z and
            subcls-Z-D: G⊢Z ≺C D and
               neq-Z-D: Z ≠ D
    show C≠D
    proof
      assume eq-C-D: C=D
      show False
      proof -
        from subcls1-C-Z eq-C-D
        have G⊢D ≺C Z
          by (auto)
        also
        from subcls-Z-D ws
        have ¬ G⊢D ≺C Z
          by (rule subcls-acyclic)
        ultimately
        show ?thesis
          by - (rule notE)
      qed
    qed
  qed
qed


lemma invert-subclseq:
⟦G⊢C ≼C D; ws-prog G⟧
 ⟹ ¬ G⊢D ≺C C
proof -
  assume       ws: ws-prog G and
     subclseq-C-D: G⊢C ≼C D
  show ?thesis
  proof (cases D=C)
    case True
    with ws
    show ?thesis
      by (auto dest: subcls-irrefl)
  next
    case False
    with subclseq-C-D
    have G⊢C ≺C D
      by (blast intro: rtrancl-into-trancl3)
    with ws
    show ?thesis
      by (blast dest: subcls-acyclic)
  qed
qed


lemma invert-subcls:
⟦G⊢C ≺C D; ws-prog G⟧
 ⟹ ¬ G⊢D ≼C C
```

proof −
  assume *ws*: *ws-prog G* **and**
    subcls-C-D: $G \vdash C \prec_C D$
  then
  have *nsubcls-D-C*: $\neg\ G \vdash D \prec_C C$
    by (*blast dest*: *subcls-acyclic*)
  show *?thesis*
  proof
    assume $G \vdash D \preceq_C C$
    then show *False*
    proof (*cases rule*: *subclseq-cases*)
      case *Eq*
      with *ws subcls-C-D*
      show *?thesis*
        by (*auto dest*: *subcls-irrefl*)
    next
      case *Subcls*
      with *nsubcls-D-C*
      show *?thesis*
        by *blast*
    qed
  qed
qed

lemma *subcls-superD*:
$[\![G \vdash C \prec_C D;\ class\ G\ C = Some\ c]\!] \Longrightarrow G \vdash (super\ c) \preceq_C D$
proof −
  assume *clsC*: *class G C = Some c*
  assume *subcls-C-C*: $G \vdash C \prec_C D$
  then obtain *S* **where**
    $G \vdash C \prec_C 1\ S$ **and**
    subclseq-S-D: $G \vdash S \preceq_C D$
    by (*blast dest*: *tranclD*)
  with *clsC*
  have *S=super c*
    by (*auto dest*: *subcls1D*)
  with *subclseq-S-D* **show** *?thesis* **by** *simp*
qed

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 −
  assume *neq-C-D*: $C \neq D$
  assume *clsC*: *class G C = Some c*
  assume *subclseq-C-D*: $G \vdash C \preceq_C D$
  then show *?thesis*
  proof (*cases rule*: *subclseq-cases*)
    case *Eq* **with** *neq-C-D* **show** *?thesis* **by** *contradiction*
  next
    case *Subcls*
    with *clsC* **show** *?thesis* **by** (*blast dest*: *subcls-superD*)
  qed
qed

implementation relation

lemma *implmt1D*: $G \vdash C \leadsto 1I \Longrightarrow C \neq Object \wedge (\exists c \in class\ G\ C\colon I \in set\ (superIfs\ c))$
apply (*unfold implmt1-def*)
apply *auto*
done

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

lemma *implmtD*: $G \vdash C \leadsto J \Longrightarrow (\exists I.\ G \vdash C \leadsto 1I \wedge G \vdash I \preceq I\ J) \vee (\exists D.\ G \vdash C \prec_C 1D \wedge G \vdash D \leadsto J)$
apply (*erule implmt.induct*)
apply *fast+*
done

lemma *implmt-ObjectE* [*elim*!]: $G \vdash Object \leadsto I \Longrightarrow R$
by (*auto dest*!: *implmtD implmt1D subcls1D*)

lemma *subcls-implmt* [*rule-format* (*no-asm*)]: $G \vdash A \preceq_C B \Longrightarrow G \vdash B \leadsto K \longrightarrow G \vdash A \leadsto K$
apply (*erule rtrancl-induct*)
apply (*auto intro*: *implmt.subcls1*)
done

lemma *implmt-subint2*: $[\![G \vdash A \leadsto J;\ G \vdash J \preceq I\ K]\!] \Longrightarrow G \vdash A \leadsto K$
apply (*erule rev-mp*, *erule implmt.induct*)
apply (*auto dest*: *implmt.subint rtrancl-trans implmt.subcls1*)
done

lemma *implmt-is-class*: $G \vdash C \leadsto I \Longrightarrow is\text{-}class\ G\ C$
apply (*erule implmt.induct*)
apply (*auto dest*: *implmt1D subcls1D*)
done

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*: $G \vdash C \leadsto I \Longrightarrow G \vdash Class\ C \preceq Iface\ I$
| *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*: $G \vdash PrimT\ x \preceq T \Longrightarrow (\exists\, y.\ T = PrimT\ y)$
apply (*ind-cases* $G \vdash PrimT\ x \preceq T$)
by *auto*

lemma *widen-PrimT2*: $G \vdash S \preceq PrimT\ x \Longrightarrow \exists\, y.\ S = PrimT\ y$
apply (*ind-cases* $G \vdash S \preceq PrimT\ x$)
by *auto*

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*: $G \vdash PrimT\ x \preceq T \Longrightarrow T = PrimT\ x$
by (*ind-cases* $G \vdash PrimT\ x \preceq T$) *simp-all*

lemma *widen-PrimT2-strong*: $G \vdash S \preceq PrimT\ x \Longrightarrow S = PrimT\ x$
by (*ind-cases* $G \vdash S \preceq PrimT\ x$) *simp-all*

Specialized versions for booleans also would work for real Java

lemma *widen-Boolean*: $G \vdash PrimT\ Boolean \preceq T \Longrightarrow T = PrimT\ Boolean$
by (*ind-cases* $G \vdash PrimT\ Boolean \preceq T$) *simp-all*

lemma *widen-Boolean2*: $G \vdash S \preceq PrimT\ Boolean \Longrightarrow S = PrimT\ Boolean$
by (*ind-cases* $G \vdash S \preceq PrimT\ Boolean$) *simp-all*

lemma *widen-RefT*: $G \vdash RefT\ R \preceq T \Longrightarrow \exists\, t.\ T{=}RefT\ t$
apply (*ind-cases* $G \vdash RefT\ R \preceq T$)
by *auto*

lemma *widen-RefT2*: $G \vdash S \preceq RefT\ R \Longrightarrow \exists\, t.\ S{=}RefT\ t$
apply (*ind-cases* $G \vdash S \preceq RefT\ R$)
by *auto*

lemma *widen-Iface*: $G \vdash Iface\ I \preceq T \Longrightarrow T{=}Class\ Object \lor (\exists\, J.\ T{=}Iface\ J)$
apply (*ind-cases* $G \vdash Iface\ I \preceq T$)
by *auto*

lemma *widen-Iface2*: $G \vdash S \preceq\ Iface\ J \Longrightarrow S = NT \lor (\exists\, I.\ S = Iface\ I) \lor (\exists\, D.\ S = Class\ D)$
apply (*ind-cases* $G \vdash S \preceq\ Iface\ J$)
by *auto*

lemma *widen-Iface-Iface*: $G \vdash Iface\ I \preceq\ Iface\ J \Longrightarrow G \vdash I \preceq I\ J$
apply (*ind-cases* $G \vdash Iface\ I \preceq\ Iface\ J$)
by *auto*

lemma *widen-Iface-Iface-eq* [*simp*]: $G \vdash Iface\ I \preceq\ Iface\ J\ =\ G \vdash I \preceq I\ J$
apply (*rule iffI*)
apply (*erule widen-Iface-Iface*)
apply (*erule widen.subint*)
done

lemma *widen-Class*: $G \vdash Class\ C \preceq T \implies (\exists D.\ T = Class\ D) \vee (\exists I.\ T = Iface\ I)$
apply (*ind-cases* $G \vdash Class\ C \preceq T$)
by *auto*

lemma *widen-Class2*: $G \vdash S \preceq\ Class\ C \implies C\ =\ Object \vee S\ =\ NT \vee (\exists D.\ S\ =\ Class\ D)$
apply (*ind-cases* $G \vdash S \preceq\ Class\ C$)
by *auto*

lemma *widen-Class-Class*: $G \vdash Class\ C \preceq\ Class\ cm \implies G \vdash C \preceq_C\ cm$
apply (*ind-cases* $G \vdash Class\ C \preceq\ Class\ cm$)
by *auto*

lemma *widen-Class-Class-eq* [*simp*]: $G \vdash Class\ C \preceq\ Class\ cm\ =\ G \vdash C \preceq_C\ cm$
apply (*rule iffI*)
apply (*erule widen-Class-Class*)
apply (*erule widen.subcls*)
done

lemma *widen-Class-Iface*: $G \vdash Class\ C \preceq\ Iface\ I \implies G \vdash C \leadsto I$
apply (*ind-cases* $G \vdash Class\ C \preceq\ Iface\ I$)
by *auto*

lemma *widen-Class-Iface-eq* [*simp*]: $G \vdash Class\ C \preceq\ Iface\ I\ =\ G \vdash C \leadsto I$
apply (*rule iffI*)
apply (*erule widen-Class-Iface*)
apply (*erule widen.implmt*)
done

lemma *widen-Array*: $G \vdash S.[] \preceq T \implies T = Class\ Object \vee (\exists T'.\ T = T'.[] \wedge G \vdash S \preceq T')$
apply (*ind-cases* $G \vdash S.[] \preceq T$)
by *auto*

lemma *widen-Array2*: $G \vdash S \preceq T.[] \implies S\ =\ NT \vee (\exists S'.\ S = S'.[] \wedge G \vdash S' \preceq T)$
apply (*ind-cases* $G \vdash S \preceq T.[]$)
by *auto*

lemma *widen-ArrayPrimT*: $G \vdash PrimT\ t.[] \preceq T \implies T = Class\ Object \vee T = PrimT\ t.[]$
apply (*ind-cases* $G \vdash PrimT\ t.[] \preceq T$)
by *auto*

lemma *widen-ArrayRefT*:

$G \vdash RefT\ t.[] \preceq T \Longrightarrow T = Class\ Object \lor (\exists s.\ T = RefT\ s.[] \land G \vdash RefT\ t \preceq RefT\ s)$
apply (*ind-cases* $G \vdash RefT\ t.[] \preceq T$)
by *auto*

lemma *widen-ArrayRefT-ArrayRefT-eq* [*simp*]:
$G \vdash RefT\ T.[] \preceq RefT\ T'.[] = G \vdash RefT\ T \preceq RefT\ T'$
apply (*rule iffI*)
apply (*drule widen-ArrayRefT*)
apply *simp*
apply (*erule widen.array*)
done

lemma *widen-Array-Array*: $G \vdash T.[] \preceq T'.[] \Longrightarrow G \vdash T \preceq T'$
apply (*drule widen-Array*)
apply *auto*
done

lemma *widen-Array-Class*: $G \vdash S.[] \preceq Class\ C \Longrightarrow C = Object$
by (*auto dest*: *widen-Array*)

lemma *widen-NT2*: $G \vdash S \preceq NT \Longrightarrow S = NT$
apply (*ind-cases* $G \vdash S \preceq NT$)
by *auto*

lemma *widen-Object*:
assumes *isrtype G T* **and** *ws-prog G*
shows $G \vdash RefT\ T \preceq Class\ Object$
proof (*cases T*)
case (*ClassT C*) **with** *assms* **have** $G \vdash C \preceq_C Object$ **by** (*auto intro*: *subcls-ObjectI*)
with *ClassT* **show** *?thesis* **by** *auto*
qed *simp-all*

lemma *widen-trans-lemma* [*rule-format* (*no-asm*)]:
$[\![G \vdash S \preceq U; \forall C.\ is\text{-}class\ G\ C \longrightarrow G \vdash C \preceq_C Object]\!] \Longrightarrow \forall T.\ G \vdash U \preceq T \longrightarrow G \vdash S \preceq T$
apply (*erule widen.induct*)
apply *safe*
prefer *5* **apply** (*drule widen-RefT*) **apply** *clarsimp*
apply (*frule-tac* [*1*] *widen-Iface*)
apply (*frule-tac* [*2*] *widen-Class*)
apply (*frule-tac* [*3*] *widen-Class*)
apply (*frule-tac* [*4*] *widen-Iface*)
apply (*frule-tac* [*5*] *widen-Class*)
apply (*frule-tac* [*6*] *widen-Array*)
apply *safe*
apply (*rule widen.int-obj*)
prefer *6* **apply** (*drule implmt-is-class*) **apply** *simp*
apply (*erule-tac* [!] *thin-rl*)
prefer *6* **apply** *simp*
apply (*rule-tac* [*9*] *widen.arr-obj*)
apply (*rotate-tac* [*9*] *−1*)
apply (*frule-tac* [*9*] *widen-RefT*)
apply (*auto elim*!: *rtrancl-trans subcls-implmt implmt-subint2*)

done

lemma *ws-widen-trans*: $[\![G \vdash S \preceq U;\ G \vdash U \preceq T;\ ws\text{-}prog\ G]\!] \Longrightarrow G \vdash S \preceq T$
by (*auto intro*: *widen-trans-lemma subcls-ObjectI*)

lemma *widen-antisym-lemma* [*rule-format* (*no-asm*)]: $[\![G \vdash S \preceq T;$
$\forall I\ J.\ G \vdash I \preceq I\ J \wedge G \vdash J \preceq I\ I \longrightarrow I = J;$
$\forall C\ D.\ G \vdash C \preceq_C\ D \wedge G \vdash D \preceq_C\ C \longrightarrow C = D;$
$\forall I\ .\ G \vdash Object \rightsquigarrow I \quad \longrightarrow False]\!] \Longrightarrow G \vdash T \preceq S \longrightarrow S = T$
apply (*erule widen.induct*)
apply (*auto dest*: *widen-Iface widen-NT2 widen-Class*)
done

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

lemma *widen-antisym*: $[\![G \vdash S \preceq T;\ G \vdash T \preceq S;\ ws\text{-}prog\ G]\!] \Longrightarrow S = T$
by (*fast elim*: *widen-antisym-lemma subint-antisym* [*THEN antisymD*]
subcls-antisym [*THEN antisymD*])

lemma *widen-ObjectD* [*dest*!]: $G \vdash Class\ Object \preceq T \Longrightarrow T = Class\ Object$
apply (*frule widen-Class*)
apply (*fast dest*: *widen-Class-Class widen-Class-Iface*)
done

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

lemma *widens-Nil* [*simp*]: $G \vdash [][\preceq][]$
apply (*unfold widens-def*)
apply *auto*
done

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

narrowing relation

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

$\neg G\vdash I\preceq I\ J \implies G\vdash \quad Iface\ I\succ Iface\ J$
| *array*: $G\vdash RefT\ S\succ RefT\ T \implies G\vdash \quad RefT\ S.[]\succ RefT\ T.[]$

lemma *narrow-RefT*: $G\vdash RefT\ R\succ T \implies \exists\, t.\ T{=}RefT\ t$
apply (*ind-cases* $G\vdash RefT\ R\succ T$)
by *auto*

lemma *narrow-RefT2*: $G\vdash S\succ RefT\ R \implies \exists\, t.\ S{=}RefT\ t$
apply (*ind-cases* $G\vdash S\succ RefT\ R$)
by *auto*

lemma *narrow-PrimT*: $G\vdash PrimT\ pt\succ T \implies \exists\, t.\ T{=}PrimT\ t$
by (*ind-cases* $G\vdash PrimT\ pt\succ T$)

lemma *narrow-PrimT2*: $G\vdash S\succ PrimT\ pt \implies$
$\exists\, t.\ S{=}PrimT\ t \wedge G\vdash PrimT\ t\preceq PrimT\ pt$
by (*ind-cases* $G\vdash S\succ PrimT\ pt$)

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*: $G\vdash PrimT\ pt\succ T \implies T{=}PrimT\ pt$
by (*ind-cases* $G\vdash PrimT\ pt\succ T$)

lemma *narrow-PrimT2-strong*: $G\vdash S\succ PrimT\ pt \implies S{=}PrimT\ pt$
by (*ind-cases* $G\vdash S\succ PrimT\ pt$)

Specialized versions for booleans also would work for real Java

lemma *narrow-Boolean*: $G\vdash PrimT\ Boolean\succ T \implies T{=}PrimT\ Boolean$
by (*ind-cases* $G\vdash PrimT\ Boolean\succ T$)

lemma *narrow-Boolean2*: $G\vdash S\succ PrimT\ Boolean \implies S{=}PrimT\ Boolean$
by (*ind-cases* $G\vdash S\succ PrimT\ Boolean$)

casting relation

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

lemma *cast-RefT*: $G\vdash RefT\ R\preceq?\ T \implies \exists\, t.\ T{=}RefT\ t$
apply (*ind-cases* $G\vdash RefT\ R\preceq?\ T$)
by (*auto dest*: *widen-RefT narrow-RefT*)

lemma *cast-RefT2*: $G \vdash S \preceq ?\ RefT\ R \Longrightarrow \exists\, t.\ S{=}RefT\ t$
apply (*ind-cases* $G \vdash S \preceq ?\ RefT\ R$)
by (*auto dest*: *widen-RefT2 narrow-RefT2*)

lemma *cast-PrimT*: $G \vdash PrimT\ pt \preceq ?\ T \Longrightarrow \exists\, t.\ T{=}PrimT\ t$
apply (*ind-cases* $G \vdash PrimT\ pt \preceq ?\ T$)
by (*auto dest*: *widen-PrimT narrow-PrimT*)

lemma *cast-PrimT2*: $G \vdash S \preceq ?\ PrimT\ pt \Longrightarrow \exists\, t.\ S{=}PrimT\ t \wedge G \vdash PrimT\ t \preceq PrimT\ pt$
apply (*ind-cases* $G \vdash S \preceq ?\ PrimT\ pt$)
by (*auto dest*: *widen-PrimT2 narrow-PrimT2*)

lemma *cast-Boolean*:
  assumes *bool-cast*: $G \vdash PrimT\ Boolean \preceq ?\ T$
  shows $T{=}PrimT\ Boolean$
using *bool-cast*
proof (*cases*)
  case *widen*
  hence $G \vdash PrimT\ Boolean \preceq\ T$
    by *simp*
  thus *?thesis* **by** (*rule widen-Boolean*)
next
  case *narrow*
  hence $G \vdash PrimT\ Boolean \succ T$
    by *simp*
  thus *?thesis* **by** (*rule narrow-Boolean*)
qed

lemma *cast-Boolean2*:
  assumes *bool-cast*: $G \vdash S \preceq ?\ PrimT\ Boolean$
  shows $S = PrimT\ Boolean$
using *bool-cast*
proof (*cases*)
  case *widen*
  hence $G \vdash S \preceq\ PrimT\ Boolean$
    by *simp*
  thus *?thesis* **by** (*rule widen-Boolean2*)
next
  case *narrow*
  hence $G \vdash S \succ PrimT\ Boolean$
    by *simp*
  thus *?thesis* **by** (*rule narrow-Boolean2*)
qed

end

Chapter 10

DeclConcepts

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

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

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

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

2 accessibility of types (cf. 6.6.1)

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

primrec
  accessible-in :: *prog* ⇒ *ty* ⇒ *pname* ⇒ *bool* (- ⊢ - *accessible′-in* - [*61*,*61*,*61*] *60*) **and**
  rt-accessible-in :: *prog* ⇒ *ref-ty* ⇒ *pname* ⇒ *bool* (- ⊢ - *accessible′-in″* - [*61*,*61*,*61*] *60*)
where
  G⊢(*PrimT p*) *accessible-in pack* = *True*
| *accessible-in-RefT-simp*:
  G⊢(*RefT r*) *accessible-in pack* = *G*⊢*r accessible-in′ pack*
| *G*⊢(*NullT*) *accessible-in′ pack* = *True*
| *G*⊢(*IfaceT I*) *accessible-in′ pack* = ((*pid I* = *pack*) ∨ *is-public G I*)
| *G*⊢(*ClassT C*) *accessible-in′ pack* = ((*pid C* = *pack*) ∨ *is-public G C*)
| *G*⊢(*ArrayT ty*) *accessible-in′ pack* = *G*⊢*ty accessible-in pack*

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

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

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

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

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

lemma *is-acc-classD*:
is-acc-class G P C ⟹ *is-class G C* ∧ *G*⊢(*Class C*) *accessible-in P*
by (*simp add*: *is-acc-class-def*)

lemma *is-acc-class-is-class*: *is-acc-class G P C* ⟹ *is-class G C*
by (*auto simp add*: *is-acc-class-def*)

lemma *is-acc-ifaceD*:
is-acc-iface G P I ⟹ *is-iface G I* ∧ *G*⊢(*Iface I*) *accessible-in P*
by (*simp add*: *is-acc-iface-def*)

lemma *is-acc-typeD*:
is-acc-type G P T ⟹ *is-type G T* ∧ *G*⊢*T accessible-in P*
by (*simp add*: *is-acc-type-def*)

lemma *is-acc-reftypeD*:
is-acc-reftype G P T ⟹ *isrtype G T* ∧ *G*⊢*T accessible-in′ P*
by (*simp add*: *is-acc-reftype-def*)

3 accessibility of members

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

Various technical conversion and selection functions

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

class *has-accmodi* =
fixes *accmodi*:: *′a* ⇒ *acc-modi*

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

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

instance ..

end

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

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

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

instance ..

end

lemma *decl-acc-modi-simp*[*simp*]: *accmodi* (*d*::(*'a*::*type*) *decl-scheme*) = *access d*
by (*simp add*: *decl-acc-modi-def*)

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

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

instance ..

end

lemma *pair-acc-modi-simp*[*simp*]: *accmodi* (*x*,*a*) = (*accmodi a*)
by (*simp add*: *pair-acc-modi-def*)

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

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

instance ..

end

lemma *memberdecl-fdecl-acc-modi-simp*[*simp*]:
accmodi (*fdecl m*) = *accmodi m*
by (*simp add*: *memberdecl-acc-modi-def*)

lemma *memberdecl-mdecl-acc-modi-simp*[*simp*]:
accmodi (*mdecl m*) = *accmodi m*
by (*simp add*: *memberdecl-acc-modi-def*)

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

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

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

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

instance ..

end

lemma *qtname-declclass-def*:
  declclass q ≡ (*q*::*qtname*)
  by (*induct q*) (*simp add*: *declclass-qtname-ext-def*)

lemma *qtname-declclass-simp*[*simp*]: *declclass* (*q*::*qtname*) = *q*
by (*simp add*: *qtname-declclass-def*)

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

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

instance ..

end

lemma *pair-declclass-simp*[*simp*]: *declclass* (*c*,*x*) = *declclass c*
by (*simp add*: *pair-declclass-def*)

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

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

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

instance ..

end

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

instance ..

end

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

lemma *member-is-static-simp*: *is-static* (*m*::*'a member-scheme*) = *static m*
by (*simp add*: *static-field-type-is-static-def*)

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

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

instance ..

end

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

lemma *pair-is-static-simp1*: *is-static p* = *is-static* (*snd p*)
by (*simp add*: *pair-is-static-def*)

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

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

instance ..

end

lemma *memberdecl-is-static-fdecl-simp*[*simp*]:
is-static (*fdecl f*) = *is-static f*
by (*simp add*: *memberdecl-is-static-def*)

lemma *memberdecl-is-static-mdecl-simp*[*simp*]:
is-static (*mdecl m*) = *is-static m*
by (*simp add*: *memberdecl-is-static-def*)

lemma *mhead-static-simp* [*simp*]: *is-static* (*mhead m*) = *is-static m*
by (*cases m*) (*simp add*: *mhead-def member-is-static-simp*)

— some mnemotic selectors for various pairs

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

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

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

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

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

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

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

lemma *decliface-simp*[*simp*]: *decliface* (*I*,*m*) = *I*
by (*simp add*: *decliface-def*)

lemma *mbr-simp*[*simp*]: *mbr* (*C*,*m*) = *m*
by (*simp add*: *mbr-def*)

lemma *access-mbr-simp* [*simp*]: (*accmodi* (*mbr m*)) = *accmodi m*
by (*cases m*) (*simp add*: *mbr-def*)

lemma *mthd-simp*[*simp*]: *mthd* (*C*,*m*) = *m*
by (*simp add*: *mthd-def*)

lemma *fld-simp*[*simp*]: *fld* (*C*,*f*) = *f*
by (*simp add*: *fld-def*)

lemma *accmodi-simp*[*simp*]: *accmodi* (*C*,*m*) = *access m*
by (*simp*)

lemma *access-mthd-simp* [*simp*]: (*access* (*mthd m*)) = *accmodi m*
by (*cases m*) (*simp add*: *mthd-def*)

lemma *access-fld-simp* [*simp*]: (*access* (*fld f*)) = *accmodi f*
by (*cases f*) (*simp add*: *fld-def*)

lemma *static-mthd-simp*[*simp*]: *static* (*mthd m*) = *is-static m*
by (*cases m*) (*simp add*: *mthd-def member-is-static-simp*)

lemma *mthd-is-static-simp* [*simp*]: *is-static* (*mthd m*) = *is-static m*
by (*cases m*) *simp*

lemma *static-fld-simp*[*simp*]: *static* (*fld f*) = *is-static f*
by (*cases f*) (*simp add*: *fld-def member-is-static-simp*)

lemma *ext-field-simp* [*simp*]: (*declclass f*,*fld f*) = *f*
by (*cases f*) (*simp add*: *fld-def*)

lemma *ext-method-simp* [*simp*]: (*declclass m*,*mthd m*) = *m*
by (*cases m*) (*simp add*: *mthd-def*)

lemma *ext-mbr-simp* [*simp*]: (*declclass m*,*mbr m*) = *m*
by (*cases m*) (*simp add*: *mbr-def*)

lemma *fname-simp*[*simp*]:*fname* (*n*,*c*) = *n*
by (*simp add*: *fname-def*)

lemma *declclassf-simp*[*simp*]:*declclassf* (*n*,*c*) = *c*
by (*simp add*: *declclassf-def*)

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

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

definition
fldclass :: *vname* × *qtname* ⇒ *qtname*
where *fldclass* = *snd*

lemma *fldname-simp*[*simp*]: *fldname* (*n*,*c*) = *n*
by (*simp add*: *fldname-def*)

lemma *fldclass-simp*[*simp*]: *fldclass* (*n*,*c*) = *c*
by (*simp add*: *fldclass-def*)

lemma *ext-fieldname-simp*[*simp*]: (*fldname f*,*fldclass f*) = *f*
by (*simp add*: *fldname-def fldclass-def*)

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

definition
methdMembr :: *qtname* × *mdecl* ⇒ *qtname* × *memberdecl*
where *methdMembr m* = (*fst m*, *mdecl* (*snd m*))

lemma *methdMembr-simp*[*simp*]: *methdMembr* (*c*,*m*) = (*c*,*mdecl m*)
by (*simp add*: *methdMembr-def*)

lemma *accmodi-methdMembr-simp*[*simp*]: *accmodi* (*methdMembr m*) = *accmodi m*
by (*cases m*) (*simp add*: *methdMembr-def*)

lemma *is-static-methdMembr-simp*[*simp*]: *is-static* (*methdMembr m*) = *is-static m*
by (*cases m*) (*simp add*: *methdMembr-def*)

lemma *declclass-methdMembr-simp*[*simp*]: *declclass* (*methdMembr m*) = *declclass m*
by (*cases m*) (*simp add*: *methdMembr-def*)

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

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

lemma *method-simp*[*simp*]: *method sig* (*C*,*m*) = (*C*,*mdecl* (*sig*,*m*))
by (*simp add*: *method-def*)

lemma *accmodi-method-simp*[*simp*]: *accmodi* (*method sig m*) = *accmodi m*
by (*simp add*: *method-def*)

lemma *declclass-method-simp*[*simp*]: *declclass* (*method sig m*) = *declclass m*
by (*simp add*: *method-def*)

lemma *is-static-method-simp*[*simp*]: *is-static* (*method sig m*) = *is-static m*
by (*cases m*) (*simp add*: *method-def*)

lemma *mbr-method-simp*[*simp*]: *mbr* (*method sig m*) = *mdecl* (*sig*,*mthd m*)
by (*simp add*: *mbr-def method-def*)

lemma *memberid-method-simp*[*simp*]: *memberid* (*method sig m*) = *mid sig*
by (*simp add*: *method-def*)

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

lemma *fieldm-simp*[*simp*]: *fieldm n* (*C*,*f*) = (*C*,*fdecl* (*n*,*f*))
by (*simp add*: *fieldm-def*)

lemma *accmodi-fieldm-simp*[*simp*]: *accmodi* (*fieldm n f*) = *accmodi f*
by (*simp add*: *fieldm-def*)

lemma *declclass-fieldm-simp*[*simp*]: *declclass* (*fieldm n f*) = *declclass f*
by (*simp add*: *fieldm-def*)

lemma *is-static-fieldm-simp*[*simp*]: *is-static* (*fieldm n f*) = *is-static f*
by (*cases f*) (*simp add*: *fieldm-def*)

lemma *mbr-fieldm-simp*[*simp*]: *mbr* (*fieldm n f*) = *fdecl* (*n*,*fld f*)
by (*simp add*: *mbr-def fieldm-def*)

lemma *memberid-fieldm-simp*[*simp*]: *memberid* (*fieldm n f*) = *fid n*
by (*simp add*: *fieldm-def*)

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

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

lemma *msig-simp*[*simp*]: *msig* (*c*,(*s*,*m*)) = *s*
by (*simp add*: *msig-def*)

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

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

lemma *qmdecl-simp*[*simp*]: *qmdecl sig* (*C*,*m*) = (*C*,(*sig*,*m*))
by (*simp add*: *qmdecl-def*)

lemma *declclass-qmdecl-simp*[*simp*]: *declclass* (*qmdecl sig m*) = *declclass m*
by (*simp add*: *qmdecl-def*)

lemma *accmodi-qmdecl-simp*[*simp*]: *accmodi* (*qmdecl sig m*) = *accmodi m*
by (*simp add*: *qmdecl-def*)

lemma *is-static-qmdecl-simp*[*simp*]: *is-static* (*qmdecl sig m*) = *is-static m*
by (*cases m*) (*simp add*: *qmdecl-def*)

lemma *msig-qmdecl-simp*[*simp*]: *msig* (*qmdecl sig m*) = *sig*
by (*simp add*: *qmdecl-def*)

lemma *mdecl-qmdecl-simp*[*simp*]:
mdecl (*mthd* (*qmdecl sig new*)) = *mdecl* (*sig*, *mthd new*)
by (*simp add*: *qmdecl-def*)

lemma *methdMembr-qmdecl-simp* [*simp*]:
methdMembr (*qmdecl sig old*) = *method sig old*
by (*simp add*: *methdMembr-def qmdecl-def method-def*)

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

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

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

instance ..

end

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

instance ..

end

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

instance ..

end

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

lemma *mhead-resTy-simp*: *resTy* (*m*::′*a mhead-scheme*) = *resT m*
by (*simp add*: *mhead-ext-type-resTy-def*)

lemma *resTy-mhead* [*simp*]:*resTy* (*mhead m*) = *resTy m*
by (*simp add*: *mhead-def mhead-resTy-simp*)

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

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

instance ..

end

lemma *pair-resTy-simp*[*simp*]: *resTy* (*x*,*m*) = *resTy m*
by (*simp add*: *pair-resTy-def*)

lemma *qmdecl-resTy-simp* [*simp*]: *resTy* (*qmdecl sig m*) = *resTy m*
by (*cases m*) (*simp*)

lemma *resTy-mthd* [*simp*]:*resTy* (*mthd m*) = *resTy m*
by (*cases m*) (*simp add*: *mthd-def*)

inheritable-in

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

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

definition
inheritable-in :: *prog* ⇒ (*qtname* × *memberdecl*) ⇒ *pname* ⇒ *bool* (- ⊢ - *inheritable*′*-in* - [*61*,*61*,*61*] *60*)
where
G⊢*membr inheritable-in pack* =
(*case* (*accmodi membr*) *of*

Private ⇒ *False*
| *Package* ⇒ (*pid* (*declclass membr*)) = *pack*
| *Protected* ⇒ *True*
| *Public* ⇒ *True*)

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

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

declared-in/undeclared-in

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

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

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

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

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

lemma *declared-in-classD*:
G⊢*m declared-in C* ⟹ *is-class G C*
by (*cases m*)
(*auto simp add*: *declared-in-def cdeclaredmethd-def cdeclaredfield-def*)

definition
undeclared-in :: *prog* ⇒ *memberid* ⇒ *qtname* ⇒ *bool* (-⊢ - *undeclared*′-*in* - [*61*,*61*,*61*] *60*)
where

$G \vdash m$ *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: ⟦$G \vdash$*mbr m declared-in C*;*declclass m* = *C*⟧ ⟹ $G \vdash m$ *member-of C*
| *Inherited*: ⟦$G \vdash m$ *inheritable-in* (*pid C*); $G \vdash$*memberid m undeclared-in C*;
$G \vdash C \prec_C 1$ *S*; $G \vdash$(*Class S*) *accessible-in* (*pid C*);$G \vdash m$ *member-of S*
⟧ ⟹ $G \vdash m$ *member-of C*

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

abbreviation
method-member-of:: *prog* ⇒ (*qtname* × *mdecl*) ⇒ *qtname* ⇒ *bool*
(- ⊢*Method* - *member′-of* - [61,61,61] 60)
where $G \vdash$*Method m member-of C* == $G \vdash$(*methdMembr m*) *member-of C*

abbreviation
methd-member-of:: *prog* ⇒ *sig* ⇒ (*qtname* × *methd*) ⇒ *qtname* ⇒ *bool*
(- ⊢*Methd* - - *member′-of* - [61,61,61,61] 60)
where $G \vdash$*Methd s m member-of C* == $G \vdash$(*method s m*) *member-of C*

abbreviation
fieldm-member-of:: *prog* ⇒ *vname* ⇒ (*qtname* × *field*) ⇒ *qtname* ⇒ *bool*
(- ⊢*Field* - - *member′-of* - [61,61,61] 60)
where $G \vdash$*Field n f member-of C* == $G \vdash$*fieldm n f member-of C*

definition
inherits :: *prog* ⇒ *qtname* ⇒ (*qtname* × *memberdecl*) ⇒ *bool* (- ⊢ - *inherits* - [61,61,61] 60)
where
$G \vdash C$ *inherits m* =
($G \vdash m$ *inheritable-in* (*pid C*) ∧ $G \vdash$*memberid m undeclared-in C* ∧
(∃ *S*. $G \vdash C \prec_C 1$ *S* ∧ $G \vdash$(*Class S*) *accessible-in* (*pid C*) ∧ $G \vdash m$ *member-of S*))

lemma *inherits-member*: $G \vdash C$ *inherits m* ⟹ $G \vdash m$ *member-of C*
by (*auto simp add*: *inherits-def intro*: *members.Inherited*)

definition
member-in :: *prog* ⇒ (*qtname* × *memberdecl*) ⇒ *qtname* ⇒ *bool* (- ⊢ - *member′-in* - [61,61,61] 60)
where $G \vdash m$ *member-in C* = (∃ *provC*. $G \vdash C \preceq_C$ *provC* ∧ $G \vdash m$ *member-of provC*)

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

abbreviation

method-member-in:: *prog* ⇒ (*qtname* × *mdecl*) ⇒ *qtname* ⇒ *bool*
(- ⊢*Method* - *member'-in* - [61,61,61] 60)
where *G*⊢*Method m member-in C* == *G*⊢(*methdMembr m*) *member-in C*

abbreviation
methd-member-in:: *prog* ⇒ *sig* ⇒ (*qtname* × *methd*) ⇒ *qtname* ⇒ *bool*
(- ⊢*Methd* - - *member'-in* - [61,61,61,61] 60)
where *G*⊢*Methd s m member-in C* == *G*⊢(*method s m*) *member-in C*

lemma *member-inD*: *G*⊢*m member-in C*
⟹ ∃ *provC*. *G*⊢ *C* $\preceq_C$ *provC* ∧ *G* ⊢ *m member-of provC*
by (*auto simp add*: *member-in-def*)

lemma *member-inI*: ⟦*G* ⊢ *m member-of provC*;*G*⊢ *C* $\preceq_C$ *provC*⟧ ⟹ *G*⊢*m member-in C*
by (*auto simp add*: *member-in-def*)

lemma *member-of-to-member-in*: *G* ⊢ *m member-of C* ⟹ *G* ⊢*m member-in C*
by (*auto intro*: *member-inI*)

overriding

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

Static overriding (used during the typecheck of the compiler)

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

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

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

Dynamic overriding (used during the typecheck of the compiler)

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

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

$G \vdash resTy\ new \preceq resTy\ old$
$\rrbracket \Longrightarrow G \vdash new\ overrides\ old$

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

abbreviation (*input*)
sig-stat-overrides::
$prog \Rightarrow sig \Rightarrow (qtname \times methd) \Rightarrow (qtname \times methd) \Rightarrow bool$
$(\text{-,-}\vdash \text{ - } overrides_S \text{ - } [61,61,61,61]\ 60)$
where $G,s \vdash new\ overrides_S\ old == G \vdash (qmdecl\ s\ new)\ overrides_S\ (qmdecl\ s\ old)$

abbreviation (*input*)
sig-overrides:: $prog \Rightarrow sig \Rightarrow (qtname \times methd) \Rightarrow (qtname \times methd) \Rightarrow bool$
$(\text{-,-}\vdash \text{ - } overrides \text{ - } [61,61,61,61]\ 60)$
where $G,s \vdash new\ overrides\ old == G \vdash (qmdecl\ s\ new)\ overrides\ (qmdecl\ s\ old)$

Hiding

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

abbreviation
sig-hides:: $prog \Rightarrow sig \Rightarrow (qtname \times methd) \Rightarrow (qtname \times methd) \Rightarrow bool$
$(\text{-,-}\vdash \text{ - } hides \text{ - } [61,61,61,61]\ 60)$
where $G,s \vdash new\ hides\ old == G \vdash (qmdecl\ s\ new)\ hides\ (qmdecl\ s\ old)$

lemma *hidesI*:
$\llbracket is\text{-}static\ new;\ msig\ new = msig\ old;$
$G \vdash (declclass\ new) \prec_C (declclass\ old);$
$G \vdash Method\ new\ declared\text{-}in\ (declclass\ new);$
$G \vdash Method\ old\ declared\text{-}in\ (declclass\ old);$
$G \vdash Method\ old\ inheritable\text{-}in\ pid\ (declclass\ new)$
$\rrbracket \Longrightarrow G \vdash new\ hides\ old$
by (*auto simp add*: *hides-def*)

lemma *hidesD*:
$\llbracket G \vdash new\ hides\ old \rrbracket \Longrightarrow$
$declclass\ new \neq Object \wedge is\text{-}static\ new \wedge msig\ new = msig\ old \wedge$
$G \vdash (declclass\ new) \prec_C (declclass\ old) \wedge$
$G \vdash Method\ new\ declared\text{-}in\ (declclass\ new) \wedge$
$G \vdash Method\ old\ declared\text{-}in\ (declclass\ old)$
by (*auto simp add*: *hides-def*)

lemma *overrides-commonD*:
$\llbracket G \vdash new\ overrides\ old \rrbracket \Longrightarrow$
$declclass\ new \neq Object \wedge \neg\ is\text{-}static\ new \wedge \neg\ is\text{-}static\ old \wedge$
$accmodi\ new \neq Private \wedge$
$msig\ new = msig\ old\ \wedge$

$G \vdash (declclass\ new) \prec_C (declclass\ old) \land$
$G \vdash Method\ new\ declared\text{-}in\ (declclass\ new) \land$
$G \vdash Method\ old\ declared\text{-}in\ (declclass\ old)$
by (*induct set*: *overridesR*) (*auto intro*: *trancl-trans*)

lemma *ws-overrides-commonD*:
$[\![G \vdash new\ overrides\ old; ws\text{-}prog\ G]\!] \Longrightarrow$
$declclass\ new \neq Object \land \neg\ is\text{-}static\ new \land \neg\ is\text{-}static\ old \land$
$accmodi\ new \neq Private \land G \vdash resTy\ new \preceq resTy\ old \land$
$msig\ new = msig\ old\ \land$
$G \vdash (declclass\ new) \prec_C (declclass\ old) \land$
$G \vdash Method\ new\ declared\text{-}in\ (declclass\ new) \land$
$G \vdash Method\ old\ declared\text{-}in\ (declclass\ old)$
by (*induct set*: *overridesR*) (*auto intro*: *trancl-trans ws-widen-trans*)

lemma *overrides-eq-sigD*:
$[\![G \vdash new\ overrides\ old]\!] \Longrightarrow msig\ old = msig\ new$
by (*auto dest*: *overrides-commonD*)

lemma *hides-eq-sigD*:
$[\![G \vdash new\ hides\ old]\!] \Longrightarrow msig\ old = msig\ new$
by (*auto simp add*: *hides-def*)

permits access

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

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

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

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

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

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

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

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

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

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

| *Overriding*: !!*class*. ⟦*G*⊢*membr member-in class*;
membr=(*C*,*mdecl new*);

$G \vdash (C, new)$ *overrides old*;
$G \vdash$ *class* $\prec_C$ *supr*;
$G \vdash$ *Method old in supr dyn-accessible-from accclass*
$\rrbracket \Longrightarrow G \vdash$ *membr in class dyn-accessible-from accclass*

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

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

lemma *accessible-from-commonD*: *G*⊢*m of C accessible-from S*
⟹ *G*⊢*m member-of C* ∧ *G*⊢(*Class C*) *accessible-in* (*pid S*)
by (*auto elim*: *accessible-fromR.induct*)

lemma *unique-declaration*:
⟦*G*⊢*m declared-in C*; *G*⊢*n declared-in C*; *memberid m* = *memberid n* ⟧
⟹ *m* = *n*
apply (*cases m*)
apply (*cases n*,
auto simp add: *declared-in-def cdeclaredmethd-def cdeclaredfield-def*)+
done

lemma *declared-not-undeclared*:
G⊢*m declared-in C* ⟹ ¬ *G*⊢ *memberid m undeclared-in C*
by (*cases m*) (*auto simp add*: *declared-in-def undeclared-in-def*)

lemma *undeclared-not-declared*:
G⊢ *memberid m undeclared-in C* ⟹ ¬ *G*⊢ *m declared-in C*
by (*cases m*) (*auto simp add*: *declared-in-def undeclared-in-def*)

lemma *not-undeclared-declared*:
¬ *G*⊢ *membr-id undeclared-in C* ⟹ (∃ *m*. *G*⊢*m declared-in C* ∧
membr-id = *memberid m*)
proof −
assume *not-undecl*:¬ *G*⊢ *membr-id undeclared-in C*
show *?thesis* (**is** *?P membr-id*)
proof (*cases membr-id*)
case (*fid vname*)
with *not-undecl*
obtain *fld* **where**
G⊢*fdecl* (*vname*,*fld*) *declared-in C*
by (*auto simp add*: *undeclared-in-def declared-in-def*

```
                  cdeclaredfield-def)
    with fid show ?thesis
      by auto
  next
    case (mid sig)
    with not-undecl
    obtain mthd where
      G⊢mdecl (sig,mthd) declared-in C
      by (auto simp add: undeclared-in-def declared-in-def
                  cdeclaredmethd-def)
    with mid show ?thesis
      by auto
  qed
qed


lemma unique-declared-in:
 ⟦G⊢m declared-in C; G⊢n declared-in C; memberid m = memberid n⟧
 ⟹ m = n
by (auto simp add: declared-in-def cdeclaredmethd-def cdeclaredfield-def
         split: memberdecl.splits)


lemma unique-member-of:
  assumes n: G⊢n member-of C and
          m: G⊢m member-of C and
       eqid: memberid n = memberid m
  shows n=m
proof −
  from n m eqid
  show n=m
  proof (induct)
    case (Immediate n C)
    assume member-n: G⊢ mbr n declared-in C declclass n = C
    assume eqid: memberid n = memberid m
    assume G ⊢ m member-of C
    then show n=m
    proof (cases)
      case Immediate
      with eqid member-n
      show ?thesis
        by (cases n, cases m)
           (auto simp add: declared-in-def
                         cdeclaredmethd-def cdeclaredfield-def
                   split: memberdecl.splits)
    next
      case Inherited
      with eqid member-n
      show ?thesis
        by (cases n) (auto dest: declared-not-undeclared)
    qed
  next
    case (Inherited n C S)
    assume undecl: G⊢ memberid n undeclared-in C
    assume  super: G⊢C≺C1S
    assume    hyp: ⟦G ⊢ m member-of S; memberid n = memberid m⟧ ⟹ n = m
    assume   eqid: memberid n = memberid m
    assume G ⊢ m member-of C
    then show n=m
```

```
    proof (cases)
      case Immediate
      then have G⊢ mbr m declared-in C by simp
      with eqid undecl
      show ?thesis
        by (cases m) (auto dest: declared-not-undeclared)
    next
      case Inherited
      with super have G ⊢ m member-of S
        by (auto dest!: subcls1D)
      with eqid hyp
      show ?thesis
        by blast
    qed
  qed
qed
```

```
lemma member-of-is-classD: G⊢m member-of C ⟹ is-class G C
proof (induct set: members)
  case (Immediate m C)
  assume G⊢ mbr m declared-in C
  then show is-class G C
    by (cases mbr m)
       (auto simp add: declared-in-def cdeclaredmethd-def cdeclaredfield-def)
next
  case (Inherited m C S)
  assume G⊢C≺C1S and is-class G S
  then show is-class G C
    by − (rule subcls-is-class2,auto)
qed
```

```
lemma member-of-declC:
 G⊢m member-of C
  ⟹ G⊢mbr m declared-in (declclass m)
by (induct set: members) auto
```

```
lemma member-of-member-of-declC:
 G⊢m member-of C
  ⟹ G⊢m member-of (declclass m)
by (auto dest: member-of-declC intro: members.Immediate)
```

```
lemma member-of-class-relation:
  G⊢m member-of C ⟹ G⊢C ≼C declclass m
proof (induct set: members)
  case (Immediate m C)
  then show G⊢C ≼C declclass m by simp
next
  case (Inherited m C S)
  then show G⊢C ≼C declclass m
    by (auto dest: r-into-rtrancl intro: rtrancl-trans)
qed
```

```
lemma member-in-class-relation:
  G⊢m member-in C ⟹ G⊢C ≼C declclass m
```

by (*auto dest*: *member-inD member-of-class-relation*
intro: *rtrancl-trans*)

lemma *stat-override-declclasses-relation*:
⟦*G*⊢(*declclass new*) $\prec_C 1$ *superNew*; *G* ⊢*Method old member-of superNew* ⟧
⟹ *G*⊢(*declclass new*) $\prec_C$ (*declclass old*)
apply (*rule trancl-rtrancl-trancl*)
apply (*erule r-into-trancl*)
apply (*cases old*)
apply (*auto dest*: *member-of-class-relation*)
done

lemma *stat-overrides-commonD*:
⟦*G*⊢*new overrides*$_S$ *old*⟧ ⟹
declclass new ≠ *Object* ∧ ¬ *is-static new* ∧ *msig new* = *msig old* ∧
G⊢(*declclass new*) $\prec_C$ (*declclass old*) ∧
G⊢*Method new declared-in* (*declclass new*) ∧
G⊢*Method old declared-in* (*declclass old*)
apply (*induct set*: *stat-overridesR*)
apply (*frule* (*1*) *stat-override-declclasses-relation*)
apply (*auto intro*: *trancl-trans*)
done

lemma *member-of-Package*:
assumes *G*⊢*m member-of C*
and *accmodi m* = *Package*
shows *pid* (*declclass m*) = *pid C*
using *assms*
proof *induct*
case *Immediate*
then show *?case* **by** *simp*
next
case *Inherited*
then show *?case* **by** (*auto simp add*: *inheritable-in-def*)
qed

lemma *member-in-declC*: *G*⊢*m member-in C*⟹ *G*⊢*m member-in* (*declclass m*)
proof −
assume *member-in-C*: *G*⊢*m member-in C*
from *member-in-C*
obtain *provC* **where**
subclseq-C-provC: *G*⊢ *C* $\preceq_C$ *provC* **and**
member-of-provC: *G* ⊢ *m member-of provC*
by (*auto simp add*: *member-in-def*)
from *member-of-provC*
have *G* ⊢ *m member-of declclass m*
by (*rule member-of-member-of-declC*)
moreover
from *member-in-C*
have *G*⊢*C* $\preceq_C$ *declclass m*
by (*rule member-in-class-relation*)
ultimately
show *?thesis*
by (*auto simp add*: *member-in-def*)
qed

lemma *dyn-accessible-from-commonD*: *G*⊢*m in C dyn-accessible-from S*
$\Longrightarrow$ *G*⊢*m member-in C*
by (*auto elim*: *dyn-accessible-fromR.induct*)

lemma *no-Private-stat-override*:
$[\![$*G*⊢*new overrides*$_S$ *old*$]\!]$ $\Longrightarrow$ *accmodi old* ≠ *Private*
by (*induct set*: *stat-overridesR*) (*auto simp add*: *inheritable-in-def*)

lemma *no-Private-override*: $[\![$*G*⊢*new overrides old*$]\!]$ $\Longrightarrow$ *accmodi old* ≠ *Private*
by (*induct set*: *overridesR*) (*auto simp add*: *inheritable-in-def*)

lemma *permits-acc-inheritance*:
$[\![$*G*⊢*m in statC permits-acc-from accC*; *G*⊢*dynC* $\preceq_C$ *statC*
$]\!]$ $\Longrightarrow$ *G*⊢*m in dynC permits-acc-from accC*
by (*cases accmodi m*)
(*auto simp add*: *permits-acc-def*
intro: *subclseq-trans*)

lemma *permits-acc-static-declC*:
$[\![$*G*⊢*m in C permits-acc-from accC*; *G*⊢*m member-in C*; *is-static m*
$]\!]$ $\Longrightarrow$ *G*⊢*m in* (*declclass m*) *permits-acc-from accC*
by (*cases accmodi m*) (*auto simp add*: *permits-acc-def*)

lemma *dyn-accessible-from-static-declC*:
assumes *acc-C*: *G*⊢*m in C dyn-accessible-from accC* **and**
static: *is-static m*
shows *G*⊢*m in* (*declclass m*) *dyn-accessible-from accC*
proof −
from *acc-C static*
show *G*⊢*m in* (*declclass m*) *dyn-accessible-from accC*
proof (*induct*)
case (*Immediate m C*)
then show *?case*
by (*auto intro*!: *dyn-accessible-fromR.Immediate*
dest: *member-in-declC permits-acc-static-declC*)
next
case (*Overriding m C declCNew new old sup*)
then have ¬ *is-static m*
by (*auto dest*: *overrides-commonD*)
moreover
assume *is-static m*
ultimately show *?case*
by *contradiction*
qed
qed

lemma *field-accessible-fromD*:
$[\![$*G*⊢*membr of C accessible-from accC*;*is-field membr*$]\!]$
$\Longrightarrow$ *G*⊢*membr member-of C* ∧
G⊢(*Class C*) *accessible-in* (*pid accC*) ∧
G⊢*membr in C permits-acc-from accC*

by (*cases set*: *accessible-fromR*)
  (*auto simp add*: *is-field-def split*: *memberdecl.splits*)

lemma *field-accessible-from-permits-acc-inheritance*:
$[\![G\vdash membr\ of\ statC\ accessible\text{-}from\ accC;\ is\text{-}field\ membr;\ G \vdash dynC \preceq_C statC]\!]$
$\Longrightarrow G\vdash membr\ in\ dynC\ permits\text{-}acc\text{-}from\ accC$
by (*auto dest*: *field-accessible-fromD intro*: *permits-acc-inheritance*)

lemma *accessible-fieldD*:
$[\![G\vdash membr\ of\ C\ accessible\text{-}from\ accC;\ is\text{-}field\ membr]\!]$
$\Longrightarrow G\vdash membr\ member\text{-}of\ C \wedge$
  $G\vdash(Class\ C)\ accessible\text{-}in\ (pid\ accC) \wedge$
  $G\vdash membr\ in\ C\ permits\text{-}acc\text{-}from\ accC$
by (*induct rule*: *accessible-fromR.induct*) (*auto dest*: *is-fieldD*)

lemma *member-of-Private*:
$[\![G\vdash m\ member\text{-}of\ C;\ accmodi\ m = Private]\!] \Longrightarrow declclass\ m = C$
by (*induct set*: *members*) (*auto simp add*: *inheritable-in-def*)

lemma *member-of-subclseq-declC*:
  $G\vdash m\ member\text{-}of\ C \Longrightarrow G\vdash C \preceq_C declclass\ m$
by (*induct set*: *members*) (*auto dest*: *r-into-rtrancl intro*: *rtrancl-trans*)

lemma *member-of-inheritance*:
  assumes *m*: $G\vdash m\ member\text{-}of\ D$ **and**
    subclseq-D-C: $G\vdash D \preceq_C C$ **and**
    subclseq-C-m: $G\vdash C \preceq_C declclass\ m$ **and**
    ws: *ws-prog G*
  shows $G\vdash m\ member\text{-}of\ C$
proof −
  from *m subclseq-D-C subclseq-C-m*
  show *?thesis*
  proof (*induct*)
    case (*Immediate m D*)
    assume $declclass\ m = D$ **and**
      $G\vdash D\preceq_C C$ **and** $G\vdash C\preceq_C declclass\ m$
    with *ws* **have** $D=C$
      by (*auto intro*: *subclseq-acyclic*)
    with *Immediate*
    show $G\vdash m\ member\text{-}of\ C$
      by (*auto intro*: *members.Immediate*)
  next
    case (*Inherited m D S*)
    assume *member-of-D-props*:
      $G \vdash m\ inheritable\text{-}in\ pid\ D$
      $G\vdash memberid\ m\ undeclared\text{-}in\ D$
      $G \vdash Class\ S\ accessible\text{-}in\ pid\ D$
      $G \vdash m\ member\text{-}of\ S$
    assume *super*: $G\vdash D\prec_C 1 S$

```
    assume hyp: ⟦G⊢S≼C C; G⊢C≼C declclass m⟧ ⟹ G ⊢ m member-of C
    assume subclseq-C-m: G⊢C≼C declclass m
    assume G⊢D≼C C
    then show G⊢m member-of C
    proof (cases rule: subclseq-cases)
      case Eq
      assume D=C
      with super member-of-D-props
      show ?thesis
        by (auto intro: members.Inherited)
    next
      case Subcls
      assume G⊢D≺C C
      with super
      have G⊢S≼C C
        by (auto dest: subcls1D subcls-superD)
      with subclseq-C-m hyp show ?thesis
        by blast
    qed
  qed
qed


lemma member-of-subcls:
  assumes     old: G⊢old member-of C and
              new: G⊢new member-of D and
             eqid: memberid new = memberid old and
      subclseq-D-C: G⊢D ≼C C and
    subcls-new-old: G⊢declclass new ≺C declclass old and
               ws: ws-prog G
  shows G⊢D ≺C C
proof -
  from old
  have subclseq-C-old: G⊢C ≼C declclass old
    by (auto dest: member-of-subclseq-declC)
  from new
  have subclseq-D-new: G⊢D ≼C declclass new
    by (auto dest: member-of-subclseq-declC)
  from subcls-new-old ws
  have neq-new-old: new≠old
    by (cases new,cases old) (auto dest: subcls-irrefl)
  from subclseq-D-new subclseq-D-C
  have G⊢(declclass new) ≼C C ∨ G⊢C ≼C (declclass new)
    by (rule subcls-compareable)
  then have G⊢(declclass new) ≼C C
  proof
    assume G⊢declclass new≼C C then show ?thesis .
  next
    assume G⊢C ≼C (declclass new)
    with new subclseq-D-C ws
    have G⊢new member-of C
      by (blast intro: member-of-inheritance)
    with eqid old
    have new=old
      by (blast intro: unique-member-of)
    with neq-new-old
    show ?thesis
      by contradiction
  qed
```

then show *?thesis*
proof (*cases rule*: *subclseq-cases*)
case *Eq*
assume *declclass new* = *C*
with *new* **have** *G*⊢*new member-of C*
by (*auto dest*: *member-of-member-of-declC*)
with *eqid old*
have *new*=*old*
by (*blast intro*: *unique-member-of*)
with *neq-new-old*
show *?thesis*
by *contradiction*
next
case *Subcls*
assume *G*⊢*declclass new*$\prec_C$ *C*
with *subclseq-D-new*
show *G*⊢*D*$\prec_C$ *C*
by (*rule rtrancl-trancl-trancl*)
qed
qed

corollary *member-of-overrides-subcls*:
⟦*G*⊢*Methd sig old member-of C*; *G*⊢*Methd sig new member-of D*;*G*⊢*D* $\preceq_C$ *C*;
G,*sig*⊢*new overrides old*; *ws-prog G*⟧
⟹ *G*⊢*D* $\prec_C$ *C*
by (*drule overrides-commonD*) (*auto intro*: *member-of-subcls*)

corollary *member-of-stat-overrides-subcls*:
⟦*G*⊢*Methd sig old member-of C*; *G*⊢*Methd sig new member-of D*;*G*⊢*D* $\preceq_C$ *C*;
G,*sig*⊢*new overrides*$_S$ *old*; *ws-prog G*⟧
⟹ *G*⊢*D* $\prec_C$ *C*
by (*drule stat-overrides-commonD*) (*auto intro*: *member-of-subcls*)

lemma *inherited-field-access*:
assumes *stat-acc*: *G*⊢*membr of statC accessible-from accC* **and**
is-field: *is-field membr* **and**
subclseq: *G* ⊢ *dynC* $\preceq_C$ *statC*
shows *G*⊢*membr in dynC dyn-accessible-from accC*
proof −
from *stat-acc is-field subclseq*
show *?thesis*
by (*auto dest*: *accessible-fieldD*
intro: *dyn-accessible-fromR.Immediate*
member-inI
permits-acc-inheritance)
qed

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

have *member-statC*: *G*⊢*m member-of statC*
  by (*auto dest*: *accessible-from-commonD*)
from *stat-acc*
show *?thesis*
proof (*cases*)
  case *Immediate*
  with *member-dynC member-statC subclseq dynC-acc*
  show *?thesis*
    by (*auto intro*: *accessible-fromR.Immediate permits-acc-inheritance*)
next
  case *Overriding*
  with *member-dynC subclseq dynC-acc*
  show *?thesis*
    by (*auto intro*: *accessible-fromR.Overriding rtrancl-trancl-trancl*)
qed
qed

fields and methods

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

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

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

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

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

only returns imethds if the interface is accessible

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

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

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

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

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

definition

```
dynmethd :: prog ⇒ qtname ⇒ qtname ⇒ (sig,qtname × methd) table where
dynmethd G statC dynC =
  (λsig.
    (if G⊢dynC ≼C statC
      then (case methd G statC sig of
              None ⇒ None
            | Some statM
                ⇒ (class-rec G dynC Map.empty
                     (λC c subcls-mthds.
                        subcls-mthds
                        ++
                        (filter-tab
                          (λ - dynM. G,sig⊢dynM overrides statM ∨ dynM=statM)
                          (methd G C) ))
                   ) sig
            )
      else None))
```

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

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

definition

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

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

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

definition

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

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

definition

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

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

definition
accfield :: *prog* ⇒ *qtname* ⇒ *qtname* ⇒ (*vname*, *qtname* × *field*) *table* **where**
accfield G S C =
(*let field-tab* = *table-of*((*map* (λ((*n*,*d*),*f*).(*n*,(*d*,*f*)))) (*fields G C*))
in filter-tab (λ*n* (*declC*,*f*). *G*⊢ (*declC*,*fdecl* (*n*,*f*)) *of C accessible-from S*)
field-tab)

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

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

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

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

lemma *efname-simp*[*simp*]:*efname* (*n*,*f*) = *n*
by (*simp add*: *efname-def*)

4 imethds

lemma *imethds-rec*: ⟦*iface G I* = *Some i*; *ws-prog G*⟧ ⟹
imethds G I = *Un-tables* ((λ*J. imethds G J*)‘*set* (*isuperIfs i*)) ⊕⊕
(*set-option* ∘ *table-of* (*map* (λ(*s*,*mh*). (*s*,*I*,*mh*)) (*imethods i*)))
apply (*unfold imethds-def*)
apply (*rule iface-rec* [*THEN trans*])
apply *auto*
done

lemma *imethds-norec*:
⟦*iface G md* = *Some i*; *ws-prog G*; *table-of* (*imethods i*) *sig* = *Some mh*⟧ ⟹
(*md*, *mh*) ∈ *imethds G md sig*
apply (*subst imethds-rec*)
apply *assumption+*
apply (*rule iffD2*)
apply (*rule overrides-t-Some-iff*)
apply (*rule disjI1*)
apply (*auto elim*: *table-of-map-SomeI*)
done

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*
apply (*erule rev-mp*)
apply (*rule ws-subint1-induct*, *assumption*, *assumption*)
apply (*subst imethds-rec*, *erule conjunct1*, *assumption*)
apply (*force elim*: *imethds-norec intro*: *rtrancl-into-rtrancl2*)
done

lemma *imethds-cases*:
assumes *im*: *im* ∈ *imethds G I sig*
and *ifI*: *iface G I* = *Some i*
and *ws*: *ws-prog G*
obtains (*NewMethod*) *table-of* (*map* (λ(*s*, *mh*). (*s*, *I*, *mh*)) (*imethods i*)) *sig* = *Some im*
| (*InheritedMethod*) *J* **where** *J* ∈ *set* (*isuperIfs i*) **and** *im* ∈ *imethds G J sig*
using *assms* **by** (*auto simp add*: *imethds-rec*)

5 accimethd

lemma *accimethds-simp* [*simp*]:
G⊢*Iface I accessible-in pack* ⟹ *accimethds G pack I* = *imethds G I*
by (*simp add*: *accimethds-def*)

lemma *accimethdsD*:
im ∈ *accimethds G pack I sig*
⟹ *im* ∈ *imethds G I sig* ∧ *G*⊢*Iface I accessible-in pack*
by (*auto simp add*: *accimethds-def*)

lemma *accimethdsI*:
⟦*im* ∈ *imethds G I sig*;*G*⊢*Iface I accessible-in pack*⟧
⟹ *im* ∈ *accimethds G pack I sig*
by *simp*

6 methd

lemma *methd-rec*: ⟦*class G C* = *Some c*; *ws-prog G*⟧ ⟹
methd G C
= (*if C* = *Object*
then Map.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*))
apply (*unfold methd-def*)
apply (*erule class-rec* [*THEN trans*], *assumption*)
apply (*simp*)
done

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*)
apply (*simp only*: *methd-rec*)
apply (*rule disjI1* [*THEN map-add-Some-iff* [*THEN iffD2*]])
apply (*auto elim*: *table-of-map-SomeI*)
done

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* $\preceq_C$ (*declclass m*) ∧ *methd G* (*declclass m*) *sig* = *Some m*
apply (*erule rev-mp*)
apply (*rule ws-subcls1-induct*, *assumption*, *assumption*)
apply (*subst methd-rec*, *assumption*)
apply (*case-tac Ca*=*Object*)
apply (*force elim*: *methd-norec*)

apply *simp*
apply (*case-tac table-of* (*map* (λ(*s*, *m*). (*s*, *Ca*, *m*)) (*methods c*)) *sig*)
apply (*force intro*: *rtrancl-into-rtrancl2*)

apply (*auto intro*: *methd-norec*)
done

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*)
by (*auto simp add*: *methd-rec*)

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*
by (*auto simp add*: *methd-rec*)

lemma *method-declared-inI*:
⟦*table-of* (*methods c*) *sig* = *Some m*; *class G C* = *Some c*⟧
⟹ *G*⊢*mdecl* (*sig*,*m*) *declared-in C*
by (*auto simp add*: *cdeclaredmethd-def declared-in-def*)

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*)
by (*auto dest*: *methd-declC method-declared-inI*)

lemma *member-methd*:
assumes *member-of*: *G*⊢*Methd sig m member-of C* **and**
ws: *ws-prog G*
shows *methd G C sig* = *Some m*
proof −
from *member-of*
have *iscls-C*: *is-class G C*
by (*rule member-of-is-classD*)
from *iscls-C ws member-of*
show *?thesis* (**is** *?Methd C*)
proof (*induct rule*: *ws-class-induct′*)
case (*Object co*)
assume *G* ⊢*Methd sig m member-of Object*

then have *G*⊢*Methd sig m declared-in Object* ∧ *declclass m* = *Object*
by (*cases set*: *members*) (*cases m*, *auto dest*: *subcls1D*)
with *ws Object*
show *?Methd Object*
by (*cases m*)
(*auto simp add*: *declared-in-def cdeclaredmethd-def methd-rec*
intro: *table-of-mapconst-SomeI*)
next
case (*Subcls C c*)
assume *clsC*: *class G C* = *Some c* **and**
neq-C-Obj: *C* ≠ *Object* **and**
hyp: *G* ⊢*Methd sig m member-of super c* ⟹ *?Methd* (*super c*) **and**
member-of: *G* ⊢*Methd sig m member-of C*
from *member-of*
show *?Methd C*
proof (*cases*)
case *Immediate*
with *clsC*
have *table-of* (*map* (λ(*s*, *m*). (*s*, *C*, *m*)) (*methods c*)) *sig* = *Some m*
by (*cases m*)
(*auto simp add*: *declared-in-def cdeclaredmethd-def*
intro: *table-of-mapconst-SomeI*)
with *clsC neq-C-Obj ws*
show *?thesis*
by (*simp add*: *methd-rec*)
next
case (*Inherited S*)
with *clsC*
have *undecl*: *G*⊢*mid sig undeclared-in C* **and**
super: *G* ⊢*Methd sig m member-of* (*super c*)
by (*auto dest*: *subcls1D*)
from *clsC undecl*
have *table-of* (*map* (λ(*s*, *m*). (*s*, *C*, *m*)) (*methods c*)) *sig* = *None*
by (*auto simp add*: *undeclared-in-def cdeclaredmethd-def*
intro: *table-of-mapconst-NoneI*)
moreover
from *Inherited* **have** *G* ⊢ *C inherits* (*method sig m*)
by (*auto simp add*: *inherits-def*)
moreover
note *clsC neq-C-Obj ws super hyp*
ultimately
show *?thesis*
by (*auto simp add*: *methd-rec intro*: *filter-tab-SomeI*)
qed
qed
qed

lemma *finite-methd*:*ws-prog G* ⟹ *finite* {*methd G C sig* |*sig C*. *is-class G C*}
apply (*rule finite-is-class* [*THEN finite-SetCompr2*])
apply (*intro strip*)
apply (*erule-tac ws-subcls1-induct*, *assumption*)
apply (*subst methd-rec*)
apply (*assumption*)
apply (*auto intro*!: *finite-range-map-of finite-range-filter-tab finite-range-map-of-map-add*)
done

lemma *finite-dom-methd*:
⟦*ws-prog G*; *is-class G C*⟧ ⟹ *finite* (*dom* (*methd G C*))
apply (*erule-tac ws-subcls1-induct*)
apply *assumption*
apply (*subst methd-rec*)
apply (*assumption*)
apply (*auto intro*!: *finite-dom-map-of finite-dom-filter-tab*)
done

7 accmethd

lemma *accmethd-SomeD*:
accmethd G S C sig = *Some m*
⟹ *methd G C sig* = *Some m* ∧ *G*⊢*method sig m of C accessible-from S*
by (*auto simp add*: *accmethd-def*)

lemma *accmethd-SomeI*:
⟦*methd G C sig* = *Some m*; *G*⊢*method sig m of C accessible-from S*⟧
⟹ *accmethd G S C sig* = *Some m*
by (*auto simp add*: *accmethd-def intro*: *filter-tab-SomeI*)

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⊢*C* $\preceq_C$ (*declclass m*) ∧ *methd G* (*declclass m*) *sig* = *Some m* ∧
G⊢*method sig m of C accessible-from S*
by (*auto dest*: *accmethd-SomeD methd-declC accmethd-SomeI*)

lemma *finite-dom-accmethd*:
⟦*ws-prog G*; *is-class G C*⟧ ⟹ *finite* (*dom* (*accmethd G S C*))
by (*auto simp add*: *accmethd-def intro*: *finite-dom-filter-tab finite-dom-methd*)

8 dynmethd

lemma *dynmethd-rec*:
⟦*class G dynC* = *Some c*; *ws-prog G*⟧ ⟹
dynmethd G statC dynC sig
= (*if G*⊢*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 −
assume *clsDynC*: *class G dynC* = *Some c* **and**
ws: *ws-prog G*
then show *?Dynmethd-def dynC sig* = *?Dynmethd-rec dynC c sig*

```
proof (induct rule: ws-class-induct'')
  case (Object co)
  show ?Dynmethd-def Object sig = ?Dynmethd-rec Object co sig
  proof (cases G⊢Object ≼C statC)
    case False
    then show ?thesis by (simp add: dynmethd-def)
  next
    case True
    then have eq-statC-Obj: statC = Object ..
    show ?thesis
    proof (cases methd G statC sig)
      case None then show ?thesis by (simp add: dynmethd-def)
    next
      case Some
      with True Object ws eq-statC-Obj
      show ?thesis
        by (auto simp add: dynmethd-def class-rec
                  intro: filter-tab-SomeI)
    qed
  qed
next
  case (Subcls dynC c sc)
  show ?Dynmethd-def dynC sig = ?Dynmethd-rec dynC c sig
  proof (cases G⊢dynC ≼C statC)
    case False
    then show ?thesis by (simp add: dynmethd-def)
  next
    case True
    note subclseq-dynC-statC = True
    show ?thesis
    proof (cases methd G statC sig)
      case None then show ?thesis by (simp add: dynmethd-def)
    next
      case (Some statM)
      note statM = Some
      let ?filter =
            λC. filter-tab
              (λ- dynM. G,sig ⊢ dynM overrides statM ∨ dynM = statM)
              (methd G C)
      let ?class-rec =
            λC. class-rec G C Map.empty
                        (λC c subcls-mthds. subcls-mthds ++ (?filter C))
      from statM Subcls ws subclseq-dynC-statC
      have dynmethd-dynC-def:
           ?Dynmethd-def dynC sig =
             ((?class-rec (super c))
              ++
             (?filter dynC)) sig
       by (simp (no-asm-simp) only: dynmethd-def class-rec)
          auto
      show ?thesis
      proof (cases dynC = statC)
        case True
        with subclseq-dynC-statC statM dynmethd-dynC-def
        have ?Dynmethd-def dynC sig = Some statM
          by (auto intro: map-add-find-right filter-tab-SomeI)
        with subclseq-dynC-statC True Some
        show ?thesis
          by auto
```

```
        next
          case False
          with subclseq-dynC-statC Subcls
          have subclseq-super-statC: G⊢(super c) ≼C statC
            by (blast dest: subclseq-superD)
          show ?thesis
          proof (cases methd G dynC sig)
            case None
            then have ?filter dynC sig = None
              by (rule filter-tab-None)
            then have ?Dynmethd-def dynC sig=?class-rec (super c) sig
              by (simp add: dynmethd-dynC-def)
            with  subclseq-super-statC statM None
            have ?Dynmethd-def dynC sig = ?Dynmethd-def (super c) sig
              by (auto simp add: empty-def dynmethd-def)
            with None subclseq-dynC-statC statM
            show ?thesis
              by simp
          next
            case (Some dynM)
            note dynM = Some
            let ?Termination = G ⊢ qmdecl sig dynM overrides qmdecl sig statM ∨
                                  dynM = statM
            show ?thesis
            proof (cases ?filter dynC sig)
              case None
              with dynM
              have no-termination: ¬ ?Termination
                by (simp add: filter-tab-def)
              from None
              have ?Dynmethd-def dynC sig=?class-rec (super c) sig
                by (simp add: dynmethd-dynC-def)
              with subclseq-super-statC statM dynM no-termination
              show ?thesis
                by (auto simp add: empty-def dynmethd-def)
            next
              case Some
              with dynM
              have termination: ?Termination
                by (auto)
              with Some dynM
              have ?Dynmethd-def dynC sig=Some dynM
               by (auto simp add: dynmethd-dynC-def)
              with subclseq-super-statC statM dynM termination
              show ?thesis
                by (auto simp add: dynmethd-def)
            qed
          qed
        qed
      qed
    qed
  qed
qed
```

```
lemma dynmethd-C-C:⟦is-class G C; ws-prog G⟧
⟹ dynmethd G C C sig = methd G C sig
apply (auto simp add: dynmethd-rec)
done
```

```
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)
by (auto simp add: dynmethd-rec)
```

```
lemma dynmethd-Some-cases:
  assumes dynM: dynmethd G statC dynC sig = Some dynM
    and is-cls-dynC: is-class G dynC
    and ws: ws-prog G
  obtains (Static) methd G statC sig = Some dynM
    | (Overrides) statM
      where methd G statC sig = Some statM
        and dynM ≠ statM
        and G,sig⊢dynM overrides statM
proof −
  from is-cls-dynC obtain dc where clsDynC: class G dynC = Some dc by blast
  from clsDynC ws dynM Static Overrides
  show ?thesis
  proof (induct rule: ws-class-induct)
    case (Object co)
    with ws  have statC = Object
      by (auto simp add: dynmethd-rec)
    with ws Object show ?thesis by (auto simp add: dynmethd-C-C)
  next
    case (Subcls C c)
    with ws show ?thesis
      by (auto simp add: dynmethd-rec)
  qed
qed
```

```
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 −
  from is-cls-dynC obtain dc where clsDynC: class G dynC = Some dc by blast
  from clsDynC ws dynM statM neq-dynM-statM
  show ?thesis (is ?P dynC)
  proof (induct rule: ws-class-induct)
    case (Object co)
    with ws  have statC = Object
      by (auto simp add: dynmethd-rec)
    with ws Object show ?P Object by (auto simp add: dynmethd-C-C)
  next
    case (Subcls dynC c)
    with ws show ?P dynC
      by (auto simp add: dynmethd-rec)
  qed
qed
```

lemma *dynmethd-Some-rec-cases*:
  assumes *dynM*: *dynmethd G statC dynC sig = Some dynM*
    and *clsDynC*: *class G dynC = Some c*
    and *ws*: *ws-prog G*
  obtains (*Static*) *methd G statC sig = Some dynM*
    | (*Override*) *statM* **where** *methd G statC sig = Some statM*
        and *methd G dynC sig = Some dynM* **and** $statM \neq dynM$
        and *G*,*sig*⊢ *dynM overrides statM*
    | (*Recursion*) $dynC \neq Object$ **and** *dynmethd G statC* (*super c*) *sig = Some dynM*
proof −
  from *clsDynC* **have** ∗: *is-class G dynC* **by** *simp*
  from *ws clsDynC dynM Static Override Recursion*
  show *?thesis*
    by (*auto simp add*: *dynmethd-rec dest*: *no-override-in-Object* [*OF dynM* ∗ *ws*])
qed

lemma *dynmethd-declC*:
⟦*dynmethd G statC dynC sig = Some m*;
  is-class G statC;*ws-prog G*
 ⟧ ⟹
  (∃ *d*. *class G* (*declclass m*)=*Some d* ∧ *table-of* (*methods d*) *sig*=*Some* (*mthd m*)) ∧
  G⊢*dynC* $\preceq_C$ (*declclass m*) ∧ *methd G* (*declclass m*) *sig = Some m*
proof −
  assume *is-cls-statC*: *is-class G statC*
  assume *ws*: *ws-prog G*
  assume *m*: *dynmethd G statC dynC sig = Some m*
  from *m*
  have *G*⊢*dynC* $\preceq_C$ *statC* **by** (*auto simp add*: *dynmethd-def*)
  from *this is-cls-statC*
  have *is-cls-dynC*: *is-class G dynC* **by** (*rule subcls-is-class2*)
  from *is-cls-dynC ws m*
  show *?thesis* (**is** *?P dynC*)
  proof (*induct rule*: *ws-class-induct′*)
    case (*Object co*)
    with *ws* **have** *statC*=*Object* **by** (*auto simp add*: *dynmethd-rec*)
    with *ws Object*
    show *?P Object*
      by (*auto simp add*: *dynmethd-C-C dest*: *methd-declC*)
  next
    case (*Subcls dynC c*)
    assume *hyp*: *dynmethd G statC* (*super c*) *sig = Some m* ⟹ *?P* (*super c*) **and**
          clsDynC: *class G dynC = Some c* **and**
              m′: *dynmethd G statC dynC sig = Some m* **and**
    neq-dynC-Obj: $dynC \neq Object$
    from *ws this* **obtain** *statM* **where**
      subclseq-dynC-statC: *G*⊢*dynC* $\preceq_C$ *statC* **and**
                   statM: *methd G statC sig = Some statM*
      by (*blast dest*: *dynmethdSomeD*)
    from *clsDynC neq-dynC-Obj*
    have *subclseq-dynC-super*: *G*⊢*dynC* $\preceq_C$ (*super c*)
      by (*auto intro*: *subcls1I*)
    from *m′ clsDynC ws*
    show *?P dynC*
    proof (*cases rule*: *dynmethd-Some-rec-cases*)
      case *Static*
      with *is-cls-statC ws subclseq-dynC-statC*
      show *?thesis*
        by (*auto intro*: *rtrancl-trans dest*: *methd-declC*)

```
    next
      case Override
      with clsDynC ws
      show ?thesis
        by (auto dest: methd-declC)
    next
      case Recursion
      with hyp subclseq-dynC-super
      show ?thesis
        by (auto intro: rtrancl-trans)
    qed
  qed
qed


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 −
  from subclseq is-cls-statC
  have is-cls-dynC: is-class G dynC by (rule subcls-is-class2)
  then obtain dc where
    clsDynC: class G dynC = Some dc by blast
  from clsDynC ws subclseq
  show ?thesis
  proof (induct rule: ws-class-induct)
    case (Object co)
    with ws  have statC = Object
      by (auto)
    with ws Object statM
    show ?P Object
      by (auto simp add: dynmethd-C-C)
  next
    case (Subcls dynC dc)
    assume clsDynC': class G dynC = Some dc
    assume neq-dynC-Obj: dynC ≠ Object
    assume hyp: G⊢super dc≼C statC ⟹ ?P (super dc)
    assume subclseq': G⊢dynC≼C statC
    then
    show ?P dynC
    proof (cases rule: subclseq-cases)
      case Eq
      with ws statM clsDynC'
      show ?thesis
        by (auto simp add: dynmethd-rec)
    next
      case Subcls
      assume G⊢dynC≺C statC
      from this clsDynC'
      have G⊢super dc≼C statC by (rule subcls-superD)
      with hyp ws clsDynC' subclseq' statM
      show ?thesis
        by (auto simp add: dynmethd-rec)
    qed
  qed
```

qed

lemma *dynmethd-cases*:
  assumes *statM*: *methd G statC sig = Some statM*
    and *subclseq*: $G \vdash dynC \preceq_C statC$
    and *is-cls-statC*: *is-class G statC*
    and *ws*: *ws-prog G*
  obtains (*Static*) *dynmethd G statC dynC sig = Some statM*
    | (*Overrides*) *dynM* **where** *dynmethd G statC dynC sig = Some dynM*
      and $dynM \neq statM$ **and** $G,sig \vdash dynM$ *overrides statM*
proof −
  note *hyp-static = Static* **and** *hyp-override = Overrides*
  from *subclseq is-cls-statC*
  have *is-cls-dynC*: *is-class G dynC* **by** (*rule subcls-is-class2*)
  then obtain *dc* **where**
    clsDynC: *class G dynC = Some dc* **by** *blast*
  from *statM subclseq is-cls-statC ws*
  obtain *dynM* **where** *dynM*: *dynmethd G statC dynC sig = Some dynM*
    by (*blast dest*: *methd-Some-dynmethd-Some*)
  from *dynM is-cls-dynC ws*
  show *?thesis*
  proof (*cases rule*: *dynmethd-Some-cases*)
    case *Static*
    with *hyp-static dynM statM* **show** *?thesis* **by** *simp*
  next
    case *Overrides*
    with *hyp-override dynM statM* **show** *?thesis* **by** *simp*
  qed
qed

lemma *ws-dynmethd*:
  assumes *statM*: *methd G statC sig = Some statM* **and**
      subclseq: $G \vdash dynC \preceq_C statC$ **and**
    is-cls-statC: *is-class G statC* **and**
        ws: *ws-prog G*
  shows
    $\exists$ *dynM*. *dynmethd G statC dynC sig = Some dynM* $\wedge$
        is-static dynM = is-static statM $\wedge$ $G \vdash resTy\ dynM \preceq resTy\ statM$
proof −
  from *statM subclseq is-cls-statC ws*
  show *?thesis*
  proof (*cases rule*: *dynmethd-cases*)
    case *Static*
    with *statM*
    show *?thesis*
      by *simp*
  next
    case *Overrides*
    with *ws*
    show *?thesis*
      by (*auto dest*: *ws-overrides-commonD*)
  qed
qed

9 dynlookup

lemma *dynlookup-cases*:

assumes *dynlookup G statT dynC sig* = *x*
obtains (*NullT*) *statT* = *NullT* **and** *Map.empty sig* = *x*
| (*IfaceT*) *I* **where** *statT* = *IfaceT I* **and** *dynimethd G I dynC sig* = *x*
| (*ClassT*) *statC* **where** *statT* = *ClassT statC* **and** *dynmethd G statC dynC sig* = *x*
| (*ArrayT*) *ty* **where** *statT* = *ArrayT ty* **and** *dynmethd G Object dynC sig* = *x*
using *assms* **by** (*cases statT*) (*auto simp add*: *dynlookup-def*)

10 fields

lemma *fields-rec*: ⟦*class G C* = *Some c*; *ws-prog G*⟧ ⟹
fields G C = *map* (λ(*fn*,*ft*). ((*fn*,*C*),*ft*)) (*cfields c*) @
(*if C* = *Object then* [] *else fields G* (*super c*))
apply (*simp only*: *fields-def*)
apply (*erule class-rec* [*THEN trans*])
apply *assumption*
apply *clarsimp*
done

lemma *fields-norec*:
⟦*class G fd* = *Some c*; *ws-prog G*; *table-of* (*cfields c*) *fn* = *Some f*⟧
⟹ *table-of* (*fields G fd*) (*fn*,*fd*) = *Some f*
apply (*subst fields-rec*)
apply *assumption*+
apply (*subst map-of-append*)
apply (*rule disjI1* [*THEN map-add-Some-iff* [*THEN iffD2*]])
apply (*auto elim*: *table-of-map2-SomeI*)
done

lemma *table-of-fieldsD*:
table-of (*map* (λ(*fn*,*ft*). ((*fn*,*C*),*ft*)) (*cfields c*)) *efn* = *Some f*
⟹ (*declclassf efn*) = *C* ∧ *table-of* (*cfields c*) (*fname efn*) = *Some f*
apply (*case-tac efn*)
by *auto*

lemma *fields-declC*:
⟦*table-of* (*fields G C*) *efn* = *Some f*; *ws-prog G*; *is-class G C*⟧ ⟹
(∃ *d*. *class G* (*declclassf efn*) = *Some d* ∧
table-of (*cfields d*) (*fname efn*)=*Some f*) ∧
G⊢*C* $\preceq_C$ (*declclassf efn*) ∧ *table-of* (*fields G* (*declclassf efn*)) *efn* = *Some f*
apply (*erule rev-mp*)
apply (*rule ws-subcls1-induct*, *assumption*, *assumption*)
apply (*subst fields-rec*, *assumption*)
apply *clarify*
apply (*simp only*: *map-of-append*)
apply (*case-tac table-of* (*map* (*case-prod* (λ*fn*. *Pair* (*fn*, *Ca*))) (*cfields c*)) *efn*)
apply (*force intro*:*rtrancl-into-rtrancl2 simp add*: *map-add-def*)

apply (*frule-tac fd*=*Ca* **in** *fields-norec*)
apply *assumption*
apply *blast*
apply (*frule table-of-fieldsD*)
apply (*frule-tac n*=*table-of* (*map* (*case-prod* (λ*fn*. *Pair* (*fn*, *Ca*))) (*cfields c*))
and *m*=*table-of* (*if Ca* = *Object then* [] *else fields G* (*super c*))
in *map-add-find-right*)

apply (*case-tac efn*)
apply (*simp*)
done

lemma *fields-emptyI*: $\bigwedge y$. $[\![$*ws-prog G*; *class G C* = *Some c*;*cfields c* = []; *C* $\neq$ *Object* $\longrightarrow$ *class G* (*super c*) = *Some y* $\wedge$ *fields G* (*super c*) = []$]\!]$ $\Longrightarrow$ *fields G C* = []
apply (*subst fields-rec*)
apply *assumption*
apply *auto*
done

lemma *fields-mono-lemma*:
$[\![$*x* $\in$ *set* (*fields G C*); *G*$\vdash$*D* $\preceq_C$ *C*; *ws-prog G*$]\!]$ $\Longrightarrow$ *x* $\in$ *set* (*fields G D*)
apply (*erule rev-mp*)
apply (*erule converse-rtrancl-induct*)
apply *fast*
apply (*drule subcls1D*)
apply *clarsimp*
apply (*subst fields-rec*)
apply *auto*
done

lemma *ws-unique-fields-lemma*:
$[\![$(*efn*,*fd*) $\in$ *set* (*fields G* (*super c*)); *fc* $\in$ *set* (*cfields c*); *ws-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*
apply (*frule-tac ws-prog-cdeclD* [*THEN conjunct2*], *assumption*, *assumption*)
apply (*drule-tac weak-map-of-SomeI*)
apply (*frule-tac subcls1I* [*THEN subcls1-irrefl*], *assumption*, *assumption*)
apply (*auto dest*: *fields-declC* [*THEN conjunct2* [*THEN conjunct1*[*THEN rtranclD*]]])
done

lemma *ws-unique-fields*: $[\![$*is-class G C*; *ws-prog G*; $\bigwedge$*C c*. $[\![$*class G C* = *Some c*$]\!]$ $\Longrightarrow$ *unique* (*cfields c*) $]\!]$ $\Longrightarrow$ *unique* (*fields G C*)
apply (*rule ws-subcls1-induct*, *assumption*, *assumption*)
apply (*subst fields-rec*, *assumption*)
apply (*auto intro*!: *unique-map-inj inj-onI*
elim!: *unique-append ws-unique-fields-lemma fields-norec*)
done

11 accfield

lemma *accfield-fields*:
accfield G S C fn = *Some f*
$\Longrightarrow$ *table-of* (*fields G C*) (*fn*, *declclass f*) = *Some* (*fld f*)
apply (*simp only*: *accfield-def Let-def*)
apply (*rule table-of-remap-SomeD*)
apply *auto*
done

lemma *accfield-declC-is-class*:
$[\![$*is-class G C*; *accfield G S C en* = *Some* (*fd*, *f*); *ws-prog G*$]\!] \Longrightarrow$
is-class G fd
apply (*drule accfield-fields*)
apply (*drule fields-declC* [*THEN conjunct1*], *assumption*)
apply *auto*
done

lemma *accfield-accessibleD*:
accfield G S C fn = *Some f* $\Longrightarrow$ *G*⊢*Field fn f of C accessible-from S*
by (*auto simp add*: *accfield-def Let-def*)

12 is methd

lemma *is-methdI*:
$[\![$*class G C* = *Some y*; *methd G C sig* = *Some b*$]\!] \Longrightarrow$ *is-methd G C sig*
apply (*unfold is-methd-def*)
apply *auto*
done

lemma *is-methdD*:
is-methd G C sig $\Longrightarrow$ *class G C* $\neq$ *None* $\wedge$ *methd G C sig* $\neq$ *None*
apply (*unfold is-methd-def*)
apply *auto*
done

lemma *finite-is-methd*:
ws-prog G $\Longrightarrow$ *finite* (*Collect* (*case-prod* (*is-methd G*)))
apply (*unfold is-methd-def*)
apply (*subst Collect-case-prod-Sigma*)
apply (*rule finite-is-class* [*THEN finite-SigmaI*])
apply (*simp only*: *mem-Collect-eq*)
apply (*fold dom-def*)
apply (*erule finite-dom-methd*)
apply *assumption*
done

calculation of the superclasses of a class

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

lemma *superclasses-rec*: $[\![$*class G C* = *Some c*; *ws-prog G*$]\!] \Longrightarrow$
superclasses G C
= (*if* (*C*=*Object*)
then {}
else insert (*super c*) (*superclasses G* (*super c*)))
apply (*unfold superclasses-def*)
apply (*erule class-rec* [*THEN trans*], *assumption*)

```
apply (simp)
done


lemma superclasses-mono:
  assumes clsrel: G⊢C≺C D
  and ws: ws-prog G
  and cls-C: class G C = Some c
  and wf: ⋀C c. ⟦class G C = Some c; C ≠ Object⟧
    ⟹ ∃ sc. class G (super c) = Some sc
  and x: x∈superclasses G D
  shows x∈superclasses G C using clsrel cls-C x
proof (induct arbitrary: c rule: converse-trancl-induct)
  case (base C)
  with wf ws show ?case
    by (auto    intro: no-subcls1-Object
             simp add: superclasses-rec subcls1-def)
next
  case (step C S)
  moreover note wf ws
  moreover from calculation
  have x∈superclasses G S
    by (force intro: no-subcls1-Object simp add: subcls1-def)
  moreover from calculation
  have super c = S
    by (auto intro: no-subcls1-Object simp add: subcls1-def)
  ultimately show ?case
    by (auto intro: no-subcls1-Object simp add: superclasses-rec)
qed


lemma subclsEval:
  assumes clsrel: G⊢C≺C D
  and ws: ws-prog G
  and cls-C: class G C = Some c
  and wf: ⋀C c. ⟦class G C = Some c; C ≠ Object⟧
    ⟹ ∃ sc. class G (super c) = Some sc
  shows D∈superclasses G C using clsrel cls-C
proof (induct arbitrary: c rule: converse-trancl-induct)
  case (base C)
  with ws wf show ?case
    by (auto intro: no-subcls1-Object simp add: superclasses-rec subcls1-def)
next
  case (step C S)
  with ws wf show ?case
    by − (rule superclasses-mono,
          auto dest: no-subcls1-Object simp add: subcls1-def )
qed

end
```

Chapter 11

WellType

1 Well-typedness of Java programs

theory *WellType*
imports *DeclConcepts*
begin

improvements over Java Specification 1.0:

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

simplifications:

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

design issues:

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

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

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

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

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

Static overloading: maximally specific methods

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

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

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

lemma *declrefT-simp*[*simp*]:*declrefT* (*r*,*m*) = *r*
by (*simp add*: *declrefT-def*)

lemma *mhd-simp*[*simp*]:*mhd* (*r*,*m*) = *m*
by (*simp add*: *mhd-def*)

lemma *static-mhd-simp*[*simp*]: *static* (*mhd m*) = *is-static m*
by (*cases m*) (*simp add*: *member-is-static-simp mhd-def*)

lemma *mhd-resTy-simp* [*simp*]: *resTy* (*mhd m*) = *resTy m*
by (*cases m*) *simp*

lemma *mhd-is-static-simp* [*simp*]: *is-static* (*mhd m*) = *is-static m*
by (*cases m*) *simp*

lemma *mhd-accmodi-simp* [*simp*]: *accmodi* (*mhd m*) = *accmodi m*
by (*cases m*) *simp*

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

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

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

primrec *mheads* :: *prog* ⇒ *qtname* ⇒ *ref-ty* ⇒ *sig* ⇒ *emhead set*
where

$mheads\ G\ S\ \ NullT\ \ \ \ \ = (\lambda sig.\ \{\})$
$|\ mheads\ G\ S\ (IfaceT\ I) = (\lambda sig.\ (\lambda(I,h).(IfaceT\ I,h))$
$\quad ` accimethds\ G\ (pid\ S)\ I\ sig \cup$
$\quad accObjectmheads\ G\ S\ (IfaceT\ I)\ sig)$
$|\ mheads\ G\ S\ (ClassT\ C) = cmheads\ G\ S\ C$
$|\ mheads\ G\ S\ (ArrayT\ T) = accObjectmheads\ G\ S\ (ArrayT\ T)$

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

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

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

lemma *max-spec2appl-meths*:
$x \in max\text{-}spec\ G\ S\ T\ sig \Longrightarrow x \in appl\text{-}methds\ G\ S\ T\ sig$
by (*auto simp*: *max-spec-def*)

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

lemma *max-spec2mheads*:
$max\text{-}spec\ G\ S\ rt\ (|name=mn,parTs=pTs|) = insert\ (mh,\ pTs')\ A$
$\Longrightarrow mh \in mheads\ G\ S\ rt\ (|name=mn,parTs=pTs'|) \wedge G\vdash pTs[\preceq]pTs'$
apply (*auto dest*: *equalityD2 subsetD max-spec2appl-meths appl-methsD*)
done

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

definition
invmode :: ('*a*::*type*)*member-scheme* ⇒ *expr* ⇒ *inv-mode* **where**
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*
apply (*unfold invmode-def*)

apply (*simp* (*no-asm*) *add*: *member-is-static-simp*)
done

lemma *invmode-Static-eq* [*simp*]: (*invmode m e* = *Static*) = *is-static m*
apply (*unfold invmode-def*)
apply (*simp* (*no-asm*))
done

lemma *invmode-IntVir-eq*: (*invmode m e* = *IntVir*) = (¬(*is-static m*) ∧ *e*≠*Super*)
apply (*unfold invmode-def*)
apply (*simp* (*no-asm*))
done

lemma *Null-staticD*:
a'=*Null* ⟶ (*is-static m*) ⟹ *invmode m e* = *IntVir* ⟶ $a' \neq$ *Null*
apply (*clarsimp simp add*: *invmode-IntVir-eq*)
done

Typing for unary operations

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

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

Typing for binary operations

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

| binop-type BitXor = Integer
| binop-type Xor = Boolean
| binop-type BitOr = Integer
| binop-type Or = Boolean
| binop-type CondAnd = Boolean
| binop-type CondOr = Boolean

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

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

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

E,dt⊨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*: $E,dt\models c::\surd \Longrightarrow$
$E,dt\models l\bullet\ c::\surd$

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

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

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

— cf. 14.16
| *Throw*: $[\![E,dt\models e::-Class\ tn;$
$prg\ E\vdash tn\preceq_C\ SXcpt\ Throwable]\!] \Longrightarrow$
$E,dt\models Throw\ e::\surd$
— cf. 14.18
| *Try*: $[\![E,dt\models c1::\surd;\ prg\ E\vdash tn\preceq_C\ SXcpt\ Throwable;$
$lcl\ E\ (VName\ vn)=None;\ E\ (\!|lcl := (lcl\ E)(VName\ vn\mapsto Class\ tn)|\!),dt\models c2::\surd]\!]$
$\Longrightarrow$
$E,dt\models Try\ c1\ Catch(tn\ vn)\ c2::\surd$

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

| *Init*: $[\![is\text{-}class\ (prg\ E)\ C]\!] \Longrightarrow$
$E,dt\models Init\ C::\surd$
— *Init* is created on the fly during evaluation (see Eval.thy). The class isn't necessarily accessible from the points *Init* is called. Therefor we only demand *is-class* and not *is-acc-class* here.

— well-typed expressions

— cf. 15.8
| *NewC*: $[\![is\text{-}acc\text{-}class\ (prg\ E)\ (pkg\ E)\ C]\!] \Longrightarrow$
$E,dt\models NewC\ C::-Class\ C$
— cf. 15.9
| *NewA*: $[\![is\text{-}acc\text{-}type\ (prg\ E)\ (pkg\ E)\ T;$
$E,dt\models i::-PrimT\ Integer]\!] \Longrightarrow$
$E,dt\models New\ T[i]::-T.[]$

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

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

— cf. 15.7.1
| *Lit*: ⟦*typeof dt x = Some T*⟧ ⟹
$E,dt \models Lit\ x{::}{-}T$

| *UnOp*: ⟦$E,dt \models e{::}{-}Te$; *wt-unop unop Te*; $T{=}PrimT$ (*unop-type unop*)⟧
⟹
$E,dt \models UnOp\ unop\ e{::}{-}T$

| *BinOp*: ⟦$E,dt \models e1{::}{-}T1$; $E,dt \models e2{::}{-}T2$; *wt-binop* (*prg E*) *binop T1 T2*;
$T{=}PrimT$ (*binop-type binop*)⟧
⟹
$E,dt \models BinOp\ binop\ e1\ e2{::}{-}T$

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

— cf. 15.13.1, 15.10.1, 15.12
| *Acc*: ⟦$E,dt \models va{::}{=}T$⟧ ⟹
$E,dt \models Acc\ va{::}{-}T$

— cf. 15.25, 15.25.1
| *Ass*: ⟦$E,dt \models va{::}{=}T$; $va \neq LVar\ This$;
$E,dt \models v\ {::}{-}T'$;
$prg\ E \vdash T' \preceq T$⟧ ⟹
$E,dt \models va{:=}v{::}{-}T'$

— cf. 15.24
| *Cond*: ⟦$E,dt \models e0{::}{-}PrimT\ Boolean$;
$E,dt \models e1{::}{-}T1$; $E,dt \models e2{::}{-}T2$;
$prg\ E \vdash T1 \preceq T2 \wedge T = T2 \vee prg\ E \vdash T2 \preceq T1 \wedge T = T1$⟧ ⟹
$E,dt \models e0\ ?\ e1 : e2{::}{-}T$

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

| *Methd*: ⟦*is-class* (*prg E*) *C*;
methd (*prg E*) *C sig = Some m*;
$E,dt \models Body$ (*declclass m*) (*stmt* (*mbody* (*mthd m*))) $::-T$⟧ ⟹
$E,dt \models Methd\ C\ sig{::}{-}T$
— The class *C* is the dynamic class of the method call (cf. Eval.thy). It hasn't got to be directly accessible from the current package *pkg E*. Only the static class must be accessible (enshured indirectly by *Call*). Note that l is just a dummy value. It is only used in the smallstep semantics. To proof typesafety directly for the smallstep semantics we would have to assume conformance of l here!

| *Body*: ⟦*is-class* (*prg E*) *D*;
$E,dt \models blk{::}\surd$;
(*lcl E*) *Result = Some T*;
is-type (*prg E*) *T*⟧ ⟹
$E,dt \models Body\ D\ blk{::}{-}T$
— The class *D* implementing the method must not directly be accessible from the current package *pkg E*, but can also be indirectly accessible due to inheritance (enshured in *Call*) The result type hasn't got to be accessible in Java! (If it is not accessible you can only assign it to Object). For dummy value l see rule *Methd*.

— well-typed variables

— cf. 15.13.1
| *LVar*: ⟦*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*

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

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

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

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

abbreviation
ty-exprs-syntax :: *env* ⇒ [*expr list*, *ty list*] ⇒ *bool* (-⊢-::≐- [*51*,*51*,*51*] *50*)
where *E*⊢*e*::≐*T* == *E*⊢*In3 e* :: *Inr T*

notation (*ASCII*)
wt-syntax (-|−-::- [*51*,*51*,*51*] *50*) **and**
wt-stmt-syntax (-|−-:<> [*51*,*51*] *50*) **and**
ty-expr-syntax (-|−-:−- [*51*,*51*,*51*] *50*) **and**
ty-var-syntax (-|−-:=- [*51*,*51*,*51*] *50*) **and**
ty-exprs-syntax (-|−-:#- [*51*,*51*,*51*] *50*)

declare *not-None-eq* [*simp del*]
declare *if-split* [*split del*] *if-split-asm* [*split del*]
declare *split-paired-All* [*simp del*] *split-paired-Ex* [*simp del*]
setup ‹*map-theory-simpset* (*fn ctxt* => *ctxt delloop split-all-tac*)›

inductive-cases *wt-elim-cases* [*cases set*]:
E,*dt*⊨*In2* (*LVar vn*) ::*T*
E,*dt*⊨*In2* ({*accC*,*statDeclC*,*s*}*e*..*fn*)::*T*
E,*dt*⊨*In2* (*e*.[*i*]) ::*T*
E,*dt*⊨*In1l* (*NewC C*) ::*T*
E,*dt*⊨*In1l* (*New T'*[*i*]) ::*T*
E,*dt*⊨*In1l* (*Cast T' e*) ::*T*
E,*dt*⊨*In1l* (*e InstOf T'*) ::*T*
E,*dt*⊨*In1l* (*Lit x*) ::*T*
E,*dt*⊨*In1l* (*UnOp unop e*) ::*T*
E,*dt*⊨*In1l* (*BinOp binop e1 e2*) ::*T*
E,*dt*⊨*In1l* (*Super*) ::*T*
E,*dt*⊨*In1l* (*Acc va*) ::*T*
E,*dt*⊨*In1l* (*Ass va v*) ::*T*
E,*dt*⊨*In1l* (*e0 ? e1 : e2*) ::*T*
E,*dt*⊨*In1l* ({*accC*,*statT*,*mode*}*e*·*mn*({*pT'*}*p*))::*T*
E,*dt*⊨*In1l* (*Methd C sig*) ::*T*
E,*dt*⊨*In1l* (*Body D blk*) ::*T*
E,*dt*⊨*In3* ([]) ::*Ts*
E,*dt*⊨*In3* (*e*#*es*) ::*Ts*
E,*dt*⊨*In1r* *Skip* ::*x*
E,*dt*⊨*In1r* (*Expr e*) ::*x*
E,*dt*⊨*In1r* (*c1*;; *c2*) ::*x*
E,*dt*⊨*In1r* (*l*· *c*) ::*x*
E,*dt*⊨*In1r* (*If*(*e*) *c1 Else c2*) ::*x*
E,*dt*⊨*In1r* (*l*· *While*(*e*) *c*) ::*x*
E,*dt*⊨*In1r* (*Jmp jump*) ::*x*
E,*dt*⊨*In1r* (*Throw e*) ::*x*
E,*dt*⊨*In1r* (*Try c1 Catch*(*tn vn*) *c2*)::*x*
E,*dt*⊨*In1r* (*c1 Finally c2*) ::*x*
E,*dt*⊨*In1r* (*Init C*) ::*x*
declare *not-None-eq* [*simp*]
declare *if-split* [*split*] *if-split-asm* [*split*]
declare *split-paired-All* [*simp*] *split-paired-Ex* [*simp*]
setup ‹*map-theory-simpset* (*fn ctxt* => *ctxt addloop* (*split-all-tac*, *split-all-tac*))›

lemma *is-acc-class-is-accessible*:
is-acc-class G P C ⟹ *G*⊢(*Class C*) *accessible-in P*
by (*auto simp add*: *is-acc-class-def*)

lemma *is-acc-iface-is-iface*: *is-acc-iface G P I* ⟹ *is-iface G I*
by (*auto simp add*: *is-acc-iface-def*)

lemma *is-acc-iface-Iface-is-accessible*:
is-acc-iface G P I ⟹ *G*⊢(*Iface I*) *accessible-in P*
by (*auto simp add*: *is-acc-iface-def*)

lemma *is-acc-type-is-type*: *is-acc-type G P T* ⟹ *is-type G T*
by (*auto simp add*: *is-acc-type-def*)

lemma *is-acc-iface-is-accessible*:
is-acc-type G P T ⟹ *G*⊢*T accessible-in P*
by (*auto simp add*: *is-acc-type-def*)

lemma *wt-Methd-is-methd*:
E⊢*In1l* (*Methd C sig*)::*T* ⟹ *is-methd* (*prg E*) *C sig*
apply (*erule-tac wt-elim-cases*)
apply *clarsimp*
apply (*erule is-methdI*, *assumption*)
done

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

lemma *wt-Call*:
⟦*E*,*dt*⊨*e*::−*RefT statT*; *E*,*dt*⊨*ps*::≐*pTs*;
max-spec (*prg E*) (*cls E*) *statT* (|*name*=*mn*,*parTs*=*pTs*|)
= {((*statDeclC*,*m*),*pTs'*)};*rT*=(*resTy m*);*accC*=*cls E*;
mode = *invmode m e*⟧ ⟹ *E*,*dt*⊨{*accC*,*statT*,*mode*}*e*·*mn*({*pTs'*}*ps*)::−*rT*
by (*auto elim*: *wt.Call*)

lemma *invocationTypeExpr-noClassD*:
⟦ *E*⊢*e*::−*RefT statT*⟧
⟹ (∀ *statC*. *statT* ≠ *ClassT statC*) ⟶ *invmode m e* ≠ *SuperM*
proof −
assume *wt*: *E*⊢*e*::−*RefT statT*
show *?thesis*
proof (*cases e*=*Super*)
case *True*
with *wt* **obtain** *C* **where** *statT* = *ClassT C* **by** (*blast elim*: *wt-elim-cases*)
then show *?thesis* **by** *blast*
next
case *False* **then show** *?thesis*
by (*auto simp add*: *invmode-def*)
qed
qed

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*
by (*auto elim*: *wt.Super*)

lemma *wt-FVar*:
⟦*E*,*dt*⊨*e*::−*Class C*; *accfield* (*prg E*) (*cls E*) *C fn* = *Some* (*statDeclC*,*f*);
sf=*is-static f*; *fT*=(*type f*); *accC*=*cls E*⟧
⟹ *E*,*dt*⊨{*accC*,*statDeclC*,*sf*}*e*..*fn*::=*fT*
by (*auto dest*: *wt.FVar*)

lemma *wt-init* [*iff*]: *E*,*dt*⊨*Init C*::√ = *is-class* (*prg E*) *C*
by (*auto elim*: *wt-elim-cases intro*: *wt.Init*)

declare *wt.Skip* [*iff*]

lemma *wt-StatRef*:
is-acc-type (*prg E*) (*pkg E*) (*RefT rt*) ⟹ *E*⊢*StatRef rt*::−*RefT rt*

apply (*rule wt.Cast*)
apply (*rule wt.Lit*)
apply (*simp* (*no-asm*))
apply (*simp* (*no-asm-simp*))
apply (*rule cast.widen*)
apply (*simp* (*no-asm*))
done

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

— In the special syntax to distinguish the typing judgements for expressions, statements, variables and expression lists the kind of term corresponds to the kind of type in the end e.g. An statement (injection *In3* into terms, always has type void (injection *Inl* into the generalised types. The following simplification procedures establish these kinds of correlation.

lemma *wt-expr-eq*: $E,dt \models In1l\ t::U = (\exists\ T.\ U{=}Inl\ T \wedge E,dt \models t::-T)$
by (*auto*, *frule wt-Inj-elim*, *auto*)

lemma *wt-var-eq*: $E,dt \models In2\ t::U = (\exists\ T.\ U{=}Inl\ T \wedge E,dt \models t::{=}T)$
by (*auto*, *frule wt-Inj-elim*, *auto*)

lemma *wt-exprs-eq*: $E,dt \models In3\ t::U = (\exists\ Ts.\ U{=}Inr\ Ts \wedge E,dt \models t::\doteq Ts)$
by (*auto*, *frule wt-Inj-elim*, *auto*)

lemma *wt-stmt-eq*: $E,dt \models In1r\ t::U = (U{=}Inl(PrimT\ Void) \wedge E,dt \models t::\surd)$
by (*auto*, *frule wt-Inj-elim*, *auto*, *frule wt-Inj-elim*, *auto*)

simproc-setup *wt-expr* ($E,dt \models In1l\ t::U$) = ‹
K (*K* (*fn ct =>*
(*case Thm.term-of ct of*
(- $ - $ - $ - $ (*Const* - $ -)) => *NONE*
| - => *SOME* (*mk-meta-eq* @{*thm wt-expr-eq*}))))›

simproc-setup *wt-var* ($E,dt \models In2\ t::U$) = ‹
K (*K* (*fn ct =>*
(*case Thm.term-of ct of*
(- $ - $ - $ - $ (*Const* - $ -)) => *NONE*
| - => *SOME* (*mk-meta-eq* @{*thm wt-var-eq*}))))›

simproc-setup *wt-exprs* ($E,dt \models In3\ t::U$) = ‹
K (*K* (*fn ct =>*
(*case Thm.term-of ct of*
(- $ - $ - $ - $ (*Const* - $ -)) => *NONE*
| - => *SOME* (*mk-meta-eq* @{*thm wt-exprs-eq*}))))›

simproc-setup *wt-stmt* (*E*,*dt*⊨*In1r t*::*U*) = ‹
  K (*K* (*fn ct* =>
    (*case Thm.term-of ct of*
      (- $ - $ - $ - $ (*Const* - $ -)) => *NONE*
    | - => *SOME* (*mk-meta-eq* @{*thm wt-stmt-eq*}))))›

lemma *wt-elim-BinOp*:
  ⟦*E*,*dt*⊨*In1l* (*BinOp binop e1 e2*)::*T*;
    ⋀*T1 T2 T3*.
      ⟦*E*,*dt*⊨*e1*::−*T1*; *E*,*dt*⊨*e2*::−*T2*; *wt-binop* (*prg E*) *binop T1 T2*;
        E,*dt*⊨(*if b then In1l e2 else In1r Skip*)::*T3*;
        T = *Inl* (*PrimT* (*binop-type binop*))⟧
      ⟹ *P*⟧
⟹ *P*
apply (*erule wt-elim-cases*)
apply (*cases b*)
apply *auto*
done

lemma *Inj-eq-lemma* [*simp*]:
  (∀ *T*. (∃ *T′*. *T* = *Inj T′* ∧ *P T′*) ⟶ *Q T*) = (∀ *T′*. *P T′* ⟶ *Q* (*Inj T′*))
by *auto*

lemma *single-valued-tys-lemma* [*rule-format* (*no-asm*)]:
  ∀ *S T*. *G*⊢*S*≼*T* ⟶ *G*⊢*T*≼*S* ⟶ *S* = *T* ⟹ *E*,*dt*⊨*t*::*T* ⟹
    G = *prg E* ⟶ (∀ *T′*. *E*,*dt*⊨*t*::*T′* ⟶ *T* = *T′*)
apply (*cases E*, *erule wt.induct*)
apply (*safe del*: *disjE*)
apply (*simp-all* (*no-asm-use*) *split del*: *if-split-asm*)
apply (*safe del*: *disjE*)

apply (*tactic* ‹*ALLGOALS* (*fn i* =>
  if i = *11 then EVERY′*
   [*Rule-Insts.thin-tac* ***context*** *E*,*dt*⊨*e0*::−*PrimT Boolean* [(***binding***‹*E*›, *NONE*, *NoSyn*)],
     Rule-Insts.thin-tac ***context*** *E*,*dt*⊨*e1*::−*T1* [(***binding***‹*E*›, *NONE*, *NoSyn*), (***binding***‹*T1*›, *NONE*, *NoSyn*)],
     Rule-Insts.thin-tac ***context*** *E*,*dt*⊨*e2*::−*T2* [(***binding***‹*E*›, *NONE*, *NoSyn*), (***binding***‹*T2*›, *NONE*, *NoSyn*)]] *i*
  else Rule-Insts.thin-tac ***context*** *All P* [(***binding***‹*P*›, *NONE*, *NoSyn*)] *i*)›)

apply (*tactic* ‹*ALLGOALS* (*eresolve-tac* ***context*** @{*thms wt-elim-cases*})›)
apply (*simp-all* (*no-asm-use*) *split del*: *if-split-asm*)
apply (*erule-tac* [*12*] *V* = *All P* **for** *P* **in** *thin-rl*)
apply (*blast del*: *equalityCE dest*: *sym* [*THEN trans*])+
done

lemma *single-valued-tys*:
ws-prog (*prg E*) ⟹ *single-valued* {(*t*,*T*). *E*,*dt*⊨*t*::*T*}
apply (*unfold single-valued-def*)
apply *clarsimp*
apply (*rule single-valued-tys-lemma*)
apply (*auto intro*!: *widen-antisym*)
done

lemma *typeof-empty-is-type*: $typeof\ (\lambda a.\ None)\ v = Some\ T \Longrightarrow is\text{-}type\ G\ T$
by (*induct v*) *auto*

lemma *typeof-is-type*: $(\forall\, a.\ v \neq Addr\ a) \Longrightarrow \exists\, T.\ typeof\ dt\ v = Some\ T \wedge is\text{-}type\ G\ T$
by (*induct v*) *auto*

end

Chapter 12

DefiniteAssignment

1 Definite Assignment

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

Definite Assignment Analysis (cf. 16)

The definite assignment analysis approximates the sets of local variables that will be assigned at a certain point of evaluation, and ensures that we will only read variables which previously were assigned. It should conform to the following idea: If the evaluation of a term completes normally (no abruption (exception, break, continue, return) appeared) , the set of local variables calculated by the analysis is a subset of the variables that were actually assigned during evaluation.

To get more precise information about the sets of assigned variables the analysis includes the following optimisations:

- Inside of a while loop we also take care of the variables assigned before break statements, since the break causes the while loop to continue normally.
- For conditional statements we take care of constant conditions to statically determine the path of evaluation.
- Inside a distinct path of a conditional statements we know to which boolean value the condition has evaluated to, and so can retrieve more information about the variables assigned during evaluation of the boolean condition.

Since in our model of Java the return values of methods are stored in a local variable we also ensure that every path of (normal) evaluation will assign the result variable, or in the sense of real Java every path ends up in and return instruction.

Not covered yet:

- analysis of definite unassigned
- special treatment of final fields

Correct nesting of jump statements

For definite assignment it becomes crucial, that jumps (break, continue, return) are nested correctly i.e. a continue jump is nested in a matching while statement, a break jump is nested in a proper label statement, a class initialiser does not terminate abruptly with a return. With this we can for example ensure that evaluation of an expression will never end up with a jump, since no breaks, continues or returns are allowed in an expression.

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

where
jumpNestingOkS jmps (*Skip*) = *True*
| *jumpNestingOkS jmps* (*Expr e*) = *True*
| *jumpNestingOkS jmps* (*j*· *s*) = *jumpNestingOkS* ({*j*} ∪ *jmps*) *s*
| *jumpNestingOkS jmps* (*c1*;;*c2*) = (*jumpNestingOkS jmps c1* ∧
jumpNestingOkS jmps c2)
| *jumpNestingOkS jmps* (*If*(*e*) *c1 Else c2*) = (*jumpNestingOkS jmps c1* ∧
jumpNestingOkS jmps c2)
| *jumpNestingOkS jmps* (*l*· *While*(*e*) *c*) = *jumpNestingOkS* ({*Cont l*} ∪ *jmps*) *c*
— The label of the while loop only handles continue jumps. Breaks are only handled by *Lab*
| *jumpNestingOkS jmps* (*Jmp j*) = (*j* ∈ *jmps*)
| *jumpNestingOkS jmps* (*Throw e*) = *True*
| *jumpNestingOkS jmps* (*Try c1 Catch*(*C vn*) *c2*) = (*jumpNestingOkS jmps c1* ∧
jumpNestingOkS jmps c2)
| *jumpNestingOkS jmps* (*c1 Finally c2*) = (*jumpNestingOkS jmps c1* ∧
jumpNestingOkS jmps c2)
| *jumpNestingOkS jmps* (*Init C*) = *True*
— wellformedness of the program must enshure that for all initializers jumpNestingOkS holds
— Dummy analysis for intermediate smallstep term *FinA*
| *jumpNestingOkS jmps* (*FinA a c*) = *False*

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

lemma *jumpNestingOk-expr-simp* [*simp*]: *jumpNestingOk jmps* (*In1l e*) = *True*
by (*simp add*: *jumpNestingOk-def*)

lemma *jumpNestingOk-expr-simp1* [*simp*]: *jumpNestingOk jmps* ⟨*e*::*expr*⟩ = *True*
by (*simp add*: *inj-term-simps*)

lemma *jumpNestingOk-stmt-simp* [*simp*]:
jumpNestingOk jmps (*In1r s*) = *jumpNestingOkS jmps s*
by (*simp add*: *jumpNestingOk-def*)

lemma *jumpNestingOk-stmt-simp1* [*simp*]:
jumpNestingOk jmps ⟨*s*::*stmt*⟩ = *jumpNestingOkS jmps s*
by (*simp add*: *inj-term-simps*)

lemma *jumpNestingOk-var-simp* [*simp*]: *jumpNestingOk jmps* (*In2 v*) = *True*
by (*simp add*: *jumpNestingOk-def*)

lemma *jumpNestingOk-var-simp1* [*simp*]: *jumpNestingOk jmps* ⟨*v*::*var*⟩ = *True*
by (*simp add*: *inj-term-simps*)

lemma *jumpNestingOk-expr-list-simp* [*simp*]: *jumpNestingOk jmps* (*In3 es*) = *True*
by (*simp add*: *jumpNestingOk-def*)

lemma *jumpNestingOk-expr-list-simp1* [*simp*]:
jumpNestingOk jmps ⟨*es*::*expr list*⟩ = *True*
by (*simp add*: *inj-term-simps*)

Calculation of assigned variables for boolean expressions

2 Very restricted calculation fallback calculation

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

primrec *assignsE* :: *expr* ⇒ *lname set*
and *assignsV* :: *var* ⇒ *lname set*
and *assignsEs*:: *expr list* ⇒ *lname set*
where
assignsE (*NewC c*) = {}
| *assignsE* (*NewA t e*) = *assignsE e*
| *assignsE* (*Cast t e*) = *assignsE e*
| *assignsE* (*e InstOf r*) = *assignsE e*
| *assignsE* (*Lit val*) = {}
| *assignsE* (*UnOp unop e*) = *assignsE e*
| *assignsE* (*BinOp binop e1 e2*) = (*if binop*=*CondAnd* ∨ *binop*=*CondOr*
then (*assignsE e1*)
else (*assignsE e1*) ∪ (*assignsE e2*))
| *assignsE* (*Super*) = {}
| *assignsE* (*Acc v*) = *assignsV v*
| *assignsE* (*v*:=*e*) = (*assignsV v*) ∪ (*assignsE e*) ∪
(*if* ∃ *n*. *v*=(*LVar n*) *then* {*the-LVar-name v*}
else {})
| *assignsE* (*b*? *e1* : *e2*) = (*assignsE b*) ∪ ((*assignsE e1*) ∩ (*assignsE e2*))
| *assignsE* ({*accC*,*statT*,*mode*}*objRef*·*mn*({*pTs*}*args*))
= (*assignsE objRef*) ∪ (*assignsEs args*)
— Only dummy analysis for intermediate expressions *Methd*, *Body*, *InsInitE* and *Callee*
| *assignsE* (*Methd C sig*) = {}
| *assignsE* (*Body C s*) = {}
| *assignsE* (*InsInitE s e*) = {}
| *assignsE* (*Callee l e*) = {}

| *assignsV* (*LVar n*) = {}
| *assignsV* ({*accC*,*statDeclC*,*stat*}*objRef*..*fn*) = *assignsE objRef*
| *assignsV* (*e1*.[*e2*]) = *assignsE e1* ∪ *assignsE e2*

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

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

lemma *assigns-expr-simp* [*simp*]: *assigns* (*In1l e*) = *assignsE e*
by (*simp add*: *assigns-def*)

lemma *assigns-expr-simp1* [*simp*]: *assigns* (⟨*e*⟩) = *assignsE e*
by (*simp add*: *inj-term-simps*)

lemma *assigns-stmt-simp* [*simp*]: *assigns* (*In1r s*) = {}
by (*simp add*: *assigns-def*)

lemma *assigns-stmt-simp1* [*simp*]: *assigns* (⟨*s::stmt*⟩) = {}
by (*simp add*: *inj-term-simps*)

lemma *assigns-var-simp* [*simp*]: *assigns* (*In2 v*) = *assignsV v*
by (*simp add*: *assigns-def*)

lemma *assigns-var-simp1* [*simp*]: *assigns* (⟨*v*⟩) = *assignsV v*
by (*simp add*: *inj-term-simps*)

lemma *assigns-expr-list-simp* [*simp*]: *assigns* (*In3 es*) = *assignsEs es*
by (*simp add*: *assigns-def*)

lemma *assigns-expr-list-simp1* [*simp*]: *assigns* (⟨*es*⟩) = *assignsEs es*
by (*simp add*: *inj-term-simps*)

3 Analysis of constant expressions

```
primrec constVal :: expr ⇒ val option
where
  constVal (NewC c)      = None
| constVal (NewA t e)    = None
| constVal (Cast t e)    = None
| constVal (Inst e r)    = None
| constVal (Lit val)     = Some val
| constVal (UnOp unop e) = (case (constVal e) of
                              None   ⇒ None
                            | Some v ⇒ Some (eval-unop unop v))
| constVal (BinOp binop e1 e2) = (case (constVal e1) of
                                    None    ⇒ None
                                  | Some v1 ⇒ (case (constVal e2) of
                                                 None    ⇒ None
                                               | Some v2 ⇒ Some (eval-binop
                                                                  binop v1 v2)))
| constVal (Super)         = None
| constVal (Acc v)         = None
| constVal (Ass v e)       = None
| constVal (Cond b e1 e2)  = (case (constVal b) of
                                None   ⇒ None
                              | Some bv⇒ (case the-Bool bv of
                                           True ⇒ (case (constVal e2) of
                                                     None   ⇒ None
                                                   | Some v ⇒ constVal e1)
                                         | False⇒ (case (constVal e1) of
                                                     None   ⇒ None
                                                   | Some v ⇒ constVal e2)))
```

— Note that *constVal* (*Cond b e1 e2*) is stricter as it could be. It requires that all tree expressions are

constant even if we can decide which branch to choose, provided the constant value of *b*
| *constVal* (*Call accC statT mode objRef mn pTs args*) = *None*
| *constVal* (*Methd C sig*) = *None*
| *constVal* (*Body C s*) = *None*
| *constVal* (*InsInitE s e*) = *None*
| *constVal* (*Callee l e*) = *None*

lemma *constVal-Some-induct* [*consumes 1*, *case-names Lit UnOp BinOp CondL CondR*]:
 assumes *const*: *constVal e* = *Some v* **and**
 hyp-Lit: $\bigwedge$ *v*. *P* (*Lit v*) **and**
 hyp-UnOp: $\bigwedge$ *unop e'*. *P e'* $\Longrightarrow$ *P* (*UnOp unop e'*) **and**
 hyp-BinOp: $\bigwedge$ *binop e1 e2*. ⟦*P e1*; *P e2*⟧ $\Longrightarrow$ *P* (*BinOp binop e1 e2*) **and**
 hyp-CondL: $\bigwedge$ *b bv e1 e2*. ⟦*constVal b* = *Some bv*; *the-Bool bv*; *P b*; *P e1*⟧
 $\Longrightarrow$ *P* (*b*? *e1* : *e2*) **and**
 hyp-CondR: $\bigwedge$ *b bv e1 e2*. ⟦*constVal b* = *Some bv*; ¬*the-Bool bv*; *P b*; *P e2*⟧
 $\Longrightarrow$ *P* (*b*? *e1* : *e2*)
 shows *P e*
proof −
 have $\bigwedge$ *v*. *constVal e* = *Some v* $\Longrightarrow$ *P e*
 proof (*induct e*)
 case *Lit*
 show *?case* **by** (*rule hyp-Lit*)
 next
 case *UnOp*
 thus *?case*
 by (*auto intro*: *hyp-UnOp*)
 next
 case *BinOp*
 thus *?case*
 by (*auto intro*: *hyp-BinOp*)
 next
 case (*Cond b e1 e2*)
 then obtain *v* **where** *v*: *constVal* (*b* ? *e1* : *e2*) = *Some v*
 by *blast*
 then obtain *bv* **where** *bv*: *constVal b* = *Some bv*
 by *simp*
 show *?case*
 proof (*cases the-Bool bv*)
 case *True*
 with *Cond* **show** *?thesis* **using** *v bv*
 by (*auto intro*: *hyp-CondL*)
 next
 case *False*
 with *Cond* **show** *?thesis* **using** *v bv*
 by (*auto intro*: *hyp-CondR*)
 qed
 qed (*simp-all add*: *hyp-Lit*)
 with *const*
 show *?thesis*
 by *blast*
qed

lemma *assignsE-const-simp*: *constVal e* = *Some v* $\Longrightarrow$ *assignsE e* = {}
 by (*induct rule*: *constVal-Some-induct*) *simp-all*

4 Main analysis for boolean expressions

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

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

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

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

```
  have ⋀ b. ?Const b e ⟹ ?Ass b e
  proof (induct e)
    case Lit
    thus ?case by simp
  next
    case UnOp
    thus ?case by simp
  next
    case (BinOp binop)
    thus ?case
      by (cases binop) (simp-all)
  next
    case (Cond c e1 e2 b)
    note hyp-c = ‹⋀ b. ?Const b c ⟹ ?Ass b c›
    note hyp-e1 = ‹⋀ b. ?Const b e1 ⟹ ?Ass b e1›
    note hyp-e2 = ‹⋀ b. ?Const b e2 ⟹ ?Ass b e2›
    note const = ‹constVal (c ? e1 : e2) = Some (Bool b)›
    then obtain bv where bv: constVal c = Some bv
      by simp
    hence emptyC: assignsE c = {} by (rule assignsE-const-simp)
    show ?case
    proof (cases the-Bool bv)
      case True
      with const bv
      have ?Const b e1 by simp
      hence ?Ass b e1 by (rule hyp-e1)
      with emptyC bv True
      show ?thesis
        by simp
    next
      case False
      with const bv
      have ?Const b e2 by simp
      hence ?Ass b e2 by (rule hyp-e2)
      with emptyC bv False
      show ?thesis
        by simp
    qed
  qed (simp-all)
  with boolConst
  show ?thesis
    by blast
qed


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 –
  have ⋀ b. ?Const b e ⟹ ?Ass b e
  proof (induct e)
    case Lit
    thus ?case by simp
  next
    case UnOp
    thus ?case by simp
  next
    case (BinOp binop)
    thus ?case
```

by (*cases binop*) (*simp-all*)
next
case (*Cond c e1 e2 b*)
note *hyp-c* = ‹⋀ *b*. *?Const b c* ⟹ *?Ass b c*›
note *hyp-e1* = ‹⋀ *b*. *?Const b e1* ⟹ *?Ass b e1*›
note *hyp-e2* = ‹⋀ *b*. *?Const b e2* ⟹ *?Ass b e2*›
note *const* = ‹*constVal* (*c ? e1 : e2*) = *Some* (*Bool b*)›
then obtain *bv* **where** *bv*: *constVal c* = *Some bv*
by *simp*
show *?case*
proof (*cases the-Bool bv*)
case *True*
with *const bv*
have *?Const b e1* **by** *simp*
hence *?Ass b e1* **by** (*rule hyp-e1*)
with *bv True*
show *?thesis*
by *simp*
next
case *False*
with *const bv*
have *?Const b e2* **by** *simp*
hence *?Ass b e2* **by** (*rule hyp-e2*)
with *bv False*
show *?thesis*
by *simp*
qed
qed (*simp-all*)
with *boolConst*
show *?thesis*
by *blast*
qed

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

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

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

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

Binary union of tables

lemma *union-ts-iff* [*simp*]: (*c* ∈ (*A* ⇒∪ *B*) *k*) = (*c* ∈ *A k* ∨ *c* ∈ *B k*)
by (*unfold union-ts-def*) *blast*

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

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

lemma *union-tsCI* [*intro*!]: $(c \notin B\ k \Longrightarrow c \in A\ k) \Longrightarrow c \in (A \Rightarrow\cup B)\ k$
by *auto*

lemma *union-tsE* [*elim*!]:
$[\![c \in (A \Rightarrow\cup B)\ k;\ (c \in A\ k \Longrightarrow P);\ (c \in B\ k \Longrightarrow P)]\!] \Longrightarrow P$
by (*unfold union-ts-def*) *blast*

Binary intersection of tables

lemma *intersect-ts-iff* [*simp*]: $c \in (A \Rightarrow\cap B)\ k = (c \in A\ k \wedge c \in B\ k)$
by (*unfold intersect-ts-def*) *blast*

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

lemma *intersect-tsD1*: $c \in (A \Rightarrow\cap B)\ k \Longrightarrow c \in A\ k$
by *simp*

lemma *intersect-tsD2*: $c \in (A \Rightarrow\cap B)\ k \Longrightarrow c \in B\ k$
by *simp*

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

All-Union of tables and set

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

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

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

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

lemma *all-union-tsE* [*elim*!]:
$[\![c \in (A \Rightarrow\cup_\forall B)\ k;\ (c \in A\ k \Longrightarrow P);\ (c \in B \Longrightarrow P)]\!] \Longrightarrow P$
by (*unfold all-union-ts-def*) *blast*

The rules of definite assignment

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

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

definition
$rmlab :: 'a \Rightarrow ('a,'b)\ tables \Rightarrow ('a,'b)\ tables$
where $rmlab\ k\ A = (\lambda x.\ if\ x{=}k\ then\ UNIV\ else\ A\ x)$

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

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

inductive
$da :: env \Rightarrow lname\ set \Rightarrow term \Rightarrow assigned \Rightarrow bool\ (\text{-}\vdash\ \text{-}\ »\text{-}»\ \text{-}\ [65,65,65,65]\ 71)$
where
$Skip$: $Env\vdash B\ »\langle Skip\rangle»\ (\!|nrm{=}B,brk{=}\lambda\ l.\ UNIV|\!)$

| $Expr$: $Env\vdash B\ »\langle e\rangle»\ A$
$\Longrightarrow$
$Env\vdash B\ »\langle Expr\ e\rangle»\ A$
| Lab: $[\![Env\vdash B\ »\langle c\rangle»\ C;\ nrm\ A = nrm\ C \cap (brk\ C)\ l;\ brk\ A = rmlab\ l\ (brk\ C)]\!]$
$\Longrightarrow$
$Env\vdash B\ »\langle Break\ l\cdot\ c\rangle»\ A$

| $Comp$: $[\![Env\vdash B\ »\langle c1\rangle»\ C1;\ Env\vdash nrm\ C1\ »\langle c2\rangle»\ C2;$
$nrm\ A = nrm\ C2;\ brk\ A = (brk\ C1) \Rightarrow\cap (brk\ C2)]\!]$
$\Longrightarrow$
$Env\vdash B\ »\langle c1;;\ c2\rangle»\ A$

| If: $[\![Env\vdash B\ »\langle e\rangle»\ E;$
$Env\vdash (B \cup assigns\text{-}if\ True\ \ e)\ »\langle c1\rangle»\ C1;$
$Env\vdash (B \cup assigns\text{-}if\ False\ e)\ »\langle c2\rangle»\ C2;$
$nrm\ A = nrm\ C1 \cap nrm\ C2;$
$brk\ A = brk\ C1 \Rightarrow\cap brk\ C2\ ]\!]$
$\Longrightarrow$
$Env\vdash B\ »\langle If(e)\ c1\ Else\ c2\rangle»\ A$

— Note that E is not further used, because we take the specialized sets that also consider if the expression evaluates to true or false. Inside of e there is no `break` or `finally`, so the break map of E will be the trivial one. So $Env\vdash B\ »\langle e\rangle»\ E$ 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 $assigns\text{-}if\ False\ e = UNIV$ and therefor $nrm\ C2 = UNIV$. So finally $nrm\ A = nrm\ C1$. For the break maps this trick workd too, because the trivial break map will map all labels to $UNIV$. In the example, if no break occurs in $c2$ the break maps will trivially map to $UNIV$ and if a break occurs it will map to $UNIV$ too, because $assigns\text{-}if\ False\ e = UNIV$. So in the intersection of the break maps the path $c2$ will have no contribution.

| $Loop$: $[\![Env\vdash B\ »\langle e\rangle»\ E;$
$Env\vdash (B \cup assigns\text{-}if\ True\ e)\ »\langle c\rangle»\ C;$
$nrm\ A = nrm\ C \cap (B \cup assigns\text{-}if\ False\ e);$
$brk\ A = brk\ C]\!]$
$\Longrightarrow$

Env⊢ *B* »⟨*l*· *While*(*e*) *c*⟩» *A*

— The *Loop* rule resembles some of the ideas of the *If* rule. For the *nrm A* the set *B* ∪ *assigns-if False e* will be *UNIV* if the condition is constantly true. To normally exit the while loop, we must consider the body *c* to be completed normally (*nrm C*) or with a break. But in this model, the label *l* of the loop only handles continue labels, not break labels. The break label will be handled by an enclosing *Lab* statement. So we don't have to handle the breaks specially.

```
| Jmp: ⟦jump=Ret ⟶ Result ∈ B;
       nrm A = UNIV;
       brk A = (case jump of
                   Break l ⇒ λ k. if k=l then B else UNIV
                 | Cont l  ⇒ λ k. UNIV
                 | Ret     ⇒ λ k. UNIV)⟧
       ⟹
       Env⊢ B »⟨Jmp jump⟩» A
```

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

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

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

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

— The set of assigned variables before execution *c2* are the same as before execution *c1*, because *c1* could throw an exception and so we can't guarantee that any variable will be assigned in *c1*. The *Finally* statement completes normally if both *c1* and *c2* complete normally. If *c1* completes abruptly with a break, then *c2* also will be executed and may terminate normally or with a break. The overall break map then is the intersection of the maps of both paths. If *c2* terminates normally we have to extend all break sets in *brk C1* with *nrm C2* (⇒∪∀). If *c2* exits with a break this break will appear in the overall result state. We don't know if *c1* completed normally or abruptly (maybe with an exception not only a break) so *c1* has no contribution to the break map following this path.

— Evaluation of expressions and the break sets of definite assignment: Thinking of a Java expression we assume that we can never have a break statement inside of a expression. So for all expressions the break sets could be set to the trivial one: $\lambda l.$ *UNIV*. But we can't trivially proof, that evaluating an expression will never result in a break, allthough Java expressions allready syntactically don't allow nested stetements in them. The reason are the nested class initialzation statements which are inserted by the evaluation rules. So to proof the absence of a break we need to ensure, that the initialization statements will never end up in a break. In a wellfromed initialization statement, of course, were breaks are nested correctly inside of *Lab* or *Loop* statements evaluation of the whole initialization statement will never result in a break, because this break will be handled inside of the statement. But for simplicity we haven't added the analysis of the correct nesting of breaks in the typing judgments right now. So we have decided to adjust the rules of definite assignment to fit to these circumstances. If an initialization is involved during evaluation of the expression (evaluation rules *FVar*, *NewC* and *NewA*

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

— Wellformedness of a program will ensure, that every static initialiser is definetly assigned and the jumps are nested correctly. The case here for *Init* is just for convenience, to get a proper precondition for the induction hypothesis in various proofs, so that we don't have to expand the initialisation on every point where it is triggerred by the evaluation rules.

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

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

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

| *UnOp*: $Env\vdash B \gg\langle e\rangle\gg A$
$\Longrightarrow$
$Env\vdash B \gg\langle UnOp\ unop\ e\rangle\gg A$

| *CondAnd*: $[\![Env\vdash B \gg\langle e1\rangle\gg E1;\ Env\vdash (B \cup assigns\text{-}if\ True\ e1) \gg\langle e2\rangle\gg E2;$
$nrm\ A = B \cup (assigns\text{-}if\ True\ (BinOp\ CondAnd\ e1\ e2) \cap$
$assigns\text{-}if\ False\ (BinOp\ CondAnd\ e1\ e2));$
$brk\ A = (\lambda\ l.\ UNIV)\ ]\!]$
$\Longrightarrow$
$Env\vdash B \gg\langle BinOp\ CondAnd\ e1\ e2\rangle\gg A$

| *CondOr*: $[\![Env\vdash B \gg\langle e1\rangle\gg E1;\ Env\vdash (B \cup assigns\text{-}if\ False\ e1) \gg\langle e2\rangle\gg E2;$
$nrm\ A = B \cup (assigns\text{-}if\ True\ (BinOp\ CondOr\ e1\ e2) \cap$
$assigns\text{-}if\ False\ (BinOp\ CondOr\ e1\ e2));$
$brk\ A = (\lambda\ l.\ UNIV)\ ]\!]$
$\Longrightarrow$
$Env\vdash B \gg\langle BinOp\ CondOr\ e1\ e2\rangle\gg A$

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

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

| *AccLVar*: $[\![vn \in B;$
$nrm\ A = B;\ brk\ A = (\lambda\ k.\ UNIV)]\!]$
$\Longrightarrow$
$Env\vdash B \gg\langle Acc\ (LVar\ vn)\rangle\gg A$

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

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

| *AssLVar*: $[\![Env\vdash B \gg\langle e\rangle\gg E;\ nrm\ A = nrm\ E \cup \{vn\};\ brk\ A = brk\ E]\!]$

$$\begin{array}{l}\Longrightarrow \\ Env\vdash B \gg\langle(LVar\ vn) := e\rangle\gg A\end{array}$$

$$\begin{array}{l}|\ Ass: [\![\forall\ vn.\ v \neq LVar\ vn;\ Env\vdash B \gg\langle v\rangle\gg V;\ Env\vdash nrm\ V \gg\langle e\rangle\gg A]\!] \\ \quad\Longrightarrow \\ \quad Env\vdash B \gg\langle v := e\rangle\gg A\end{array}$$

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

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

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

— 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.

$$\begin{array}{l}|\ Methd: [\![methd\ (prg\ Env)\ D\ sig = Some\ m; \\ \quad Env\vdash B \gg\langle Body\ (declclass\ m)\ (stmt\ (mbody\ (mthd\ m)))\rangle\gg A \\ \quad]\!] \\ \quad\Longrightarrow \\ \quad Env\vdash B \gg\langle Methd\ D\ sig\rangle\gg A\end{array}$$

$$\begin{array}{l}|\ Body: [\![Env\vdash B \gg\langle c\rangle\gg C;\ jumpNestingOkS\ \{Ret\}\ c;\ Result \in nrm\ C; \\ \quad nrm\ A = B;\ brk\ A = (\lambda\ l.\ UNIV)]\!] \\ \quad\Longrightarrow \\ \quad Env\vdash B \gg\langle Body\ D\ c\rangle\gg A\end{array}$$

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

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

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

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

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

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

declare *inj-term-sym-simps* [*simp*]
declare *assigns-if*.*simps* [*simp del*]
declare *split-paired-All* [*simp del*] *split-paired-Ex* [*simp del*]
setup ‹*map-theory-simpset* (*fn ctxt* => *ctxt delloop split-all-tac*)›

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*]
setup ‹*map-theory-simpset* (*fn ctxt* => *ctxt addloop* (*split-all-tac*, *split-all-tac*))›

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

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

lemma *da-Lit*: *A* = (|*nrm*=*B*,*brk*=λ *l*. *UNIV*|) ⟹ *Env*⊢ *B* »⟨*Lit v*⟩» *A*
by (*auto intro*: *da.Lit*)

lemma *da-Super*: ⟦*This* ∈ *B*;*A* = (|*nrm*=*B*,*brk*=λ *l*. *UNIV*|)⟧ ⟹ *Env*⊢ *B* »⟨*Super*⟩» *A*
by (*auto intro*: *da.Super*)

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

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 −
obtain *vv* **where** *ex-lit*: *E*⊢*Lit vv*::− *PrimT Boolean*
using *typeof.simps*(*2*) *wt.Lit* **by** *blast*

```
have ?Boolean e ⟹ ?Incl e
proof (induct e)
  case (Cast T e)
  have E⊢e::− (PrimT Boolean)
  proof −
    from ‹E⊢(Cast T e)::− (PrimT Boolean)›
    obtain Te where E⊢e::−Te
                          prg E⊢Te≼? PrimT Boolean
      by cases simp
    thus ?thesis
      by − (drule cast-Boolean2,simp)
  qed
  with Cast.hyps
  show ?case
    by simp
next
  case (Lit val)
  thus ?case
    by − (erule wt-elim-cases, cases val, auto simp add: empty-dt-def)
next
  case (UnOp unop e)
  thus ?case
    by − (erule wt-elim-cases,cases unop,
          (fastforce simp add: assignsE-const-simp)+)
next
  case (BinOp binop e1 e2)
  from BinOp.prems obtain e1T e2T
    where E⊢e1::−e1T and E⊢e2::−e2T and wt-binop (prg E) binop e1T e2T
          and (binop-type binop) = Boolean
    by (elim wt-elim-cases) simp
  with BinOp.hyps
  show ?case
    by − (cases binop, auto simp add: assignsE-const-simp)
next
  case (Cond c e1 e2)
  note hyp-c = ‹?Boolean c ⟹ ?Incl c›
  note hyp-e1 = ‹?Boolean e1 ⟹ ?Incl e1›
  note hyp-e2 = ‹?Boolean e2 ⟹ ?Incl e2›
  note wt = ‹E⊢(c ? e1 : e2)::−PrimT Boolean›
  then obtain
    boolean-c: E⊢c::−PrimT Boolean and
    boolean-e1: E⊢e1::−PrimT Boolean and
    boolean-e2: E⊢e2::−PrimT Boolean
    by (elim wt-elim-cases) (auto dest: widen-Boolean2)
  show ?case
  proof (cases constVal c)
    case None
    with boolean-e1 boolean-e2
    show ?thesis
      using hyp-e1 hyp-e2
      by (auto)
  next
    case (Some bv)
    show ?thesis
    proof (cases the-Bool bv)
      case True
      with Some show ?thesis using hyp-e1 boolean-e1 by auto
    next
      case False
```

with *Some* **show** *?thesis* **using** *hyp-e2 boolean-e2* **by** *auto*
qed
qed
next
show $\bigwedge x.\ E\vdash Lit\ vv{::}{-}PrimT\ Boolean$
by (*rule ex-lit*)
qed (*simp-all add*: *ex-lit*)
with *boolEx*
show *?thesis*
by *blast*
qed

lemma *rmlab-same-label* [*simp*]: $(rmlab\ l\ A)\ l = UNIV$
by (*simp add*: *rmlab-def*)

lemma *rmlab-same-label1* [*simp*]: $l{=}l' \Longrightarrow (rmlab\ l\ A)\ l' = UNIV$
by (*simp add*: *rmlab-def*)

lemma *rmlab-other-label* [*simp*]: $l{\neq}l'{\Longrightarrow} (rmlab\ l\ A)\ l' = A\ l'$
by (*auto simp add*: *rmlab-def*)

lemma *range-inter-ts-subseteq* [*intro*]: $\forall\ k.\ A\ k \subseteq B\ k \Longrightarrow {\Rightarrow}\bigcap A \subseteq {\Rightarrow}\bigcap B$
by (*auto simp add*: *range-inter-ts-def*)

lemma *range-inter-ts-subseteq'*: $\forall\ k.\ A\ k \subseteq B\ k \Longrightarrow x \in {\Rightarrow}\bigcap A \Longrightarrow x \in {\Rightarrow}\bigcap B$
by (*auto simp add*: *range-inter-ts-def*)

lemma *da-monotone*:
assumes *da*: $Env\vdash\ B\ \ »t»\ A$ **and**
$B \subseteq B'$ **and**
da': $Env\vdash\ B'\ »t»\ A'$
shows $(nrm\ A \subseteq nrm\ A') \wedge (\forall\ l.\ (brk\ A\ l \subseteq brk\ A'\ l))$
proof −
from *da*
have $\bigwedge\ B'\ A'.\ [\![Env\vdash\ B'\ »t»\ A';\ B \subseteq B']\!]$
$\Longrightarrow (nrm\ A \subseteq nrm\ A') \wedge (\forall\ l.\ (brk\ A\ l \subseteq brk\ A'\ l))$
(**is** *PROP ?Hyp Env B t A*)
proof (*induct*)
case *Skip*
then show *?case* **by** *cases simp*
next
case *Expr*
from *Expr.prems Expr.hyps*
show *?case* **by** *cases simp*
next
case (*Lab Env B c C A l B' A'*)
note $A = ‹nrm\ A = nrm\ C \cap brk\ C\ l›\ ‹brk\ A = rmlab\ l\ (brk\ C)›$
note $‹PROP\ ?Hyp\ Env\ B\ \langle c\rangle\ C›$
moreover
note $‹B \subseteq B'›$

moreover
obtain *C′*
where *Env*⊢ *B′* »⟨*c*⟩» *C′*
and *A′*: *nrm A′* = *nrm C′* ∩ *brk C′ l brk A′* = *rmlab l* (*brk C′*)
using *Lab.prems*
by *cases simp*
ultimately
have *nrm C* ⊆ *nrm C′* **and** *hyp-brk*: (∀ *l*. *brk C l* ⊆ *brk C′ l*) **by** *auto*
then
have *nrm C* ∩ *brk C l* ⊆ *nrm C′* ∩ *brk C′ l* **by** *auto*
moreover
{
fix *l′*
from *hyp-brk*
have *rmlab l* (*brk C*) *l′* ⊆ *rmlab l* (*brk C′*) *l′*
by (*cases l*=*l′*) *simp-all*
}
moreover note *A A′*
ultimately show *?case*
by *simp*
next
case (*Comp Env B c1 C1 c2 C2 A B′ A′*)
note *A* = ‹*nrm A* = *nrm C2*› ‹*brk A* = *brk C1* ⇒∩ *brk C2*›
from ‹*Env*⊢ *B′* »⟨*c1*;; *c2*⟩» *A′*›
obtain *C1′ C2′*
where *da-c1*: *Env*⊢ *B′* »⟨*c1*⟩» *C1′* **and**
da-c2: *Env*⊢ *nrm C1′* »⟨*c2*⟩» *C2′* **and**
A′: *nrm A′* = *nrm C2′ brk A′* = *brk C1′* ⇒∩ *brk C2′*
by *cases auto*
note ‹*PROP ?Hyp Env B* ⟨*c1*⟩ *C1*›
moreover note ‹*B* ⊆ *B′*›
moreover note *da-c1*
ultimately have *C1′*: *nrm C1* ⊆ *nrm C1′* (∀ *l*. *brk C1 l* ⊆ *brk C1′ l*)
by *auto*
note ‹*PROP ?Hyp Env* (*nrm C1*) ⟨*c2*⟩ *C2*›
with *da-c2 C1′*
have *C2′*: *nrm C2* ⊆ *nrm C2′* (∀ *l*. *brk C2 l* ⊆ *brk C2′ l*)
by *auto*
with *A A′ C1′*
show *?case*
by *auto*
next
case (*If Env B e E c1 C1 c2 C2 A B′ A′*)
note *A* = ‹*nrm A* = *nrm C1* ∩ *nrm C2*› ‹*brk A* = *brk C1* ⇒∩ *brk C2*›
from ‹*Env*⊢ *B′* »⟨*If*(*e*) *c1 Else c2*⟩» *A′*›
obtain *C1′ C2′*
where *da-c1*: *Env*⊢ *B′* ∪ *assigns-if True e* »⟨*c1*⟩» *C1′* **and**
da-c2: *Env*⊢ *B′* ∪ *assigns-if False e* »⟨*c2*⟩» *C2′* **and**
A′: *nrm A′* = *nrm C1′* ∩ *nrm C2′ brk A′* = *brk C1′* ⇒∩ *brk C2′*
by *cases auto*
note ‹*PROP ?Hyp Env* (*B* ∪ *assigns-if True e*) ⟨*c1*⟩ *C1*›
moreover note *B′* = ‹*B* ⊆ *B′*›
moreover note *da-c1*
ultimately obtain *C1′*: *nrm C1* ⊆ *nrm C1′* (∀ *l*. *brk C1 l* ⊆ *brk C1′ l*)
by *blast*
note ‹*PROP ?Hyp Env* (*B* ∪ *assigns-if False e*) ⟨*c2*⟩ *C2*›
with *da-c2 B′*
obtain *C2′*: *nrm C2* ⊆ *nrm C2′* (∀ *l*. *brk C2 l* ⊆ *brk C2′ l*)
by *blast*

```
  with A A' C1'
  show ?case
    by auto
next
  case (Loop Env B e E c C A l B' A')
  note A = ‹nrm A = nrm C ∩ (B ∪ assigns-if False e)› ‹brk A = brk C›
  from ‹Env⊢ B' »⟨l• While(e) c⟩» A'›
  obtain C'
    where
     da-c': Env⊢ B' ∪ assigns-if True e »⟨c⟩» C' and
        A': nrm A' = nrm C' ∩ (B' ∪ assigns-if False e)
            brk A' = brk C'
    by cases auto
  note ‹PROP ?Hyp Env (B ∪ assigns-if True e) ⟨c⟩ C›
  moreover note B' = ‹B ⊆ B'›
  moreover note da-c'
  ultimately obtain C': nrm C ⊆ nrm C' (∀ l. brk C l ⊆ brk C' l)
    by blast
  with A A' B'
  have nrm A ⊆ nrm A'
    by blast
  moreover
  { fix l'
    have  brk A l' ⊆ brk A' l'
    proof (cases constVal e)
      case None
      with A A' C'
      show ?thesis
         by (cases l=l') auto
    next
      case (Some bv)
      with A A' C'
      show ?thesis
        by (cases the-Bool bv, cases l=l') auto
    qed
  }
  ultimately show ?case
    by auto
next
  case (Jmp jump B A Env B' A')
  thus ?case by (elim da-elim-cases) (auto split: jump.splits)
next
  case Throw thus ?case by (elim da-elim-cases) auto
next
  case (Try Env B c1 C1 vn C c2 C2 A B' A')
  note A = ‹nrm A = nrm C1 ∩ nrm C2› ‹brk A = brk C1 ⇒∩  brk C2›
  from ‹Env⊢ B' »⟨Try c1 Catch(C vn) c2⟩» A'›
  obtain C1' C2'
    where da-c1': Env⊢ B' »⟨c1⟩» C1' and
          da-c2': Env(|lcl := (lcl Env)(VName vn↦Class C)|)⊢ B' ∪ {VName vn}
                   »⟨c2⟩» C2' and
          A': nrm A' = nrm C1' ∩ nrm C2'
              brk A' = brk C1' ⇒∩  brk C2'
    by cases auto
  note ‹PROP ?Hyp Env B ⟨c1⟩ C1›
  moreover note B' = ‹B ⊆ B'›
  moreover note da-c1'
  ultimately obtain C1': nrm C1 ⊆ nrm C1' (∀ l. brk C1 l ⊆ brk C1' l)
    by blast
```

```
    note ‹PROP ?Hyp (Env(|lcl := (lcl Env)(VName vn↦Class C)|))
                (B ∪ {VName vn}) ⟨c2⟩ C2›
    with B' da-c2'
    obtain nrm C2 ⊆ nrm C2' (∀ l. brk C2 l ⊆ brk C2' l)
      by blast
    with C1' A A'
    show ?case
      by auto
  next
    case (Fin Env B c1 C1 c2 C2 A B' A')
    note A = ‹nrm A = nrm C1 ∪ nrm C2›
      ‹brk A = (brk C1 ⇒∪∀ nrm C2) ⇒∩ (brk C2)›
    from ‹Env⊢ B' »⟨c1 Finally c2⟩» A'›
    obtain C1' C2'
      where  da-c1': Env⊢ B' »⟨c1⟩» C1' and
             da-c2': Env⊢ B' »⟨c2⟩» C2' and
             A':  nrm A' = nrm C1' ∪ nrm C2'
                  brk A' = (brk C1' ⇒∪∀ nrm C2') ⇒∩ (brk C2')
      by cases auto
    note ‹PROP ?Hyp Env B ⟨c1⟩ C1›
    moreover note B' = ‹B ⊆ B'›
    moreover note da-c1'
    ultimately obtain C1': nrm C1 ⊆ nrm C1' (∀ l. brk C1 l ⊆ brk C1' l)
      by blast
    note hyp-c2 = ‹PROP ?Hyp Env B ⟨c2⟩ C2›
    from da-c2' B'
     obtain nrm C2 ⊆ nrm C2' (∀ l. brk C2 l ⊆ brk C2' l)
       by − (drule hyp-c2,auto)
     with A A' C1'
     show ?case
       by auto
   next
     case Init thus ?case by (elim da-elim-cases) auto
   next
     case NewC thus ?case by (elim da-elim-cases) auto
   next
     case NewA thus ?case by (elim da-elim-cases) auto
   next
     case Cast thus ?case by (elim da-elim-cases) auto
   next
     case Inst thus ?case by (elim da-elim-cases) auto
   next
     case Lit thus ?case by (elim da-elim-cases) auto
   next
     case UnOp thus ?case by (elim da-elim-cases) auto
   next
     case (CondAnd Env B e1 E1 e2 E2 A B' A')
     note A = ‹nrm A = B ∪
                  assigns-if True (BinOp CondAnd e1 e2) ∩
                  assigns-if False (BinOp CondAnd e1 e2)›
           ‹brk A = (λl. UNIV)›
     from ‹Env⊢ B' »⟨BinOp CondAnd e1 e2⟩» A'›
     obtain  A': nrm A' = B' ∪
                          assigns-if True (BinOp CondAnd e1 e2) ∩
                          assigns-if False (BinOp CondAnd e1 e2)
                   brk A' = (λl. UNIV)
       by cases auto
     note B' = ‹B ⊆ B'›
     with A A' show ?case
```

```
    by auto
  next
    case CondOr thus ?case by (elim da-elim-cases) auto
  next
    case BinOp thus ?case by (elim da-elim-cases) auto
  next
    case Super thus ?case by (elim da-elim-cases) auto
  next
    case AccLVar thus ?case by (elim da-elim-cases) auto
  next
    case Acc thus ?case by (elim da-elim-cases) auto
  next
    case AssLVar thus ?case by (elim da-elim-cases) auto
  next
    case Ass thus ?case by (elim da-elim-cases) auto
  next
    case (CondBool Env c e1 e2 B C E1 E2 A B' A')
    note A = ‹nrm A = B ∪
                   assigns-if True (c ? e1 : e2) ∩
                   assigns-if False (c ? e1 : e2)›
          ‹brk A = (λl. UNIV)›
    note ‹Env⊢ (c ? e1 : e2)::− (PrimT Boolean)›
    moreover
    note ‹Env⊢ B' »⟨c ? e1 : e2⟩» A'›
    ultimately
    obtain A': nrm A' = B' ∪
                             assigns-if True (c ? e1 : e2) ∩
                             assigns-if False (c ? e1 : e2)
                 brk A' = (λl. UNIV)
      by (elim da-elim-cases) (auto simp add: inj-term-simps)

    note B' = ‹B ⊆ B'›
    with A A' show ?case
      by auto
  next
    case (Cond Env c e1 e2 B C E1 E2 A B' A')
    note A = ‹nrm A = nrm E1 ∩ nrm E2› ‹brk A = (λl. UNIV)›
    note not-bool = ‹¬ Env⊢ (c ? e1 : e2)::− (PrimT Boolean)›
    from ‹Env⊢ B' »⟨c ? e1 : e2⟩» A'›
    obtain E1' E2'
      where da-e1': Env⊢ B' ∪ assigns-if True c »⟨e1⟩» E1' and
            da-e2': Env⊢ B' ∪ assigns-if False c »⟨e2⟩» E2' and
              A': nrm A' = nrm E1' ∩ nrm E2'
                brk A' = (λl. UNIV)
      using not-bool
      by (elim da-elim-cases) (auto simp add: inj-term-simps)

    note ‹PROP ?Hyp Env (B ∪ assigns-if True c) ⟨e1⟩ E1›
    moreover note B' = ‹B ⊆ B'›
    moreover note da-e1'
    ultimately obtain E1': nrm E1 ⊆ nrm E1' (∀ l. brk E1 l ⊆ brk E1' l)
      by blast
    note ‹PROP ?Hyp Env (B ∪ assigns-if False c) ⟨e2⟩ E2›
    with B' da-e2'
    obtain nrm E2 ⊆ nrm E2' (∀ l. brk E2 l ⊆ brk E2' l)
      by blast
   with E1' A A'
   show ?case
     by auto
```

```
  next
    case Call
    from Call.prems and Call.hyps
    show ?case by cases auto
  next
    case Methd thus ?case by (elim da-elim-cases) auto
  next
    case Body thus ?case by (elim da-elim-cases) auto
  next
    case LVar thus ?case by (elim da-elim-cases) auto
  next
    case FVar thus ?case by (elim da-elim-cases) auto
  next
    case AVar thus ?case by (elim da-elim-cases) auto
  next
    case Nil thus ?case by (elim da-elim-cases) auto
  next
    case Cons thus ?case by (elim da-elim-cases) auto
  qed
  from this [OF da' ‹B ⊆ B'›] show ?thesis .
qed


lemma da-weaken:
  assumes da: Env⊢ B »t» A and B ⊆ B'
  shows ∃ A'. Env ⊢ B' »t» A'
proof −
  note assigned.select-convs [Pure.intro]
  from da
  have ⋀ B'. B ⊆ B' ⟹ ∃ A'. Env⊢ B' »t» A' (is PROP ?Hyp Env B t)
  proof (induct)
    case Skip thus ?case by (iprover intro: da.Skip)
  next
    case Expr thus ?case by (iprover intro: da.Expr)
  next
    case (Lab Env B c C A l B')
    note ‹PROP ?Hyp Env B ⟨c⟩›
    moreover
    note B' = ‹B ⊆ B'›
    ultimately obtain C' where Env⊢ B' »⟨c⟩» C'
      by iprover
    then obtain A' where Env⊢ B' »⟨Break l· c⟩» A'
      by (iprover intro: da.Lab)
    thus ?case ..
  next
    case (Comp Env B c1 C1 c2 C2 A B')
    note da-c1 = ‹Env⊢ B »⟨c1⟩» C1›
    note ‹PROP ?Hyp Env B ⟨c1⟩›
    moreover
    note B' = ‹B ⊆ B'›
    ultimately obtain C1' where da-c1': Env⊢ B' »⟨c1⟩» C1'
      by iprover
    with da-c1 B'
    have
      nrm C1 ⊆ nrm C1'
      by (rule da-monotone [elim-format]) simp
    moreover
    note ‹PROP ?Hyp Env (nrm C1) ⟨c2⟩›
    ultimately obtain C2' where Env⊢ nrm C1' »⟨c2⟩» C2'
```

```
      by iprover
    with da-c1' obtain A' where Env⊢ B' »⟨c1;; c2⟩» A'
      by (iprover intro: da.Comp)
    thus ?case ..
  next
    case (If Env B e E c1 C1 c2 C2 A B')
    note B' = ‹B ⊆ B'›
    obtain E' where Env⊢ B' »⟨e⟩» E'
    proof –
      have PROP ?Hyp Env B ⟨e⟩ by (rule If.hyps)
      with B'
      show ?thesis using that by iprover
    qed
    moreover
    obtain C1' where Env⊢ (B' ∪ assigns-if True e) »⟨c1⟩» C1'
    proof –
      from B'
      have (B ∪ assigns-if True e) ⊆ (B' ∪ assigns-if True e)
        by blast
      moreover
      have PROP ?Hyp Env (B ∪ assigns-if True e) ⟨c1⟩ by (rule If.hyps)
      ultimately
      show ?thesis using that by iprover
    qed
    moreover
    obtain C2' where Env⊢ (B' ∪ assigns-if False e) »⟨c2⟩» C2'
    proof –
      from B' have (B ∪ assigns-if False e) ⊆ (B' ∪ assigns-if False e)
        by blast
      moreover
      have PROP ?Hyp Env (B ∪ assigns-if False e) ⟨c2⟩ by (rule If.hyps)
      ultimately
      show ?thesis using that by iprover
    qed
    ultimately
    obtain A' where Env⊢ B' »⟨If(e) c1 Else c2⟩» A'
      by (iprover intro: da.If)
    thus ?case ..
  next
    case (Loop Env B e E c C A l B')
    note B' = ‹B ⊆ B'›
    obtain E' where Env⊢ B' »⟨e⟩» E'
    proof –
      have PROP ?Hyp Env B ⟨e⟩ by (rule Loop.hyps)
      with B'
      show ?thesis using that by iprover
    qed
    moreover
    obtain C' where Env⊢ (B' ∪ assigns-if True e) »⟨c⟩» C'
    proof –
      from B'
      have (B ∪ assigns-if True e) ⊆ (B' ∪ assigns-if True e)
        by blast
      moreover
      have PROP ?Hyp Env (B ∪ assigns-if True e) ⟨c⟩ by (rule Loop.hyps)
      ultimately
      show ?thesis using that by iprover
    qed
    ultimately
```

```
    obtain A' where Env⊢ B' »⟨l• While(e) c⟩» A'
      by (iprover intro: da.Loop )
    thus ?case ..
  next
    case (Jmp jump B A Env B')
    note B' = ‹B ⊆ B'›
    with Jmp.hyps have jump = Ret ⟶ Result ∈ B'
      by auto
    moreover
    obtain A'::assigned
              where  nrm A' = UNIV
                     brk A' = (case jump of
                                 Break l ⇒ λk. if k = l then B' else UNIV
                               | Cont l ⇒ λk. UNIV
                               | Ret ⇒ λk. UNIV)

      by  iprover
    ultimately have Env⊢ B' »⟨Jmp jump⟩» A'
      by (rule da.Jmp)
    thus ?case ..
  next
    case Throw thus ?case by (iprover intro: da.Throw )
  next
    case (Try Env B c1 C1 vn C c2 C2 A B')
    note B' = ‹B ⊆ B'›
    obtain C1' where Env⊢ B' »⟨c1⟩» C1'
    proof -
      have PROP ?Hyp Env B ⟨c1⟩ by (rule Try.hyps)
      with B'
      show ?thesis using that by iprover
    qed
    moreover
    obtain C2' where
      Env(|lcl := (lcl Env)(VName vn↦Class C)|)⊢ B' ∪ {VName vn} »⟨c2⟩» C2'
    proof -
      from B' have B ∪ {VName vn} ⊆ B' ∪ {VName vn} by blast
      moreover
      have PROP ?Hyp (Env(|lcl := (lcl Env)(VName vn↦Class C)|))
                      (B ∪ {VName vn}) ⟨c2⟩
        by (rule Try.hyps)
      ultimately
      show ?thesis using that by iprover
    qed
    ultimately
    obtain A' where Env⊢ B' »⟨Try c1 Catch(C vn) c2⟩» A'
      by (iprover intro: da.Try )
    thus ?case ..
  next
    case (Fin Env B c1 C1 c2 C2 A B')
    note B' = ‹B ⊆ B'›
    obtain C1' where C1': Env⊢ B' »⟨c1⟩» C1'
    proof -
      have PROP ?Hyp Env B ⟨c1⟩ by (rule Fin.hyps)
      with B'
      show ?thesis using that by iprover
    qed
    moreover
    obtain C2' where Env⊢ B' »⟨c2⟩» C2'
    proof -
```

```
    have PROP ?Hyp Env B ⟨c2⟩ by (rule Fin.hyps)
    with B'
    show ?thesis using that by iprover
  qed
  ultimately
  obtain A' where Env⊢ B' »⟨c1 Finally c2⟩» A'
    by (iprover intro: da.Fin )
  thus ?case ..
next
  case Init thus ?case by (iprover intro: da.Init)
next
  case NewC thus ?case by (iprover intro: da.NewC)
next
  case NewA thus ?case by (iprover intro: da.NewA)
next
  case Cast thus ?case by (iprover intro: da.Cast)
next
  case Inst thus ?case by (iprover intro: da.Inst)
next
  case Lit thus ?case by (iprover intro: da.Lit)
next
  case UnOp thus ?case by (iprover intro: da.UnOp)
next
  case (CondAnd Env B e1 E1 e2 E2 A B')
  note B' = ‹B ⊆ B'›
  obtain E1' where Env⊢ B' »⟨e1⟩» E1'
  proof –
    have PROP ?Hyp Env B ⟨e1⟩ by (rule CondAnd.hyps)
    with B'
    show ?thesis using that by iprover
  qed
  moreover
  obtain E2' where Env⊢ B' ∪  assigns-if True e1 »⟨e2⟩» E2'
  proof –
    from B' have B ∪ assigns-if True e1 ⊆ B' ∪  assigns-if True e1
      by blast
    moreover
    have PROP ?Hyp Env (B ∪ assigns-if True e1) ⟨e2⟩ by (rule CondAnd.hyps)
    ultimately show ?thesis using that by iprover
  qed
  ultimately
  obtain A' where Env⊢ B' »⟨BinOp CondAnd e1 e2⟩» A'
    by (iprover intro: da.CondAnd)
  thus ?case ..
next
  case (CondOr Env B e1 E1 e2 E2 A B')
  note B' = ‹B ⊆ B'›
  obtain E1' where Env⊢ B' »⟨e1⟩» E1'
  proof –
    have PROP ?Hyp Env B ⟨e1⟩ by (rule CondOr.hyps)
    with B'
    show ?thesis using that by iprover
  qed
  moreover
  obtain E2' where Env⊢ B' ∪ assigns-if False e1 »⟨e2⟩» E2'
  proof –
    from B' have B ∪ assigns-if False e1 ⊆ B' ∪  assigns-if False e1
      by blast
    moreover
```

```
      have PROP ?Hyp Env (B ∪ assigns-if False e1) ⟨e2⟩ by (rule CondOr.hyps)
      ultimately show ?thesis using that by iprover
    qed
    ultimately
    obtain A′ where Env⊢ B′ »⟨BinOp CondOr e1 e2⟩» A′
      by (iprover intro: da.CondOr)
    thus ?case ..
  next
    case (BinOp Env B e1 E1 e2 A binop B′)
    note B′ = ‹B ⊆ B′›
    obtain E1′ where E1′: Env⊢ B′ »⟨e1⟩» E1′
    proof –
      have PROP ?Hyp Env B ⟨e1⟩ by (rule BinOp.hyps)
      with B′
      show ?thesis using that by iprover
    qed
    moreover
    obtain A′ where Env⊢ nrm E1′ »⟨e2⟩» A′
    proof –
      have Env⊢ B »⟨e1⟩» E1 by (rule BinOp.hyps)
      from this B′ E1′
      have nrm E1 ⊆ nrm E1′
        by (rule da-monotone [THEN conjE])
      moreover
      have PROP ?Hyp Env (nrm E1) ⟨e2⟩ by (rule BinOp.hyps)
      ultimately show ?thesis using that by iprover
    qed
    ultimately
    have Env⊢ B′ »⟨BinOp binop e1 e2⟩» A′
      using BinOp.hyps by (iprover intro: da.BinOp)
    thus ?case ..
  next
    case (Super B Env B′)
    note B′ = ‹B ⊆ B′›
    with Super.hyps have This ∈ B′
      by auto
    thus ?case by (iprover intro: da.Super)
  next
    case (AccLVar vn B A Env B′)
    note ‹vn ∈ B›
    moreover
    note ‹B ⊆ B′›
    ultimately have vn ∈ B′ by auto
    thus ?case by (iprover intro: da.AccLVar)
  next
    case Acc thus ?case by (iprover intro: da.Acc)
  next
    case (AssLVar Env B e E A vn B′)
    note B′ = ‹B ⊆ B′›
    then obtain E′ where Env⊢ B′ »⟨e⟩» E′
      by (rule AssLVar.hyps [elim-format]) iprover
    then obtain A′ where
      Env⊢ B′ »⟨LVar vn:=e⟩» A′
      by (iprover intro: da.AssLVar)
    thus ?case ..
  next
    case (Ass v Env B V e A B′)
    note B′ = ‹B ⊆ B′›
    note ‹∀ vn. v ≠ LVar vn›
```

moreover
obtain V' **where** V': *Env*⊢ B' »⟨*v*⟩» V'
proof −
have *PROP ?Hyp Env B* ⟨*v*⟩ **by** (*rule Ass.hyps*)
with B'
show *?thesis* **using** *that* **by** *iprover*
qed
moreover
obtain A' **where** *Env*⊢ *nrm* V' »⟨*e*⟩» A'
proof −
have *Env*⊢ *B* »⟨*v*⟩» *V* **by** (*rule Ass.hyps*)
from *this* B' V'
have *nrm V* ⊆ *nrm* V'
by (*rule da-monotone* [*THEN conjE*])
moreover
have *PROP ?Hyp Env* (*nrm V*) ⟨*e*⟩ **by** (*rule Ass.hyps*)
ultimately show *?thesis* **using** *that* **by** *iprover*
qed
ultimately
have *Env*⊢ B' »⟨*v* := *e*⟩» A'
by (*iprover intro*: *da.Ass*)
thus *?case* ..
next
case (*CondBool Env c e1 e2 B C E1 E2 A* B')
note B' = ‹*B* ⊆ B'›
note ‹*Env*⊢(*c ? e1 : e2*)::−(*PrimT Boolean*)›
moreover obtain C' **where** C': *Env*⊢ B' »⟨*c*⟩» C'
proof −
have *PROP ?Hyp Env B* ⟨*c*⟩ **by** (*rule CondBool.hyps*)
with B'
show *?thesis* **using** *that* **by** *iprover*
qed
moreover
obtain *E1*′ **where** *Env*⊢ B' ∪ *assigns-if True c* »⟨*e1*⟩» *E1*′
proof −
from B'
have (*B* ∪ *assigns-if True c*) ⊆ (B' ∪ *assigns-if True c*)
by *blast*
moreover
have *PROP ?Hyp Env* (*B* ∪ *assigns-if True c*) ⟨*e1*⟩ **by** (*rule CondBool.hyps*)
ultimately
show *?thesis* **using** *that* **by** *iprover*
qed
moreover
obtain *E2*′ **where** *Env*⊢ B' ∪ *assigns-if False c* »⟨*e2*⟩» *E2*′
proof −
from B'
have (*B* ∪ *assigns-if False c*) ⊆ (B' ∪ *assigns-if False c*)
by *blast*
moreover
have *PROP ?Hyp Env* (*B* ∪ *assigns-if False c*) ⟨*e2*⟩ **by**(*rule CondBool.hyps*)
ultimately
show *?thesis* **using** *that* **by** *iprover*
qed
ultimately
obtain A' **where** *Env*⊢ B' »⟨*c ? e1 : e2*⟩» A'
by (*iprover intro*: *da.CondBool*)
thus *?case* ..
next

```
  case (Cond Env c e1 e2 B C E1 E2 A B′)
  note B′ = ‹B ⊆ B′›
  note ‹¬ Env⊢(c ? e1 : e2)::−(PrimT Boolean)›
  moreover obtain C′ where C′: Env⊢ B′ »⟨c⟩» C′
  proof −
    have PROP ?Hyp Env B ⟨c⟩ by (rule Cond.hyps)
    with B′
    show ?thesis using that by iprover
  qed
  moreover
  obtain E1′ where Env⊢ B′ ∪ assigns-if True c »⟨e1⟩» E1′
  proof −
    from B′
    have (B ∪ assigns-if True c) ⊆ (B′ ∪ assigns-if True c)
      by blast
    moreover
    have PROP ?Hyp Env (B ∪ assigns-if True c) ⟨e1⟩ by (rule Cond.hyps)
    ultimately
    show ?thesis using that by iprover
  qed
  moreover
  obtain E2′ where Env⊢ B′ ∪ assigns-if False c »⟨e2⟩» E2′
  proof −
    from B′
    have (B ∪ assigns-if False c) ⊆ (B′ ∪ assigns-if False c)
      by blast
    moreover
    have PROP ?Hyp Env (B ∪ assigns-if False c) ⟨e2⟩ by (rule Cond.hyps)
    ultimately
    show ?thesis using that by iprover
  qed
  ultimately
  obtain A′ where Env⊢ B′ »⟨c ? e1 : e2⟩» A′
    by (iprover intro: da.Cond)
  thus ?case ..
next
  case (Call Env B e E args A accC statT mode mn pTs B′)
  note B′ = ‹B ⊆ B′›
  obtain E′ where E′: Env⊢ B′ »⟨e⟩» E′
  proof −
    have PROP ?Hyp Env B ⟨e⟩ by (rule Call.hyps)
    with B′
    show ?thesis using that by iprover
  qed
  moreover
  obtain A′ where Env⊢ nrm E′ »⟨args⟩» A′
  proof −
    have Env⊢ B »⟨e⟩» E by (rule Call.hyps)
    from this B′ E′
    have nrm E ⊆ nrm E′
      by (rule da-monotone [THEN conjE])
    moreover
    have PROP ?Hyp Env (nrm E) ⟨args⟩ by (rule Call.hyps)
    ultimately show ?thesis using that by iprover
  qed
  ultimately
  have Env⊢ B′ »⟨{accC,statT,mode}e·mn( {pTs}args)⟩» A′
    by (iprover intro: da.Call)
  thus ?case ..
```

```
next
  case Methd thus ?case by (iprover intro: da.Methd)
next
  case (Body Env B c C A D B')
  note B' = ‹B ⊆ B'›
  obtain C' where C': Env⊢ B' »⟨c⟩» C' and nrm-C': nrm C ⊆ nrm C'
  proof −
    have Env⊢ B »⟨c⟩» C by (rule Body.hyps)
    moreover note B'
    moreover
    from B' obtain C' where da-c: Env⊢ B' »⟨c⟩» C'
      by (rule Body.hyps [elim-format]) blast
    ultimately
    have nrm C ⊆ nrm C'
      by (rule da-monotone [THEN conjE])
    with da-c that show ?thesis by iprover
  qed
  moreover
  note ‹Result ∈ nrm C›
  with nrm-C' have Result ∈ nrm C'
    by blast
  moreover note ‹jumpNestingOkS {Ret} c›
  ultimately obtain A' where
    Env⊢ B' »⟨Body D c⟩» A'
    by (iprover intro: da.Body)
  thus ?case ..
next
  case LVar thus ?case by (iprover intro: da.LVar)
next
  case FVar thus ?case by (iprover intro: da.FVar)
next
  case (AVar Env B e1 E1 e2 A B')
  note B' = ‹B ⊆ B'›
  obtain E1' where E1': Env⊢ B' »⟨e1⟩» E1'
  proof −
    have PROP ?Hyp Env B ⟨e1⟩ by (rule AVar.hyps)
    with B'
    show ?thesis using that by iprover
  qed
  moreover
  obtain A' where Env⊢ nrm E1' »⟨e2⟩» A'
  proof −
    have Env⊢ B »⟨e1⟩» E1 by (rule AVar.hyps)
    from this B' E1'
    have nrm E1 ⊆ nrm E1'
      by (rule da-monotone [THEN conjE])
    moreover
    have PROP ?Hyp Env (nrm E1) ⟨e2⟩ by (rule AVar.hyps)
    ultimately show ?thesis using that by iprover
  qed
  ultimately
  have Env⊢ B' »⟨e1.[e2]⟩» A'
    by (iprover intro: da.AVar)
  thus ?case ..
next
  case Nil thus ?case by (iprover intro: da.Nil)
next
  case (Cons Env B e E es A B')
  note B' = ‹B ⊆ B'›
```

obtain *E′* **where** *E′*: *Env*⊢ *B′* »⟨*e*⟩» *E′*
proof −
have *PROP ?Hyp Env B* ⟨*e*⟩ **by** (*rule Cons.hyps*)
with *B′*
show *?thesis* **using** *that* **by** *iprover*
qed
moreover
obtain *A′* **where** *Env*⊢ *nrm E′* »⟨*es*⟩» *A′*
proof −
have *Env*⊢ *B* »⟨*e*⟩» *E* **by** (*rule Cons.hyps*)
from *this B′ E′*
have *nrm E* ⊆ *nrm E′*
by (*rule da-monotone* [*THEN conjE*])
moreover
have *PROP ?Hyp Env* (*nrm E*) ⟨*es*⟩ **by** (*rule Cons.hyps*)
ultimately show *?thesis* **using** *that* **by** *iprover*
qed
ultimately
have *Env*⊢ *B′* »⟨*e* # *es*⟩» *A′*
by (*iprover intro*: *da.Cons*)
thus *?case* **..**
qed
from *this* [*OF* ‹*B* ⊆ *B′*›] **show** *?thesis* **.**
qed

corollary *da-weakenE* [*consumes 2*]:
assumes *da*: *Env*⊢ *B* »*t*» *A* **and**
B′: *B* ⊆ *B′* **and**
ex-mono: ⋀ *A′*. ⟦*Env*⊢ *B′* »*t*» *A′*; *nrm A* ⊆ *nrm A′*;
⋀ *l*. *brk A l* ⊆ *brk A′ l*⟧ ⟹ *P*
shows *P*
proof −
from *da B′*
obtain *A′* **where** *A′*: *Env*⊢ *B′* »*t*» *A′*
by (*rule da-weaken* [*elim-format*]) *iprover*
with *da B′*
have *nrm A* ⊆ *nrm A′* ∧ (∀ *l*. *brk A l* ⊆ *brk A′ l*)
by (*rule da-monotone*)
with *A′ ex-mono*
show *?thesis*
by *iprover*
qed

end

Chapter 13

WellForm

1 Well-formedness of Java programs

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

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

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

simplifications:

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

well-formed field declarations

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

definition
  $wf\text{-}fdecl :: prog \Rightarrow pname \Rightarrow fdecl \Rightarrow bool$
  where $wf\text{-}fdecl\ G\ P = (\lambda(fn,f).\ is\text{-}acc\text{-}type\ G\ P\ (type\ f))$

lemma $wf\text{-}fdecl\text{-}def2$: $\bigwedge fd.\ wf\text{-}fdecl\ G\ P\ fd = is\text{-}acc\text{-}type\ G\ P\ (type\ (snd\ fd))$
apply (*unfold wf-fdecl-def*)
apply *simp*
done

well-formed method declarations

A method head is wellformed if:

- the signature and the method head agree in the number of parameters
- all types of the parameters are visible

- the result type is visible
- the parameter names are unique

definition
wf-mhead :: *prog* ⇒ *pname* ⇒ *sig* ⇒ *mhead* ⇒ *bool* **where**
wf-mhead G P = (λ *sig mh. length* (*parTs sig*) = *length* (*pars mh*) ∧
(∀ *T*∈*set* (*parTs sig*). *is-acc-type G P T*) ∧
is-acc-type G P (*resTy mh*) ∧
distinct (*pars mh*))

A method declaration is wellformed if:

- the method head is wellformed
- the names of the local variables are unique
- the types of the local variables must be accessible
- the local variables don't shadow the parameters
- the class of the method is defined
- the body statement is welltyped with respect to the modified environment of local names, were the local variables, the parameters the special result variable (Res) and This are assoziated with there types.

definition
callee-lcl :: *qtname* ⇒ *sig* ⇒ *methd* ⇒ *lenv* **where**
callee-lcl C sig m =
(λ*k*. (*case k of*
EName e
⇒ (*case e of*
VNam v
⇒((*table-of* (*lcls* (*mbody m*)))(*pars m* [↦] *parTs sig*)) *v*
| *Res* ⇒ *Some* (*resTy m*))
| *This* ⇒ *if is-static m then None else Some* (*Class C*)))

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

definition
wf-mdecl :: *prog* ⇒ *qtname* ⇒ *mdecl* ⇒ *bool* **where**
wf-mdecl G C =
(λ(*sig*,*m*).
wf-mhead G (*pid C*) *sig* (*mhead m*) ∧
unique (*lcls* (*mbody m*)) ∧
(∀ (*vn*,*T*)∈*set* (*lcls* (*mbody m*)). *is-acc-type G* (*pid C*) *T*) ∧
(∀ *pn*∈*set* (*pars m*). *table-of* (*lcls* (*mbody m*)) *pn* = *None*) ∧
jumpNestingOkS {*Ret*} (*stmt* (*mbody m*)) ∧
is-class G C ∧
(|*prg*=*G*,*cls*=*C*,*lcl*=*callee-lcl C sig m*|)⊢(*stmt* (*mbody m*))::√ ∧
(∃ *A*. (|*prg*=*G*,*cls*=*C*,*lcl*=*callee-lcl C sig m*|)
⊢ *parameters m* »⟨*stmt* (*mbody m*)⟩» *A*
∧ *Result* ∈ *nrm A*))

lemma *callee-lcl-VNam-simp* [*simp*]:

callee-lcl C sig m (*EName* (*VNam v*))
  = ((*table-of* (*lcls* (*mbody m*)))(*pars m* [↦] *parTs sig*)) *v*
by (*simp add*: *callee-lcl-def*)

lemma *callee-lcl-Res-simp* [*simp*]:
callee-lcl C sig m (*EName Res*) = *Some* (*resTy m*)
by (*simp add*: *callee-lcl-def*)

lemma *callee-lcl-This-simp* [*simp*]:
callee-lcl C sig m (*This*) = (*if is-static m then None else Some* (*Class C*))
by (*simp add*: *callee-lcl-def*)

lemma *callee-lcl-This-static-simp*:
is-static m ⟹ *callee-lcl C sig m* (*This*) = *None*
by *simp*

lemma *callee-lcl-This-not-static-simp*:
¬ *is-static m* ⟹ *callee-lcl C sig m* (*This*) = *Some* (*Class C*)
by *simp*

lemma *wf-mheadI*:
⟦*length* (*parTs sig*) = *length* (*pars m*); ∀ *T*∈*set* (*parTs sig*). *is-acc-type G P T*;
  is-acc-type G P (*resTy m*); *distinct* (*pars m*)⟧ ⟹
  wf-mhead G P sig m
apply (*unfold wf-mhead-def*)
apply (*simp* (*no-asm-simp*))
done

lemma *wf-mdeclI*: ⟦
  wf-mhead G (*pid C*) *sig* (*mhead m*); *unique* (*lcls* (*mbody m*));
  (∀ *pn*∈*set* (*pars m*). *table-of* (*lcls* (*mbody m*)) *pn* = *None*);
  ∀ (*vn*,*T*)∈*set* (*lcls* (*mbody m*)). *is-acc-type G* (*pid C*) *T*;
  jumpNestingOkS {*Ret*} (*stmt* (*mbody m*));
  is-class G C;
  ⦇*prg*=*G*,*cls*=*C*,*lcl*=*callee-lcl C sig m*⦈⊢(*stmt* (*mbody m*))::√;
  (∃ *A*. ⦇*prg*=*G*,*cls*=*C*,*lcl*=*callee-lcl C sig m*⦈ ⊢ *parameters m* »⟨*stmt* (*mbody m*)⟩» *A*
       ∧ *Result* ∈ *nrm A*)
  ⟧ ⟹
  wf-mdecl G C (*sig*,*m*)
apply (*unfold wf-mdecl-def*)
apply *simp*
done

lemma *wf-mdeclE* [*consumes 1*]:
  ⟦*wf-mdecl G C* (*sig*,*m*);
    ⟦*wf-mhead G* (*pid C*) *sig* (*mhead m*); *unique* (*lcls* (*mbody m*));
     ∀ *pn*∈*set* (*pars m*). *table-of* (*lcls* (*mbody m*)) *pn* = *None*;
     ∀ (*vn*,*T*)∈*set* (*lcls* (*mbody m*)). *is-acc-type G* (*pid C*) *T*;
     jumpNestingOkS {*Ret*} (*stmt* (*mbody m*));
     is-class G C;
     ⦇*prg*=*G*,*cls*=*C*,*lcl*=*callee-lcl C sig m*⦈⊢(*stmt* (*mbody m*))::√;
   (∃ *A*. ⦇*prg*=*G*,*cls*=*C*,*lcl*=*callee-lcl C sig m*⦈⊢ *parameters m* »⟨*stmt* (*mbody m*)⟩» *A*

∧ *Result* ∈ *nrm A*)
⟧ ⟹ *P*
⟧ ⟹ *P*
by (*unfold wf-mdecl-def*) *simp*

lemma *wf-mdeclD1*:
wf-mdecl G C (*sig*,*m*) ⟹
wf-mhead G (*pid C*) *sig* (*mhead m*) ∧ *unique* (*lcls* (*mbody m*)) ∧
(∀ *pn*∈*set* (*pars m*). *table-of* (*lcls* (*mbody m*)) *pn* = *None*) ∧
(∀ (*vn*,*T*)∈*set* (*lcls* (*mbody m*)). *is-acc-type G* (*pid C*) *T*)
apply (*unfold wf-mdecl-def*)
apply *simp*
done

lemma *wf-mdecl-bodyD*:
wf-mdecl G C (*sig*,*m*) ⟹
(∃ *T*. ⦇*prg*=*G*,*cls*=*C*,*lcl*=*callee-lcl C sig m*⦈⊢*Body C* (*stmt* (*mbody m*))::−*T* ∧
G⊢*T*≼(*resTy m*))
apply (*unfold wf-mdecl-def*)
apply *clarify*
apply (*rule-tac x*=(*resTy m*) **in** *exI*)
apply (*unfold wf-mhead-def*)
apply (*auto simp add*: *wf-mhead-def is-acc-type-def intro*: *wt.Body*)
done

lemma *rT-is-acc-type*:
wf-mhead G P sig m ⟹ *is-acc-type G P* (*resTy m*)
apply (*unfold wf-mhead-def*)
apply *auto*
done

well-formed interface declarations

A interface declaration is wellformed if:

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

definition

```
  wf-idecl :: prog  ⇒ idecl ⇒ bool where
 wf-idecl G =
    (λ(I,i).
        ws-idecl G I (isuperIfs i) ∧
        ¬is-class G I ∧
        (∀ (sig,mh)∈set (imethods i). wf-mhead G (pid I) sig mh ∧
                                     ¬is-static mh ∧
                                      accmodi mh = Public) ∧
        unique (imethods i) ∧
        (∀ J∈set (isuperIfs i). is-acc-iface G (pid I) J) ∧
        (table-of (imethods i)
          hiding (methd G Object)
          under  (λ new old. accmodi old ≠ Private)
          entails (λnew old. G⊢resTy new≼resTy old ∧
                           is-static new = is-static old)) ∧
        (set-option ∘ table-of (imethods i)
               hidings Un-tables((λJ.(imethds G J))‘set (isuperIfs i))
               entails (λnew 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
apply (unfold wf-idecl-def)
apply auto
done


lemma wf-idecl-hidings:
wf-idecl G (I, i) ⟹
  (λs. set-option (table-of (imethods i) s))
  hidings Un-tables ((λJ. imethds G J) ‘ set (isuperIfs i))
  entails λnew old. G⊢resTy new≼resTy old
apply (unfold wf-idecl-def o-def)
apply simp
done


lemma wf-idecl-hiding:
wf-idecl G (I, i) ⟹
 (table-of (imethods i)
           hiding (methd G Object)
           under  (λ new old. accmodi old ≠ Private)
           entails (λnew old. G⊢resTy new≼resTy old ∧
                            is-static new = is-static old))
apply (unfold wf-idecl-def)
apply simp
done


lemma wf-idecl-supD:
⟦wf-idecl G (I,i); J ∈ set (isuperIfs i)⟧
 ⟹ is-acc-iface G (pid I) J ∧ (J, I) ∉ (subint1 G)⁺
apply (unfold wf-idecl-def ws-idecl-def)
apply auto
done
```

well-formed class declarations

A class declaration is wellformed if:

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

definition
entails :: ('*a*,'*b*) *table* ⇒ ('*b* ⇒ *bool*) ⇒ *bool* (- *entails* - *20*)
where (*t entails P*) = (∀ *k*. ∀ *x* ∈ *t k*: *P x*)

lemma *entailsD*:
⟦*t entails P*; *t k* = *Some x*⟧ ⟹ *P x*
by (*simp add*: *entails-def*)

lemma *empty-entails*[*simp*]: *Map.empty entails P*
by (*simp add*: *entails-def*)

definition
wf-cdecl :: *prog* ⇒ *cdecl* ⇒ *bool* **where**
wf-cdecl G =
(λ(*C*,*c*).
¬*is-iface G C* ∧
(∀ *I*∈*set* (*superIfs c*). *is-acc-iface G* (*pid C*) *I* ∧
(∀ *s*. ∀ *im* ∈ *imethds G I s*.
(∃ *cm* ∈ *methd G C s*: *G*⊢*resTy cm*≼*resTy im* ∧
¬ *is-static cm* ∧
accmodi im ≤ *accmodi cm*))) ∧
(∀ *f*∈*set* (*cfields c*). *wf-fdecl G* (*pid C*) *f*) ∧ *unique* (*cfields c*) ∧

```
(∀ m∈set (methods c). wf-mdecl G C m) ∧ unique (methods c) ∧
jumpNestingOkS {} (init c) ∧
(∃ A. (|prg=G,cls=C,lcl=Map.empty|)⊢ {} »⟨init c⟩» A) ∧
(|prg=G,cls=C,lcl=Map.empty|)⊢(init c)::√ ∧ ws-cdecl G C (super c) ∧
(C ≠ Object ⟶
      (is-acc-class G (pid C) (super c) ∧
      (table-of (map (λ (s,m). (s,C,m)) (methods c))
       entails (λ new. ∀ old sig.
                (G,sig⊢new overrides_S old
                 ⟶ (G⊢resTy new≼resTy old ∧
                      accmodi old ≤ accmodi new ∧
                      ¬is-static old)) ∧
                (G,sig⊢new hides old
                  ⟶ (accmodi old ≤ accmodi new ∧
                       is-static old))))
      )))
```

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

```
⟦wf-cdecl G (C,c);
  ⟦¬is-iface G C;
   (∀ I∈set (superIfs c). is-acc-iface G (pid C) I ∧
       (∀ s. ∀ im ∈ imethds G I s.
           (∃ cm ∈ methd  G C s: G⊢resTy cm≼resTy im ∧
                                  ¬ is-static cm ∧
                                  accmodi im ≤ accmodi cm)));
    ∀ f∈set (cfields c). wf-fdecl G (pid C) f; unique (cfields c);
    ∀ m∈set (methods c). wf-mdecl G C m; unique (methods c);
    jumpNestingOkS {} (init c);
    ∃ A. (|prg=G,cls=C,lcl=Map.empty|)⊢ {} »⟨init c⟩» A;
    (|prg=G,cls=C,lcl=Map.empty|)⊢(init c)::√;
    ws-cdecl G C (super c);
    (C ≠ Object ⟶
          (is-acc-class G (pid C) (super c) ∧
          (table-of (map (λ (s,m). (s,C,m)) (methods c))
           entails (λ new. ∀ old sig.
                    (G,sig⊢new overrides_S old
                     ⟶ (G⊢resTy new≼resTy old ∧
                          accmodi old ≤ accmodi new ∧
                          ¬is-static old)) ∧
                    (G,sig⊢new hides old
                      ⟶ (accmodi old ≤ accmodi new ∧
                           is-static old))))
          ))⟧ ⟹ P
  ⟧ ⟹ P
```

by (*unfold wf-cdecl-def*) *simp*

lemma *wf-cdecl-unique*:
wf-cdecl G (*C*,*c*) ⟹ *unique* (*cfields c*) ∧ *unique* (*methods c*)
apply (*unfold wf-cdecl-def*)
apply *auto*
done

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

apply (*unfold wf-cdecl-def*)
apply *auto*
done

lemma *wf-cdecl-mdecl*:
⟦*wf-cdecl G* (*C*,*c*); *m*∈*set* (*methods c*)⟧ ⟹ *wf-mdecl G C m*
apply (*unfold wf-cdecl-def*)
apply *auto*
done

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*))
apply (*unfold wf-cdecl-def*)
apply *auto*
done

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))))
apply (*unfold wf-cdecl-def ws-cdecl-def*)
apply *auto*
done

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*
apply (*drule* (*1*) *wf-cdecl-supD*)
apply (*clarify*)
apply (*drule entailsD*)
apply (*blast intro*: *table-of-map-SomeI*)
apply (*drule-tac x*=*old* **in** *spec*)
apply (*auto dest*: *overrides-eq-sigD simp add*: *msig-def*)
done

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*

$\rrbracket \Longrightarrow$ *accmodi old* $\leq$ *access newM* $\wedge$
is-static old
apply (*drule* (*1*) *wf-cdecl-supD*)
apply (*clarify*)
apply (*drule entailsD*)
apply (*blast intro*: *table-of-map-SomeI*)
apply (*drule-tac x=old* **in** *spec*)
apply (*auto dest*: *hides-eq-sigD simp add*: *msig-def*)
done

lemma *wf-cdecl-wt-init*:
wf-cdecl G (*C*, *c*) $\Longrightarrow$ (|*prg=G*,*cls=C*,*lcl=Map.empty*|)⊢*init c*::√
apply (*unfold wf-cdecl-def*)
apply *auto*
done

well-formed programs

A program declaration is wellformed if:

- the class ObjectC of Object is defined
- every method of Object has an access modifier distinct from Package. This is necessary since every interface automatically inherits from Object. We must know, that every time a Object method is "overriden" by an interface method this is also overriden by the class implementing the the interface (see *implement-dynmethd and class-mheadsD*)
- all standard Exceptions are defined
- all defined interfaces are wellformed
- all defined classes are wellformed

definition
wf-prog :: *prog* ⇒ *bool* **where**
wf-prog G = (*let is* = *ifaces G*; *cs* = *classes G in*
ObjectC ∈ *set cs* ∧
(∀ *m*∈*set Object-mdecls*. *accmodi m* ≠ *Package*) ∧
(∀ *xn*. *SXcptC xn* ∈ *set cs*) ∧
(∀ *i*∈*set is*. *wf-idecl G i*) ∧ *unique is* ∧
(∀ *c*∈*set cs*. *wf-cdecl G c*) ∧ *unique cs*)

lemma *wf-prog-idecl*: ⟦*iface G I* = *Some i*; *wf-prog G*⟧ ⟹ *wf-idecl G* (*I*,*i*)
apply (*unfold wf-prog-def Let-def*)
apply *simp*
apply (*fast dest*: *map-of-SomeD*)
done

lemma *wf-prog-cdecl*: ⟦*class G C* = *Some c*; *wf-prog G*⟧ ⟹ *wf-cdecl G* (*C*,*c*)
apply (*unfold wf-prog-def Let-def*)
apply *simp*
apply (*fast dest*: *map-of-SomeD*)
done

lemma *wf-prog-Object-mdecls*:

```
wf-prog G ⟹ (∀ m∈set Object-mdecls. accmodi m ≠ Package)
apply (unfold wf-prog-def Let-def)
apply simp
done
```

```
lemma wf-prog-acc-superD:
 ⟦wf-prog G; class G C = Some c; C ≠ Object ⟧
  ⟹ is-acc-class G (pid C) (super c)
by (auto dest: wf-prog-cdecl wf-cdecl-supD)
```

```
lemma wf-ws-prog [elim!,simp]: wf-prog G ⟹ ws-prog G
apply (unfold wf-prog-def Let-def)
apply (rule ws-progI)
apply  (simp-all (no-asm))
apply  (auto simp add: is-acc-class-def is-acc-iface-def
            dest!: wf-idecl-supD wf-cdecl-supD )+
done
```

```
lemma class-Object [simp]:
wf-prog G ⟹
  class G Object = Some (|access=Public,cfields=[],methods=Object-mdecls,
                          init=Skip,super=undefined,superIfs=[]|)
apply (unfold wf-prog-def Let-def ObjectC-def)
apply (fast dest!: map-of-SomeI)
done
```

```
lemma methd-Object[simp]: wf-prog G ⟹ methd G Object =
  table-of (map (λ(s,m). (s, Object, m)) Object-mdecls)
apply (subst methd-rec)
apply (auto simp add: Let-def)
done
```

```
lemma wf-prog-Object-methd:
⟦wf-prog G; methd G Object sig = Some m⟧ ⟹ accmodi m ≠ Package
by (auto dest!: wf-prog-Object-mdecls) (auto dest!: map-of-SomeD)
```

```
lemma wf-prog-Object-is-public[intro]:
 wf-prog G ⟹ is-public G Object
by (auto simp add: is-public-def dest: class-Object)
```

```
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=[]|)
apply (unfold wf-prog-def Let-def SXcptC-def)
apply (fast dest!: map-of-SomeI)
done
```

lemma *wf-ObjectC* [*simp*]:
wf-cdecl G ObjectC = (¬*is-iface G Object* ∧ *Ball* (*set Object-mdecls*)
(*wf-mdecl G Object*) ∧ *unique Object-mdecls*)
apply (*unfold wf-cdecl-def ws-cdecl-def ObjectC-def*)
apply (*auto intro*: *da.Skip*)
done

lemma *Object-is-class* [*simp*,*elim*!]: *wf-prog G* ⟹ *is-class G Object*
apply (*simp* (*no-asm-simp*))
done

lemma *Object-is-acc-class* [*simp*,*elim*!]: *wf-prog G* ⟹ *is-acc-class G S Object*
apply (*simp* (*no-asm-simp*) *add*: *is-acc-class-def is-public-def*
accessible-in-RefT-simp)
done

lemma *SXcpt-is-class* [*simp*,*elim*!]: *wf-prog G* ⟹ *is-class G* (*SXcpt xn*)
apply (*simp* (*no-asm-simp*))
done

lemma *SXcpt-is-acc-class* [*simp*,*elim*!]:
wf-prog G ⟹ *is-acc-class G S* (*SXcpt xn*)
apply (*simp* (*no-asm-simp*) *add*: *is-acc-class-def is-public-def*
accessible-in-RefT-simp)
done

lemma *fields-Object* [*simp*]: *wf-prog G* ⟹ *DeclConcepts.fields G Object* = []
by (*force intro*: *fields-emptyI*)

lemma *accfield-Object* [*simp*]:
wf-prog G ⟹ *accfield G S Object* = *Map.empty*
apply (*unfold accfield-def*)
apply (*simp* (*no-asm-simp*) *add*: *Let-def*)
done

lemma *fields-Throwable* [*simp*]:
wf-prog G ⟹ *DeclConcepts.fields G* (*SXcpt Throwable*) = []
by (*force intro*: *fields-emptyI*)

lemma *fields-SXcpt* [*simp*]: *wf-prog G* ⟹ *DeclConcepts.fields G* (*SXcpt xn*) = []
apply (*case-tac xn* = *Throwable*)
apply (*simp* (*no-asm-simp*))
by (*force intro*: *fields-emptyI*)

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*
apply (*erule* (*2*) *widen-trans*)
done

lemma *Xcpt-subcls-Throwable* [*simp*]:
wf-prog G ⟹ *G*⊢*SXcpt xn*$\preceq_C$ *SXcpt Throwable*
apply (*rule SXcpt-subcls-Throwable-lemma*)
apply *auto*
done

lemma *unique-fields*:
⟦*is-class G C*; *wf-prog G*⟧ ⟹ *unique* (*DeclConcepts.fields G C*)
apply (*erule ws-unique-fields*)
apply (*erule wf-ws-prog*)
apply (*erule* (*1*) *wf-prog-cdecl* [*THEN wf-cdecl-unique* [*THEN conjunct1*]])
done

lemma *fields-mono*:
⟦*table-of* (*DeclConcepts.fields G C*) *fn* = *Some f*; *G*⊢*D*$\preceq_C$ *C*;
is-class G D; *wf-prog G*⟧
⟹ *table-of* (*DeclConcepts.fields G D*) *fn* = *Some f*
apply (*rule map-of-SomeI*)
apply (*erule* (*1*) *unique-fields*)
apply (*erule* (*1*) *map-of-SomeD* [*THEN fields-mono-lemma*])
apply (*erule wf-ws-prog*)
done

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*)
apply (*frule wf-ws-prog*)
apply (*force dest*: *fields-declC* [*THEN conjunct1*]
wf-prog-cdecl [*THEN wf-cdecl-fdecl*]
simp add: *wf-fdecl-def2 is-acc-type-def*)
done

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*
apply (*frule wf-ws-prog*)
apply (*drule* (*2*) *imethds-declI* [*THEN conjunct1*])
apply *clarify*
apply (*frule-tac I*=(*decliface m*) **in** *wf-prog-idecl*,*assumption*)
apply (*drule wf-idecl-mhead*)
apply (*erule map-of-SomeD*)
apply (*cases m*, *simp*)
done

lemma *methd-wf-mdecl*:
⟦*methd G C sig* = *Some m*; *wf-prog G*; *class G C* = *Some y*⟧ ⟹
G⊢*C*$\preceq_C$ (*declclass m*) ∧ *is-class G* (*declclass m*) ∧
wf-mdecl G (*declclass m*) (*sig*,(*mthd m*))
apply (*frule wf-ws-prog*)
apply (*drule* (*1*) *methd-declC*)
apply *fast*
apply *clarsimp*

apply (*frule* (*1*) *wf-prog-cdecl*, *erule wf-cdecl-mdecl*, *erule map-of-SomeD*)
done

lemma *methd-rT-is-type*:
⟦*wf-prog G*;*methd G C sig* = *Some m*;
  class G C = *Some y*⟧
⟹ *is-type G* (*resTy m*)
apply (*drule* (*2*) *methd-wf-mdecl*)
apply *clarify*
apply (*drule wf-mdeclD1*)
apply *clarify*
apply (*drule rT-is-acc-type*)
apply (*cases m*, *simp add*: *is-acc-type-def*)
done

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*)
by (*auto simp add*: *accmethd-def*
    intro: *methd-rT-is-type*)

lemma *methd-Object-SomeD*:
⟦*wf-prog G*;*methd G Object sig* = *Some m*⟧
 ⟹ *declclass m* = *Object*
by (*auto dest*: *class-Object simp add*: *methd-rec*)

lemmas *iface-rec-induct′* = *iface-rec.induct* [*of %x y z. P x y*] **for** *P*

lemma *wf-imethdsD*:
⟦*im* ∈ *imethds G I sig*;*wf-prog G*; *is-iface G I*⟧
 ⟹ ¬*is-static im* ∧ *accmodi im* = *Public*
proof −
 assume *asm*: *wf-prog G is-iface G I im* ∈ *imethds G I sig*

 have *wf-prog G* ⟶
    (∀ *i im*. *iface G I* = *Some i* ⟶ *im* ∈ *imethds G I sig*
       ⟶ ¬*is-static im* ∧ *accmodi im* = *Public*) (**is** *?P G I*)
 proof (*induct G I rule*: *iface-rec-induct′*, *intro allI impI*)
  fix *G I i im*
  assume *hyp*: ⋀ *i J*. *iface G I* = *Some i* ⟹ *ws-prog G* ⟹ *J* ∈ *set* (*isuperIfs i*)
      ⟹ *?P G J*
  assume *wf*: *wf-prog G* **and** *if-I*: *iface G I* = *Some i* **and**
    im: *im* ∈ *imethds G I sig*
  show ¬*is-static im* ∧ *accmodi im* = *Public*
  proof −
   let *?inherited* = *Un-tables* (*imethds G* ‘ *set* (*isuperIfs i*))
   let *?new* = (*set-option* ∘ *table-of* (*map* (λ(*s*, *mh*). (*s*, *I*, *mh*)) (*imethods i*)))
   from *if-I wf im* **have** *imethds*:*im* ∈ (*?inherited* ⊕⊕ *?new*) *sig*
    by (*simp add*: *imethds-rec*)
   from *wf if-I* **have**
    wf-supI: ∀ *J*. *J* ∈ *set* (*isuperIfs i*) ⟶ (∃ *j*. *iface G J* = *Some j*)

by (*blast dest*: *wf-prog-idecl wf-idecl-supD is-acc-ifaceD*)
from *wf if-I* **have**
$\forall$ *im* $\in$ *set* (*imethods i*). $\neg$ *is-static im* $\wedge$ *accmodi im* = *Public*
by (*auto dest*!: *wf-prog-idecl wf-idecl-mhead*)
then have *new-ok*: $\forall$ *im. table-of* (*imethods i*) *sig* = *Some im*
$\longrightarrow$ $\neg$ *is-static im* $\wedge$ *accmodi im* = *Public*
by (*auto dest*!: *table-of-Some-in-set*)
show *?thesis*
proof (*cases ?new sig* = {})
case *True*
from *True wf wf-supI if-I imethds hyp*
show *?thesis* **by** (*auto simp del*: *split-paired-All*)
next
case *False*
from *False wf wf-supI if-I imethds new-ok hyp*
show *?thesis* **by** (*auto dest*: *wf-idecl-hidings hidings-entailsD*)
qed
qed
qed
with *asm* **show** *?thesis* **by** (*auto simp del*: *split-paired-All*)
qed

lemma *wf-prog-hidesD*:
assumes *hides*: *G* $\vdash$*new hides old* **and** *wf*: *wf-prog G*
shows
accmodi old $\leq$ *accmodi new* $\wedge$
is-static old
proof −
from *hides*
obtain *c* **where**
clsNew: *class G* (*declclass new*) = *Some c* **and**
neqObj: *declclass new* $\neq$ *Object*
by (*auto dest*: *hidesD declared-in-classD*)
with *hides* **obtain** *newM oldM* **where**
newM: *table-of* (*methods c*) (*msig new*) = *Some newM* **and**
new: *new* = (*declclass new*,(*msig new*),*newM*) **and**
old: *old* = (*declclass old*,(*msig old*),*oldM*) **and**
msig new = *msig old*
by (*cases new*,*cases old*)
(*auto dest*: *hidesD*
simp add: *cdeclaredmethd-def declared-in-def*)
with *hides*
have *hides*′:
G,(*msig new*)$\vdash$(*declclass new*,*newM*) *hides* (*declclass old*,*oldM*)
by *auto*
from *clsNew wf*
have *wf-cdecl G* (*declclass new*,*c*) **by** (*blast intro*: *wf-prog-cdecl*)
note *wf-cdecl-hides-SomeD* [*OF this neqObj newM hides*′]
with *new old*
show *?thesis*
by (*cases new*, *cases old*) *auto*
qed

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 −
  from stat-override
  obtain c where
    clsNew: class G (declclass new) = Some c and
    neqObj: declclass new ≠ Object
    by (auto dest: stat-overrides-commonD declared-in-classD)
  with stat-override obtain newM oldM where
    newM: table-of (methods c) (msig new) = Some newM and
     new: new = (declclass new,(msig new),newM) and
     old: old = (declclass old,(msig old),oldM) and
          msig new = msig old
    by (cases new,cases old)
       (auto dest: stat-overrides-commonD
         simp add: cdeclaredmethd-def declared-in-def)
  with stat-override
  have stat-override':
        G,(msig new)⊢(declclass new,newM) overrides_S (declclass old,oldM)
    by auto
  from clsNew wf
  have wf-cdecl G (declclass new,c) by (blast intro: wf-prog-cdecl)
  note wf-cdecl-overrides-SomeD [OF this neqObj newM stat-override']
  with new old
  show ?thesis
    by (cases new, cases old) auto
qed


lemma static-to-dynamic-overriding:
  assumes stat-override: G⊢new overrides_S old and wf : wf-prog G
  shows G⊢new overrides old
proof −
  from stat-override
  show ?thesis (is ?Overrides new old)
  proof (induct)
    case (Direct new old superNew)
    then have stat-override:G⊢new overrides_S old
      by (rule stat-overridesR.Direct)
    from stat-override wf
    have resTy-widen: G⊢resTy new≼resTy old and
      not-static-old: ¬ is-static old
      by (auto dest: wf-prog-stat-overridesD)
    have not-private-new: accmodi new ≠ Private
    proof −
      from stat-override
      have accmodi old ≠ Private
        by (rule no-Private-stat-override)
      moreover
      from stat-override wf
      have accmodi old ≤ accmodi new
        by (auto dest: wf-prog-stat-overridesD)
      ultimately
```

show *?thesis*
by (*auto dest*: *acc-modi-bottom*)
qed
with *Direct resTy-widen not-static-old*
show *?Overrides new old*
by (*auto intro*: *overridesR.Direct stat-override-declclasses-relation*)
next
case (*Indirect new inter old*)
then show *?Overrides new old*
by (*blast intro*: *overridesR.Indirect*)
qed
qed

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 −
from *wf* **have** *ws*: *ws-prog G* **by** *auto*
from *old-declared* **have** *iscls-declC-old*: *is-class G* (*declclass old*)
by (*auto simp add*: *declared-in-def cdeclaredmethd-def*)
from *subcls* **have** *iscls-C*: *is-class G C*
by (*blast dest*: *subcls-is-class*)
from *iscls-C ws old-inheritable subcls*
show *?thesis* (**is** *?P C old*)
proof (*induct rule*: *ws-class-induct'*)
case *Object*
assume $G\vdash$*Object*$\prec_C$ *declclass old*
then show *?P Object old*
by *blast*
next
case (*Subcls C c*)
assume *cls-C*: *class G C* = *Some c* **and**
neq-C-Obj: *C* $\neq$ *Object* **and**
hyp: ⟦$G \vdash$*Method old inheritable-in pid* (*super c*);
$G\vdash$*super c*$\prec_C$ *declclass old*⟧ $\Longrightarrow$ *?P* (*super c*) *old* **and**
inheritable: $G \vdash$*Method old inheritable-in pid C* **and**
subclsC: $G\vdash C\prec_C$ *declclass old*
from *cls-C neq-C-Obj*
have *super*: $G\vdash C \prec_C 1$ *super c*
by (*rule subcls1I*)
from *wf cls-C neq-C-Obj*
have *accessible-super*: $G\vdash$(*Class* (*super c*)) *accessible-in* (*pid C*)
by (*auto dest*: *wf-prog-cdecl wf-cdecl-supD is-acc-classD*)
{
fix *old*
assume *member-super*: $G\vdash$*Method old member-of* (*super c*)
assume *inheritable*: $G \vdash$*Method old inheritable-in pid C*
assume *instance-method*: $\neg$ *is-static old*
from *member-super*
have *old-declared*: $G\vdash$*Method old declared-in* (*declclass old*)
by (*cases old*) (*auto dest*: *member-of-declC*)
have *?P C old*

```
      proof (cases G⊢mid (msig old) undeclared-in C)
        case True
        with inheritable super accessible-super member-super
        have G⊢Method old member-of C
          by (cases old) (auto intro: members.Inherited)
        then show ?thesis
          by auto
      next
        case False
        then obtain new-member where
             G⊢new-member declared-in C and
             mid (msig old) = memberid new-member
          by (auto dest: not-undeclared-declared)
        then obtain new where
                  new: G⊢Method new declared-in C and
               eq-sig: msig old = msig new and
             declC-new: declclass new = C
          by (cases new-member) auto
        then have member-new: G⊢Method new member-of C
          by (cases new) (auto intro: members.Immediate)
        from declC-new super member-super
        have subcls-new-old: G⊢declclass new ≺C declclass old
          by (auto dest!: member-of-subclseq-declC
                    dest: r-into-trancl intro: trancl-rtrancl-trancl)
        show ?thesis
        proof (cases is-static new)
          case False
          with eq-sig declC-new new old-declared inheritable
               super member-super subcls-new-old
          have G⊢new overridesS old
            by (auto intro!: stat-overridesR.Direct)
          with member-new show ?thesis
            by blast
        next
          case True
          with eq-sig declC-new subcls-new-old new old-declared inheritable
          have G⊢new hides old
            by (auto intro: hidesI)
          with wf
          have is-static old
            by (blast dest: wf-prog-hidesD)
          with instance-method
          show ?thesis
            by (contradiction)
        qed
      qed
    } note hyp-member-super = this
    from subclsC cls-C
    have G⊢(super c)≼C declclass old
      by (rule subcls-superD)
    then
    show ?P C old
    proof (cases rule: subclseq-cases)
      case Eq
      assume super c = declclass old
      with old-declared
      have G⊢Method old member-of (super c)
        by (cases old) (auto intro: members.Immediate)
      with inheritable instance-method
```

```
      show ?thesis
        by (blast dest: hyp-member-super)
    next
      case Subcls
      assume G⊢super c≺C declclass old
      moreover
      from inheritable accmodi-old
      have G ⊢Method old inheritable-in pid (super c)
        by (cases accmodi old) (auto simp add: inheritable-in-def)
      ultimately
      have ?P (super c) old
        by (blast dest: hyp)
      then show ?thesis
      proof
        assume G ⊢Method old member-of super c
        with inheritable instance-method
        show ?thesis
          by (blast dest: hyp-member-super)
      next
        assume ∃ new. G ⊢ new overridesS old ∧ G ⊢Method new member-of super c
        then obtain super-new where
          super-new-override: G ⊢ super-new overridesS old and
            super-new-member: G ⊢Method super-new member-of super c
          by blast
        from super-new-override wf
        have accmodi old ≤ accmodi super-new
          by (auto dest: wf-prog-stat-overridesD)
        with inheritable accmodi-old
        have G ⊢Method super-new inheritable-in pid C
          by (auto simp add: inheritable-in-def
                     split: acc-modi.splits
                      dest: acc-modi-le-Dests)
        moreover
        from super-new-override
        have ¬ is-static super-new
          by (auto dest: stat-overrides-commonD)
        moreover
        note super-new-member
        ultimately have ?P C super-new
          by (auto dest: hyp-member-super)
        then show ?thesis
        proof
          assume G ⊢Method super-new member-of C
          with super-new-override
          show ?thesis
            by blast
        next
          assume ∃ new. G ⊢ new overridesS super-new ∧
                  G ⊢Method new member-of C
          with super-new-override show ?thesis
            by (blast intro: stat-overridesR.Indirect)
        qed
      qed
    qed
  qed
qed
```

lemma *non-Package-instance-method-inheritance-cases*:

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*
obtains (*Inheritance*) $G \vdash$*Method old member-of C*
| (*Overriding*) *new* **where** $G \vdash$ *new overrides*$_S$ *old* **and** $G \vdash$*Method new member-of C*
proof −
from *old-inheritable accmodi-old instance-method subcls old-declared wf*
Inheritance Overriding
show *thesis*
by (*auto dest*: *non-Package-instance-method-inheritance*)
qed

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 −
from *dyn-override accmodi-old*
show *?thesis* (**is** *?Overrides new old*)
proof (*induct rule*: *overridesR.induct*)
case (*Direct new old*)
assume *new-declared*: $G \vdash$*Method new declared-in declclass new*
assume *eq-sig-new-old*: *msig new* = *msig old*
assume *subcls-new-old*: $G \vdash$*declclass new* $\prec_C$ *declclass old*
assume $G \vdash$*Method old inheritable-in pid* (*declclass new*) **and**
accmodi old $\neq$ *Package* **and**
$\neg$ *is-static old* **and**
$G \vdash$*declclass new*$\prec_C$ *declclass old* **and**
$G \vdash$*Method old declared-in declclass old*
from *this wf*
show *?Overrides new old*
proof (*cases rule*: *non-Package-instance-method-inheritance-cases*)
case *Inheritance*
assume $G \vdash$*Method old member-of declclass new*
then have $G \vdash$*mid* (*msig old*) *undeclared-in declclass new*
proof *cases*
case *Immediate*
with *subcls-new-old wf* **show** *?thesis*
by (*auto dest*: *subcls-irrefl*)
next
case *Inherited*
then show *?thesis*
by (*cases old*) *auto*
qed
with *eq-sig-new-old new-declared*
show *?thesis*
by (*cases old*,*cases new*) (*auto dest*!: *declared-not-undeclared*)
next
case (*Overriding new′*)
assume *stat-override-new′*: $G \vdash$ *new′ overrides*$_S$ *old*
then have *msig new′* = *msig old*
by (*auto dest*: *stat-overrides-commonD*)
with *eq-sig-new-old* **have** *eq-sig-new-new′*: *msig new*=*msig new′*
by *simp*

```
    assume G ⊢Method new' member-of declclass new
    then show ?thesis
    proof (cases)
      case Immediate
      then have declC-new: declclass new' = declclass new
        by auto
      from Immediate
      have G⊢Method new' declared-in declclass new
        by (cases new') auto
      with new-declared eq-sig-new-new' declC-new
      have new=new'
        by (cases new, cases new') (auto dest: unique-declared-in)
      with stat-override-new'
      show ?thesis
        by simp
    next
      case Inherited
      then have G⊢mid (msig new') undeclared-in declclass new
        by (cases new') (auto)
      with eq-sig-new-new' new-declared
      show ?thesis
        by (cases new,cases new') (auto dest!: declared-not-undeclared)
    qed
  qed
next
  case (Indirect new inter old)
  assume accmodi-old: accmodi old ≠ Package
  assume accmodi old ≠ Package ⟹ G ⊢ inter overrides_S old
  with accmodi-old
  have stat-override-inter-old: G ⊢ inter overrides_S old
    by blast
  moreover
  assume hyp-inter: accmodi inter ≠ Package ⟹ G ⊢ new overrides_S inter
  moreover
  have accmodi inter ≠ Package
  proof −
    from stat-override-inter-old wf
    have accmodi old ≤ accmodi inter
      by (auto dest: wf-prog-stat-overridesD)
    with stat-override-inter-old accmodi-old
    show ?thesis
      by (auto dest!: no-Private-stat-override
               split: acc-modi.splits
               dest: acc-modi-le-Dests)
  qed
  ultimately show ?Overrides new old
    by (blast intro: stat-overridesR.Indirect)
qed
qed


lemma wf-prog-dyn-override-prop:
  assumes dyn-override: G ⊢ new overrides old and
                wf: wf-prog G
  shows accmodi old ≤ accmodi new
proof (cases accmodi old = Package)
  case True
  note old-Package = this
  show ?thesis
```

proof (*cases accmodi old* ≤ *accmodi new*)
  case *True* **then show** *?thesis* **.**
next
  case *False*
  with *old-Package*
  have *accmodi new* = *Private*
    by (*cases accmodi new*) (*auto simp add*: *le-acc-def less-acc-def*)
  with *dyn-override*
  show *?thesis*
    by (*auto dest*: *overrides-commonD*)
qed
next
  case *False*
  with *dyn-override wf*
  have $G \vdash$ *new overrides*$_S$ *old*
    by (*blast intro*: *dynamic-to-static-overriding*)
  with *wf*
  show *?thesis*
    by (*blast dest*: *wf-prog-stat-overridesD*)
qed

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 (*cases accmodi old*)
  case *Private*
  with *dyn-override* **show** *?thesis*
    by (*simp add*: *no-Private-override*)
next
  case *Package*
  then show *?thesis* **.**
next
  case *Protected*
  with *dyn-override wf*
  have $G \vdash$ *new overrides*$_S$ *old*
    by (*auto intro*: *dynamic-to-static-overriding*)
  with *wf*
  have *accmodi old* ≤ *accmodi new*
    by (*auto dest*: *wf-prog-stat-overridesD*)
  with *Protected accmodi-new*
  show *?thesis*
    by (*simp add*: *less-acc-def le-acc-def*)
next
  case *Public*
  with *dyn-override wf*
  have $G \vdash$ *new overrides*$_S$ *old*
    by (*auto intro*: *dynamic-to-static-overriding*)
  with *wf*
  have *accmodi old* ≤ *accmodi new*
    by (*auto dest*: *wf-prog-stat-overridesD*)
  with *Public accmodi-new*
  show *?thesis*
    by (*simp add*: *less-acc-def le-acc-def*)
qed

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 −
from *dyn-override accmodi-old accmodi-new*
show *?thesis* (**is** *?EqPid old new*)
proof (*induct rule*: *overridesR.induct*)
case (*Direct new old*)
assume *accmodi old* = *Package*
$G \vdash$*Method old inheritable-in pid* (*declclass new*)
then show *pid* (*declclass old*) = *pid* (*declclass new*)
by (*auto simp add*: *inheritable-in-def*)
next
case (*Indirect new inter old*)
assume *accmodi-old*: *accmodi old* = *Package* **and**
accmodi-new: *accmodi new* = *Package*
assume $G \vdash$ *new overrides inter*
with *accmodi-new wf*
have *accmodi inter* = *Package*
by (*auto intro*: *overrides-Package-old*)
with *Indirect*
show *pid* (*declclass old*) = *pid* (*declclass new*)
by *auto*
qed
qed

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 −
from *dyn-override accmodi-old outside-pack*
show *?thesis* (**is** *?P new old*)
proof (*induct rule*: *overridesR.induct*)
case (*Direct new old*)
assume *accmodi-old*: *accmodi old* = *Package*
assume *outside-pack*: *pid* (*declclass old*) $\neq$ *pid* (*declclass new*)
assume $G \vdash$*Method old inheritable-in pid* (*declclass new*)
with *accmodi-old*
have *pid* (*declclass old*) = *pid* (*declclass new*)
by (*simp add*: *inheritable-in-def*)
with *outside-pack*
show *?P new old*
by (*contradiction*)
next
case (*Indirect new inter old*)
assume *accmodi-old*: *accmodi old* = *Package*
assume *outside-pack*: *pid* (*declclass old*) $\neq$ *pid* (*declclass new*)
assume *override-new-inter*: $G \vdash$ *new overrides inter*
assume *override-inter-old*: $G \vdash$ *inter overrides old*
assume *hyp-new-inter*: $[\![$*accmodi inter* = *Package*;

```
                        pid (declclass inter) ≠ pid (declclass new)⟧
                        ⟹ ?P new inter
  assume hyp-inter-old: ⟦accmodi old = Package;
                        pid (declclass old) ≠ pid (declclass inter)⟧
                        ⟹ ?P inter old
  show ?P new old
  proof (cases pid (declclass old) = pid (declclass inter))
    case True
    note same-pack-old-inter = this
    show ?thesis
    proof (cases pid (declclass inter) = pid (declclass new))
      case True
      with same-pack-old-inter outside-pack
      show ?thesis
        by auto
    next
      case False
      note diff-pack-inter-new = this
      show ?thesis
      proof (cases accmodi inter = Package)
        case True
        with diff-pack-inter-new hyp-new-inter
        obtain newinter where
          over-new-newinter: G ⊢ new overrides newinter and
          over-newinter-inter: G ⊢ newinter overrides inter and
          eq-pid: pid (declclass inter) = pid (declclass newinter) and
          accmodi-newinter: Protected ≤ accmodi newinter
          by auto
        from over-newinter-inter override-inter-old
        have G⊢newinter overrides old
          by (rule overridesR.Indirect)
        moreover
        from eq-pid same-pack-old-inter
        have pid (declclass old) = pid (declclass newinter)
          by simp
        moreover
        note over-new-newinter accmodi-newinter
        ultimately show ?thesis
          by blast
      next
        case False
        with override-new-inter
        have Protected ≤ accmodi inter
          by (cases accmodi inter) (auto dest: no-Private-override)
        with override-new-inter override-inter-old same-pack-old-inter
        show ?thesis
          by blast
      qed
    qed
  next
    case False
    with accmodi-old hyp-inter-old
    obtain newinter where
      over-inter-newinter: G ⊢ inter overrides newinter and
        over-newinter-old: G ⊢ newinter overrides old and
              eq-pid: pid (declclass old) = pid (declclass newinter) and
      accmodi-newinter: Protected ≤ accmodi newinter
      by auto
    from override-new-inter over-inter-newinter
```

have *G* ⊢ *new overrides newinter*
by (*rule overridesR.Indirect*)
with *eq-pid over-newinter-old accmodi-newinter*
show *?thesis*
by *blast*
qed
qed
qed

lemmas *class-rec-induct'* = *class-rec.induct* [*of %x y z w. P x y*] **for** *P*

lemma *declclass-widen*[*rule-format*]:
wf-prog G
⟶ (∀ *c m. class G C* = *Some c* ⟶ *methd G C sig* = *Some m*
⟶ *G*⊢*C* $\preceq_C$ *declclass m*) (**is** *?P G C*)
proof (*induct G C rule*: *class-rec-induct'*, *intro allI impI*)
fix *G C c m*
assume *Hyp*: ⋀*c. class G C* = *Some c* ⟹ *ws-prog G* ⟹ *C* ≠ *Object*
⟹ *?P G* (*super c*)
assume *wf*: *wf-prog G* **and** *cls-C*: *class G C* = *Some c* **and**
m: *methd G C sig* = *Some m*
show *G*⊢*C*$\preceq_C$ *declclass m*
proof (*cases C*=*Object*)
case *True*
with *wf m* **show** *?thesis* **by** (*simp add*: *methd-Object-SomeD*)
next
let *?filter*=*filter-tab* (λ*sig m. G*⊢*C inherits method sig m*)
let *?table* = *table-of* (*map* (λ(*s*, *m*). (*s*, *C*, *m*)) (*methods c*))
case *False*
with *cls-C wf m*
have *methd-C*: (*?filter* (*methd G* (*super c*)) ++ *?table*) *sig* = *Some m*
by (*simp add*: *methd-rec*)
show *?thesis*
proof (*cases ?table sig*)
case *None*
from *this methd-C* **have** *?filter* (*methd G* (*super c*)) *sig* = *Some m*
by *simp*
moreover
from *wf cls-C False* **obtain** *sup* **where** *class G* (*super c*) = *Some sup*
by (*blast dest*: *wf-prog-cdecl wf-cdecl-supD is-acc-class-is-class*)
moreover note *wf False cls-C*
ultimately have *G*⊢*super c* $\preceq_C$ *declclass m*
by (*auto intro*: *Hyp* [*rule-format*])
moreover from *cls-C False* **have** *G*⊢*C* $\prec_C$*1 super c* **by** (*rule subcls1I*)
ultimately show *?thesis* **by** − (*rule rtrancl-into-rtrancl2*)
next
case *Some*
from *this wf False cls-C methd-C* **show** *?thesis* **by** *auto*
qed
qed
qed

lemma *declclass-methd-Object*:
⟦*wf-prog G*; *methd G Object sig* = *Some m*⟧ ⟹ *declclass m* = *Object*
by *auto*

lemma *methd-declaredD*:
⟦*wf-prog G*; *is-class G C*;*methd G C sig* = *Some m*⟧
 ⟹ *G*⊢(*mdecl* (*sig*,*mthd m*)) *declared-in* (*declclass m*)
proof −
  assume *wf*: *wf-prog G*
  then have *ws*: *ws-prog G* **..**
  assume *clsC*: *is-class G C*
  from *clsC ws*
  show *methd G C sig* = *Some m*
      ⟹ *G*⊢(*mdecl* (*sig*,*mthd m*)) *declared-in* (*declclass m*)
  proof (*induct C rule*: *ws-class-induct′*)
    case *Object*
    assume *methd G Object sig* = *Some m*
    with *wf* **show** *?thesis*
      by − (*rule method-declared-inI*, *auto*)
  next
    case *Subcls*
    fix *C c*
    assume *clsC*: *class G C* = *Some c*
    and *m*: *methd G C sig* = *Some m*
    and *hyp*: *methd G* (*super c*) *sig* = *Some m* ⟹ *?thesis*
    let *?newMethods* = *table-of* (*map* (λ(*s*, *m*). (*s*, *C*, *m*)) (*methods c*))
    show *?thesis*
    proof (*cases ?newMethods sig*)
      case *None*
      from *None ws clsC m hyp*
      show *?thesis* **by** (*auto intro*: *method-declared-inI simp add*: *methd-rec*)
    next
      case *Some*
      from *Some ws clsC m*
      show *?thesis* **by** (*auto intro*: *method-declared-inI simp add*: *methd-rec*)
    qed
  qed
qed

lemma *methd-rec-Some-cases*:
  assumes *methd-C*: *methd G C sig* = *Some m* **and**
      ws: *ws-prog G* **and**
      clsC: *class G C* = *Some c* **and**
      neq-C-Obj: *C*≠*Object*
  obtains (*NewMethod*) *table-of* (*map* (λ(*s*, *m*). (*s*, *C*, *m*)) (*methods c*)) *sig* = *Some m*
   | (*InheritedMethod*) *G*⊢*C inherits* (*method sig m*) **and** *methd G* (*super c*) *sig* = *Some m*
proof −
  let *?inherited* = *filter-tab* (λ*sig m*. *G*⊢*C inherits method sig m*)
          (*methd G* (*super c*))
  let *?new* = *table-of* (*map* (λ(*s*, *m*). (*s*, *C*, *m*)) (*methods c*))
  from *ws clsC neq-C-Obj methd-C*
  have *methd-unfold*: (*?inherited* ++ *?new*) *sig* = *Some m*
    by (*simp add*: *methd-rec*)
  show *thesis*
  proof (*cases ?new sig*)
    case *None*
    with *methd-unfold* **have** *?inherited sig* = *Some m*
      by (*auto*)
    with *InheritedMethod* **show** *?thesis* **by** *blast*
  next
    case *Some*
    with *methd-unfold* **have** *?new sig* = *Some m*

```
      by auto
    with NewMethod show ?thesis by blast
  qed
qed


lemma methd-member-of:
  assumes wf: wf-prog G
  shows
    ⟦is-class G C; methd G C sig = Some m⟧ ⟹ G⊢Methd sig m member-of C
  (is ?Class C ⟹ ?Method C ⟹ ?MemberOf C)
proof −
  from wf  have  ws: ws-prog G ..
  assume defC: is-class G C
  from defC ws
  show ?Class C ⟹ ?Method C ⟹ ?MemberOf C
  proof (induct rule: ws-class-induct')
    case Object
    with wf have declC: Object = declclass m
      by (simp add: declclass-methd-Object)
    from Object wf have G⊢Methd sig m declared-in Object
      by (auto intro: methd-declaredD simp add: declC)
    with declC
    show ?MemberOf Object
      by (auto intro!: members.Immediate
                  simp del: methd-Object)
  next
    case (Subcls C c)
    assume  clsC: class G C = Some c and
       neq-C-Obj: C ≠ Object
    assume methd: ?Method C
    from methd ws clsC neq-C-Obj
    show ?MemberOf C
    proof (cases rule: methd-rec-Some-cases)
      case NewMethod
      with clsC show ?thesis
        by (auto dest: method-declared-inI intro!: members.Immediate)
    next
      case InheritedMethod
      then show ?thesis
        by (blast dest: inherits-member)
    qed
  qed
qed


lemma current-methd:
      ⟦table-of (methods c) sig = Some new;
        ws-prog G; class G C = Some c; C ≠ Object;
        methd G (super c) sig = Some old⟧
    ⟹ methd G C sig = Some (C,new)
by (auto simp add: methd-rec
            intro: filter-tab-SomeI map-add-find-right table-of-map-SomeI)


lemma wf-prog-staticD:
  assumes      wf: wf-prog G and
            clsC: class G C = Some c and
```

neq-C-Obj: $C \neq Object$ **and**
old: *methd G* (*super c*) *sig* = *Some old* **and**
accmodi-old: *Protected* $\leq$ *accmodi old* **and**
new: *table-of* (*methods c*) *sig* = *Some new*
shows *is-static new* = *is-static old*
proof –
from *clsC wf*
have *wf-cdecl*: *wf-cdecl G* (*C*,*c*) **by** (*rule wf-prog-cdecl*)
from *wf clsC neq-C-Obj*
have *is-cls-super*: *is-class G* (*super c*)
by (*blast dest*: *wf-prog-acc-superD is-acc-classD*)
from *wf is-cls-super old*
have *old-member-of*: *G*⊢*Methd sig old member-of* (*super c*)
by (*rule methd-member-of*)
from *old wf is-cls-super*
have *old-declared*: *G*⊢*Methd sig old declared-in* (*declclass old*)
by (*auto dest*: *methd-declared-in-declclass*)
from *new clsC*
have *new-declared*: *G*⊢*Methd sig* (*C*,*new*) *declared-in C*
by (*auto intro*: *method-declared-inI*)
note *trancl-rtrancl-tranc* = *trancl-rtrancl-trancl* [*trans*]
from *clsC neq-C-Obj*
have *subcls1-C-super*: $G \vdash C \prec_C 1$ *super c*
by (*rule subcls1I*)
then have $G \vdash C \prec_C$ *super c* **..**
also from *old wf is-cls-super*
have *G*⊢*super c* $\preceq_C$ (*declclass old*) **by** (*auto dest*: *methd-declC*)
finally have *subcls-C-old*: $G \vdash C \prec_C$ (*declclass old*) **.**
from *accmodi-old*
have *inheritable*: *G*⊢*Methd sig old inheritable-in pid C*
by (*auto simp add*: *inheritable-in-def*
dest: *acc-modi-le-Dests*)
show *?thesis*
proof (*cases is-static new*)
case *True*
with *subcls-C-old new-declared old-declared inheritable*
have *G*,*sig*⊢(*C*,*new*) *hides old*
by (*auto intro*: *hidesI*)
with *True wf-cdecl neq-C-Obj new*
show *?thesis*
by (*auto dest*: *wf-cdecl-hides-SomeD*)
next
case *False*
with *subcls-C-old new-declared old-declared inheritable subcls1-C-super*
old-member-of
have *G*,*sig*⊢(*C*,*new*) $overrides_S$ *old*
by (*auto intro*: *stat-overridesR.Direct*)
with *False wf-cdecl neq-C-Obj new*
show *?thesis*
by (*auto dest*: *wf-cdecl-overrides-SomeD*)
qed
qed

lemma *inheritable-instance-methd*:
assumes *subclseq-C-D*: $G \vdash C \preceq_C D$ **and**
is-cls-D: *is-class G D* **and**
wf: *wf-prog G* **and**
old: *methd G D sig* = *Some old* **and**

```
        accmodi-old: Protected ≤ accmodi old and
      not-static-old: ¬ is-static old
  shows
  ∃ new. methd G C sig = Some new ∧
       (new = old ∨ G,sig⊢new overrides_S old)
 (is (∃ new. (?Constraint C new old)))
proof −
  from subclseq-C-D is-cls-D
  have is-cls-C: is-class G C by (rule subcls-is-class2)
  from wf
  have ws: ws-prog G ..
  from is-cls-C ws subclseq-C-D
  show ∃ new. ?Constraint C new old
  proof (induct rule: ws-class-induct')
    case (Object co)
    then have eq-D-Obj: D=Object by auto
    with old
    have ?Constraint Object old old
      by auto
    with eq-D-Obj
    show ∃ new. ?Constraint Object new old by auto
  next
    case (Subcls C c)
    assume hyp: G⊢super c≼_C D ⟹ ∃ new. ?Constraint (super c) new old
    assume clsC: class G C = Some c
    assume neq-C-Obj: C≠Object
    from clsC wf
    have wf-cdecl: wf-cdecl G (C,c)
      by (rule wf-prog-cdecl)
    from ws clsC neq-C-Obj
    have is-cls-super: is-class G (super c)
      by (auto dest: ws-prog-cdeclD)
    from clsC wf neq-C-Obj
    have superAccessible: G⊢(Class (super c)) accessible-in (pid C) and
         subcls1-C-super: G⊢C ≺_C1 super c
      by (auto dest: wf-prog-cdecl wf-cdecl-supD is-acc-classD
              intro: subcls1I)
    show ∃ new. ?Constraint C new old
    proof (cases G⊢super c≼_C D)
      case False
      from False Subcls
      have eq-C-D: C=D
        by (auto dest: subclseq-superD)
      with old
      have ?Constraint C old old
        by auto
      with eq-C-D
      show ∃ new. ?Constraint C new old by auto
    next
      case True
      with hyp obtain super-method
        where super: ?Constraint (super c) super-method old by blast
      from super not-static-old
      have not-static-super: ¬ is-static super-method
        by (auto dest!: stat-overrides-commonD)
      from super old wf accmodi-old
      have accmodi-super-method: Protected ≤ accmodi super-method
        by (auto dest!: wf-prog-stat-overridesD)
      from super accmodi-old wf
```

have *inheritable*: *G*⊢*Methd sig super-method inheritable-in* (*pid C*)
by (*auto dest*!: *wf-prog-stat-overridesD*
acc-modi-le-Dests
simp add: *inheritable-in-def*)
from *super wf is-cls-super*
have *member*: *G*⊢*Methd sig super-method member-of* (*super c*)
by (*auto intro*: *methd-member-of*)
from *member*
have *decl-super-method*:
G⊢*Methd sig super-method declared-in* (*declclass super-method*)
by (*auto dest*: *member-of-declC*)
from *super subcls1-C-super ws is-cls-super*
have *subcls-C-super*: *G*⊢*C* $\prec_C$ (*declclass super-method*)
by (*auto intro*: *rtrancl-into-trancl2 dest*: *methd-declC*)
show ∃ *new*. *?Constraint C new old*
proof (*cases methd G C sig*)
case *None*
have *methd G* (*super c*) *sig* = *None*
proof −
from *clsC ws None*
have *no-new*: *table-of* (*methods c*) *sig* = *None*
by (*auto simp add*: *methd-rec*)
with *clsC*
have *undeclared*: *G*⊢*mid sig undeclared-in C*
by (*auto simp add*: *undeclared-in-def cdeclaredmethd-def*)
with *inheritable member superAccessible subcls1-C-super*
have *inherits*: *G*⊢*C inherits* (*method sig super-method*)
by (*auto simp add*: *inherits-def*)
with *clsC ws no-new super neq-C-Obj*
have *methd G C sig* = *Some super-method*
by (*auto simp add*: *methd-rec map-add-def intro*: *filter-tab-SomeI*)
with *None* **show** *?thesis*
by *simp*
qed
with *super* **show** *?thesis* **by** *auto*
next
case (*Some new*)
from *this ws clsC neq-C-Obj*
show *?thesis*
proof (*cases rule*: *methd-rec-Some-cases*)
case *InheritedMethod*
with *super Some* **show** *?thesis*
by *auto*
next
case *NewMethod*
assume *new*: *table-of* (*map* (λ(*s*, *m*). (*s*, *C*, *m*)) (*methods c*)) *sig*
= *Some new*
from *new*
have *declcls-new*: *declclass new* = *C*
by *auto*
from *wf clsC neq-C-Obj super new not-static-super accmodi-super-method*
have *not-static-new*: ¬ *is-static new*
by (*auto dest*: *wf-prog-staticD*)
from *clsC new*
have *decl-new*: *G*⊢*Methd sig new declared-in C*
by (*auto simp add*: *declared-in-def cdeclaredmethd-def*)
from *not-static-new decl-new decl-super-method*
member subcls1-C-super inheritable declcls-new subcls-C-super
have *G*,*sig*⊢ *new* $overrides_S$ *super-method*

```
          by (auto intro: stat-overridesR.Direct)
        with super Some
        show ?thesis
          by (auto intro: stat-overridesR.Indirect)
      qed
    qed
  qed
 qed
qed


lemma inheritable-instance-methd-cases:
 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
 obtains (Inheritance) methd G C sig = Some old
   | (Overriding) new where methd G C sig = Some new and G,sig⊢new overridesS old
proof −
 from subclseq-C-D is-cls-D wf old accmodi-old not-static-old
 show ?thesis
   by (auto dest: inheritable-instance-methd intro: Inheritance Overriding)
qed


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 −
 from subclseq-C-D is-cls-D wf old accmodi-old not-static-old
 show ?thesis
 proof (cases rule: inheritable-instance-methd-cases)
   case Inheritance
   with not-static-old accmodi-old show ?thesis by auto
 next
   case (Overriding new)
   then have ¬ is-static new by (auto dest: stat-overrides-commonD)
   with Overriding not-static-old accmodi-old wf
   show ?thesis
     by (auto dest!: wf-prog-stat-overridesD)
 qed
qed



lemma bexI': x ∈ A ⟹ P x ⟹ ∃ x∈A. P x by blast

lemma ballE': ∀ x∈A. P x ⟹ (x ∉ A ⟹ Q) ⟹ (P x ⟹ Q) ⟹ Q by blast
```

```
lemma subint-widen-imethds:
  assumes irel: G⊢I≼I J
  and wf: wf-prog G
  and is-iface: is-iface G J
  and jm: jm ∈ imethds G J sig
  shows ∃ im ∈ imethds G I sig. is-static im = is-static jm ∧
                        accmodi im = accmodi jm ∧
                        G⊢resTy im≼resTy jm
  using irel jm
proof (induct rule: converse-rtrancl-induct)
    case base
    then show ?case by  (blast elim: bexI')
  next
    case (step I SI)
    from ‹G⊢I ≺I1 SI›
    obtain i where
      ifI: iface G I = Some i and
       SI: SI ∈ set (isuperIfs i)
      by (blast dest: subint1D)

    let ?newMethods
          = (set-option ∘ table-of (map (λ(sig, mh). (sig, I, mh)) (imethods i)))
    show ?case
    proof (cases ?newMethods sig = {})
      case True
      with ifI SI step wf
      show ?thesis
        by (auto simp add: imethds-rec)
    next
      case False
      from ifI wf False
      have imethds: imethds G I sig = ?newMethods sig
        by (simp add: imethds-rec)
      from False
      obtain im where
        imdef: im ∈ ?newMethods sig
        by (blast)
      with imethds
      have im: im ∈ imethds G I sig
        by (blast)
      with im wf ifI
      obtain
         imStatic: ¬ is-static im and
         imPublic: accmodi im = Public
        by (auto dest!: imethds-wf-mhead)
      from ifI wf
      have wf-I: wf-idecl G (I,i)
        by (rule wf-prog-idecl)
      with SI wf
      obtain si where
         ifSI: iface G SI = Some si and
        wf-SI: wf-idecl G (SI,si)
        by (auto dest!: wf-idecl-supD is-acc-ifaceD
                  dest: wf-prog-idecl)
      from step
      obtain sim::qtname × mhead  where
                      sim: sim ∈ imethds G SI sig and
         eq-static-sim-jm: is-static sim = is-static jm and
```

eq-access-sim-jm: *accmodi sim* = *accmodi jm* **and**
resTy-widen-sim-jm: *G*⊢*resTy sim*⪯*resTy jm*
by *blast*
with *wf-I SI imdef sim*
have *G*⊢*resTy im*⪯*resTy sim*
by (*auto dest*!: *wf-idecl-hidings hidings-entailsD*)
with *wf resTy-widen-sim-jm*
have *resTy-widen-im-jm*: *G*⊢*resTy im*⪯*resTy jm*
by (*blast intro*: *widen-trans*)
from *sim wf ifSI*
obtain
simStatic: ¬ *is-static sim* **and**
simPublic: *accmodi sim* = *Public*
by (*auto dest*!: *imethds-wf-mhead*)
from *im*
imStatic simStatic eq-static-sim-jm
imPublic simPublic eq-access-sim-jm
resTy-widen-im-jm
show *?thesis*
by *auto*
qed
qed

lemma *implmt1-methd*:
⋀*sig*. ⟦*G*⊢*C*⇝1*I*; *wf-prog G*; *im* ∈ *imethds G I sig*⟧ ⟹
∃ *cm* ∈*methd G C sig*: ¬ *is-static cm* ∧ ¬ *is-static im* ∧
G⊢*resTy cm*⪯*resTy im* ∧
accmodi im = *Public* ∧ *accmodi cm* = *Public*
apply (*drule implmt1D*)
apply *clarify*
apply (*drule* (*2*) *wf-prog-cdecl* [*THEN wf-cdecl-impD*])
apply (*frule* (*1*) *imethds-wf-mhead*)
apply (*simp add*: *is-acc-iface-def*)
apply (*force*)
done

lemma *implmt-methd* [*rule-format* (*no-asm*)]:
⟦*wf-prog G*; *G*⊢*C*⇝*I*⟧ ⟹ *is-iface G I* ⟶
(∀ *im* ∈*imethds G I* *sig*.
∃ *cm*∈*methd G C sig*: ¬*is-static cm* ∧ ¬ *is-static im* ∧
G⊢*resTy cm*⪯*resTy im* ∧
accmodi im = *Public* ∧ *accmodi cm* = *Public*)
apply (*frule implmt-is-class*)
apply (*erule implmt.induct*)
apply *safe*
apply (*drule* (*2*) *implmt1-methd*)
apply *fast*
apply (*drule* (*1*) *subint-widen-imethds*)
apply *simp*
apply *assumption*

apply *clarify*
apply (*drule* (*2*) *implmt1-methd*)
apply (*force*)
apply (*frule subcls1D*)
apply (*drule* (*1*) *bspec*)
apply *clarify*
apply (*drule* (*3*) *r-into-rtrancl* [*THEN inheritable-instance-methd-props*,
OF - implmt-is-class])
apply *auto*
done

lemma *mheadsD* [*rule-format* (*no-asm*)]:
emh ∈ *mheads G S t sig* ⟶ *wf-prog G* ⟶
(∃ *C D m*. *t* = *ClassT C* ∧ *declrefT emh* = *ClassT D* ∧
accmethd G S C sig = *Some m* ∧
(*declclass m* = *D*) ∧ *mhead* (*mthd m*) = (*mhd emh*)) ∨
(∃ *I*. *t* = *IfaceT I* ∧ ((∃ *im*. *im* ∈ *accimethds G* (*pid S*) *I sig* ∧
mthd im = *mhd emh*) ∨
(∃ *m*. *G*⊢*Iface I accessible-in* (*pid S*) ∧ *accmethd G S Object sig* = *Some m* ∧
accmodi m ≠ *Private* ∧
declrefT emh = *ClassT Object* ∧ *mhead* (*mthd m*) = *mhd emh*))) ∨
(∃ *T m*. *t* = *ArrayT T* ∧ *G*⊢*Array T accessible-in* (*pid S*) ∧
accmethd G S Object sig = *Some m* ∧ *accmodi m* ≠ *Private* ∧
declrefT emh = *ClassT Object* ∧ *mhead* (*mthd m*) = *mhd emh*)
apply (*rule-tac ref-ty1*=*t* **in** *ref-ty-ty.induct* [*THEN conjunct1*])
apply *auto*
apply (*auto simp add*: *cmheads-def accObjectmheads-def Objectmheads-def*)
apply (*auto dest*!: *accmethd-SomeD*)
done

lemma *mheads-cases*:
assumes *emh* ∈ *mheads G S t sig* **and** *wf-prog G*
obtains (*Class-methd*) *C D m* **where**
t = *ClassT C declrefT emh* = *ClassT D accmethd G S C sig* = *Some m*
declclass m = *D mhead* (*mthd m*) = *mhd emh*
| (*Iface-methd*) *I im* **where** *t* = *IfaceT I*
im ∈ *accimethds G* (*pid S*) *I sig mthd im* = *mhd emh*
| (*Iface-Object-methd*) *I m* **where**
t = *IfaceT I G*⊢*Iface I accessible-in* (*pid S*)
accmethd G S Object sig = *Some m accmodi m* ≠ *Private*
declrefT emh = *ClassT Object mhead* (*mthd m*) = *mhd emh*
| (*Array-Object-methd*) *T m* **where**
t = *ArrayT T G*⊢*Array T accessible-in* (*pid S*)
accmethd G S Object sig = *Some m accmodi m* ≠ *Private*
declrefT emh = *ClassT Object mhead* (*mthd m*) = *mhd emh*
using *assms* **by** (*blast dest*!: *mheadsD*)

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 −
assume *wf*: *wf-prog G*
then have *ws*: *ws-prog G* **..**
assume *clsC*: *class G C* = *Some c*
from *clsC ws*

```
  show ⋀ m d. ⟦methd G C sig = Some m; class G (declclass m) = Some d⟧
       ⟹ table-of (methods d) sig  = Some (mthd m)
  proof (induct rule: ws-class-induct)
    case Object
    with wf show ?thesis m d by auto
  next
    case (Subcls C c)
    let ?newMethods = table-of (map (λ(s, m). (s, C, m)) (methods c)) sig
    show ?thesis m d
    proof (cases ?newMethods)
      case None
      from None ws Subcls
      show ?thesis by (auto simp add: methd-rec) (rule Subcls)
    next
      case Some
      from Some ws Subcls
      show ?thesis
        by (auto simp add: methd-rec
                 dest: wf-prog-cdecl wf-cdecl-supD is-acc-class-is-class)
    qed
  qed
qed


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 −
  from is-cls-C wf
  have subclseq: G⊢C ≼C Object
    by (auto intro: subcls-ObjectI)
  from wf have ws: ws-prog G
    by simp
  from wf
  have is-cls-Obj: is-class G Object
    by simp
  from statM subclseq is-cls-Obj ws private
  show ?thesis
  proof (cases rule: dynmethd-cases)
    case Static then show ?thesis .
  next
    case Overrides
    with private show ?thesis
      by (auto dest: no-Private-override)
  qed
qed


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 −
```

```
  from wf have ws: ws-prog G by simp
  {
    fix I i new
    assume ifI: iface G I = Some i
    assume new: table-of (imethods i) sig = Some new
    from ifI new not-private wf old
    have ?P (I,new)
      by (auto dest!: wf-prog-idecl wf-idecl-hiding cond-hiding-entailsD
            simp del: methd-Object)
  } note hyp-newmethod = this
  from is-if-I ws new
  show ?thesis
  proof (induct rule: ws-interface-induct)
    case (Step I i)
    assume ifI: iface G I = Some i
    assume new: new ∈ imethds G I sig
    from Step
    have hyp: ∀ J ∈ set (isuperIfs i). (new ∈ imethds G J sig ⟶ ?P new)
      by auto
    from new ifI ws
    show ?P new
    proof (cases rule: imethds-cases)
      case NewMethod
      with ifI hyp-newmethod
      show ?thesis
        by auto
    next
      case (InheritedMethod J)
      assume J ∈ set (isuperIfs i)
             new ∈ imethds G J sig
      with hyp
      show ?thesis
        by auto
    qed
  qed
qed
```

Which dynamic classes are valid to look up a member of a distinct static type? We have to distinct class members (named static members in Java) from instance members. Class members are global to all Objects of a class, instance members are local to a single Object instance. If a member is equipped with the static modifier it is a class member, else it is an instance member. The following table gives an overview of the current framework. We assume to have a reference with static type statT and a dynamic class dynC. Between both of these types the widening relation holds *G*⊢*Class dynC*≼*statT*. Unfortunately this ordinary widening relation isn't enough to describe the valid lookup classes, since we must cope the special cases of arrays and interfaces,too. If we statically expect an array or inteface we may lookup a field or a method in Object which isn't covered in the widening relation.

statT field instance method static (class) method

NullT / / / Iface / dynC Object Class dynC dynC dynC Array / Object Object

In most cases we con lookup the member in the dynamic class. But as an interface can't declare new static methods, nor an array can define new methods at all, we have to lookup methods in the base class Object.

The limitation to classes in the field column is artificial and comes out of the typing rule for the field access (see rule *FVar* in the welltyping relation *wt* in theory WellType). I stems out of the fact, that Object indeed has no non private fields. So interfaces and arrays can actually have no fields at all and a field access would be senseless. (In Java interfaces are allowed to declare new fields but in

current Bali not!). So there is no principal reason why we should not allow Objects to declare non private fields. Then we would get the following column:
statT field ——————— NullT / Iface Object Class dynC Array Object

primrec *valid-lookup-cls*:: *prog* ⇒ *ref-ty* ⇒ *qtname* ⇒ *bool* ⇒ *bool*
(-,- ⊢ - *valid′-lookup′-cls′-for* - [*61*,*61*,*61*,*61*] *60*)
where
G,*NullT* ⊢ *dynC valid-lookup-cls-for static-membr* = *False*
| *G*,*IfaceT I* ⊢ *dynC valid-lookup-cls-for static-membr*
= (*if static-membr*
then dynC=*Object*
else G⊢*Class dynC*⪯ *Iface I*)
| *G*,*ClassT C* ⊢ *dynC valid-lookup-cls-for static-membr* = *G*⊢*Class dynC*⪯ *Class C*
| *G*,*ArrayT T* ⊢ *dynC valid-lookup-cls-for static-membr* = (*dynC*=*Object*)

lemma *valid-lookup-cls-is-class*:
assumes *dynC*: *G*,*statT* ⊢ *dynC valid-lookup-cls-for static-membr* **and**
ty-statT: *isrtype G statT* **and**
wf: *wf-prog G*
shows *is-class G dynC*
proof (*cases statT*)
case *NullT*
with *dynC ty-statT* **show** *?thesis*
by (*auto dest*: *widen-NT2*)
next
case (*IfaceT I*)
with *dynC wf* **show** *?thesis*
by (*auto dest*: *implmt-is-class*)
next
case (*ClassT C*)
with *dynC ty-statT* **show** *?thesis*
by (*auto dest*: *subcls-is-class2*)
next
case (*ArrayT T*)
with *dynC wf* **show** *?thesis*
by (*auto*)
qed

declare *split-paired-All* [*simp del*] *split-paired-Ex* [*simp del*]
setup ‹*map-theory-simpset* (*fn ctxt* => *ctxt delloop split-all-tac*)›
setup ‹*map-theory-claset* (*fn ctxt* => *ctxt delSWrapper split-all-tac*)›

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 −
assume *emh*: *emh* ∈ *mheads G S statT sig*
and *wf*: *wf-prog G*
and *dynC-Prop*: *G*,*statT* ⊢ *dynC valid-lookup-cls-for* (*is-static emh*)
and *istype*: *isrtype G statT*
from *dynC-Prop istype wf*
obtain *y* **where**
dynC: *class G dynC* = *Some y*
by (*auto dest*: *valid-lookup-cls-is-class*)

```
from emh wf show ?thesis
proof (cases rule: mheads-cases)
  case Class-methd
  fix statC statDeclC sm
  assume   statC: statT = ClassT statC
  assume         accmethd G S statC sig = Some sm
  then have   sm: methd G statC sig = Some sm
    by (blast dest: accmethd-SomeD)
  assume eq-mheads: mhead (mthd sm) = mhd emh
  from statC
  have dynlookup: dynlookup G statT dynC sig = dynmethd G statC dynC sig
    by (simp add: dynlookup-def)
  from wf statC istype dynC-Prop sm
  obtain dm where
    dynmethd G statC dynC sig = Some dm
    is-static dm = is-static sm
    G⊢resTy dm≼resTy sm
    by (force dest!: ws-dynmethd accmethd-SomeD)
  with dynlookup eq-mheads
  show ?thesis
    by (cases emh type: prod) (auto)
next
  case Iface-methd
  fix I im
  assume   statI: statT = IfaceT I and
        eq-mheads: mthd im = mhd emh and
                   im ∈ accimethds G (pid S) I sig
  then have im: im ∈ imethds G I sig
    by (blast dest: accimethdsD)
  with istype statI eq-mheads wf
  have not-static-emh: ¬ is-static emh
    by (cases emh) (auto dest: wf-prog-idecl imethds-wf-mhead)
  from statI im
  have dynlookup: dynlookup G statT dynC sig = methd G dynC sig
    by (auto simp add: dynlookup-def dynimethd-def)
  from wf dynC-Prop statI istype im not-static-emh
  obtain dm where
    methd G dynC sig = Some dm
    is-static dm = is-static im
    G⊢resTy (mthd dm)≼resTy (mthd im)
    by (force dest: implmt-methd)
  with dynlookup eq-mheads
  show ?thesis
    by (cases emh type: prod) (auto)
next
  case Iface-Object-methd
  fix I sm
  assume   statI: statT = IfaceT I and
              sm: accmethd G S Object sig = Some sm and
       eq-mheads: mhead (mthd sm) = mhd emh and
           nPriv: accmodi sm ≠ Private
   show ?thesis
   proof (cases imethds G I sig = {})
     case True
     with statI
     have dynlookup: dynlookup G statT dynC sig = dynmethd G Object dynC sig
       by (simp add: dynlookup-def dynimethd-def)
     from wf dynC
     have subclsObj: G⊢dynC ≼C Object
```

```
        by (auto intro: subcls-ObjectI)
      from wf dynC dynC-Prop istype sm subclsObj
      obtain dm where
        dynmethd G Object dynC sig = Some dm
        is-static dm = is-static sm
        G⊢resTy (mthd dm)≼resTy (mthd sm)
        by (auto dest!: ws-dynmethd accmethd-SomeD
                intro: class-Object [OF wf] intro: that)
      with dynlookup eq-mheads
      show ?thesis
        by (cases emh type: prod) (auto)
    next
      case False
      with statI
      have dynlookup: dynlookup G statT dynC sig = methd G dynC sig
        by (simp add: dynlookup-def dynimethd-def)
      from istype statI
      have is-iface G I
        by auto
      with wf sm nPriv False
      obtain im where
            im: im ∈ imethds G I sig and
        eq-stat: is-static im = is-static sm and
        resProp: G⊢resTy (mthd im)≼resTy (mthd sm)
        by (auto dest: wf-imethds-hiding-objmethdsD accmethd-SomeD)
      from im wf statI istype eq-stat eq-mheads
      have not-static-sm: ¬ is-static emh
        by (cases emh) (auto dest: wf-prog-idecl imethds-wf-mhead)
      from im wf dynC-Prop dynC istype statI not-static-sm
      obtain dm where
        methd G dynC sig = Some dm
        is-static dm = is-static im
        G⊢resTy (mthd dm)≼resTy (mthd im)
        by (auto dest: implmt-methd)
      with wf eq-stat resProp dynlookup eq-mheads
      show ?thesis
        by (cases emh type: prod) (auto intro: widen-trans)
    qed
  next
    case Array-Object-methd
    fix T sm
    assume statArr: statT = ArrayT T and
              sm: accmethd G S Object sig = Some sm and
        eq-mheads: mhead (mthd sm) = mhd emh
    from statArr dynC-Prop wf
    have dynlookup: dynlookup G statT dynC sig = methd G Object sig
      by (auto simp add: dynlookup-def dynmethd-C-C)
    with sm eq-mheads sm
    show ?thesis
      by (cases emh type: prod) (auto dest: accmethd-SomeD)
  qed
qed
declare split-paired-All [simp] split-paired-Ex [simp]
setup ‹map-theory-claset (fn ctxt => ctxt addSbefore (split-all-tac, split-all-tac))›
setup ‹map-theory-simpset (fn ctxt => ctxt addloop (split-all-tac, split-all-tac))›
```

lemma *methd-declclass*:
$\llbracket$*class G C = Some c*; *wf-prog G*; *methd G C sig = Some m*$\rrbracket$
$\Longrightarrow$ *methd G* (*declclass m*) *sig = Some m*
proof –
  assume *asm*: *class G C = Some c wf-prog G methd G C sig = Some m*
  have *wf-prog G* $\longrightarrow$
      ($\forall$ *c m*. *class G C = Some c* $\longrightarrow$ *methd G C sig = Some m*
        $\longrightarrow$ *methd G* (*declclass m*) *sig = Some m*)   (**is** *?P G C*)
  proof (*induct G C rule*: *class-rec-induct′*, *intro allI impI*)
    fix *G C c m*
    assume *hyp*: $\bigwedge$*c*. *class G C = Some c* $\Longrightarrow$ *ws-prog G* $\Longrightarrow$ *C* $\neq$ *Object* $\Longrightarrow$
          ?P G (*super c*)
    assume *wf*: *wf-prog G* **and** *cls-C*: *class G C = Some c* **and**
       m: *methd G C sig = Some m*
    show *methd G* (*declclass m*) *sig = Some m*
    proof (*cases C=Object*)
      case *True*
      with *wf m* **show** *?thesis* **by** (*auto intro*: *table-of-map-SomeI*)
    next
      let *?filter=filter-tab* (λ*sig m*. *G*$\vdash$*C inherits method sig m*)
      let *?table = table-of* (*map* (λ(*s*, *m*). (*s*, *C*, *m*)) (*methods c*))
      case *False*
      with *cls-C wf m*
      have *methd-C*: (*?filter* (*methd G* (*super c*)) ++ *?table*) *sig = Some m*
        by (*simp add*: *methd-rec*)
      show *?thesis*
      proof (*cases ?table sig*)
        case *None*
        from *this methd-C* **have** *?filter* (*methd G* (*super c*)) *sig = Some m*
          by *simp*
        moreover
        from *wf cls-C False* **obtain** *sup* **where** *class G* (*super c*) = *Some sup*
          by (*blast dest*: *wf-prog-cdecl wf-cdecl-supD is-acc-class-is-class*)
        moreover note *wf False cls-C*
        ultimately show *?thesis* **by** (*auto intro*: *hyp* [*rule-format*])
      next
        case *Some*
        from *this methd-C m* **show** *?thesis* **by** *auto*
      qed
    qed
  qed
  with *asm* **show** *?thesis* **by** *auto*
qed

lemma *dynmethd-declclass*:
$\llbracket$*dynmethd G statC dynC sig = Some m*;
 wf-prog G; *is-class G statC*
$\rrbracket$ $\Longrightarrow$ *methd G* (*declclass m*) *sig = Some m*
by (*auto dest*: *dynmethd-declC*)

lemma *dynlookup-declC*:
$\llbracket$*dynlookup G statT dynC sig = Some m*; *wf-prog G*;
 is-class G dynC;*isrtype G statT*
$\rrbracket$ $\Longrightarrow$ *G*$\vdash$*dynC* $\preceq_C$ (*declclass m*) $\wedge$ *is-class G* (*declclass m*)
by (*cases statT*)

(*auto simp add*: *dynlookup-def dynimethd-def*
dest: *methd-declC dynmethd-declC*)

lemma *dynlookup-Array-declclassD* [*simp*]:
⟦*dynlookup G* (*ArrayT T*) *Object sig* = *Some dm*;*wf-prog G*⟧
⟹ *declclass dm* = *Object*
proof −
assume *dynL*: *dynlookup G* (*ArrayT T*) *Object sig* = *Some dm*
assume *wf*: *wf-prog G*
from *wf* **have** *ws*: *ws-prog G* **by** *auto*
from *wf* **have** *is-cls-Obj*: *is-class G Object* **by** *auto*
from *dynL wf*
show *?thesis*
by (*auto simp add*: *dynlookup-def dynmethd-C-C* [*OF is-cls-Obj ws*]
dest: *methd-Object-SomeD*)
qed

declare *split-paired-All* [*simp del*] *split-paired-Ex* [*simp del*]
setup ‹*map-theory-simpset* (*fn ctxt* => *ctxt delloop split-all-tac*)›
setup ‹*map-theory-claset* (*fn ctxt* => *ctxt delSWrapper split-all-tac*)›

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*)))
apply (*unfold empty-dt-def*)
apply (*erule wt.induct*)
apply (*auto split del*: *if-split-asm simp del*: *snd-conv*
simp add: *is-acc-class-def is-acc-type-def*)
apply (*erule typeof-empty-is-type*)
apply (*frule* (*1*) *wf-prog-cdecl* [*THEN wf-cdecl-supD*],
force simp del: *snd-conv*, *clarsimp simp add*: *is-acc-class-def*)
apply (*drule* (*1*) *max-spec2mheads* [*THEN conjunct1*, *THEN mheadsD*])
apply (*drule-tac* [*2*] *accfield-fields*)
apply (*frule class-Object*)
apply (*auto dest*: *accmethd-rT-is-type*
imethds-wf-mhead [*THEN conjunct1*, *THEN rT-is-acc-type*]
dest!:*accimethdsD*
simp del: *class-Object*
simp add: *is-acc-type-def*
)
done
declare *split-paired-All* [*simp*] *split-paired-Ex* [*simp*]
setup ‹*map-theory-claset* (*fn ctxt* => *ctxt addSbefore* (*split-all-tac*, *split-all-tac*))›
setup ‹*map-theory-simpset* (*fn ctxt* => *ctxt addloop* (*split-all-tac*, *split-all-tac*))›

lemma *ty-expr-is-type*:
⟦*E*⊢*e*::−*T*; *wf-prog* (*prg E*)⟧ ⟹ *is-type* (*prg E*) *T*
by (*auto dest*!: *wt-is-type*)

lemma *ty-var-is-type*:
⟦*E*⊢*v*::=*T*; *wf-prog* (*prg E*)⟧ ⟹ *is-type* (*prg E*) *T*
by (*auto dest*!: *wt-is-type*)

lemma *ty-exprs-is-type*:

⟦*E*⊢*es*::≐*Ts*; *wf-prog* (*prg E*)⟧ ⟹ *Ball* (*set Ts*) (*is-type* (*prg E*))
by (*auto dest*!: *wt-is-type*)

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*)
apply (*subgoal-tac is-static emh* ∨ *e* = *Super*)
defer apply (*force simp add*: *invmode-def*)
apply (*frule ty-expr-is-type*)
apply *simp*
apply (*case-tac is-static emh*)
apply (*frule* (*1*) *mheadsD*)
apply *clarsimp*
apply *safe*
apply *blast*
apply (*auto dest*!: *imethds-wf-mhead*
      accmethd-SomeD
      accimethdsD
    simp add: *accObjectmheads-def Objectmheads-def*)

apply (*erule wt-elim-cases*)
apply (*force simp add*: *cmheads-def*)
done

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*
apply (*frule* (*2*) *methd-wf-mdecl*, *clarify*)
apply (*force dest*!: *wf-mdecl-bodyD intro*!: *wt.Methd*)
done

2 accessibility concerns

lemma *mheads-type-accessible*:
⟦*emh* ∈ *mheads G S T sig*; *wf-prog G*⟧
⟹ *G*⊢*RefT T accessible-in* (*pid S*)
by (*erule mheads-cases*)
 (*auto dest*: *accmethd-SomeD accessible-from-commonD accimethdsD*)

lemma *static-to-dynamic-accessible-from-aux*:
⟦*G*⊢*m of C accessible-from accC*;*wf-prog G*⟧
⟹ *G*⊢*m in C dyn-accessible-from accC*
proof (*induct rule*: *accessible-fromR.induct*)
qed (*auto intro*: *dyn-accessible-fromR.intros*
      member-of-to-member-in
      static-to-dynamic-overriding)

lemma *static-to-dynamic-accessible-from*:
 assumes *stat-acc*: *G*⊢*m of statC accessible-from accC* **and**

subclseq: $G \vdash dynC \preceq_C statC$ **and**
wf: *wf-prog G*
shows $G \vdash m$ *in dynC dyn-accessible-from accC*
proof −
from *stat-acc subclseq*
show *?thesis* (**is** *?Dyn-accessible m*)
proof (*induct rule*: *accessible-fromR.induct*)
case (*Immediate m statC*)
then show *?Dyn-accessible m*
by (*blast intro*: *dyn-accessible-fromR.Immediate*
member-inI
permits-acc-inheritance)
next
case (*Overriding m - -*)
with *wf* **show** *?Dyn-accessible m*
by (*blast intro*: *dyn-accessible-fromR.Overriding*
member-inI
static-to-dynamic-overriding
rtrancl-trancl-trancl
static-to-dynamic-accessible-from-aux)
qed
qed

lemma *static-to-dynamic-accessible-from-static*:
assumes *stat-acc*: $G \vdash m$ *of statC accessible-from accC* **and**
static: *is-static m* **and**
wf: *wf-prog G*
shows $G \vdash m$ *in* (*declclass m*) *dyn-accessible-from accC*
proof −
from *stat-acc wf*
have $G \vdash m$ *in statC dyn-accessible-from accC*
by (*auto intro*: *static-to-dynamic-accessible-from*)
from *this static*
show *?thesis*
by (*rule dyn-accessible-from-static-declC*)
qed

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-in dynC*
proof −
from *m*
have *subclseq*: $G \vdash dynC \preceq_C statC$
by (*auto simp add*: *dynmethd-def*)
from *subclseq iscls-statC*
have *iscls-dynC*: *is-class G dynC*
by (*rule subcls-is-class2*)
from *iscls-dynC iscls-statC wf m*
have $G \vdash dynC \preceq_C$ (*declclass m*) ∧ *is-class G* (*declclass m*) ∧
methd G (*declclass m*) *sig* = *Some m*
by − (*drule dynmethd-declC*, *auto*)
with *wf*
show *?thesis*
by (*auto intro*: *member-inI dest*: *methd-member-of*)
qed

lemma *dynmethd-access-prop*:
assumes *statM*: *methd G statC sig = Some statM* **and**
stat-acc: *G⊢Methd sig statM of statC accessible-from accC* **and**
dynM: *dynmethd G statC dynC sig = Some dynM* **and**
wf: *wf-prog G*
shows *G⊢Methd sig dynM in dynC dyn-accessible-from accC*
proof –
from *wf* **have** *ws*: *ws-prog G* **..**
from *dynM*
have *subclseq*: $G\vdash dynC \preceq_C statC$
by (*auto simp add*: *dynmethd-def*)
from *stat-acc*
have *is-cls-statC*: *is-class G statC*
by (*auto dest*: *accessible-from-commonD member-of-is-classD*)
with *subclseq*
have *is-cls-dynC*: *is-class G dynC*
by (*rule subcls-is-class2*)
from *is-cls-statC statM wf*
have *member-statC*: *G⊢Methd sig statM member-of statC*
by (*auto intro*: *methd-member-of*)
from *stat-acc*
have *statC-acc*: *G⊢Class statC accessible-in* (*pid accC*)
by (*auto dest*: *accessible-from-commonD*)
from *statM subclseq is-cls-statC ws*
show *?thesis*
proof (*cases rule*: *dynmethd-cases*)
case *Static*
assume *dynmethd*: *dynmethd G statC dynC sig = Some statM*
with *dynM* **have** *eq-dynM-statM*: *dynM=statM*
by *simp*
with *stat-acc subclseq wf*
show *?thesis*
by (*auto intro*: *static-to-dynamic-accessible-from*)
next
case (*Overrides newM*)
assume *dynmethd*: *dynmethd G statC dynC sig = Some newM*
assume *override*: *G,sig⊢newM overrides statM*
assume *neq*: *newM≠statM*
from *dynmethd dynM*
have *eq-dynM-newM*: *dynM=newM*
by *simp*
from *dynmethd eq-dynM-newM wf is-cls-statC*
have *G⊢Methd sig dynM member-in dynC*
by (*auto intro*: *dynmethd-member-in*)
moreover
from *subclseq*
have $G\vdash dynC \prec_C statC$
proof (*cases rule*: *subclseq-cases*)
case *Eq*
assume *dynC=statC*
moreover
from *is-cls-statC* **obtain** *c*
where *class G statC = Some c*
by *auto*
moreover
note *statM ws dynmethd*
ultimately

have *newM=statM*
by (*auto simp add*: *dynmethd-C-C*)
with *neq* **show** *?thesis*
by (*contradiction*)
next
case *Subcls* **then show** *?thesis* **.**
qed
moreover
from *stat-acc wf*
have *G⊢Methd sig statM in statC dyn-accessible-from accC*
by (*blast intro*: *static-to-dynamic-accessible-from*)
moreover
note *override eq-dynM-newM*
ultimately show *?thesis*
by (*cases dynM*,*cases statM*) (*auto intro*: *dyn-accessible-fromR.Overriding*)
qed
qed

lemma *implmt-methd-access*:
fixes *accC*::*qtname*
assumes *iface-methd*: *imethds G I sig* ≠ {} **and**
implements: *G⊢dynC⇝I* **and**
isif-I: *is-iface G I* **and**
wf: *wf-prog G*
shows ∃ *dynM*. *methd G dynC sig* = *Some dynM* ∧
G⊢Methd sig dynM in dynC dyn-accessible-from accC
proof −
from *implements*
have *iscls-dynC*: *is-class G dynC* **by** (*rule implmt-is-class*)
from *iface-methd*
obtain *im*
where *im* ∈ *imethds G I sig*
by *auto*
with *wf implements isif-I*
obtain *dynM*
where *dynM*: *methd G dynC sig* = *Some dynM* **and**
pub: *accmodi dynM* = *Public*
by (*blast dest*: *implmt-methd*)
with *iscls-dynC wf*
have *G⊢Methd sig dynM in dynC dyn-accessible-from accC*
by (*auto intro*!: *dyn-accessible-fromR.Immediate*
intro: *methd-member-of member-of-to-member-in*
simp add: *permits-acc-def*)
with *dynM*
show *?thesis*
by *blast*
qed

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

have *dynimethd G I dynC sig = methd G dynC sig*
by (*simp add*: *dynimethd-def*)
with *iface-methd implements isif-I wf*
show *?thesis*
by (*simp only*:)
(*blast intro*: *implmt-methd-access*)
qed

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 −
from *emh wf*
have *statT-acc*: *G*⊢*RefT statT accessible-in* (*pid accC*)
by (*rule mheads-type-accessible*)
from *dynC-prop isT-statT wf*
have *iscls-dynC*: *is-class G dynC*
by (*rule valid-lookup-cls-is-class*)
from *emh dynC-prop isT-statT wf dynM*
have *eq-static*: *is-static emh = is-static dynM*
by (*auto dest*: *dynamic-mheadsD*)
from *emh wf* **show** *?thesis*
proof (*cases rule*: *mheads-cases*)
case (*Class-methd statC - statM*)
assume *statT*: *statT = ClassT statC*
assume *accmethd G accC statC sig = Some statM*
then have *statM*: *methd G statC sig = Some statM* **and**
stat-acc: *G*⊢*Methd sig statM of statC accessible-from accC*
by (*auto dest*: *accmethd-SomeD*)
from *dynM statT*
have *dynM'*: *dynmethd G statC dynC sig = Some dynM*
by (*simp add*: *dynlookup-def*)
from *statM stat-acc wf dynM'*
show *?thesis*
by (*auto dest*!: *dynmethd-access-prop*)
next
case (*Iface-methd I im*)
then have *iface-methd*: *imethds G I sig* ≠ {} **and**
statT-acc: *G*⊢*RefT statT accessible-in* (*pid accC*)
by (*auto dest*: *accimethdsD*)
assume *statT*: *statT = IfaceT I*
assume *im*: *im* ∈ *accimethds G* (*pid accC*) *I sig*
assume *eq-mhds*: *mthd im = mhd emh*
from *dynM statT*
have *dynM'*: *dynimethd G I dynC sig = Some dynM*
by (*simp add*: *dynlookup-def*)
from *isT-statT statT*
have *isif-I*: *is-iface G I*
by *simp*
show *?thesis*
proof (*cases is-static emh*)
case *False*
with *statT dynC-prop*
have *widen-dynC*: *G*⊢*Class dynC* ⪯ *RefT statT*

by *simp*
from *statT widen-dynC*
have *implmnt*: $G \vdash dynC \leadsto I$
by *auto*
from *eq-static False*
have *not-static-dynM*: $\neg$ *is-static dynM*
by *simp*
from *iface-methd implmnt isif-I wf dynM'*
show *?thesis*
by − (*drule implmt-dynimethd-access*, *auto*)
next
case *True*
assume *is-static emh*
moreover
from *wf isT-statT statT im*
have $\neg$ *is-static im*
by (*auto dest*: *accimethdsD wf-prog-idecl imethds-wf-mhead*)
moreover note *eq-mhds*
ultimately show *?thesis*
by (*cases emh*) *auto*
qed
next
case (*Iface-Object-methd I statM*)
assume *statT*: *statT* = *IfaceT I*
assume *accmethd G accC Object sig* = *Some statM*
then have *statM*: *methd G Object sig* = *Some statM* **and**
stat-acc: $G \vdash$ *Methd sig statM of Object accessible-from accC*
by (*auto dest*: *accmethd-SomeD*)
assume *not-Private-statM*: *accmodi statM* $\neq$ *Private*
assume *eq-mhds*: *mhead* (*mthd statM*) = *mhd emh*
from *iscls-dynC wf*
have *widen-dynC-Obj*: $G \vdash dynC \preceq_C Object$
by (*auto intro*: *subcls-ObjectI*)
show *?thesis*
proof (*cases imethds G I sig* = {})
case *True*
from *dynM statT True*
have *dynM'*: *dynmethd G Object dynC sig* = *Some dynM*
by (*simp add*: *dynlookup-def dynimethd-def*)
from *statT*
have $G \vdash$ *RefT statT* $\preceq$ *Class Object*
by *auto*
with *statM statT-acc stat-acc widen-dynC-Obj statT isT-statT*
wf dynM' eq-static dynC-prop
show *?thesis*
by − (*drule dynmethd-access-prop*,*force*+)
next
case *False*
then obtain *im* **where**
im: *im* $\in$ *imethds G I sig*
by *auto*
have *not-static-emh*: $\neg$ *is-static emh*
proof −
from *im statM statT isT-statT wf not-Private-statM*
have *is-static im* = *is-static statM*
by (*fastforce dest*: *wf-imethds-hiding-objmethdsD*)
with *wf isT-statT statT im*
have $\neg$ *is-static statM*
by (*auto dest*: *wf-prog-idecl imethds-wf-mhead*)

```
        with eq-mhds
        show ?thesis
          by (cases emh) auto
      qed
      with statT dynC-prop
      have implmnt: G⊢dynC⇝I
        by simp
      with isT-statT statT
      have isif-I: is-iface G I
        by simp
      from dynM statT
      have dynM': dynimethd G I dynC sig = Some dynM
        by (simp add: dynlookup-def)
      from False implmnt isif-I wf dynM'
      show ?thesis
        by − (drule implmt-dynimethd-access, auto)
    qed
  next
    case (Array-Object-methd T statM)
    assume statT: statT = ArrayT T
    assume accmethd G accC Object sig = Some statM
    then have    statM: methd G Object sig = Some statM and
              stat-acc: G⊢Methd sig statM of Object accessible-from accC
      by (auto dest: accmethd-SomeD)
    from statT dynC-prop
    have dynC-Obj: dynC = Object
      by simp
    then
    have widen-dynC-Obj: G⊢Class dynC ≼ Class Object
      by simp
    from dynM statT
    have dynM': dynmethd G Object dynC sig = Some dynM
      by (simp add: dynlookup-def)
    from statM statT-acc stat-acc dynM' wf widen-dynC-Obj
         statT isT-statT
    show ?thesis
      by − (drule dynmethd-access-prop, simp+)
  qed
qed


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 −
  from dynC-prop isT-statT wf
  have is-cls-dynC: is-class G dynC
    by (auto dest: valid-lookup-cls-is-class)
  with emh wf dynC-prop isT-statT
  obtain dynM where
    dynlookup G statT dynC sig = Some dynM
    by − (drule dynamic-mheadsD,auto)
  with  emh dynC-prop isT-statT wf
  show ?thesis
    by (blast intro: dynlookup-access-prop)
```

qed

lemma *stat-overrides-Package-old*:
assumes *stat-override*: $G \vdash$ *new overrides*$_S$ *old* **and**
accmodi-new: *accmodi new* = *Package* **and**
wf: *wf-prog G*
shows *accmodi old* = *Package*
proof −
from *stat-override wf*
have *accmodi old* ≤ *accmodi new*
by (*auto dest*: *wf-prog-stat-overridesD*)
with *stat-override accmodi-new* **show** *?thesis*
by (*cases accmodi old*) (*auto dest*: *no-Private-stat-override*
dest: *acc-modi-le-Dests*)
qed

Properties of dynamic accessibility

lemma *dyn-accessible-Private*:
assumes *dyn-acc*: $G \vdash$ *m in C dyn-accessible-from accC* **and**
priv: *accmodi m* = *Private*
shows *accC* = *declclass m*
proof −
from *dyn-acc priv*
show *?thesis*
proof (*induct*)
case (*Immediate m C*)
from ‹$G \vdash$ *m in C permits-acc-from accC*› **and** ‹*accmodi m* = *Private*›
show *?case*
by (*simp add*: *permits-acc-def*)
next
case *Overriding*
then show *?case*
by (*auto dest!*: *overrides-commonD*)
qed
qed

dyn-accessible-Package only works with the *wf-prog* assumption. Without it. it is easy to leaf the Package!

lemma *dyn-accessible-Package*:
⟦$G \vdash$ *m in C dyn-accessible-from accC*; *accmodi m* = *Package*;
wf-prog G⟧
⟹ *pid accC* = *pid* (*declclass m*)
proof −
assume *wf*: *wf-prog G*
assume *accessible*: $G \vdash$ *m in C dyn-accessible-from accC*
then show *accmodi m* = *Package*
⟹ *pid accC* = *pid* (*declclass m*)
(**is** *?Pack m* ⟹ *?P m*)
proof (*induct rule*: *dyn-accessible-fromR.induct*)
case (*Immediate m C*)
assume $G \vdash$*m member-in C*
$G \vdash$ *m in C permits-acc-from accC*
accmodi m = *Package*
then show *?P m*
by (*auto simp add*: *permits-acc-def*)
next
case (*Overriding new declC newm old Sup C*)

assume *member-new*: *G* ⊢ *new member-in C* **and**
new: *new* = (*declC*, *mdecl newm*) **and**
override: *G* ⊢ (*declC*, *newm*) *overrides old* **and**
subcls-C-Sup: *G*⊢*C* $\prec_C$ *Sup* **and**
acc-old: *G* ⊢ *methdMembr old in Sup dyn-accessible-from accC* **and**
hyp: *?Pack* (*methdMembr old*) ⟹ *?P* (*methdMembr old*) **and**
accmodi-new: *accmodi new* = *Package*
from *override accmodi-new new wf*
have *accmodi-old*: *accmodi old* = *Package*
by (*auto dest*: *overrides-Package-old*)
with *hyp*
have *P-sup*: *?P* (*methdMembr old*)
by (*simp*)
from *wf override new accmodi-old accmodi-new*
have *eq-pid-new-old*: *pid* (*declclass new*) = *pid* (*declclass old*)
by (*auto dest*: *dyn-override-Package*)
with *eq-pid-new-old P-sup* **show** *?P new*
by *auto*
qed
qed

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 −
from *dyn-acc pack field*
show *?thesis*
proof (*induct*)
case (*Immediate f C*)
from ‹*G* ⊢ *f in C permits-acc-from accC*› **and** ‹*accmodi f* = *Package*›
show *?case*
by (*simp add*: *permits-acc-def*)
next
case *Overriding*
then show *?case* **by** (*simp add*: *is-field-def*)
qed
qed

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* $\preceq_C$ *accC*
proof −
from *dyn-acc prot field instance-field outside*
show *?thesis*
proof (*induct*)
case (*Immediate f C*)
note ‹*G* ⊢ *f in C permits-acc-from accC*›
moreover
assume *accmodi f* = *Protected* **and** *is-field f* **and** ¬ *is-static f* **and**

pid (*declclass f*) ≠ *pid accC*
ultimately
show $G\vdash C \preceq_C accC$
by (*auto simp add*: *permits-acc-def*)
next
case *Overriding*
then show *?case* **by** (*simp add*: *is-field-def*)
qed
qed

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 $G\vdash accC \preceq_C declclass\ f \land G\vdash C \preceq_C declclass\ f$
proof −
from *dyn-acc prot field static-field outside*
show *?thesis*
proof (*induct*)
case (*Immediate f C*)
assume *accmodi f* = *Protected* **and** *is-field f* **and** *is-static f* **and**
pid (*declclass f*) ≠ *pid accC*
moreover
note ‹*G* ⊢ *f in C permits-acc-from accC*›
ultimately
have $G\vdash accC \preceq_C declclass\ f$
by (*auto simp add*: *permits-acc-def*)
moreover
from ‹*G* ⊢ *f member-in C*›
have $G\vdash C \preceq_C declclass\ f$
by (*rule member-in-class-relation*)
ultimately show *?case*
by *blast*
next
case *Overriding*
then show *?case* **by** (*simp add*: *is-field-def*)
qed
qed

end

Chapter 14

State

1 State for evaluation of Java expressions and statements

theory *State*
imports *DeclConcepts*
begin

design issues:

- all kinds of objects (class instances, arrays, and class objects) are handeled via a general object abstraction
- the heap and the map for class objects are combined into a single table (*recall* (*loc*, *obj*) *table* × (*qtname*, *obj*) *table* ~= (*loc* + *qtname*, *obj*) *table*)

objects

datatype *obj-tag* = — tag for generic object
CInst qtname — class instance
| *Arr ty int* — array with component type and length
— | CStat qtname the tag is irrelevant for a class object, i.e. the static fields of a class, since its type is given already by the reference to it (see below)

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

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

definition
the-Arr :: *obj option* ⇒ *ty* × *int* × (*vn*, *val*) *table*
where *the-Arr obj* = (*SOME* (*T*,*k*,*t*). *obj* = *Some* (|*tag*=*Arr T k*,*values*=*t*|))

lemma *the-Arr-Arr* [*simp*]: *the-Arr* (*Some* (|*tag*=*Arr T k*,*values*=*cs*|)) = (*T*,*k*,*cs*)
apply (*auto simp*: *the-Arr-def*)
done

lemma *the-Arr-Arr1* [*simp*,*intro*,*dest*]:
$[\![tag\ obj = Arr\ T\ k]\!] \Longrightarrow the\text{-}Arr\ (Some\ obj) = (T,k,values\ obj)$
apply (*auto simp add*: *the-Arr-def*)
done

definition
$upd\text{-}obj :: vn \Rightarrow val \Rightarrow obj \Rightarrow obj$
where $upd\text{-}obj\ n\ v = (\lambda obj.\ obj\ (\!|values{:=}(values\ obj)(n{\mapsto}v)|\!))$

lemma *upd-obj-def2* [*simp*]:
$upd\text{-}obj\ n\ v\ obj = obj\ (\!|values{:=}(values\ obj)(n{\mapsto}v)|\!)$
apply (*auto simp*: *upd-obj-def*)
done

definition
$obj\text{-}ty :: obj \Rightarrow ty$ **where**
$obj\text{-}ty\ obj = (case\ tag\ obj\ of$
$CInst\ C \Rightarrow Class\ C$
$\mid Arr\ T\ k \Rightarrow T.[])$

lemma *obj-ty-eq* [*intro*!]: $obj\text{-}ty\ (\!|tag{=}oi,values{=}x|\!) = obj\text{-}ty\ (\!|tag{=}oi,values{=}y|\!)$
by (*simp add*: *obj-ty-def*)

lemma *obj-ty-eq1* [*intro*!,*dest*]:
$tag\ obj = tag\ obj' \Longrightarrow obj\text{-}ty\ obj = obj\text{-}ty\ obj'$
by (*simp add*: *obj-ty-def*)

lemma *obj-ty-cong* [*simp*]:
$obj\text{-}ty\ (obj\ (\!|values{:=}vs|\!)) = obj\text{-}ty\ obj$
by *auto*

lemma *obj-ty-CInst* [*simp*]:
$obj\text{-}ty\ (\!|tag{=}CInst\ C,values{=}vs|\!) = Class\ C$
by (*simp add*: *obj-ty-def*)

lemma *obj-ty-CInst1* [*simp*,*intro*!,*dest*]:
$[\![tag\ obj = CInst\ C]\!] \Longrightarrow obj\text{-}ty\ obj = Class\ C$
by (*simp add*: *obj-ty-def*)

lemma *obj-ty-Arr* [*simp*]:
$obj\text{-}ty\ (\!|tag{=}Arr\ T\ i,values{=}vs|\!) = T.[]$
by (*simp add*: *obj-ty-def*)

lemma *obj-ty-Arr1* [*simp*,*intro*!,*dest*]:
$[\![tag\ obj = Arr\ T\ i]\!] \Longrightarrow obj\text{-}ty\ obj = T.[]$
by (*simp add*: *obj-ty-def*)

lemma *obj-ty-widenD*:
$G\vdash obj\text{-}ty\ obj\preceq RefT\ t \Longrightarrow (\exists\ C.\ tag\ obj = CInst\ C) \lor (\exists\ T\ k.\ tag\ obj = Arr\ T\ k)$

apply (*unfold obj-ty-def*)
apply (*auto split*: *obj-tag.split-asm*)
done

definition
obj-class :: *obj* ⇒ *qtname* **where**
obj-class obj = (*case tag obj of*
CInst C ⇒ *C*
| *Arr T k* ⇒ *Object*)

lemma *obj-class-CInst* [*simp*]: *obj-class* (|*tag*=*CInst C*,*values*=*vs*|) = *C*
by (*auto simp*: *obj-class-def*)

lemma *obj-class-CInst1* [*simp*,*intro*!,*dest*]:
tag obj = *CInst C* ⟹ *obj-class obj* = *C*
by (*auto simp*: *obj-class-def*)

lemma *obj-class-Arr* [*simp*]: *obj-class* (|*tag*=*Arr T k*,*values*=*vs*|) = *Object*
by (*auto simp*: *obj-class-def*)

lemma *obj-class-Arr1* [*simp*,*intro*!,*dest*]:
tag obj = *Arr T k* ⟹ *obj-class obj* = *Object*
by (*auto simp*: *obj-class-def*)

lemma *obj-ty-obj-class*: *G*⊢*obj-ty obj*≼ *Class statC* = *G*⊢*obj-class obj* $\preceq_C$ *statC*
apply (*case-tac tag obj*)
apply (*auto simp add*: *obj-ty-def obj-class-def*)
apply (*case-tac statC* = *Object*)
apply (*auto dest*: *widen-Array-Class*)
done

object references

type-synonym *oref* = *loc* + *qtname* — generalized object reference
syntax
Heap :: *loc* ⇒ *oref*
Stat :: *qtname* ⇒ *oref*

translations
Heap => *CONST Inl*
Stat => *CONST Inr*
(*type*) *oref* <= (*type*) *loc* + *qtname*

definition
fields-table :: *prog* ⇒ *qtname* ⇒ (*fspec* ⇒ *field* ⇒ *bool*) ⇒ (*fspec*, *ty*) *table* **where**
fields-table G C P =
map-option type ∘ *table-of* (*filter* (*case-prod P*) (*DeclConcepts.fields G C*))

lemma *fields-table-SomeI*:
⟦*table-of* (*DeclConcepts.fields G C*) *n* = *Some f*; *P n f*⟧
⟹ *fields-table G C P n* = *Some* (*type f*)
apply (*unfold fields-table-def*)

apply *clarsimp*
apply (*rule exI*)
apply (*rule conjI*)
apply (*erule map-of-filter-in*)
apply *assumption*
apply *simp*
done

lemma *fields-table-SomeD′*: *fields-table G C P fn* = *Some T* $\Longrightarrow$
$\exists f$. (*fn*,*f*)∈*set*(*DeclConcepts.fields G C*) ∧ *type f* = *T*
apply (*unfold fields-table-def*)
apply *clarsimp*
apply (*drule map-of-SomeD*)
apply *auto*
done

lemma *fields-table-SomeD*:
⟦*fields-table G C P fn* = *Some T*; *unique* (*DeclConcepts.fields G C*)⟧ $\Longrightarrow$
$\exists f$. *table-of* (*DeclConcepts.fields G C*) *fn* = *Some f* ∧ *type f* = *T*
apply (*unfold fields-table-def*)
apply *clarsimp*
apply (*rule exI*)
apply (*rule conjI*)
apply (*erule table-of-filter-unique-SomeD*)
apply *assumption*
apply *simp*
done

definition
in-bounds :: *int* ⇒ *int* ⇒ *bool* ((-/ *in′-bounds* -) [*50*, *51*] *50*)
where *i in-bounds k* = (*0* ≤ *i* ∧ *i* < *k*)

definition
arr-comps :: ′*a* ⇒ *int* ⇒ *int* ⇒ ′*a option*
where *arr-comps T k* = (λ*i*. *if i in-bounds k then Some T else None*)

definition
var-tys :: *prog* ⇒ *obj-tag* ⇒ *oref* ⇒ (*vn*, *ty*) *table* **where**
var-tys G oi r =
(*case r of*
Heap a ⇒ (*case oi of*
CInst C ⇒ *fields-table G C* (λ*n f*. ¬*static f*) (+) *Map.empty*
| *Arr T k* ⇒ *Map.empty* (+) *arr-comps T k*)
| *Stat C* ⇒ *fields-table G C* (λ*fn f*. *declclassf fn* = *C* ∧ *static f*)
(+) *Map.empty*)

lemma *var-tys-Some-eq*:
var-tys G oi r n = *Some T*
= (*case r of*
Inl a ⇒ (*case oi of*
CInst C ⇒ (∃ *nt*. *n* = *Inl nt* ∧ *fields-table G C* (λ*n f*.
¬*static f*) *nt* = *Some T*)
| *Arr t k* ⇒ (∃ *i*. *n* = *Inr i* ∧ *i in-bounds k* ∧ *t* = *T*))
| *Inr C* ⇒ (∃ *nt*. *n* = *Inl nt* ∧
fields-table G C (λ*fn f*. *declclassf fn* = *C* ∧ *static f*) *nt*

= *Some T*))
apply (*unfold var-tys-def arr-comps-def*)
apply (*force split*: *sum.split-asm sum.split obj-tag.split*)
done

stores

type-synonym *globs* — global variables: heap and static variables
= (*oref* , *obj*) *table*
type-synonym *heap*
= (*loc* , *obj*) *table*

translations
(*type*) *globs* <= (*type*) (*oref* , *obj*) *table*
(*type*) *heap* <= (*type*) (*loc* , *obj*) *table*

datatype *st* =
st globs locals

2 access

definition
globs :: *st* ⇒ *globs*
where *globs* = *case-st* (λ*g l*. *g*)

definition
locals :: *st* ⇒ *locals*
where *locals* = *case-st* (λ*g l*. *l*)

definition *heap* :: *st* ⇒ *heap* **where**
heap s = *globs s* ∘ *Heap*

lemma *globs-def2* [*simp*]: *globs* (*st g l*) = *g*
by (*simp add*: *globs-def*)

lemma *locals-def2* [*simp*]: *locals* (*st g l*) = *l*
by (*simp add*: *locals-def*)

lemma *heap-def2* [*simp*]: *heap s a*=*globs s* (*Heap a*)
by (*simp add*: *heap-def*)

abbreviation *val-this* :: *st* ⇒ *val*
where *val-this s* == *the* (*locals s This*)

abbreviation *lookup-obj* :: *st* ⇒ *val* ⇒ *obj*
where *lookup-obj s a′* == *the* (*heap s* (*the-Addr a′*))

3 memory allocation

definition
new-Addr :: *heap* ⇒ *loc option* **where**
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* $\Longrightarrow$ *h a* = *None*
apply (*auto simp add*: *new-Addr-def*)
apply (*erule someI*)
done

lemma *new-AddrD2*: *new-Addr h* = *Some a* $\Longrightarrow$ $\forall$ *b*. *h b* $\neq$ *None* $\longrightarrow$ *b* $\neq$ *a*
apply (*drule new-AddrD*)
apply *auto*
done

lemma *new-Addr-SomeI*: *h a* = *None* $\Longrightarrow$ $\exists$ *b*. *new-Addr h* = *Some b* $\wedge$ *h b* = *None*
apply (*simp add*: *new-Addr-def*)
apply (*fast intro*: *someI2*)
done

4 initialization

abbreviation *init-vals* :: (*'a*, *ty*) *table* $\Rightarrow$ (*'a*, *val*) *table*
where *init-vals vs* == *map-option default-val* $\circ$ *vs*

lemma *init-arr-comps-base* [*simp*]: *init-vals* (*arr-comps T 0*) = *Map.empty*
apply (*unfold arr-comps-def in-bounds-def*)
apply (*rule ext*)
apply *auto*
done

lemma *init-arr-comps-step* [*simp*]:
0 < *j* $\Longrightarrow$ *init-vals* (*arr-comps T j*) =
(*init-vals* (*arr-comps T* (*j* − *1*)))(*j* − *1*$\mapsto$*default-val T*)
apply (*unfold arr-comps-def in-bounds-def*)
apply (*rule ext*)
apply *auto*
done

5 update

definition
gupd :: *oref* $\Rightarrow$ *obj* $\Rightarrow$ *st* $\Rightarrow$ *st* (*gupd'*(-$\mapsto$-') [*10*, *10*] *1000*)
where *gupd r obj* = *case-st* (λ*g l*. *st* (*g*(*r*$\mapsto$*obj*)) *l*)

definition
lupd :: *lname* $\Rightarrow$ *val* $\Rightarrow$ *st* $\Rightarrow$ *st* (*lupd'*(-$\mapsto$-') [*10*, *10*] *1000*)
where *lupd vn v* = *case-st* (λ*g l*. *st g* (*l*(*vn*$\mapsto$*v*)))

definition
upd-gobj :: *oref* $\Rightarrow$ *vn* $\Rightarrow$ *val* $\Rightarrow$ *st* $\Rightarrow$ *st*
where *upd-gobj r n v* = *case-st* (λ*g l*. *st* (*chg-map* (*upd-obj n v*) *r g*) *l*)

definition
set-locals :: *locals* $\Rightarrow$ *st* $\Rightarrow$ *st*
where *set-locals l* = *case-st* (λ*g l'*. *st g l*)

definition

init-obj :: *prog* ⇒ *obj-tag* ⇒ *oref* ⇒ *st* ⇒ *st*
where *init-obj G oi r* = *gupd*(*r*↦(|*tag*=*oi*, *values*=*init-vals* (*var-tys G oi r*)|))

abbreviation
init-class-obj :: *prog* ⇒ *qtname* ⇒ *st* ⇒ *st*
where *init-class-obj G C* == *init-obj G undefined* (*Inr C*)

lemma *gupd-def2* [*simp*]: *gupd*(*r*↦*obj*) (*st g l*) = *st* (*g*(*r*↦*obj*)) *l*
apply (*unfold gupd-def*)
apply (*simp* (*no-asm*))
done

lemma *lupd-def2* [*simp*]: *lupd*(*vn*↦*v*) (*st g l*) = *st g* (*l*(*vn*↦*v*))
apply (*unfold lupd-def*)
apply (*simp* (*no-asm*))
done

lemma *globs-gupd* [*simp*]: *globs* (*gupd*(*r*↦*obj*) *s*) = (*globs s*)(*r*↦*obj*)
apply (*induct s*)
by (*simp add*: *gupd-def*)

lemma *globs-lupd* [*simp*]: *globs* (*lupd*(*vn*↦*v*) *s*) = *globs s*
apply (*induct s*)
by (*simp add*: *lupd-def*)

lemma *locals-gupd* [*simp*]: *locals* (*gupd*(*r*↦*obj*) *s*) = *locals s*
apply (*induct s*)
by (*simp add*: *gupd-def*)

lemma *locals-lupd* [*simp*]: *locals* (*lupd*(*vn*↦*v*) *s*) = (*locals s*)(*vn*↦*v*)
apply (*induct s*)
by (*simp add*: *lupd-def*)

lemma *globs-upd-gobj-new* [*rule-format* (*no-asm*), *simp*]:
globs s r = *None* ⟶ *globs* (*upd-gobj r n v s*) = *globs s*
apply (*unfold upd-gobj-def*)
apply (*induct s*)
apply *auto*
done

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*)
apply (*unfold upd-gobj-def*)
apply (*induct s*)
apply *auto*
done

lemma *locals-upd-gobj* [*simp*]: *locals* (*upd-gobj r n v s*) = *locals s*
apply (*induct s*)
by (*simp add*: *upd-gobj-def*)

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*)
apply (*unfold init-obj-def*)
apply (*simp* (*no-asm*))
done

lemma *locals-init-obj* [*simp*]: *locals* (*init-obj G oi r s*) = *locals s*
by (*simp add*: *init-obj-def*)

lemma *surjective-st* [*simp*]: *st* (*globs s*) (*locals s*) = *s*
apply (*induct s*)
by *auto*

lemma *surjective-st-init-obj*:
st (*globs* (*init-obj G oi r s*)) (*locals s*) = *init-obj G oi r s*
apply (*subst locals-init-obj* [*THEN sym*])
apply (*rule surjective-st*)
done

lemma *heap-heap-upd* [*simp*]:
heap (*st* (*g*(*Inl a*↦*obj*)) *l*) = (*heap* (*st g l*))(*a*↦*obj*)
apply (*rule ext*)
apply (*simp* (*no-asm*))
done

lemma *heap-stat-upd* [*simp*]: *heap* (*st* (*g*(*Inr C*↦*obj*)) *l*) = *heap* (*st g l*)
apply (*rule ext*)
apply (*simp* (*no-asm*))
done

lemma *heap-local-upd* [*simp*]: *heap* (*st g* (*l*(*vn*↦*v*))) = *heap* (*st g l*)
apply (*rule ext*)
apply (*simp* (*no-asm*))
done

lemma *heap-gupd-Heap* [*simp*]: *heap* (*gupd*(*Heap a*↦*obj*) *s*) = (*heap s*)(*a*↦*obj*)
apply (*rule ext*)
apply (*simp* (*no-asm*))
done

lemma *heap-gupd-Stat* [*simp*]: *heap* (*gupd*(*Stat C*↦*obj*) *s*) = *heap s*
apply (*rule ext*)
apply (*simp* (*no-asm*))
done

lemma *heap-lupd* [*simp*]: *heap* (*lupd*(*vn*↦*v*) *s*) = *heap s*
apply (*rule ext*)
apply (*simp* (*no-asm*))
done

lemma *heap-upd-gobj-Stat* [*simp*]: *heap* (*upd-gobj* (*Stat C*) *n v s*) = *heap s*
apply (*rule ext*)
apply (*simp* (*no-asm*))
apply (*case-tac globs s* (*Stat C*))
apply *auto*
done

lemma *set-locals-def2* [*simp*]: *set-locals l* (*st g l′*) = *st g l*
apply (*unfold set-locals-def*)
apply (*simp* (*no-asm*))
done

lemma *set-locals-id* [*simp*]: *set-locals* (*locals s*) *s* = *s*
apply (*unfold set-locals-def*)
apply (*induct-tac s*)
apply (*simp* (*no-asm*))
done

lemma *set-set-locals* [*simp*]: *set-locals l* (*set-locals l′ s*) = *set-locals l s*
apply (*unfold set-locals-def*)
apply (*induct-tac s*)
apply (*simp* (*no-asm*))
done

lemma *locals-set-locals* [*simp*]: *locals* (*set-locals l s*) = *l*
apply (*unfold set-locals-def*)
apply (*induct-tac s*)
apply (*simp* (*no-asm*))
done

lemma *globs-set-locals* [*simp*]: *globs* (*set-locals l s*) = *globs s*
apply (*unfold set-locals-def*)
apply (*induct-tac s*)
apply (*simp* (*no-asm*))
done

lemma *heap-set-locals* [*simp*]: *heap* (*set-locals l s*) = *heap s*
apply (*unfold heap-def*)
apply (*induct-tac s*)
apply (*simp* (*no-asm*))
done

abrupt completion

primrec *the-Xcpt* :: *abrupt* ⇒ *xcpt*
where *the-Xcpt* (*Xcpt x*) = *x*

primrec *the-Jump* :: *abrupt* => *jump*
where *the-Jump* (*Jump j*) = *j*

primrec *the-Loc* :: *xcpt* ⇒ *loc*
where *the-Loc* (*Loc a*) = *a*

primrec *the-Std* :: *xcpt* ⇒ *xname*
  where *the-Std* (*Std x*) = *x*

definition
  abrupt-if :: *bool* ⇒ *abopt* ⇒ *abopt* ⇒ *abopt*
  where *abrupt-if c x′ x* = (*if c* ∧ (*x* = *None*) *then x′ else x*)

lemma *abrupt-if-True-None* [*simp*]: *abrupt-if True x None* = *x*
by (*simp add*: *abrupt-if-def*)

lemma *abrupt-if-True-not-None* [*simp*]: *x* ≠ *None* ⟹ *abrupt-if True x y* ≠ *None*
by (*simp add*: *abrupt-if-def*)

lemma *abrupt-if-False* [*simp*]: *abrupt-if False x y* = *y*
by (*simp add*: *abrupt-if-def*)

lemma *abrupt-if-Some* [*simp*]: *abrupt-if c x* (*Some y*) = *Some y*
by (*simp add*: *abrupt-if-def*)

lemma *abrupt-if-not-None* [*simp*]: *y* ≠ *None* ⟹ *abrupt-if c x y* = *y*
apply (*simp add*: *abrupt-if-def*)
by *auto*

lemma *split-abrupt-if*:
P (*abrupt-if c x′ x*) =
      ((*c* ∧ *x* = *None* ⟶ *P x′*) ∧ (¬ (*c* ∧ *x* = *None*) ⟶ *P x*))
apply (*unfold abrupt-if-def*)
apply (*split if-split*)
apply *auto*
done

abbreviation *raise-if* :: *bool* ⇒ *xname* ⇒ *abopt* ⇒ *abopt*
  where *raise-if c xn* == *abrupt-if c* (*Some* (*Xcpt* (*Std xn*)))

abbreviation *np* :: *val* ⇒ *abopt* ⇒ *abopt*
  where *np v* == *raise-if* (*v* = *Null*) *NullPointer*

abbreviation *check-neg* :: *val* ⇒ *abopt* ⇒ *abopt*
  where *check-neg i′* == *raise-if* (*the-Intg i′*<*0*) *NegArrSize*

abbreviation *error-if* :: *bool* ⇒ *error* ⇒ *abopt* ⇒ *abopt*
  where *error-if c e* == *abrupt-if c* (*Some* (*Error e*))

lemma *raise-if-None* [*simp*]: (*raise-if c x y* = *None*) = (¬*c* ∧ *y* = *None*)
apply (*simp add*: *abrupt-if-def*)
by *auto*
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*)
apply (*simp add*: *abrupt-if-def*)
apply *auto*
done

lemma *raise-if-SomeD* [*dest*!]:
raise-if c x y = *Some z* ⟹ *c* ∧ *z*=(*Xcpt* (*Std x*)) ∧ *y*=*None* ∨ (*y*=*Some z*)
apply (*case-tac y*)
apply (*case-tac c*)
apply (*simp add*: *abrupt-if-def*)
apply (*simp add*: *abrupt-if-def*)
apply *auto*
done

lemma *error-if-None* [*simp*]: (*error-if c e y* = *None*) = (¬*c* ∧ *y* = *None*)
apply (*simp add*: *abrupt-if-def*)
by *auto*
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*)
apply (*simp add*: *abrupt-if-def*)
apply *auto*
done

lemma *error-if-SomeD* [*dest*!]:
error-if c e y = *Some z* ⟹ *c* ∧ *z*=(*Error e*) ∧ *y*=*None* ∨ (*y*=*Some z*)
apply (*case-tac y*)
apply (*case-tac c*)
apply (*simp add*: *abrupt-if-def*)
apply (*simp add*: *abrupt-if-def*)
apply *auto*
done

definition
absorb :: *jump* ⇒ *abopt* ⇒ *abopt*
where *absorb j a* = (*if a*=*Some* (*Jump j*) *then None else a*)

lemma *absorb-SomeD* [*dest*!]: *absorb j a* = *Some x* ⟹ *a* = *Some x*
by (*auto simp add*: *absorb-def*)

lemma *absorb-same* [*simp*]: *absorb j* (*Some* (*Jump j*)) = *None*
by (*auto simp add*: *absorb-def*)

lemma *absorb-other* [*simp*]: *a* ≠ *Some* (*Jump j*) ⟹ *absorb j a* = *a*
by (*auto simp add*: *absorb-def*)

lemma *absorb-Some-NoneD*: *absorb j* (*Some abr*) = *None* ⟹ *abr* = *Jump j*
by (*simp add*: *absorb-def*)

lemma *absorb-Some-JumpD*: *absorb j s* = *Some* (*Jump j′*) $\Longrightarrow$ *j′≠j*
by (*simp add*: *absorb-def*)

full program state

type-synonym
state = *abopt* × *st* — state including abruption information

translations
(*type*) *abopt* <= (*type*) *abrupt option*
(*type*) *state* <= (*type*) *abopt* × *st*

abbreviation
Norm :: *st* ⇒ *state*
where *Norm s* == (*None*, *s*)

abbreviation (*input*)
abrupt :: *state* ⇒ *abopt*
where *abrupt* == *fst*

abbreviation (*input*)
store :: *state* ⇒ *st*
where *store* == *snd*

lemma *single-stateE*: ∀ *Z*. *Z* = (*s*::*state*) $\Longrightarrow$ *False*
apply (*erule-tac x* = (*Some k*,*y*) **for** *k y* **in** *all-dupE*)
apply (*erule-tac x* = (*None*,*y*) **for** *y* **in** *allE*)
apply *clarify*
done

lemma *state-not-single*: *All* ((=) (*x*::*state*)) $\Longrightarrow$ *R*
apply (*drule-tac x* = (*if abrupt x* = *None then Some x′ else None*, *y*) **for** *x′ y* **in** *spec*)
apply *clarsimp*
done

definition
normal :: *state* ⇒ *bool*
where *normal* = (λ*s*. *abrupt s* = *None*)

lemma *normal-def2* [*simp*]: *normal s* = (*abrupt s* = *None*)
apply (*unfold normal-def*)
apply (*simp* (*no-asm*))
done

definition
heap-free :: *nat* ⇒ *state* ⇒ *bool*
where *heap-free n* = (λ*s*. *atleast-free* (*heap* (*store s*)) *n*)

lemma *heap-free-def2* [*simp*]: *heap-free n s* = *atleast-free* (*heap* (*store s*)) *n*
apply (*unfold heap-free-def*)
apply *simp*
done

6 update

definition
abupd :: (*abopt* ⇒ *abopt*) ⇒ *state* ⇒ *state*
where *abupd f* = *map-prod f id*

definition
supd :: (*st* ⇒ *st*) ⇒ *state* ⇒ *state*
where *supd* = *map-prod id*

lemma *abupd-def2* [*simp*]: *abupd f* (*x*,*s*) = (*f x*,*s*)
by (*simp add*: *abupd-def*)

lemma *abupd-abrupt-if-False* [*simp*]: ⋀ *s*. *abupd* (*abrupt-if False xo*) *s* = *s*
by *simp*

lemma *supd-def2* [*simp*]: *supd f* (*x*,*s*) = (*x*,*f s*)
by (*simp add*: *supd-def*)

lemma *supd-lupd* [*simp*]:
⋀ *s*. *supd* (*lupd vn v*) *s* = (*abrupt s*,*lupd vn v* (*store s*))
apply (*simp* (*no-asm-simp*) *only*: *split-tupled-all*)
apply (*simp* (*no-asm*))
done

lemma *supd-gupd* [*simp*]:
⋀ *s*. *supd* (*gupd r obj*) *s* = (*abrupt s*,*gupd r obj* (*store s*))
apply (*simp* (*no-asm-simp*) *only*: *split-tupled-all*)
apply (*simp* (*no-asm*))
done

lemma *supd-init-obj* [*simp*]:
supd (*init-obj G oi r*) *s* = (*abrupt s*,*init-obj G oi r* (*store s*))
apply (*unfold init-obj-def*)
apply (*simp* (*no-asm*))
done

lemma *abupd-store-invariant* [*simp*]: *store* (*abupd f s*) = *store s*
by (*cases s*) *simp*

lemma *supd-abrupt-invariant* [*simp*]: *abrupt* (*supd f s*) = *abrupt s*
by (*cases s*) *simp*

abbreviation *set-lvars* :: *locals* ⇒ *state* ⇒ *state*
where *set-lvars l* == *supd* (*set-locals l*)

abbreviation *restore-lvars* :: *state* ⇒ *state* ⇒ *state*
where *restore-lvars s′ s* == *set-lvars* (*locals* (*store s′*)) *s*

lemma *set-set-lvars* [*simp*]: $\bigwedge$ *s. set-lvars l* (*set-lvars l′ s*) = *set-lvars l s*
apply (*simp* (*no-asm-simp*) *only*: *split-tupled-all*)
apply (*simp* (*no-asm*))
done

lemma *set-lvars-id* [*simp*]: $\bigwedge$ *s. set-lvars* (*locals* (*store s*)) *s* = *s*
apply (*simp* (*no-asm-simp*) *only*: *split-tupled-all*)
apply (*simp* (*no-asm*))
done

initialisation test

definition
inited :: *qtname* ⇒ *globs* ⇒ *bool*
where *inited C g* = (*g* (*Stat C*) ≠ *None*)

definition
initd :: *qtname* ⇒ *state* ⇒ *bool*
where *initd C* = *inited C* ∘ *globs* ∘ *store*

lemma *not-inited-empty* [*simp*]: ¬*inited C Map.empty*
apply (*unfold inited-def*)
apply (*simp* (*no-asm*))
done

lemma *inited-gupdate* [*simp*]: *inited C* (*g*(*r*↦*obj*)) = (*inited C g* ∨ *r* = *Stat C*)
apply (*unfold inited-def*)
apply (*auto split*: *st.split*)
done

lemma *inited-init-class-obj* [*intro*!]: *inited C* (*globs* (*init-class-obj G C s*))
apply (*unfold inited-def*)
apply (*simp* (*no-asm*))
done

lemma *not-initedD*: ¬ *inited C g* ⟹ *g* (*Stat C*) = *None*
apply (*unfold inited-def*)
apply (*erule notnotD*)
done

lemma *initedD*: *inited C g* ⟹ ∃ *obj*. *g* (*Stat C*) = *Some obj*
apply (*unfold inited-def*)
apply *auto*
done

lemma *initd-def2* [*simp*]: *initd C s* = *inited C* (*globs* (*store s*))
apply (*unfold initd-def*)
apply (*simp* (*no-asm*))
done

error-free

definition
error-free :: *state* ⇒ *bool*
where *error-free s* = (¬ (∃ *err*. *abrupt s* = *Some* (*Error err*)))

lemma *error-free-Norm* [*simp*,*intro*]: *error-free* (*Norm s*)
by (*simp add*: *error-free-def*)

lemma *error-free-normal* [*simp*,*intro*]: *normal s* ⟹ *error-free s*
by (*simp add*: *error-free-def*)

lemma *error-free-Xcpt* [*simp*]: *error-free* (*Some* (*Xcpt x*),*s*)
by (*simp add*: *error-free-def*)

lemma *error-free-Jump* [*simp*,*intro*]: *error-free* (*Some* (*Jump j*),*s*)
by (*simp add*: *error-free-def*)

lemma *error-free-Error* [*simp*]: *error-free* (*Some* (*Error e*),*s*) = *False*
by (*simp add*: *error-free-def*)

lemma *error-free-Some* [*simp*,*intro*]:
¬ (∃ *err*. *x*=*Error err*) ⟹ *error-free* ((*Some x*),*s*)
by (*auto simp add*: *error-free-def*)

lemma *error-free-abupd-absorb* [*simp*,*intro*]:
error-free s ⟹ *error-free* (*abupd* (*absorb j*) *s*)
by (*cases s*)
(*auto simp add*: *error-free-def absorb-def*
split: *if-split-asm*)

lemma *error-free-absorb* [*simp*,*intro*]:
error-free (*a*,*s*) ⟹ *error-free* (*absorb j a*, *s*)
by (*auto simp add*: *error-free-def absorb-def*
split: *if-split-asm*)

lemma *error-free-abrupt-if* [*simp*,*intro*]:
⟦*error-free s*; ¬ (∃ *err*. *x*=*Error err*)⟧
⟹ *error-free* (*abupd* (*abrupt-if p* (*Some x*)) *s*)
by (*cases s*)
(*auto simp add*: *abrupt-if-def*
split: *if-split*)

lemma *error-free-abrupt-if1* [*simp*,*intro*]:
⟦*error-free* (*a*,*s*); ¬ (∃ *err*. *x*=*Error err*)⟧
⟹ *error-free* (*abrupt-if p* (*Some x*) *a*, *s*)
by (*auto simp add*: *abrupt-if-def*
split: *if-split*)

lemma *error-free-abrupt-if-Xcpt* [*simp*,*intro*]:
error-free s
⟹ *error-free* (*abupd* (*abrupt-if p* (*Some* (*Xcpt x*))) *s*)
by *simp*

lemma *error-free-abrupt-if-Xcpt1* [*simp*,*intro*]:
error-free (*a*,*s*)
⟹ *error-free* (*abrupt-if p* (*Some* (*Xcpt x*)) *a*, *s*)
by *simp*

lemma *error-free-abrupt-if-Jump* [*simp*,*intro*]:
error-free s
⟹ *error-free* (*abupd* (*abrupt-if p* (*Some* (*Jump j*))) *s*)
by *simp*

lemma *error-free-abrupt-if-Jump1* [*simp*,*intro*]:
error-free (*a*,*s*)
⟹ *error-free* (*abrupt-if p* (*Some* (*Jump j*)) *a*, *s*)
by *simp*

lemma *error-free-raise-if* [*simp*,*intro*]:
error-free s ⟹ *error-free* (*abupd* (*raise-if p x*) *s*)
by *simp*

lemma *error-free-raise-if1* [*simp*,*intro*]:
error-free (*a*,*s*) ⟹ *error-free* ((*raise-if p x a*), *s*)
by *simp*

lemma *error-free-supd* [*simp*,*intro*]:
error-free s ⟹ *error-free* (*supd f s*)
by (*cases s*) (*simp add*: *error-free-def*)

lemma *error-free-supd1* [*simp*,*intro*]:
error-free (*a*,*s*) ⟹ *error-free* (*a*,*f s*)
by (*simp add*: *error-free-def*)

lemma *error-free-set-lvars* [*simp*,*intro*]:
error-free s ⟹ *error-free* ((*set-lvars l*) *s*)
by (*cases s*) *simp*

lemma *error-free-set-locals* [*simp*,*intro*]:
error-free (*x*, *s*)
⟹ *error-free* (*x*, *set-locals l s'*)
by (*simp add*: *error-free-def*)

end

Chapter 15

Eval

1 Operational evaluation (big-step) semantics of Java expressions and statements

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

improvements over Java Specification 1.0:

- dynamic method lookup does not need to consider the return type (cf.15.11.4.4)
- throw raises a NullPointer exception if a null reference is given, and each throw of a standard exception yield a fresh exception object (was not specified)
- if there is not enough memory even to allocate an OutOfMemory exception, evaluation/execution fails, i.e. simply stops (was not specified)
- array assignment checks lhs (and may throw exceptions) before evaluating rhs
- fixed exact positions of class initializations (immediate at first active use)

design issues:

- evaluation vs. (single-step) transition semantics evaluation semantics chosen, because:
  - ++ less verbose and therefore easier to read (and to handle in proofs)
  - \+ more abstract
  - \+ intermediate values (appearing in recursive rules) need not be stored explicitly, e.g. no call body construct or stack of invocation frames containing local variables and return addresses for method calls needed
  - \+ convenient rule induction for subject reduction theorem
  - \- no interleaving (for parallelism) can be described
  - \- stating a property of infinite executions requires the meta-level argument that this property holds for any finite prefixes of it (e.g. stopped using a counter that is decremented to zero and then throwing an exception)
- unified evaluation for variables, expressions, expression lists, statements
- the value entry in statement rules is redundant
- the value entry in rules is irrelevant in case of exceptions, but its full inclusion helps to make the rule structure independent of exception occurrence.
- as irrelevant value entries are ignored, it does not matter if they are unique. For simplicity, (fixed) arbitrary values are preferred over "free" values.

- the rule format is such that the start state may contain an exception.
  - ++ faciliates exception handling
  - + symmetry
- the rules are defined carefully in order to be applicable even in not type-correct situations (yielding undefined values), e.g. *the-Addr* (*Val* (*Bool b*)) = *undefined*.
  - ++ fewer rules
  - - less readable because of auxiliary functions like *the-Addr*

  Alternative: "defensive" evaluation throwing some InternalError exception in case of (impossible, for correct programs) type mismatches
- there is exactly one rule per syntactic construct
  - + no redundancy in case distinctions
- halloc fails iff there is no free heap address. When there is only one free heap address left, it returns an OutOfMemory exception. In this way it is guaranteed that when an OutOfMemory exception is thrown for the first time, there is a free location on the heap to allocate it.
- the allocation of objects that represent standard exceptions is deferred until execution of any enclosing catch clause, which is transparent to the program.
  - - requires an auxiliary execution relation
  - ++ avoids copies of allocation code and awkward case distinctions (whether there is enough memory to allocate the exception) in evaluation rules
- unfortunately *new-Addr* is not directly executable because of Hilbert operator.

simplifications:

- local variables are initialized with default values (no definite assignment)
- garbage collection not considered, therefore also no finalizers
- stack overflow and memory overflow during class initialization not modelled
- exceptions in initializations not replaced by ExceptionInInitializerError

type-synonym *vvar* = *val* × (*val* ⇒ *state* ⇒ *state*)
type-synonym *vals* = (*val*, *vvar*, *val list*) *sum3*
translations
(*type*) *vvar* <= (*type*) *val* × (*val* ⇒ *state* ⇒ *state*)
(*type*) *vals* <= (*type*) (*val*, *vvar*, *val list*) *sum3*

To avoid redundancy and to reduce the number of rules, there is only one evaluation rule for each syntactic term. This is also true for variables (e.g. see the rules below for *LVar*, *FVar* and *AVar*). So evaluation of a variable must capture both possible further uses: read (rule *Acc*) or write (rule *Ass*) to the variable. Therefor a variable evaluates to a special value *vvar*, which is a pair, consisting of the current value (for later read access) and an update function (for later write access). Because during assignment to an array variable an exception may occur if the types don't match, the update function is very generic: it transforms the full state. This generic update function causes some technical trouble during some proofs (e.g. type safety, correctness of definite assignment). There we need to prove some additional invariant on this update function to prove the assignment correct, since the update function could potentially alter the whole state in an arbitrary manner. This invariant must be carried around through the whole induction. So for future approaches it may be better not to take such a generic update function, but only to store the address and the kind of variable (array (+ element type), local variable or field) for later assignment.

abbreviation
dummy-res :: *vals* (◇)
where ◇ == *In1 Unit*

abbreviation (*input*)
val-inj-vals ($\lfloor$-$\rfloor_e$ *1000*)
where $\lfloor e \rfloor_e$ == *In1 e*

abbreviation (*input*)
var-inj-vals ($\lfloor$-$\rfloor_v$ *1000*)
where $\lfloor v \rfloor_v$ == *In2 v*

abbreviation (*input*)
lst-inj-vals ($\lfloor$-$\rfloor_l$ *1000*)
where $\lfloor es \rfloor_l$ == *In3 es*

definition *undefined3* :: (*'al* + *'ar*, *'b*, *'c*) *sum3* ⇒ *vals* **where**
undefined3 = *case-sum3* (*In1* ∘ *case-sum* (*λx. undefined*) (*λx. Unit*))
(*λx. In2 undefined*) (*λx. In3 undefined*)

lemma [*simp*]: *undefined3* (*In1l x*) = *In1 undefined*
by (*simp add*: *undefined3-def*)

lemma [*simp*]: *undefined3* (*In1r x*) = ◇
by (*simp add*: *undefined3-def*)

lemma [*simp*]: *undefined3* (*In2 x*) = *In2 undefined*
by (*simp add*: *undefined3-def*)

lemma [*simp*]: *undefined3* (*In3 x*) = *In3 undefined*
by (*simp add*: *undefined3-def*)

exception throwing and catching

definition
throw :: *val* ⇒ *abopt* ⇒ *abopt* **where**
throw a' x = *abrupt-if True* (*Some* (*Xcpt* (*Loc* (*the-Addr a'*)))) (*np a' x*)

lemma *throw-def2*:
throw a' x = *abrupt-if True* (*Some* (*Xcpt* (*Loc* (*the-Addr a'*)))) (*np a' x*)
apply (*unfold throw-def*)
apply (*simp* (*no-asm*))
done

definition
fits :: *prog* ⇒ *st* ⇒ *val* ⇒ *ty* ⇒ *bool* (-,-⊢- *fits* -[*61*,*61*,*61*,*61*]*60*)
where *G*,*s*⊢*a' fits T* = ((∃*rt*. *T*=*RefT rt*) ⟶ *a'*=*Null* ∨ *G*⊢*obj-ty*(*lookup-obj s a'*)≼*T*)

lemma *fits-Null* [*simp*]: *G*,*s*⊢*Null fits T*

by (*simp add*: *fits-def*)

lemma *fits-Addr-RefT* [*simp*]:
  $G,s\vdash Addr\ a\ fits\ RefT\ t = G\vdash obj\text{-}ty\ (the\ (heap\ s\ a))\preceq RefT\ t$
by (*simp add*: *fits-def*)

lemma *fitsD*: $\bigwedge X.\ G,s\vdash a'\ fits\ T \Longrightarrow (\exists pt.\ T = PrimT\ pt) \vee$
  $(\exists t.\ T = RefT\ t) \wedge a' = Null \vee$
  $(\exists t.\ T = RefT\ t) \wedge a' \neq Null \wedge\ G\vdash obj\text{-}ty\ (lookup\text{-}obj\ s\ a')\preceq T$
apply (*unfold fits-def*)
apply (*case-tac* $\exists pt.\ T = PrimT\ pt$)
apply *simp-all*
apply (*case-tac T*)
defer
apply (*case-tac* $a' = Null$)
apply *simp-all*
done

definition
  catch :: *prog* ⇒ *state* ⇒ *qtname* ⇒ *bool* (-,-⊢*catch* -[*61*,*61*,*61*]*60*) **where**
  $G,s\vdash catch\ C = (\exists xc.\ abrupt\ s{=}Some\ (Xcpt\ xc) \wedge$
      $G,store\ s\vdash Addr\ (the\text{-}Loc\ xc)\ fits\ Class\ C)$

lemma *catch-Norm* [*simp*]: $\neg G,Norm\ s\vdash catch\ tn$
apply (*unfold catch-def*)
apply (*simp* (*no-asm*))
done

lemma *catch-XcptLoc* [*simp*]:
  $G,(Some\ (Xcpt\ (Loc\ a)),s)\vdash catch\ C = G,s\vdash Addr\ a\ fits\ Class\ C$
apply (*unfold catch-def*)
apply (*simp* (*no-asm*))
done

lemma *catch-Jump* [*simp*]: $\neg G,(Some\ (Jump\ j),s)\vdash catch\ tn$
apply (*unfold catch-def*)
apply (*simp* (*no-asm*))
done

lemma *catch-Error* [*simp*]: $\neg G,(Some\ (Error\ e),s)\vdash catch\ tn$
apply (*unfold catch-def*)
apply (*simp* (*no-asm*))
done

definition
  new-xcpt-var :: *vname* ⇒ *state* ⇒ *state* **where**
  $new\text{-}xcpt\text{-}var\ vn = (\lambda(x,s).\ Norm\ (lupd(VName\ vn\mapsto Addr\ (the\text{-}Loc\ (the\text{-}Xcpt\ (the\ x))))\ s))$

lemma *new-xcpt-var-def2* [*simp*]:
 $new\text{-}xcpt\text{-}var\ vn\ (x,s) =$
   $Norm\ (lupd(VName\ vn\mapsto Addr\ (the\text{-}Loc\ (the\text{-}Xcpt\ (the\ x))))\ s)$

apply (*unfold new-xcpt-var-def*)
apply (*simp* (*no-asm*))
done

misc

definition
assign :: ($'a \Rightarrow state \Rightarrow state$) $\Rightarrow$ $'a \Rightarrow state \Rightarrow state$ **where**
assign f v = ($\lambda(x,s)$. *let* (x',s') = (*if* $x = None$ *then f v else id*) (x,s)
in (x',*if* $x' = None$ *then* s' *else* s))

lemma *assign-Norm-Norm* [*simp*]:
f v (*Norm s*) = *Norm* s' $\Longrightarrow$ *assign f v* (*Norm s*) = *Norm* s'
by (*simp add*: *assign-def Let-def*)

lemma *assign-Norm-Some* [*simp*]:
⟦*abrupt* (*f v* (*Norm s*)) = *Some y*⟧
$\Longrightarrow$ *assign f v* (*Norm s*) = (*Some y*,*s*)
by (*simp add*: *assign-def Let-def split-beta*)

lemma *assign-Some* [*simp*]:
assign f v (*Some x*,*s*) = (*Some x*,*s*)
by (*simp add*: *assign-def Let-def split-beta*)

lemma *assign-Some1* [*simp*]: $\neg$ *normal s* $\Longrightarrow$ *assign f v s* = *s*
by (*auto simp add*: *assign-def Let-def split-beta*)

lemma *assign-supd* [*simp*]:
assign (λv. *supd* (*f v*)) *v* (*x*,*s*)
= (*x*, *if* $x = None$ *then f v s else s*)
apply *auto*
done

lemma *assign-raise-if* [*simp*]:
assign (λv (*x*,*s*). ((*raise-if* (*b s v*) *xcpt*) *x*, *f v s*)) *v* (*x*, *s*) =
(*raise-if* (*b s v*) *xcpt x*, *if* x=*None* $\wedge$ $\neg$*b s v then f v s else s*)
apply (*case-tac* $x = None$)
apply *auto*
done

definition
init-comp-ty :: *ty* $\Rightarrow$ *stmt*
where *init-comp-ty T* = (*if* ($\exists$ *C*. *T* = *Class C*) *then Init* (*the-Class T*) *else Skip*)

```
lemma init-comp-ty-PrimT [simp]: init-comp-ty (PrimT pt) = Skip
apply (unfold init-comp-ty-def)
apply (simp (no-asm))
done

definition
  invocation-class :: inv-mode ⇒ st ⇒ val ⇒ ref-ty ⇒ qtname where
 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'))

definition
  invocation-declclass :: prog ⇒ inv-mode ⇒ st ⇒ val ⇒ ref-ty ⇒ sig ⇒ qtname where
  invocation-declclass G m s a' statT sig =
    declclass (the (dynlookup G statT
                              (invocation-class m s a' statT)
                              sig))


lemma invocation-class-IntVir [simp]:
invocation-class IntVir s a' statT = obj-class (lookup-obj s a')
by (simp add: invocation-class-def)


lemma dynclass-SuperM [simp]:
 invocation-class SuperM s a' statT = the-Class (RefT statT)
by (simp add: invocation-class-def)


lemma invocation-class-Static [simp]:
  invocation-class Static s a' statT = (if (∃ statC. statT = ClassT statC)
                                        then the-Class (RefT statT)
                                        else Object)
by (simp add: invocation-class-def)

definition
  init-lvars :: prog ⇒ qtname ⇒ sig ⇒ inv-mode ⇒ val ⇒ val list ⇒ state ⇒ state
where
  init-lvars G C sig mode a' pvs =
    (λ(x,s).
      let m = mthd (the (methd G C sig));
          l = λ k.
              (case k of
                 EName e
                   ⇒ (case e of
                         VNam v ⇒ (Map.empty ((pars m)[↦]pvs)) v
                       | Res    ⇒ None)
               | This
                   ⇒ (if mode=Static then None else Some a'))
      in set-lvars l (if mode = Static then x else np a' x,s))



lemma init-lvars-def2: — better suited for simplification
```

```
init-lvars G C sig mode a' pvs (x,s) =
  set-lvars
    (λ k.
       (case k of
          EName e
            ⇒ (case e of
                  VNam v
                  ⇒ (Map.empty ((pars (mthd (the (methd G C sig))))[↦]pvs)) v
                | Res ⇒ None)
        | This
            ⇒ (if mode=Static then None else Some a')))
    (if mode = Static then x else np a' x,s)
```
apply (*unfold init-lvars-def*)
apply (*simp* (*no-asm*) *add*: *Let-def*)
done

definition
```
  body :: prog ⇒ qtname ⇒ sig ⇒ expr where
 body G C sig =
    (let m = the (methd G C sig)
     in Body (declclass m) (stmt (mbody (mthd m))))
```

lemma *body-def2*: — better suited for simplification
```
body G C sig = Body  (declclass (the (methd G C sig)))
                          (stmt (mbody (mthd (the (methd G C sig)))))
```
apply (*unfold body-def Let-def*)
apply *auto*
done

variables

definition
```
  lvar :: lname ⇒ st ⇒ vvar
  where lvar vn s = (the (locals s vn), λv. supd (lupd(vn↦v)))
```

definition
```
  fvar :: qtname ⇒ bool ⇒ vname ⇒ val ⇒ state ⇒ vvar × state where
 fvar C stat fn a' s =
   (let (oref,xf) = if stat then (Stat C,id)
                           else (Heap (the-Addr a'),np a');
                  n = Inl (fn,C);
                  f = (λv. supd (upd-gobj oref n v))
    in ((the (values (the (globs (store s) oref)) n),f),abupd xf s))
```

definition
```
  avar :: prog ⇒ val ⇒ val ⇒ state ⇒ vvar × state where
  avar G i' a' s =
    (let   oref = Heap (the-Addr a');
              i = the-Intg i';
              n = Inr i;
       (T,k,cs) = the-Arr (globs (store s) oref);
              f = (λv (x,s). (raise-if (¬G,s⊢v fits T)
                                           ArrStore x
                             ,upd-gobj oref n v s))
     in ((the (cs n),f),abupd (raise-if (¬i in-bounds k) IndOutBound ∘ np a') s))
```

lemma *fvar-def2*: — better suited for simplification

fvar C stat fn a′ s =
  ((*the*
    (*values*
     (*the* (*globs* (*store s*) (*if stat then Stat C else Heap* (*the-Addr a′*))))
     (*Inl* (*fn*,*C*)))
   ,(λ*v*. *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*)

apply (*unfold fvar-def*)
apply (*simp* (*no-asm*) *add*: *Let-def split-beta*)
done

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′*)))
   ,(λ*v* (*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)
apply (*unfold avar-def*)
apply (*simp* (*no-asm*) *add*: *Let-def split-beta*)
done

definition
  check-field-access :: *prog* ⇒ *qtname* ⇒ *qtname* ⇒ *vname* ⇒ *bool* ⇒ *val* ⇒ *state* ⇒ *state* **where**
  check-field-access G accC statDeclC fn stat a′ s =
   (*let oref* = *if stat then Stat statDeclC*
             else Heap (*the-Addr a′*);
      dynC = *case oref of*
              Heap a ⇒ *obj-class* (*the* (*globs* (*store s*) *oref*))
             | *Stat C* ⇒ *C*;
      f = (*the* (*table-of* (*DeclConcepts.fields G dynC*) (*fn*,*statDeclC*)))
    in abupd
      (*error-if* (¬ *G*⊢*Field fn* (*statDeclC*,*f*) *in dynC dyn-accessible-from accC*)
              AccessViolation)
      s)

definition
  check-method-access :: *prog* ⇒ *qtname* ⇒ *ref-ty* ⇒ *inv-mode* ⇒ *sig* ⇒ *val* ⇒ *state* ⇒ *state* **where**
  check-method-access G accC statT mode sig a′ s =
   (*let invC* = *invocation-class mode* (*store s*) *a′ statT*;
     dynM = *the* (*dynlookup G statT invC sig*)
    in abupd
      (*error-if* (¬ *G*⊢*Methd sig dynM in invC dyn-accessible-from accC*)
              AccessViolation)
      s)

evaluation judgments

inductive
  halloc :: [*prog*,*state*,*obj-tag*,*loc*,*state*]⇒*bool* (-⊢- −*halloc* -≻-→ -[*61*,*61*,*61*,*61*,*61*]*60*) **for** *G*::*prog*

where — allocating objects on the heap, cf. 12.5

Abrupt:
G⊢(Some x,s) −halloc oi≻undefined→ (Some x,s)

| *New*: ⟦new-Addr (heap s) = Some a;
(x,oi′) = (if atleast-free (heap s) (Suc (Suc 0)) then (None,oi)
else (Some (Xcpt (Loc a)),CInst (SXcpt OutOfMemory)))⟧
⟹
G⊢Norm s −halloc oi≻a→ (x,init-obj G oi′ (Heap a) s)

inductive *sxalloc* :: [prog,state,state]⇒bool (-⊢- −sxalloc→ -[61,61,61]60) **for** G::prog
where — allocating exception objects for standard exceptions (other than OutOfMemory)

Norm: G⊢ Norm s −sxalloc→ Norm s

| *Jmp*: G⊢(Some (Jump j), s) −sxalloc→ (Some (Jump j), s)

| *Error*: G⊢(Some (Error e), s) −sxalloc→ (Some (Error e), s)

| *XcptL*: G⊢(Some (Xcpt (Loc a)),s) −sxalloc→ (Some (Xcpt (Loc a)),s)

| *SXcpt*: ⟦G⊢Norm s0 −halloc (CInst (SXcpt xn))≻a→ (x,s1)⟧ ⟹
G⊢(Some (Xcpt (Std xn)),s0) −sxalloc→ (Some (Xcpt (Loc a)),s1)

inductive
eval :: [prog,state,term,vals,state]⇒bool (-⊢- −-≻→ ′(-, -′) [61,61,80,0,0]60)
and *exec* ::[prog,state,stmt ,state]⇒bool(-⊢- −-→ - [61,61,65, 61]60)
and *evar* ::[prog,state,var ,vvar,state]⇒bool(-⊢- −-=≻-→ -[61,61,90,61,61]60)
and *eval′*::[prog,state,expr ,val ,state]⇒bool(-⊢- −-−≻-→ -[61,61,80,61,61]60)
and *evals*::[prog,state,expr list ,
val list ,state]⇒bool(-⊢- −-≐≻-→ -[61,61,61,61,61]60)
for G::prog
where

G⊢s −c → s′ ≡ G⊢s −In1r c≻→ (◇, s′)
| G⊢s −e−≻v → s′ ≡ G⊢s −In1l e≻→ (In1 v, s′)
| G⊢s −e=≻vf→ s′ ≡ G⊢s −In2 e≻→ (In2 vf, s′)
| G⊢s −e≐≻v → s′ ≡ G⊢s −In3 e≻→ (In3 v, s′)

— propagation of abrupt completion

— cf. 14.1, 15.5
| *Abrupt*:
G⊢(Some xc,s) −t≻→ (undefined3 t, (Some xc, s))

— execution of statements

— cf. 14.5
| *Skip*: G⊢Norm s −Skip→ Norm s

— cf. 14.7
| *Expr*: ⟦G⊢Norm s0 −e−≻v→ s1⟧ ⟹
G⊢Norm s0 −Expr e→ s1

| *Lab*: ⟦G⊢Norm s0 −c → s1⟧ ⟹
G⊢Norm s0 −l• c→ abupd (absorb l) s1

— cf. 14.2
| *Comp*: $[\![G\vdash Norm\ s0\ -c1\ \rightarrow\ s1;$
$G\vdash\ \ s1\ -c2\ \rightarrow\ s2]\!]\Longrightarrow$
$G\vdash Norm\ s0\ -c1;;\ c2\rightarrow\ s2$

— 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*: $[\![G\vdash Norm\ s0\ -e-\succ b\rightarrow\ s1;$
$if\ the\text{-}Bool\ b$
$then\ (G\vdash s1\ -c\rightarrow\ s2\ \wedge$
$G\vdash(abupd\ (absorb\ (Cont\ l))\ s2)\ -l\cdot\ While(e)\ c\rightarrow\ s3)$
$else\ s3\ =\ s1]\!]\Longrightarrow$
$G\vdash Norm\ s0\ -l\cdot\ While(e)\ c\rightarrow\ s3$

| *Jmp*: $G\vdash Norm\ s\ -Jmp\ j\rightarrow\ (Some\ (Jump\ j),\ s)$

— cf. 14.16
| *Throw*: $[\![G\vdash Norm\ s0\ -e-\succ a'\rightarrow\ s1]\!]\Longrightarrow$
$G\vdash Norm\ s0\ -Throw\ e\rightarrow\ abupd\ (throw\ a')\ s1$

— 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*: $[\![the\ (class\ G\ C)\ =\ c;$
$if\ inited\ C\ (globs\ s0)\ then\ s3\ =\ Norm\ s0$
$else\ (G\vdash Norm\ (init\text{-}class\text{-}obj\ G\ C\ s0)$
$-(if\ C\ =\ Object\ then\ Skip\ else\ Init\ (super\ c))\rightarrow\ s1\ \wedge$
$G\vdash set\text{-}lvars\ Map.empty\ s1\ -init\ c\rightarrow\ s2\ \wedge\ s3\ =\ restore\text{-}lvars\ s1\ s2)]\!]$
$\Longrightarrow$
$G\vdash Norm\ s0\ -Init\ C\rightarrow\ s3$
— This class initialisation rule is a little bit inaccurate. Look at the exact sequence: (1) The current class object (the static fields) are initialised (*init-class-obj*), (2) the superclasses are initialised, (3) the static initialiser of the current class is invoked. More precisely we should expect another ordering, namely 2 1 3. But we can't just naively toggle 1 and 2. By calling *init-class-obj* before initialising the superclasses, we also implicitly record that we have started to initialise the current class (by setting an value for the class object). This becomes crucial for the completeness proof of the axiomatic semantics *AxCompl.thy*. Static initialisation requires an induction on the number of classes not yet initialised (or to be more precise, classes were the initialisation has not yet begun). So we could first assign a dummy value to the class before superclass initialisation and afterwards set the correct values. But as long as we don't take memory overflow into account when allocating class objects, we can leave things as they are for convenience.

— evaluation of expressions

— cf. 15.8.1, 12.4.1
| *NewC*: ⟦G⊢Norm s0 −Init C→ s1;
G⊢ s1 −halloc (CInst C)≻a→ s2⟧ ⟹
G⊢Norm s0 −NewC C−≻Addr a→ s2

— cf. 15.9.1, 12.4.1
| *NewA*: ⟦G⊢Norm s0 −init-comp-ty T→ s1; G⊢s1 −e−≻i′→ s2;
G⊢abupd (check-neg i′) s2 −halloc (Arr T (the-Intg i′))≻a→ s3⟧ ⟹
G⊢Norm s0 −New T[e]−≻Addr a→ s3

— cf. 15.15
| *Cast*: ⟦G⊢Norm s0 −e−≻v→ s1;
s2 = abupd (raise-if (¬G,store s1⊢v fits T) ClassCast) s1⟧ ⟹
G⊢Norm s0 −Cast T e−≻v→ s2

— cf. 15.19.2
| *Inst*: ⟦G⊢Norm s0 −e−≻v→ s1;
b = (v≠Null ∧ G,store s1⊢v fits RefT T)⟧ ⟹
G⊢Norm s0 −e InstOf T−≻Bool b→ s1

— cf. 15.7.1
| *Lit*: G⊢Norm s −Lit v−≻v→ Norm s

| *UnOp*: ⟦G⊢Norm s0 −e−≻v→ s1⟧
⟹ G⊢Norm s0 −UnOp unop e−≻(eval-unop unop v)→ s1

| *BinOp*: ⟦G⊢Norm s0 −e1−≻v1→ s1;
G⊢s1 −(if need-second-arg binop v1 then (In1l e2) else (In1r Skip))
≻→ (In1 v2, s2)
⟧
⟹ G⊢Norm s0 −BinOp binop e1 e2−≻(eval-binop binop v1 v2)→ s2

— cf. 15.10.2
| *Super*: G⊢Norm s −Super−≻val-this s→ Norm s

— cf. 15.2
| *Acc*: ⟦G⊢Norm s0 −va=≻(v,f)→ s1⟧ ⟹
G⊢Norm s0 −Acc va−≻v→ s1

— cf. 15.25.1
| *Ass*: ⟦G⊢Norm s0 −va=≻(w,f)→ s1;
G⊢ s1 −e−≻v → s2⟧ ⟹
G⊢Norm s0 −va:=e−≻v→ assign f v s2

— cf. 15.24
| *Cond*: ⟦G⊢Norm s0 −e0−≻b→ s1;
G⊢ s1 −(if the-Bool b then e1 else e2)−≻v→ s2⟧ ⟹
G⊢Norm s0 −e0 ? e1 : e2−≻v→ s2

— The interplay of *Call*, *Methd* and *Body*: Method invocation is split up into these three rules:

Call Calculates the target address and evaluates the arguments of the method, and then performs dynamic or static lookup of the method, corresponding to the call mode. Then the *Methd* rule is evaluated on the calculated declaration class of the method invocation.

Methd A syntactic bridge for the folded method body. It is used by the axiomatic semantics to add the proper hypothesis for recursive calls of the method.

Body An extra syntactic entity for the unfolded method body was introduced to properly trigger class initialisation. Without class initialisation we could just evaluate the body statement.

— cf. 15.11.4.1, 15.11.4.2, 15.11.4.4, 15.11.4.5
| *Call*:
$[\![G\vdash Norm\ s0\ -e-\succ a'\rightarrow\ s1;\ G\vdash s1\ -args\dot{=}\succ vs\rightarrow\ s2;$
$D = invocation\text{-}declclass\ G\ mode\ (store\ s2)\ a'\ statT\ (\!|name{=}mn,parTs{=}pTs|\!);$
$s3{=}init\text{-}lvars\ G\ D\ (\!|name{=}mn,parTs{=}pTs|\!)\ mode\ a'\ vs\ s2;$
$s3' = check\text{-}method\text{-}access\ G\ accC\ statT\ mode\ (\!|name{=}mn,parTs{=}pTs|\!)\ a'\ s3;$
$G\vdash s3'\ -Methd\ D\ (\!|name{=}mn,parTs{=}pTs|\!)-\succ v\rightarrow\ s4]\!]$
$\Longrightarrow$
$G\vdash Norm\ s0\ -\{accC,statT,mode\}e{\cdot}mn(\{pTs\}args)-\succ v\rightarrow\ (restore\text{-}lvars\ s2\ s4)$
— The accessibility check is after *init-lvars*, to keep it simple. *init-lvars* already tests for the absence of a null-pointer reference in case of an instance method invocation.

| *Methd*: $[\![G\vdash Norm\ s0\ -body\ G\ D\ sig-\succ v\rightarrow\ s1]\!] \Longrightarrow$
$G\vdash Norm\ s0\ -Methd\ D\ sig-\succ v\rightarrow\ s1$

| *Body*: $[\![G\vdash Norm\ s0\ -Init\ D\rightarrow\ s1;\ G\vdash s1\ -c\rightarrow\ s2;$
$s3 = (if\ (\exists\ l.\ abrupt\ s2 = Some\ (Jump\ (Break\ l))\ \vee$
$abrupt\ s2 = Some\ (Jump\ (Cont\ l)))$
$then\ abupd\ (\lambda\ x.\ Some\ (Error\ CrossMethodJump))\ s2$
$else\ s2)]\!] \Longrightarrow$
$G\vdash Norm\ s0\ -Body\ D\ c-\succ the\ (locals\ (store\ s2)\ Result)$
$\rightarrow abupd\ (absorb\ Ret)\ s3$
— cf. 14.15, 12.4.1
— We filter out a break/continue in *s2*, so that we can proof definite assignment correct, without the need of conformance of the state. By this the different parts of the typesafety proof can be disentangled a little.

— evaluation of variables

— cf. 15.13.1, 15.7.2
| *LVar*: $G\vdash Norm\ s\ -LVar\ vn{=}\succ lvar\ vn\ s\rightarrow\ Norm\ s$

— cf. 15.10.1, 12.4.1
| *FVar*: $[\![G\vdash Norm\ s0\ -Init\ statDeclC\rightarrow\ s1;\ G\vdash s1\ -e-\succ a\rightarrow\ s2;$
$(v,s2') = fvar\ statDeclC\ stat\ fn\ a\ s2;$
$s3 = check\text{-}field\text{-}access\ G\ accC\ statDeclC\ fn\ stat\ a\ s2'\ ]\!] \Longrightarrow$
$G\vdash Norm\ s0\ -\{accC,statDeclC,stat\}e..fn{=}\succ v\rightarrow\ s3$
— The accessibility check is after *fvar*, to keep it simple. *fvar* already tests for the absence of a null-pointer reference in case of an instance field

— cf. 15.12.1, 15.25.1
| *AVar*: $[\![G\vdash\ Norm\ s0\ -e1-\succ a\rightarrow\ s1;\ G\vdash s1\ -e2-\succ i\rightarrow\ s2;$
$(v,s2') = avar\ G\ i\ a\ s2]\!] \Longrightarrow$
$G\vdash Norm\ s0\ -e1.[e2]{=}\succ v\rightarrow\ s2'$

— evaluation of expression lists

— cf. 15.11.4.2
| *Nil*:
$G\vdash Norm\ s0\ -[]\dot{=}\succ[]\rightarrow\ Norm\ s0$

— cf. 15.6.4
| *Cons*: $[\![G\vdash Norm\ s0\ -e\ -\succ\ v\ \rightarrow\ s1;$
$G\vdash\quad s1\ -es\dot{=}\succ vs\rightarrow\ s2]\!] \Longrightarrow$
$G\vdash Norm\ s0\ -e\#es\dot{=}\succ v\#vs\rightarrow\ s2$

ML ‹
ML-Thms.bind-thm (*eval-induct*, *rearrange-prems*
[*0*,*1*,*2*,*8*,*4*,*30*,*31*,*27*,*15*,*16*,
17,*18*,*19*,*20*,*21*,*3*,*5*,*25*,*26*,*23*,*6*,
7,*11*,*9*,*13*,*14*,*12*,*22*,*10*,*28*,
29,*24*] @{*thm eval.induct*})
›

declare *if-split* [*split del*] *if-split-asm* [*split del*]
option.split [*split del*] *option.split-asm* [*split del*]
inductive-cases *halloc-elim-cases*:
$G\vdash(Some\ xc,s)\ -halloc\ oi\succ a\rightarrow\ s'$
$G\vdash(Norm\ \ s)\ -halloc\ oi\succ a\rightarrow\ s'$

inductive-cases *sxalloc-elim-cases*:
$G\vdash\ Norm\ \ s\ -sxalloc\rightarrow\ s'$
$G\vdash(Some\ (Jump\ j),s)\ -sxalloc\rightarrow\ s'$
$G\vdash(Some\ (Error\ e),s)\ -sxalloc\rightarrow\ s'$
$G\vdash(Some\ (Xcpt\ (Loc\ a\ )),s)\ -sxalloc\rightarrow\ s'$
$G\vdash(Some\ (Xcpt\ (Std\ xn)),s)\ -sxalloc\rightarrow\ s'$
inductive-cases *sxalloc-cases*: $G\vdash s\ -sxalloc\rightarrow\ s'$

lemma *sxalloc-elim-cases2*: $[\![G\vdash s\ -sxalloc\rightarrow\ s';$
$\bigwedge s.\ \ [\![s' = Norm\ s]\!] \Longrightarrow P;$
$\bigwedge j\ s.\ [\![s' = (Some\ (Jump\ j),s)]\!] \Longrightarrow P;$
$\bigwedge e\ s.\ [\![s' = (Some\ (Error\ e),s)]\!] \Longrightarrow P;$
$\bigwedge a\ s.\ [\![s' = (Some\ (Xcpt\ (Loc\ a)),s)]\!] \Longrightarrow P$
$]\!] \Longrightarrow P$
apply *cut-tac*
apply (*erule sxalloc-cases*)
apply *blast+*
done

declare *not-None-eq* [*simp del*]
declare *split-paired-All* [*simp del*] *split-paired-Ex* [*simp del*]
setup ‹*map-theory-simpset* (*fn ctxt* => *ctxt delloop split-all-tac*)›

inductive-cases *eval-cases*: $G\vdash s\ -t\succ\rightarrow\ (v,\ s')$

inductive-cases *eval-elim-cases* [*cases set*]:
$G\vdash(Some\ xc,s)\ -t\ \succ\rightarrow\ (v,\ s')$
$G\vdash Norm\ s\ -In1r\ Skip\ \succ\rightarrow\ (x,\ s')$
$G\vdash Norm\ s\ -In1r\ (Jmp\ j)\ \succ\rightarrow\ (x,\ s')$
$G\vdash Norm\ s\ -In1r\ (l\bullet\ c)\ \succ\rightarrow\ (x,\ s')$
$G\vdash Norm\ s\ -In3\ ([])\ \succ\rightarrow\ (v,\ s')$
$G\vdash Norm\ s\ -In3\ (e\#es)\ \succ\rightarrow\ (v,\ s')$
$G\vdash Norm\ s\ -In1l\ (Lit\ w)\ \succ\rightarrow\ (v,\ s')$
$G\vdash Norm\ s\ -In1l\ (UnOp\ unop\ e)\ \succ\rightarrow\ (v,\ s')$
$G\vdash Norm\ s\ -In1l\ (BinOp\ binop\ e1\ e2)\ \succ\rightarrow\ (v,\ s')$
$G\vdash Norm\ s\ -In2\ (LVar\ vn)\ \succ\rightarrow\ (v,\ s')$
$G\vdash Norm\ s\ -In1l\ (Cast\ T\ e)\ \succ\rightarrow\ (v,\ s')$
$G\vdash Norm\ s\ -In1l\ (e\ InstOf\ T)\ \succ\rightarrow\ (v,\ s')$
$G\vdash Norm\ s\ -In1l\ (Super)\ \succ\rightarrow\ (v,\ s')$
$G\vdash Norm\ s\ -In1l\ (Acc\ va)\ \succ\rightarrow\ (v,\ s')$

$G \vdash Norm\ s\ -In1r\ (Expr\ e) \succ\rightarrow (x,\ s')$
$G \vdash Norm\ s\ -In1r\ (c1;;\ c2) \succ\rightarrow (x,\ s')$
$G \vdash Norm\ s\ -In1l\ (Methd\ C\ sig) \succ\rightarrow (x,\ s')$
$G \vdash Norm\ s\ -In1l\ (Body\ D\ c) \succ\rightarrow (x,\ s')$
$G \vdash Norm\ s\ -In1l\ (e0\ ?\ e1\ :\ e2) \succ\rightarrow (v,\ s')$
$G \vdash Norm\ s\ -In1r\ (If(e)\ c1\ Else\ c2) \succ\rightarrow (x,\ s')$
$G \vdash Norm\ s\ -In1r\ (l\bullet\ While(e)\ c) \succ\rightarrow (x,\ s')$
$G \vdash Norm\ s\ -In1r\ (c1\ Finally\ c2) \succ\rightarrow (x,\ s')$
$G \vdash Norm\ s\ -In1r\ (Throw\ e) \succ\rightarrow (x,\ s')$
$G \vdash Norm\ s\ -In1l\ (NewC\ C) \succ\rightarrow (v,\ s')$
$G \vdash Norm\ s\ -In1l\ (New\ T[e]) \succ\rightarrow (v,\ s')$
$G \vdash Norm\ s\ -In1l\ (Ass\ va\ e) \succ\rightarrow (v,\ s')$
$G \vdash Norm\ s\ -In1r\ (Try\ c1\ Catch(tn\ vn)\ c2) \succ\rightarrow (x,\ s')$
$G \vdash Norm\ s\ -In2\ (\{accC,statDeclC,stat\}e..fn) \succ\rightarrow (v,\ s')$
$G \vdash Norm\ s\ -In2\ (e1.[e2]) \succ\rightarrow (v,\ s')$
$G \vdash Norm\ s\ -In1l\ (\{accC,statT,mode\}e\cdot mn(\{pT\}p)) \succ\rightarrow (v,\ s')$
$G \vdash Norm\ s\ -In1r\ (Init\ C) \succ\rightarrow (x,\ s')$

declare *not-None-eq* [*simp*]
declare *split-paired-All* [*simp*] *split-paired-Ex* [*simp*]
declaration ‹*K* (*Simplifier.map-ss* (*fn ss* => *ss addloop* (*split-all-tac*, *split-all-tac*)))›
declare *if-split* [*split*] *if-split-asm* [*split*]
option.split [*split*] *option.split-asm* [*split*]

lemma *eval-Inj-elim*:
$G \vdash s\ -t \succ\rightarrow (w,s')$
$\Longrightarrow case\ t\ of$
$In1\ ec \Rightarrow (case\ ec\ of$
$Inl\ e \Rightarrow (\exists\, v.\ w = In1\ v)$
$|\ Inr\ c \Rightarrow w = \diamondsuit)$
$|\ In2\ e \Rightarrow (\exists\, v.\ w = In2\ v)$
$|\ In3\ e \Rightarrow (\exists\, v.\ w = In3\ v)$
apply (*erule eval-cases*)
apply *auto*
apply (*induct-tac t*)
apply (*rename-tac a*, *induct-tac a*)
apply *auto*
done

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*: $G \vdash s\ -In1l\ t \succ\rightarrow (w,\ s') = (\exists\, v.\ w{=}In1\ v \wedge G \vdash s\ -t{-}\succ v \rightarrow s')$
by (*auto*, *frule eval-Inj-elim*, *auto*)

lemma *eval-var-eq*: $G \vdash s\ -In2\ t \succ\rightarrow (w,\ s') = (\exists\, vf.\ w{=}In2\ vf \wedge G \vdash s\ -t{=}\succ vf \rightarrow s')$
by (*auto*, *frule eval-Inj-elim*, *auto*)

lemma *eval-exprs-eq*: $G \vdash s\ -In3\ t \succ\rightarrow (w,\ s') = (\exists\, vs.\ w{=}In3\ vs \wedge G \vdash s\ -t \doteq\succ vs \rightarrow s')$
by (*auto*, *frule eval-Inj-elim*, *auto*)

lemma *eval-stmt-eq*: $G \vdash s\ -In1r\ t \succ\rightarrow (w,\ s') = (w{=}\diamondsuit \wedge G \vdash s\ -t \rightarrow s')$
by (*auto*, *frule eval-Inj-elim*, *auto*, *frule eval-Inj-elim*, *auto*)

simproc-setup *eval-expr* $(G \vdash s\ -In1l\ t \succ\rightarrow (w,\ s'))$ = ‹

```
  K (K (fn ct =>
    (case Thm.term-of ct of
      (- $ - $ - $ - $ (Const - $ -) $ -) => NONE
    | - => SOME (mk-meta-eq @{thm eval-expr-eq}))))›

simproc-setup eval-var (G⊢s −In2 t≻→ (w, s′)) = ‹
  K (K (fn ct =>
    (case Thm.term-of ct of
      (- $ - $ - $ - $ (Const - $ -) $ -) => NONE
    | - => SOME (mk-meta-eq @{thm eval-var-eq}))))›

simproc-setup eval-exprs (G⊢s −In3 t≻→ (w, s′)) = ‹
  K (K (fn ct =>
    (case Thm.term-of ct of
      (- $ - $ - $ - $ (Const - $ -) $ -) => NONE
    | - => SOME (mk-meta-eq @{thm eval-exprs-eq}))))›

simproc-setup eval-stmt (G⊢s −In1r t≻→ (w, s′)) = ‹
  K (K (fn ct =>
    (case Thm.term-of ct of
      (- $ - $ - $ - $ (Const - $ -) $ -) => NONE
    | - => SOME (mk-meta-eq @{thm eval-stmt-eq}))))›

ML ‹
ML-Thms.bind-thms (AbruptIs, sum3-instantiate context @{thm eval.Abrupt})
›

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⊢Norm s−Callee l e−≻v→ s′ = False
proof −
  { fix s t v s′
    assume eval: G⊢s −t≻→ (v,s′) and
          normal: normal s and
          callee: t=In1l (Callee l e)
    then have False by induct auto
  }
  then show ?thesis
    by (cases s′) fastforce
qed


lemma eval-InsInitE: G⊢Norm s−InsInitE c e−≻v→ s′ = False
proof −
  { fix s t v s′
    assume eval: G⊢s −t≻→ (v,s′) and
          normal: normal s and
          callee: t=In1l (InsInitE c e)
    then have False by induct auto
  }
  then show ?thesis
    by (cases s′) fastforce
qed


lemma eval-InsInitV: G⊢Norm s−InsInitV c w=≻v→ s′ = False
```

proof –
  { **fix** *s t v s'*
    assume *eval*: *G⊢s −t≻→ (v,s')* **and**
      normal: *normal s* **and**
      callee: *t=In2 (InsInitV c w)*
    then have *False* **by** *induct auto*
  }
  then show *?thesis*
    by (*cases s'*) *fastforce*
qed

lemma *eval-FinA*: *G⊢Norm s−FinA a c→ s' = False*
proof –
  { **fix** *s t v s'*
    assume *eval*: *G⊢s −t≻→ (v,s')* **and**
      normal: *normal s* **and**
      callee: *t=In1r (FinA a c)*
    then have *False* **by** *induct auto*
  }
  then show *?thesis*
    by (*cases s'*) *fastforce*
qed

lemma *eval-no-abrupt-lemma*:
  ⋀s s'. G⊢s −t≻→ (w,s') ⟹ normal s' ⟶ normal s
by (*erule eval-cases, auto*)

lemma *eval-no-abrupt*:
  G⊢(x,s) −t≻→ (w,Norm s') =
    (x = None ∧ G⊢Norm s −t≻→ (w,Norm s'))
apply *auto*
apply (*frule eval-no-abrupt-lemma, auto*)+
done

simproc-setup *eval-no-abrupt (G⊢(x,s) −e≻→ (w,Norm s')) =* ‹
  K (K (fn ct =>
    (case Thm.term-of ct of
      *(- $ - $ (Const (**const-name** ‹Pair›, -) $ (Const (**const-name** ‹None›, -)) $ -) $ - $ - $ -) => NONE*
    | - => SOME (mk-meta-eq @{thm eval-no-abrupt}))))
›

lemma *eval-abrupt-lemma*:
  G⊢s −t≻→ (v,s') ⟹ abrupt s=Some xc ⟶ s'= s ∧ v = undefined3 t
by (*erule eval-cases, auto*)

lemma *eval-abrupt*:
  G⊢(Some xc,s) −t≻→ (w,s') =
    (s'=(Some xc,s) ∧ w=undefined3 t ∧
    G⊢(Some xc,s) −t≻→ (undefined3 t,(Some xc,s)))
apply *auto*
apply (*frule eval-abrupt-lemma, auto*)+
done

simproc-setup *eval-abrupt* (*G*⊢(*Some xc*,*s*) −*e*≻→ (*w*,*s*′)) = ‹
K (*K* (*fn ct* =>
(*case Thm.term-of ct of*
(- $ - $ - $ - $ - $ (*Const* (***const-name*** ‹*Pair*›, -) $ (*Const* (***const-name*** ‹*Some*›, -) $ -)$ -)) =>
NONE
| - => *SOME* (*mk-meta-eq* @{*thm eval-abrupt*}))))
›

lemma *LitI*: *G*⊢*s* −*Lit v*−≻(*if normal s then v else undefined*)→ *s*
apply (*case-tac s*, *case-tac a* = *None*)
by (*auto intro*!: *eval.Lit*)

lemma *SkipI* [*intro*!]: *G*⊢*s* −*Skip*→ *s*
apply (*case-tac s*, *case-tac a* = *None*)
by (*auto intro*!: *eval.Skip*)

lemma *ExprI*: *G*⊢*s* −*e*−≻*v*→ *s*′ ⟹ *G*⊢*s* −*Expr e*→ *s*′
apply (*case-tac s*, *case-tac a* = *None*)
by (*auto intro*!: *eval.Expr*)

lemma *CompI*: ⟦*G*⊢*s* −*c1*→ *s1*; *G*⊢*s1* −*c2*→ *s2*⟧ ⟹ *G*⊢*s* −*c1*;; *c2*→ *s2*
apply (*case-tac s*, *case-tac a* = *None*)
by (*auto intro*!: *eval.Comp*)

lemma *CondI*:
⋀*s1*. ⟦*G*⊢*s* −*e*−≻*b*→ *s1*; *G*⊢*s1* −(*if the-Bool b then e1 else e2*)−≻*v*→ *s2*⟧ ⟹
G⊢*s* −*e* ? *e1* : *e2*−≻(*if normal s1 then v else undefined*)→ *s2*
apply (*case-tac s*, *case-tac a* = *None*)
by (*auto intro*!: *eval.Cond*)

lemma *IfI*: ⟦*G*⊢*s* −*e*−≻*v*→ *s1*; *G*⊢*s1* −(*if the-Bool v then c1 else c2*)→ *s2*⟧
⟹ *G*⊢*s* −*If*(*e*) *c1 Else c2*→ *s2*
apply (*case-tac s*, *case-tac a* = *None*)
by (*auto intro*!: *eval.If*)

lemma *MethdI*: *G*⊢*s* −*body G C sig*−≻*v*→ *s*′
⟹ *G*⊢*s* −*Methd C sig*−≻*v*→ *s*′
apply (*case-tac s*, *case-tac a* = *None*)
by (*auto intro*!: *eval.Methd*)

lemma *eval-Call*:
⟦*G*⊢*Norm s0* −*e*−≻*a*′→ *s1*; *G*⊢*s1* −*ps*≐≻*pvs*→ *s2*;
D = *invocation-declclass G mode* (*store s2*) *a*′ *statT* (|*name*=*mn*,*parTs*=*pTs*|);
s3 = *init-lvars G D* (|*name*=*mn*,*parTs*=*pTs*|) *mode a*′ *pvs s2*;
s3′ = *check-method-access G accC statT mode* (|*name*=*mn*,*parTs*=*pTs*|) *a*′ *s3*;
G⊢*s3*′−*Methd D* (|*name*=*mn*,*parTs*=*pTs*|)−≻ *v*→ *s4*;
s4′ = *restore-lvars s2 s4*⟧ ⟹
G⊢*Norm s0* −{*accC*,*statT*,*mode*}*e*·*mn*({*pTs*}*ps*)−≻*v*→ *s4*′
apply (*drule eval.Call*, *assumption*)
apply (*rule HOL.refl*)
apply *simp*+

done

lemma *eval-Init*:
⟦*if inited C* (*globs s0*) *then s3* = *Norm s0*
else G⊢*Norm* (*init-class-obj G C s0*)
−(*if C* = *Object then Skip else Init* (*super* (*the* (*class G C*))))→ *s1* ∧
G⊢*set-lvars Map.empty s1* −(*init* (*the* (*class G C*)))→ *s2* ∧
s3 = *restore-lvars s1 s2*⟧ ⟹
G⊢*Norm s0* −*Init C*→ *s3*
apply (*rule eval.Init*)
apply *auto*
done

lemma *init-done*: *initd C s* ⟹ *G*⊢*s* −*Init C*→ *s*
apply (*case-tac s*, *simp*)
apply (*case-tac a*)
apply *safe*
apply (*rule eval-Init*)
apply *auto*
done

lemma *eval-StatRef*:
G⊢*s* −*StatRef rt*−≻(*if abrupt s*=*None then Null else undefined*)→ *s*
apply (*case-tac s*, *simp*)
apply (*case-tac a* = *None*)
apply (*auto del*: *eval.Abrupt intro*!: *eval.intros*)
done

lemma *SkipD* [*dest*!]: *G*⊢*s* −*Skip*→ *s'* ⟹ *s'* = *s*
apply (*erule eval-cases*)
by *auto*

lemma *Skip-eq* [*simp*]: *G*⊢*s* −*Skip*→ *s'* = (*s* = *s'*)
by *auto*

lemma *init-retains-locals* [*rule-format* (*no-asm*)]: *G*⊢*s* −*t*≻→ (*w*,*s'*) ⟹
(∀ *C*. *t*=*In1r* (*Init C*) ⟶ *locals* (*store s*) = *locals* (*store s'*))
apply (*erule eval.induct*)
apply (*simp* (*no-asm-use*) *split del*: *if-split-asm option.split-asm*)+
apply *auto*
done

lemma *halloc-xcpt* [*dest*!]:
⋀*s'*. *G*⊢(*Some xc*,*s*) −*halloc oi*≻*a*→ *s'* ⟹ *s'*=(*Some xc*,*s*)
apply (*erule-tac halloc-elim-cases*)
by *auto*

lemma *eval-Methd*:
  $G \vdash s\ -In1l(body\ G\ C\ sig) \succ\rightarrow (w,s')$
  $\Longrightarrow G \vdash s\ -In1l(Methd\ C\ sig) \succ\rightarrow (w,s')$
apply (*case-tac s*)
apply (*case-tac a*)
apply *clarsimp+*
apply (*erule eval.Methd*)
apply (*drule eval-abrupt-lemma*)
apply *force*
done

lemma *eval-Body*: $[\![G \vdash Norm\ s0\ -Init\ D \rightarrow s1;\ G \vdash s1\ -c \rightarrow s2;$
  $res = the\ (locals\ (store\ s2)\ Result);$
  $s3 = (if\ (\exists\ l.\ abrupt\ s2 = Some\ (Jump\ (Break\ l)) \lor$
  $abrupt\ s2 = Some\ (Jump\ (Cont\ l)))$
  $then\ abupd\ (\lambda\ x.\ Some\ (Error\ CrossMethodJump))\ s2$
  $else\ s2);$
  $s4 = abupd\ (absorb\ Ret)\ s3]\!] \Longrightarrow$
 $G \vdash Norm\ s0\ -Body\ D\ c - \succ res \rightarrow s4$
by (*auto elim*: *eval.Body*)

lemma *eval-binop-arg2-indep*:
$\neg\ need\text{-}second\text{-}arg\ binop\ v1 \Longrightarrow eval\text{-}binop\ binop\ v1\ x = eval\text{-}binop\ binop\ v1\ y$
by (*cases binop*)
  (*simp-all add*: *need-second-arg-def*)

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 −
  from *eval-e1*
  have *?EvalBinOp Unit*
    by (*rule eval.BinOp*)
      (*simp add*: *no-need*)
  moreover
  from *no-need*
  have *eval-binop binop v1 Unit = eval-binop binop v1 v2*
    by (*simp add*: *eval-binop-arg2-indep*)
  ultimately
  show *?thesis*
    by *simp*
qed

single valued

lemma *unique-halloc* [*rule-format* (*no-asm*)]:
  $G \vdash s\ -halloc\ oi \succ a \rightarrow s' \Longrightarrow G \vdash s\ -halloc\ oi \succ a' \rightarrow s'' \longrightarrow a' = a \land s'' = s'$
apply (*erule halloc.induct*)
apply (*auto elim!*: *halloc-elim-cases split del*: *if-split if-split-asm*)
apply (*drule trans* [*THEN sym*], *erule sym*)
defer
apply (*drule trans* [*THEN sym*], *erule sym*)
apply *auto*

done

lemma *single-valued-halloc*:
single-valued $\{((s,oi),(a,s')).\ G\vdash s\ {-halloc}\ oi\succ a \to s'\}$
apply (*unfold single-valued-def*)
by (*clarsimp*, *drule* (*1*) *unique-halloc*, *auto*)

lemma *unique-sxalloc* [*rule-format* (*no-asm*)]:
$G\vdash s\ {-sxalloc}\!\to s' \Longrightarrow G\vdash s\ {-sxalloc}\!\to s'' \longrightarrow s'' = s'$
apply (*erule sxalloc.induct*)
apply (*auto dest*: *unique-halloc elim*!: *sxalloc-elim-cases*
split del: *if-split if-split-asm*)
done

lemma *single-valued-sxalloc*: *single-valued* $\{(s,s').\ G\vdash s\ {-sxalloc}\!\to s'\}$
apply (*unfold single-valued-def*)
apply (*blast dest*: *unique-sxalloc*)
done

lemma *split-pairD*: $(x,y) = p \Longrightarrow x = fst\ p\ \&\ y = snd\ p$
by *auto*

lemma *unique-eval* [*rule-format* (*no-asm*)]:
$G\vdash s\ {-t}\succ\!\to (w,\ s') \Longrightarrow (\forall\, w'\ s''.\ G\vdash s\ {-t}\succ\!\to (w',\ s'') \longrightarrow w' = w \wedge s'' = s')$
apply (*erule eval-induct*)
apply (*tactic* ‹*ALLGOALS* (*EVERY'*
[*strip-tac* ***context***, *rotate-tac* ~*1*, *eresolve-tac* ***context*** @{*thms eval-elim-cases*}])›)

prefer *28*
apply (*simp* (*no-asm-use*) *only*: *split*: *if-split-asm*)

prefer *30*
apply (*case-tac inited C* (*globs s0*), (*simp only*: *if-True if-False simp-thms*)+)
prefer *26*
apply (*simp* (*no-asm-use*) *only*: *split*: *if-split-asm*, *blast*)

apply (*blast dest*: *unique-sxalloc unique-halloc split-pairD*)+
done

lemma *single-valued-eval*:
single-valued $\{((s,\ t),\ (v,\ s')).\ G\vdash s\ {-t}\succ\!\to (v,\ s')\}$
apply (*unfold single-valued-def*)
by (*clarify*, *drule* (*1*) *unique-eval*, *auto*)

end

Chapter 16

Example

1 Example Bali program

theory *Example*
imports *Eval WellForm*
begin

The following example Bali program includes:

- class and interface declarations with inheritance, hiding of fields, overriding of methods (with refined result type), array type,
- method call (with dynamic binding), parameter access, return expressions,
- expression statements, sequential composition, literal values, local assignment, local access, field assignment, type cast,
- exception generation and propagation, try and catch statement, throw statement
- instance creation and (default) static initialization

```
package java_lang

public interface HasFoo {
  public Base foo(Base z);
}

public class Base implements HasFoo {
  static boolean arr[] = new boolean[2];
  public HasFoo vee;
  public Base foo(Base z) {
    return z;
  }
}

public class Ext extends Base {
  public int vee;
  public Ext foo(Base z) {
    ((Ext)z).vee = 1;
    return null;
  }
}
```

```
public class Main {
  public static void main(String args[]) throws Throwable {
    Base e = new Ext();
    try {e.foo(null); }
    catch(NullPointerException z) {
      while(Ext.arr[2]) ;
    }
  }
}
```

declare *widen.null* [*intro*]

lemma *wf-fdecl-def2*: $\bigwedge fd.$ *wf-fdecl G P fd* = *is-acc-type G P* (*type* (*snd fd*))
apply (*unfold wf-fdecl-def*)
apply (*simp* (*no-asm*))
done

declare *wf-fdecl-def2* [*iff*]

type and expression names

datatype *tnam'* = *HasFoo'* | *Base'* | *Ext'* | *Main'*
datatype *vnam'* = *arr'* | *vee'* | *z'* | *e'*
datatype *label'* = *lab1'*

axiomatization
tnam' :: *tnam'* ⇒ *tnam* **and**
vnam' :: *vnam'* ⇒ *vname* **and**
label':: *label'* ⇒ *label*
where

inj-tnam' [*simp*]: $\bigwedge x\ y.$ (*tnam'* x = *tnam'* y) = ($x = y$) **and**
inj-vnam' [*simp*]: $\bigwedge x\ y.$ (*vnam'* x = *vnam'* y) = ($x = y$) **and**
inj-label' [*simp*]: $\bigwedge x\ y.$ (*label'* x = *label'* y) = ($x = y$) **and**

surj-tnam': $\bigwedge n.\ \exists m.\ n =$ *tnam'* m **and**
surj-vnam': $\bigwedge n.\ \exists m.\ n =$ *vnam'* m **and**
surj-label': $\bigwedge n.\ \exists m.\ n =$ *label'* m

abbreviation
HasFoo :: *qtname* **where**
HasFoo == (|*pid*=*java-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*
by (*simp add*: *Object-def*)

lemma *neq-Ext-Object* [*simp*]: *Ext*≠*Object*
by (*simp add*: *Object-def*)

lemma *neq-Main-Object* [*simp*]: *Main*≠*Object*
by (*simp add*: *Object-def*)

lemma *neq-Base-SXcpt* [*simp*]: *Base*≠*SXcpt xn*
by (*simp add*: *SXcpt-def*)

lemma *neq-Ext-SXcpt* [*simp*]: *Ext*≠*SXcpt xn*
by (*simp add*: *SXcpt-def*)

lemma *neq-Main-SXcpt* [*simp*]: *Main*≠*SXcpt xn*
by (*simp add*: *SXcpt-def*)

classes and interfaces

overloading
Object-mdecls ≡ *Object-mdecls*
SXcpt-mdecls ≡ *SXcpt-mdecls*
begin
definition *Object-mdecls* ≡ []
definition *SXcpt-mdecls* ≡ []
end

axiomatization
foo :: *mname*

definition
foo-sig :: *sig*

```
  where foo-sig = (|name=foo,parTs=[Class Base]|)

definition
  foo-mhead :: mhead
  where foo-mhead = (|access=Public,static=False,pars=[z],resT=Class Base|)

definition
  Base-foo :: mdecl
  where Base-foo = (foo-sig, (|access=Public,static=False,pars=[z],resT=Class Base,
                    mbody=(|lcls=[],stmt=Return (!!z)|)|))

definition Ext-foo :: mdecl
  where Ext-foo = (foo-sig,
          (|access=Public,static=False,pars=[z],resT=Class Ext,
           mbody=(|lcls=[]
                ,stmt=Expr({Ext,Ext,False}Cast (Class Ext) (!!z)..vee :=
                                             Lit (Intg 1)) ;;
                        Return (Lit Null)|)
          |))

definition
  arr-viewed-from :: qtname ⇒ qtname ⇒ var
  where arr-viewed-from accC C = {accC,Base,True}StatRef (ClassT C)..arr

definition
  BaseCl :: class where
  BaseCl = (|access=Public,
        cfields=[(arr, (|access=Public,static=True ,type=PrimT Boolean.[]|)),
               (vee, (|access=Public,static=False,type=Iface HasFoo    |))],
        methods=[Base-foo],
        init=Expr(arr-viewed-from Base Base
               :=New (PrimT Boolean)[Lit (Intg 2)]),
        super=Object,
        superIfs=[HasFoo]|)

definition
  ExtCl :: class where
  ExtCl = (|access=Public,
        cfields=[(vee, (|access=Public,static=False,type= PrimT Integer|))],
        methods=[Ext-foo],
        init=Skip,
        super=Base,
        superIfs=[]|)

definition
  MainCl :: class where
  MainCl = (|access=Public,
        cfields=[],
        methods=[],
        init=Skip,
        super=Object,
        superIfs=[]|)


definition
  HasFooInt :: iface
  where HasFooInt = (|access=Public,imethods=[(foo-sig, foo-mhead)],isuperIfs=[]|)

definition
```

Ifaces ::*idecl list*
where *Ifaces* = [(*HasFoo*,*HasFooInt*)]

definition
Classes ::*cdecl list*
where *Classes* = [(*Base*,*BaseCl*),(*Ext*,*ExtCl*),(*Main*,*MainCl*)]@*standard-classes*

lemmas *table-classes-defs* =
Classes-def standard-classes-def ObjectC-def SXcptC-def

lemma *table-ifaces* [*simp*]: *table-of Ifaces* = *Map.empty*(*HasFoo*↦*HasFooInt*)
apply (*unfold Ifaces-def*)
apply (*simp* (*no-asm*))
done

lemma *table-classes-Object* [*simp*]:
table-of Classes Object = *Some* (|*access*=*Public*,*cfields*=[]
,*methods*=*Object-mdecls*
,*init*=*Skip*,*super*=*undefined*,*superIfs*=[]|)
apply (*unfold table-classes-defs*)
apply (*simp* (*no-asm*) *add*:*Object-def*)
done

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=[]|)
apply (*unfold table-classes-defs*)
apply (*induct-tac xn*)
apply (*simp add*: *Object-def SXcpt-def*)+
done

lemma *table-classes-HasFoo* [*simp*]: *table-of Classes HasFoo* = *None*
apply (*unfold table-classes-defs*)
apply (*simp* (*no-asm*) *add*: *Object-def SXcpt-def*)
done

lemma *table-classes-Base* [*simp*]: *table-of Classes Base* = *Some BaseCl*
apply (*unfold table-classes-defs*)
apply (*simp* (*no-asm*) *add*: *Object-def SXcpt-def*)
done

lemma *table-classes-Ext* [*simp*]: *table-of Classes Ext* = *Some ExtCl*
apply (*unfold table-classes-defs*)
apply (*simp* (*no-asm*) *add*: *Object-def SXcpt-def*)
done

lemma *table-classes-Main* [*simp*]: *table-of Classes Main* = *Some MainCl*
apply (*unfold table-classes-defs*)
apply (*simp* (*no-asm*) *add*: *Object-def SXcpt-def*)

done

program

abbreviation
tprg :: *prog* **where**
tprg == (|*ifaces*=*Ifaces*,*classes*=*Classes*|)

definition
test :: (*ty*)*list* ⇒ *stmt* **where**
test pTs = (*e*:==*NewC Ext*;;
Try Expr({*Main*,*ClassT Base*,*IntVir*}!!*e*·*foo*({*pTs*}[*Lit Null*]))
Catch((*SXcpt NullPointer*) *z*)
(*lab1*· *While*(*Acc*
(*Acc* (*arr-viewed-from Main Ext*).[*Lit* (*Intg 2*)])) *Skip*))

well-structuredness

lemma *not-Object-subcls-any* [*elim*!]: (*Object*, *C*) ∈ (*subcls1 tprg*)$^+$ ⟹ *R*
apply (*auto dest*!: *tranclD subcls1D*)
done

lemma *not-Throwable-subcls-SXcpt* [*elim*!]:
(*SXcpt Throwable*, *SXcpt xn*) ∈ (*subcls1 tprg*)$^+$ ⟹ *R*
apply (*auto dest*!: *tranclD subcls1D*)
apply (*simp add*: *Object-def SXcpt-def*)
done

lemma *not-SXcpt-n-subcls-SXcpt-n* [*elim*!]:
(*SXcpt xn*, *SXcpt xn*) ∈ (*subcls1 tprg*)$^+$ ⟹ *R*
apply (*auto dest*!: *tranclD subcls1D*)
apply (*drule rtranclD*)
apply *auto*
done

lemma *not-Base-subcls-Ext* [*elim*!]: (*Base*, *Ext*) ∈ (*subcls1 tprg*)$^+$ ⟹ *R*
apply (*auto dest*!: *tranclD subcls1D simp add*: *BaseCl-def*)
done

lemma *not-TName-n-subcls-TName-n* [*rule-format* (*no-asm*), *elim*!]:
((|*pid*=*java-lang*,*tid*=*TName tn*|), (|*pid*=*java-lang*,*tid*=*TName tn*|))
∈ (*subcls1 tprg*)$^+$ ⟶ *R*
apply (*rule-tac n1* = *tn* **in** *surj-tnam'* [*THEN exE*])
apply (*erule ssubst*)
apply (*rule tnam'.induct*)
apply *safe*
apply (*auto dest*!: *tranclD subcls1D simp add*: *BaseCl-def ExtCl-def MainCl-def*)
apply (*drule rtranclD*)
apply *auto*
done

lemma *ws-idecl-HasFoo*: *ws-idecl tprg HasFoo* []
apply (*unfold ws-idecl-def*)

apply (*simp* (*no-asm*))
done

lemma *ws-cdecl-Object*: *ws-cdecl tprg Object any*
apply (*unfold ws-cdecl-def*)
apply *auto*
done

lemma *ws-cdecl-Throwable*: *ws-cdecl tprg* (*SXcpt Throwable*) *Object*
apply (*unfold ws-cdecl-def*)
apply *auto*
done

lemma *ws-cdecl-SXcpt*: *ws-cdecl tprg* (*SXcpt xn*) (*SXcpt Throwable*)
apply (*unfold ws-cdecl-def*)
apply *auto*
done

lemma *ws-cdecl-Base*: *ws-cdecl tprg Base Object*
apply (*unfold ws-cdecl-def*)
apply *auto*
done

lemma *ws-cdecl-Ext*: *ws-cdecl tprg Ext Base*
apply (*unfold ws-cdecl-def*)
apply *auto*
done

lemma *ws-cdecl-Main*: *ws-cdecl tprg Main Object*
apply (*unfold ws-cdecl-def*)
apply *auto*
done

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* $\Longrightarrow$ ($\forall\,(I,i)\in$*set Ifaces*. *ws-idecl G I* (*isuperIfs i*))
apply (*simp* (*no-asm*) *add*: *Ifaces-def HasFooInt-def*)
apply (*auto intro*!: *ws-idecl-HasFoo*)
done

lemma *ws-cdecl-all*: G=*tprg* $\Longrightarrow$ ($\forall\,(C,c)\in$*set Classes*. *ws-cdecl G C* (*super c*))
apply (*simp* (*no-asm*) *add*: *Classes-def BaseCl-def ExtCl-def MainCl-def*)
apply (*auto intro*!: *ws-cdecls simp add*: *standard-classes-def ObjectC-def*
SXcptC-def)

done

lemma *ws-tprg*: *ws-prog tprg*
apply (*unfold ws-prog-def*)
apply (*auto intro*!: *ws-idecl-all ws-cdecl-all*)
done

misc program properties (independent of well-structuredness)

lemma *single-iface* [*simp*]: *is-iface tprg I* = (*I* = *HasFoo*)
apply (*unfold Ifaces-def*)
apply (*simp* (*no-asm*))
done

lemma *empty-subint1* [*simp*]: *subint1 tprg* = {}
apply (*unfold subint1-def Ifaces-def HasFooInt-def*)
apply *auto*
done

lemma *unique-ifaces*: *unique Ifaces*
apply (*unfold Ifaces-def*)
apply (*simp* (*no-asm*))
done

lemma *unique-classes*: *unique Classes*
apply (*unfold table-classes-defs*)
apply (*simp*)
done

lemma *SXcpt-subcls-Throwable* [*simp*]: $tprg \vdash SXcpt\ xn \preceq_C SXcpt\ Throwable$
apply (*rule SXcpt-subcls-Throwable-lemma*)
apply *auto*
done

lemma *Ext-subclseq-Base* [*simp*]: $tprg \vdash Ext \preceq_C Base$
apply (*rule subcls-direct1*)
apply (*simp* (*no-asm*) *add*: *ExtCl-def*)
apply (*simp add*: *Object-def*)
apply (*simp* (*no-asm*))
done

lemma *Ext-subcls-Base* [*simp*]: $tprg \vdash Ext \prec_C Base$
apply (*rule subcls-direct2*)
apply (*simp* (*no-asm*) *add*: *ExtCl-def*)
apply (*simp add*: *Object-def*)
apply (*simp* (*no-asm*))
done

fields and method lookup

lemma *fields-tprg-Object* [*simp*]: *DeclConcepts.fields tprg Object* = []
by (*rule ws-tprg* [*THEN fields-emptyI*], *force*+)

lemma *fields-tprg-Throwable* [*simp*]:
DeclConcepts.fields tprg (*SXcpt Throwable*) = []
by (*rule ws-tprg* [*THEN fields-emptyI*], *force*+)

lemma *fields-tprg-SXcpt* [*simp*]: *DeclConcepts.fields tprg* (*SXcpt xn*) = []
apply (*case-tac xn* = *Throwable*)
apply (*simp* (*no-asm-simp*))
by (*rule ws-tprg* [*THEN fields-emptyI*], *force*+)

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* |))]
apply (*subst fields-rec′*)
apply (*auto simp add*: *BaseCl-def*)
done

lemma *fields-Ext* [*simp*]:
DeclConcepts.fields tprg Ext
= [((*vee*,*Ext*), (|*access*=*Public*,*static*=*False*,*type*= *PrimT Integer*|))]
@ *DeclConcepts.fields tprg Base*
apply (*rule trans*)
apply (*rule fields-rec′*)
apply (*auto simp add*: *ExtCl-def Object-def*)
done

lemmas *imethds-rec′* = *imethds-rec* [*OF - ws-tprg*]
lemmas *methd-rec′* = *methd-rec* [*OF - ws-tprg*]

lemma *imethds-HasFoo* [*simp*]:
imethds tprg HasFoo = *set-option* ∘ *Map.empty*(*foo-sig*↦(*HasFoo*, *foo-mhead*))
apply (*rule trans*)
apply (*rule imethds-rec′*)
apply (*auto simp add*: *HasFooInt-def*)
done

lemma *methd-tprg-Object* [*simp*]: *methd tprg Object* = *Map.empty*
apply (*subst methd-rec′*)
apply (*auto simp add*: *Object-mdecls-def*)
done

lemma *methd-Base* [*simp*]:
methd tprg Base = *table-of* [(λ(*s*,*m*). (*s*, *Base*, *m*)) *Base-foo*]
apply (*rule trans*)
apply (*rule methd-rec′*)
apply (*auto simp add*: *BaseCl-def*)
done

lemma *memberid-Base-foo-simp* [*simp*]:
memberid (*mdecl Base-foo*) = *mid foo-sig*
by (*simp add*: *Base-foo-def*)

lemma *memberid-Ext-foo-simp* [*simp*]:
memberid (*mdecl Ext-foo*) = *mid foo-sig*
by (*simp add*: *Ext-foo-def*)

lemma *Base-declares-foo*:
tprg⊢*mdecl Base-foo declared-in Base*
by (*auto simp add*: *declared-in-def cdeclaredmethd-def BaseCl-def Base-foo-def*)

lemma *foo-sig-not-undeclared-in-Base*:
¬ *tprg*⊢*mid foo-sig undeclared-in Base*
proof −
from *Base-declares-foo*
show *?thesis*
by (*auto dest*!: *declared-not-undeclared*)
qed

lemma *Ext-declares-foo*:
tprg⊢*mdecl Ext-foo declared-in Ext*
by (*auto simp add*: *declared-in-def cdeclaredmethd-def ExtCl-def Ext-foo-def*)

lemma *foo-sig-not-undeclared-in-Ext*:
¬ *tprg*⊢*mid foo-sig undeclared-in Ext*
proof −
from *Ext-declares-foo*
show *?thesis*
by (*auto dest*!: *declared-not-undeclared*)
qed

lemma *Base-foo-not-inherited-in-Ext*:
¬ *tprg* ⊢ *Ext inherits* (*Base*,*mdecl Base-foo*)
by (*auto simp add*: *inherits-def foo-sig-not-undeclared-in-Ext*)

lemma *Ext-method-inheritance*:
filter-tab (λ*sig m*. *tprg* ⊢ *Ext inherits method sig m*)
(*Map.empty*(*fst* ((λ(*s*, *m*). (*s*, *Base*, *m*)) *Base-foo*)↦
snd ((λ(*s*, *m*). (*s*, *Base*, *m*)) *Base-foo*)))
= *Map.empty*
proof −
from *Base-foo-not-inherited-in-Ext*
show *?thesis*
by (*auto intro*: *filter-tab-all-False simp add*: *Base-foo-def*)
qed

lemma *methd-Ext* [*simp*]: *methd tprg Ext* =
table-of [(λ(*s*,*m*). (*s*, *Ext*, *m*)) *Ext-foo*]
apply (*rule trans*)

apply (*rule methd-rec′*)
apply (*auto simp add*: *ExtCl-def Object-def Ext-method-inheritance*)
done

accessibility

lemma *classesDefined*:
$[\![$*class tprg C* = *Some c*; *C*≠*Object*$]\!]$ $\Longrightarrow$ ∃ *sc*. *class tprg* (*super c*) = *Some sc*
apply (*auto simp add*: *Classes-def standard-classes-def*
BaseCl-def ExtCl-def MainCl-def
SXcptC-def ObjectC-def)
done

lemma *superclassesBase* [*simp*]: *superclasses tprg Base*={*Object*}
proof −
have *ws*: *ws-prog tprg* **by** (*rule ws-tprg*)
then show *?thesis*
by (*auto simp add*: *superclasses-rec BaseCl-def*)
qed

lemma *superclassesExt* [*simp*]: *superclasses tprg Ext*={*Base*,*Object*}
proof −
have *ws*: *ws-prog tprg* **by** (*rule ws-tprg*)
then show *?thesis*
by (*auto simp add*: *superclasses-rec ExtCl-def BaseCl-def*)
qed

lemma *superclassesMain* [*simp*]: *superclasses tprg Main*={*Object*}
proof −
have *ws*: *ws-prog tprg* **by** (*rule ws-tprg*)
then show *?thesis*
by (*auto simp add*: *superclasses-rec MainCl-def*)
qed

lemma *HasFoo-accessible*[*simp*]:*tprg*⊢(*Iface HasFoo*) *accessible-in P*
by (*simp add*: *accessible-in-RefT-simp is-public-def HasFooInt-def*)

lemma *HasFoo-is-acc-iface*[*simp*]: *is-acc-iface tprg P HasFoo*
by (*simp add*: *is-acc-iface-def*)

lemma *HasFoo-is-acc-type*[*simp*]: *is-acc-type tprg P* (*Iface HasFoo*)
by (*simp add*: *is-acc-type-def*)

lemma *Base-accessible*[*simp*]:*tprg*⊢(*Class Base*) *accessible-in P*
by (*simp add*: *accessible-in-RefT-simp is-public-def BaseCl-def*)

lemma *Base-is-acc-class*[*simp*]: *is-acc-class tprg P Base*
by (*simp add*: *is-acc-class-def*)

lemma *Base-is-acc-type*[*simp*]: *is-acc-type tprg P* (*Class Base*)

by (*simp add*: *is-acc-type-def*)

lemma *Ext-accessible*[*simp*]:*tprg*⊢(*Class Ext*) *accessible-in P*
by (*simp add*: *accessible-in-RefT-simp is-public-def ExtCl-def*)

lemma *Ext-is-acc-class*[*simp*]: *is-acc-class tprg P Ext*
by (*simp add*: *is-acc-class-def*)

lemma *Ext-is-acc-type*[*simp*]: *is-acc-type tprg P* (*Class Ext*)
by (*simp add*: *is-acc-type-def*)

lemma *accmethd-tprg-Object* [*simp*]: *accmethd tprg S Object* = *Map.empty*
apply (*unfold accmethd-def*)
apply (*simp*)
done

lemma *snd-special-simp*: *snd* ((λ(*s*, *m*). (*s*, *a*, *m*)) *x*) = (*a*,*snd x*)
by (*cases x*) (*auto*)

lemma *fst-special-simp*: *fst* ((λ(*s*, *m*). (*s*, *a*, *m*)) *x*) = *fst x*
by (*cases x*) (*auto*)

lemma *foo-sig-undeclared-in-Object*:
tprg⊢*mid foo-sig undeclared-in Object*
by (*auto simp add*: *undeclared-in-def cdeclaredmethd-def Object-mdecls-def*)

lemma *unique-sig-Base-foo*:
tprg⊢ *mdecl* (*sig*, *snd Base-foo*) *declared-in Base* ⟹ *sig*=*foo-sig*
by (*auto simp add*: *declared-in-def cdeclaredmethd-def*
Base-foo-def BaseCl-def)

lemma *Base-foo-no-override*:
tprg,*sig*⊢(*Base*,(*snd Base-foo*)) *overrides old* ⟹ *P*
apply (*drule overrides-commonD*)
apply (*clarsimp*)
apply (*frule subclsEval*)
apply (*rule ws-tprg*)
apply (*simp*)
apply (*rule classesDefined*)
apply *assumption*+
apply (*frule unique-sig-Base-foo*)
apply (*auto dest*!: *declared-not-undeclared intro*: *foo-sig-undeclared-in-Object*
dest: *unique-sig-Base-foo*)
done

lemma *Base-foo-no-stat-override*:
tprg,*sig*⊢(*Base*,(*snd Base-foo*)) $overrides_S$ *old* ⟹ *P*
apply (*drule stat-overrides-commonD*)
apply (*clarsimp*)

apply (*frule subclsEval*)
apply (*rule ws-tprg*)
apply (*simp*)
apply (*rule classesDefined*)
apply *assumption+*
apply (*frule unique-sig-Base-foo*)
apply (*auto dest*!: *declared-not-undeclared intro*: *foo-sig-undeclared-in-Object*
dest: *unique-sig-Base-foo*)
done

lemma *Base-foo-no-hide*:
tprg,*sig*⊢(*Base*,(*snd Base-foo*)) *hides old* ⟹ *P*
by (*auto dest*: *hidesD simp add*: *Base-foo-def member-is-static-simp*)

lemma *Ext-foo-no-hide*:
tprg,*sig*⊢(*Ext*,(*snd Ext-foo*)) *hides old* ⟹ *P*
by (*auto dest*: *hidesD simp add*: *Ext-foo-def member-is-static-simp*)

lemma *unique-sig-Ext-foo*:
tprg⊢ *mdecl* (*sig*, *snd Ext-foo*) *declared-in Ext* ⟹ *sig*=*foo-sig*
by (*auto simp add*: *declared-in-def cdeclaredmethd-def*
Ext-foo-def ExtCl-def)

lemma *Ext-foo-override*:
tprg,*sig*⊢(*Ext*,(*snd Ext-foo*)) *overrides old*
⟹ *old* = (*Base*,(*snd Base-foo*))
apply (*drule overrides-commonD*)
apply (*clarsimp*)
apply (*frule subclsEval*)
apply (*rule ws-tprg*)
apply (*simp*)
apply (*rule classesDefined*)
apply *assumption+*
apply (*frule unique-sig-Ext-foo*)
apply (*case-tac old*)
apply (*insert Base-declares-foo foo-sig-undeclared-in-Object*)
apply (*auto simp add*: *ExtCl-def Ext-foo-def*
BaseCl-def Base-foo-def Object-mdecls-def
split-paired-all
member-is-static-simp
dest: *declared-not-undeclared unique-declaration*)
done

lemma *Ext-foo-stat-override*:
tprg,*sig*⊢(*Ext*,(*snd Ext-foo*)) *overrides*$_S$ *old*
⟹ *old* = (*Base*,(*snd Base-foo*))
apply (*drule stat-overrides-commonD*)
apply (*clarsimp*)
apply (*frule subclsEval*)
apply (*rule ws-tprg*)
apply (*simp*)
apply (*rule classesDefined*)
apply *assumption+*

apply (*frule unique-sig-Ext-foo*)
apply (*case-tac old*)
apply (*insert Base-declares-foo foo-sig-undeclared-in-Object*)
apply (*auto simp add*: *ExtCl-def Ext-foo-def*
BaseCl-def Base-foo-def Object-mdecls-def
split-paired-all
member-is-static-simp
dest: *declared-not-undeclared unique-declaration*)
done

lemma *Base-foo-member-of-Base*:
tprg⊢(*Base*,*mdecl Base-foo*) *member-of Base*
by (*auto intro*!: *members.Immediate Base-declares-foo*)

lemma *Base-foo-member-in-Base*:
tprg⊢(*Base*,*mdecl Base-foo*) *member-in Base*
by (*rule member-of-to-member-in* [*OF Base-foo-member-of-Base*])

lemma *Ext-foo-member-of-Ext*:
tprg⊢(*Ext*,*mdecl Ext-foo*) *member-of Ext*
by (*auto intro*!: *members.Immediate Ext-declares-foo*)

lemma *Ext-foo-member-in-Ext*:
tprg⊢(*Ext*,*mdecl Ext-foo*) *member-in Ext*
by (*rule member-of-to-member-in* [*OF Ext-foo-member-of-Ext*])

lemma *Base-foo-permits-acc*:
tprg ⊢ (*Base*, *mdecl Base-foo*) *in Base permits-acc-from S*
by (*simp add*: *permits-acc-def Base-foo-def*)

lemma *Base-foo-accessible* [*simp*]:
tprg⊢(*Base*,*mdecl Base-foo*) *of Base accessible-from S*
by (*auto intro*: *accessible-fromR.Immediate*
Base-foo-member-of-Base Base-foo-permits-acc)

lemma *Base-foo-dyn-accessible* [*simp*]:
tprg⊢(*Base*,*mdecl Base-foo*) *in Base dyn-accessible-from S*
apply (*rule dyn-accessible-fromR.Immediate*)
apply (*rule Base-foo-member-in-Base*)
apply (*rule Base-foo-permits-acc*)
done

lemma *accmethd-Base* [*simp*]:
accmethd tprg S Base = *methd tprg Base*
apply (*simp add*: *accmethd-def*)
apply (*rule filter-tab-all-True*)
apply (*simp add*: *snd-special-simp fst-special-simp*)
done

lemma *Ext-foo-permits-acc*:

tprg ⊢ (*Ext*, *mdecl Ext-foo*) *in Ext permits-acc-from S*
by (*simp add*: *permits-acc-def Ext-foo-def*)

lemma *Ext-foo-accessible* [*simp*]:
tprg⊢(*Ext*,*mdecl Ext-foo*) *of Ext accessible-from S*
by (*auto intro*: *accessible-fromR.Immediate*
Ext-foo-member-of-Ext Ext-foo-permits-acc)

lemma *Ext-foo-dyn-accessible* [*simp*]:
tprg⊢(*Ext*,*mdecl Ext-foo*) *in Ext dyn-accessible-from S*
apply (*rule dyn-accessible-fromR.Immediate*)
apply (*rule Ext-foo-member-in-Ext*)
apply (*rule Ext-foo-permits-acc*)
done

lemma *Ext-foo-overrides-Base-foo*:
tprg⊢(*Ext*,*Ext-foo*) *overrides* (*Base*,*Base-foo*)
proof (*rule overridesR.Direct*, *simp-all*)
show ¬ *is-static Ext-foo*
by (*simp add*: *member-is-static-simp Ext-foo-def*)
show ¬ *is-static Base-foo*
by (*simp add*: *member-is-static-simp Base-foo-def*)
show *accmodi Ext-foo* ≠ *Private*
by (*simp add*: *Ext-foo-def*)
show *msig* (*Ext*, *Ext-foo*) = *msig* (*Base*, *Base-foo*)
by (*simp add*: *Ext-foo-def Base-foo-def*)
show *tprg*⊢*mdecl Ext-foo declared-in Ext*
by (*auto intro*: *Ext-declares-foo*)
show *tprg*⊢*mdecl Base-foo declared-in Base*
by (*auto intro*: *Base-declares-foo*)
show *tprg* ⊢(*Base*, *mdecl Base-foo*) *inheritable-in java-lang*
by (*simp add*: *inheritable-in-def Base-foo-def*)
show *tprg*⊢*resTy Ext-foo*⪯*resTy Base-foo*
by (*simp add*: *Ext-foo-def Base-foo-def mhead-resTy-simp*)
qed

lemma *accmethd-Ext* [*simp*]:
accmethd tprg S Ext = *methd tprg Ext*
apply (*simp add*: *accmethd-def*)
apply (*rule filter-tab-all-True*)
apply (*auto simp add*: *snd-special-simp fst-special-simp*)
done

lemma *cls-Ext*: *class tprg Ext* = *Some ExtCl*
by *simp*

lemma *dynmethd-Ext-foo*:
dynmethd tprg Base Ext ⦇*name* = *foo*, *parTs* = [*Class Base*]⦈
= *Some* (*Ext*,*snd Ext-foo*)
proof −
have *methd tprg Base* ⦇*name* = *foo*, *parTs* = [*Class Base*]⦈
= *Some* (*Base*,*snd Base-foo*) **and**
methd tprg Ext ⦇*name* = *foo*, *parTs* = [*Class Base*]⦈
= *Some* (*Ext*,*snd Ext-foo*)

```
    by (auto simp add: Ext-foo-def Base-foo-def foo-sig-def)
  with cls-Ext ws-tprg Ext-foo-overrides-Base-foo
  show ?thesis
    by (auto simp add: dynmethd-rec simp add: Ext-foo-def Base-foo-def)
qed


lemma Base-fields-accessible[simp]:
 accfield tprg S Base
  = table-of((map (λ((n,d),f).(n,(d,f)))) (DeclConcepts.fields tprg Base))
apply (auto simp add: accfield-def fun-eq-iff Let-def
                      accessible-in-RefT-simp
                      is-public-def
                      BaseCl-def
                      permits-acc-def
                      declared-in-def
                      cdeclaredfield-def
               intro!: filter-tab-all-True-Some filter-tab-None
                       accessible-fromR.Immediate
               intro: members.Immediate)
done


lemma arr-member-of-Base:
  tprg⊢(Base, fdecl (arr,
                  (|access = Public, static = True, type = PrimT Boolean.[]|)))
         member-of Base
by (auto intro: members.Immediate
       simp add: declared-in-def cdeclaredfield-def BaseCl-def)


lemma arr-member-in-Base:
  tprg⊢(Base, fdecl (arr,
                  (|access = Public, static = True, type = PrimT Boolean.[]|)))
         member-in Base
by (rule member-of-to-member-in [OF arr-member-of-Base])


lemma arr-member-of-Ext:
  tprg⊢(Base, fdecl (arr,
                    (|access = Public, static = True, type = PrimT Boolean.[]|)))
            member-of Ext
apply (rule members.Inherited)
apply   (simp add: inheritable-in-def)
apply   (simp add: undeclared-in-def cdeclaredfield-def ExtCl-def)
apply   (auto intro: arr-member-of-Base simp add: subcls1-def ExtCl-def)
done


lemma arr-member-in-Ext:
  tprg⊢(Base, fdecl (arr,
                  (|access = Public, static = True, type = PrimT Boolean.[]|)))
         member-in Ext
by (rule member-of-to-member-in [OF arr-member-of-Ext])


lemma Ext-fields-accessible[simp]:
accfield tprg S Ext
```

= *table-of*((*map* (λ((*n*,*d*),*f*).(*n*,(*d*,*f*)))) (*DeclConcepts.fields tprg Ext*))
apply (*auto simp add*: *accfield-def fun-eq-iff Let-def*
accessible-in-RefT-simp
is-public-def
BaseCl-def
ExtCl-def
permits-acc-def
intro!: *filter-tab-all-True-Some filter-tab-None*
accessible-fromR.Immediate)
apply (*auto intro*: *members.Immediate arr-member-of-Ext*
simp add: *declared-in-def cdeclaredfield-def ExtCl-def*)
done

lemma *arr-Base-dyn-accessible* [*simp*]:
tprg⊢(*Base*, *fdecl* (*arr*, (|*access*=*Public*,*static*= *True* ,*type*=*PrimT Boolean*.[]|)))
in Base dyn-accessible-from S
apply (*rule dyn-accessible-fromR.Immediate*)
apply (*rule arr-member-in-Base*)
apply (*simp add*: *permits-acc-def*)
done

lemma *arr-Ext-dyn-accessible*[*simp*]:
tprg⊢(*Base*, *fdecl* (*arr*, (|*access*=*Public*,*static*= *True* ,*type*=*PrimT Boolean*.[]|)))
in Ext dyn-accessible-from S
apply (*rule dyn-accessible-fromR.Immediate*)
apply (*rule arr-member-in-Ext*)
apply (*simp add*: *permits-acc-def*)
done

lemma *array-of-PrimT-acc* [*simp*]:
is-acc-type tprg java-lang (*PrimT t*.[])
apply (*simp add*: *is-acc-type-def accessible-in-RefT-simp*)
done

lemma *PrimT-acc* [*simp*]:
is-acc-type tprg java-lang (*PrimT t*)
apply (*simp add*: *is-acc-type-def accessible-in-RefT-simp*)
done

lemma *Object-acc* [*simp*]:
is-acc-class tprg java-lang Object
apply (*auto simp add*: *is-acc-class-def accessible-in-RefT-simp is-public-def*)
done

well-formedness

lemma *wf-HasFoo*: *wf-idecl tprg* (*HasFoo*, *HasFooInt*)
apply (*unfold wf-idecl-def HasFooInt-def*)
apply (*auto intro*!: *wf-mheadI ws-idecl-HasFoo*
simp add: *foo-sig-def foo-mhead-def mhead-resTy-simp*
member-is-static-simp)
done

declare *member-is-static-simp* [*simp*]
declare *wt.Skip* [*rule del*] *wt.Init* [*rule del*]
ML ‹*ML-Thms.bind-thms* (*wt-intros*, *map* (*rewrite-rule* ***context*** @{*thms id-def*}) @{*thms wt.intros*})›
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*
apply (*unfold Base-foo-defs*)
apply (*auto intro*!: *wf-mdeclI wf-mheadI intro*!: *wtIs*
simp add: *mhead-resTy-simp*)

apply (*rule exI*)
apply (*simp add*: *parameters-def*)
apply (*rule conjI*)
apply (*rule da.Comp*)
apply (*rule da.Expr*)
apply (*rule da.AssLVar*)
apply (*rule da.AccLVar*)
apply (*simp*)
apply (*rule assigned.select-convs*)
apply (*simp*)
apply (*rule assigned.select-convs*)
apply (*simp*)
apply (*simp*)
apply (*rule da.Jmp*)
apply (*simp*)
apply (*rule assigned.select-convs*)
apply (*simp*)
apply (*rule assigned.select-convs*)
apply (*simp*)
apply (*simp*)
done

lemma *wf-Ext-foo*: *wf-mdecl tprg Ext Ext-foo*
apply (*unfold Ext-foo-defs*)
apply (*auto intro*!: *wf-mdeclI wf-mheadI intro*!: *wtIs*
simp add: *mhead-resTy-simp*)
apply (*rule wt.Cast*)
prefer *2*
apply *simp*
apply (*rule-tac* [*2*] *narrow.subcls* [*THEN cast.narrow*])
apply (*auto intro*!: *wtIs*)

apply (*rule exI*)
apply (*simp add*: *parameters-def*)
apply (*rule conjI*)
apply (*rule da.Comp*)
apply (*rule da.Expr*)
apply (*rule da.Ass*)
apply *simp*

apply (*rule da.FVar*)
apply (*rule da.Cast*)
apply (*rule da.AccLVar*)
apply *simp*
apply (*rule assigned.select-convs*)
apply *simp*
apply (*rule da-Lit*)
apply (*simp*)
apply (*rule da.Comp*)
apply (*rule da.Expr*)
apply (*rule da.AssLVar*)
apply (*rule da.Lit*)
apply (*rule assigned.select-convs*)
apply *simp*
apply (*rule da.Jmp*)
apply *simp*
apply (*rule assigned.select-convs*)
apply *simp*
apply (*rule assigned.select-convs*)
apply (*simp*)
apply (*rule assigned.select-convs*)
apply *simp*
apply *simp*
done

declare *mhead-resTy-simp* [*simp add*]

lemma *wf-BaseC*: *wf-cdecl tprg* (*Base*,*BaseCl*)
apply (*unfold wf-cdecl-def BaseCl-def arr-viewed-from-def*)
apply (*auto intro*!: *wf-Base-foo*)
apply (*auto intro*!: *ws-cdecl-Base simp add*: *Base-foo-def foo-mhead-def*)
apply (*auto intro*!: *wtIs*)

apply (*rule exI*)
apply (*rule da.Expr*)
apply (*rule da.Ass*)
apply (*simp*)
apply (*rule da.FVar*)
apply (*rule da.Cast*)
apply (*rule da-Lit*)
apply *simp*
apply (*rule da.NewA*)
apply (*rule da.Lit*)
apply (*auto simp add*: *Base-foo-defs entails-def Let-def*)
apply (*insert Base-foo-no-stat-override*, *simp add*: *Base-foo-def*,*blast*)+
apply (*insert Base-foo-no-hide* , *simp add*: *Base-foo-def*,*blast*)
done

lemma *wf-ExtC*: *wf-cdecl tprg* (*Ext*,*ExtCl*)
apply (*unfold wf-cdecl-def ExtCl-def*)
apply (*auto intro*!: *wf-Ext-foo ws-cdecl-Ext*)
apply (*auto simp add*: *entails-def snd-special-simp*)
apply (*insert Ext-foo-stat-override*)
apply (*rule exI*,*rule da.Skip*)
apply (*force simp add*: *qmdecl-def Ext-foo-def Base-foo-def*)
apply (*force simp add*: *qmdecl-def Ext-foo-def Base-foo-def*)

apply (*force simp add*: *qmdecl-def Ext-foo-def Base-foo-def*)
apply (*insert Ext-foo-no-hide*)
apply (*simp-all add*: *qmdecl-def*)
apply *blast+*
done

lemma *wf-MainC*: *wf-cdecl tprg* (*Main*,*MainCl*)
apply (*unfold wf-cdecl-def MainCl-def*)
apply (*auto intro*: *ws-cdecl-Main*)
apply (*rule exI*,*rule da.Skip*)
done

lemma *wf-idecl-all*: *p*=*tprg* $\Longrightarrow$ *Ball* (*set Ifaces*) (*wf-idecl p*)
apply (*simp* (*no-asm*) *add*: *Ifaces-def*)
apply (*simp* (*no-asm-simp*))
apply (*rule wf-HasFoo*)
done

lemma *wf-cdecl-all-standard-classes*:
Ball (*set standard-classes*) (*wf-cdecl tprg*)
apply (*unfold standard-classes-def Let-def*
ObjectC-def SXcptC-def Object-mdecls-def SXcpt-mdecls-def)
apply (*simp* (*no-asm*) *add*: *wf-cdecl-def ws-cdecls*)
apply (*auto simp add*:*is-acc-class-def accessible-in-RefT-simp SXcpt-def*
intro: *da.Skip*)
apply (*auto simp add*: *Object-def Classes-def standard-classes-def*
SXcptC-def SXcpt-def)
done

lemma *wf-cdecl-all*: *p*=*tprg* $\Longrightarrow$ *Ball* (*set Classes*) (*wf-cdecl p*)
apply (*simp* (*no-asm*) *add*: *Classes-def*)
apply (*simp* (*no-asm-simp*))
apply (*rule wf-BaseC* [*THEN conjI*])
apply (*rule wf-ExtC* [*THEN conjI*])
apply (*rule wf-MainC* [*THEN conjI*])
apply (*rule wf-cdecl-all-standard-classes*)
done

theorem *wf-tprg*: *wf-prog tprg*
apply (*unfold wf-prog-def Let-def*)
apply (*simp* (*no-asm*) *add*: *unique-ifaces unique-classes*)
apply (*rule conjI*)
apply ((*simp* (*no-asm*) *add*: *Classes-def standard-classes-def*))
apply (*rule conjI*)
apply (*simp add*: *Object-mdecls-def*)
apply *safe*
apply (*cut-tac xn-cases*)
apply (*simp* (*no-asm-simp*) *add*: *Classes-def standard-classes-def*)
apply (*insert wf-idecl-all*)
apply (*insert wf-cdecl-all*)
apply *auto*
done

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*])}
apply (*unfold appl-methds-def*)
apply (*simp* (*no-asm*))
apply (*subgoal-tac tprg*⊢*NT*≼ *Class Base*)
apply (*auto simp add*: *cmheads-def Base-foo-defs*)
done

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*])}
by (*simp add*: *max-spec-def appl-methds-Base-foo Collect-conv-if*)

well-typedness

schematic-goal *wt-test*: (|*prg*=*tprg*,*cls*=*Main*,*lcl*=*Map.empty*(*VName e*↦*Class Base*)|)⊢*test ?pTs*::√
apply (*unfold test-def arr-viewed-from-def*)

apply (*rule wtIs*)
apply (*rule wtIs*)
apply (*rule wtIs*)
apply (*rule wtIs*)
apply (*simp*)
apply (*simp*)
apply (*simp*)
apply (*rule wtIs*)
apply (*simp*)
apply (*simp*)
apply (*rule wtIs*)
prefer *4*
apply (*simp*)
defer
apply (*rule wtIs*)
apply (*rule wtIs*)
apply (*rule wtIs*)
apply (*rule wtIs*)
apply (*simp*)
apply (*simp*)
apply (*rule wtIs*)
apply (*rule wtIs*)
apply (*simp*)
apply (*rule wtIs*)
apply (*simp*)
apply (*rule max-spec-Base-foo*)
apply (*simp*)
apply (*simp*)
apply (*simp*)
apply (*simp*)
apply (*simp*)
apply (*rule wtIs*)
apply (*rule wtIs*)
apply (*rule wtIs*)
apply (*rule wtIs*)
apply (*rule wtIs*)

apply (*rule wtIs*)
apply (*simp*)
apply (*simp*)
apply (*simp*)
apply (*simp*)
apply (*simp*)
apply (*rule wtIs*)
apply (*simp*)
apply (*rule wtIs*)
done

definite assignment

schematic-goal *da-test*: ⦇*prg*=*tprg*,*cls*=*Main*,*lcl*=*Map.empty*(*VName e*↦*Class Base*)⦈
⊢{} »⟨*test ?pTs*⟩» ⦇*nrm*={*VName e*},*brk*=λ *l*. *UNIV*⦈
apply (*unfold test-def arr-viewed-from-def*)
apply (*rule da.Comp*)
apply (*rule da.Expr*)
apply (*rule da.AssLVar*)
apply (*rule da.NewC*)
apply (*rule assigned.select-convs*)
apply (*simp*)
apply (*rule da.Try*)
apply (*rule da.Expr*)
apply (*rule da.Call*)
apply (*rule da.AccLVar*)
apply (*simp*)
apply (*rule assigned.select-convs*)
apply (*simp*)
apply (*rule da.Cons*)
apply (*rule da.Lit*)
apply (*rule da.Nil*)
apply (*rule da.Loop*)
apply (*rule da.Acc*)
apply (*simp*)
apply (*rule da.AVar*)
apply (*rule da.Acc*)
apply *simp*
apply (*rule da.FVar*)
apply (*rule da.Cast*)
apply (*rule da.Lit*)
apply (*rule da.Lit*)
apply (*rule da-Skip*)
apply (*simp*)
apply (*simp*,*rule assigned.select-convs*)
apply (*simp*)
apply (*simp*,*rule assigned.select-convs*)
apply (*simp*)
apply *simp*
apply *simp*
apply (*simp add*: *intersect-ts-def*)
done

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*
apply (*frule atleast-free-SucD*)
apply (*drule atleast-free-Suc* [*THEN iffD1*])

apply *clarsimp*
apply (*frule new-Addr-SomeI*)
apply *force*
done

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 ‹*ML-Thms.bind-thms* (*eval-intros*, *map*
(*simplify* (***context*** *delsimps* @{*thms Skip-eq*} *addsimps* @{*thms lvar-def*}) *o*
rewrite-rule **context** [@{*thm assign-def*}, @{*thm Let-def*}]) @{*thms eval.intros*})›
lemmas *eval-Is* = *eval-Init eval-StatRef AbruptIs eval-intros*

axiomatization
a :: *loc* **and**
b :: *loc* **and**
c :: *loc*

abbreviation *one* == *Suc 0*
abbreviation *two* == *Suc one*
abbreviation *three* == *Suc two*
abbreviation *four* == *Suc three*

abbreviation
obj-a == (|*tag*=*Arr* (*PrimT Boolean*) *2*
,*values*= *Map.empty*(*Inr 0*↦*Bool False*, *Inr 1*↦*Bool False*)|)

abbreviation
obj-b == (|*tag*=*CInst Ext*
,*values*=(*Map.empty*(*Inl* (*vee*, *Base*)↦*Null*, *Inl* (*vee*, *Ext*)↦*Intg 0*))|)

abbreviation
obj-c == (|*tag*=*CInst* (*SXcpt NullPointer*),*values*=*CONST Map.empty*|)

abbreviation *arr-N* == *Map.empty*(*Inl* (*arr*, *Base*)↦*Null*)
abbreviation *arr-a* == *Map.empty*(*Inl* (*arr*, *Base*)↦*Addr a*)

abbreviation
globs1 == *Map.empty*(*Inr Ext* ↦(|*tag*=*undefined*, *values*=*Map.empty*|),
Inr Base ↦(|*tag*=*undefined*, *values*=*arr-N*|),
Inr Object↦(|*tag*=*undefined*, *values*=*Map.empty*|))

abbreviation
globs2 == *Map.empty*(*Inr Ext* ↦(|*tag*=*undefined*, *values*=*Map.empty*|),
Inr Object↦(|*tag*=*undefined*, *values*=*Map.empty*|),
Inl a↦*obj-a*,
Inr Base ↦(|*tag*=*undefined*, *values*=*arr-a*|))

abbreviation *globs3* == *globs2*(*Inl b*↦*obj-b*)
abbreviation *globs8* == *globs3*(*Inl c*↦*obj-c*)
abbreviation *locs3* == *Map.empty*(*VName e*↦*Addr b*)
abbreviation *locs8* == *locs3*(*VName z*↦*Addr c*)

abbreviation *s0* == *st Map.empty Map.empty*
abbreviation *s0'* == *Norm s0*
abbreviation *s1* == *st globs1 Map.empty*
abbreviation *s1'* == *Norm s1*

abbreviation *s2* == *st globs2 Map.empty*
abbreviation *s2′* == *Norm s2*
abbreviation *s3* == *st globs3 locs3*
abbreviation *s3′* == *Norm s3*
abbreviation *s7′* == (*Some* (*Xcpt* (*Std NullPointer*)), *s3*)
abbreviation *s8* == *st globs8 locs8*
abbreviation *s8′* == *Norm s8*
abbreviation *s9′* == (*Some* (*Xcpt* (*Std IndOutBound*)), *s8*)

declare *prod.inject* [*simp del*]
schematic-goal *exec-test*:
⟦*the* (*new-Addr* (*heap s1*)) = *a*;
the (*new-Addr* (*heap ?s2*)) = *b*;
the (*new-Addr* (*heap ?s3*)) = *c*⟧ ⟹
atleast-free (*heap s0*) *four* ⟹
tprg⊢*s0′* −*test* [*Class Base*]→ *?s9′*
apply (*unfold test-def arr-viewed-from-def*)

apply (*simp* (*no-asm-use*))
apply (*drule* (*1*) *alloc-one*,*clarsimp*)
apply (*rule eval-Is*)
apply (*erule-tac V* = *the* (*new-Addr* -) = *c* **in** *thin-rl*)
apply (*erule-tac* [*2*] *V* = *new-Addr* - = *Some a* **in** *thin-rl*)
apply (*erule-tac* [*2*] *V* = *atleast-free* - *four* **in** *thin-rl*)
apply (*rule eval-Is*)
apply (*rule eval-Is*)
apply (*rule eval-Is*)
apply (*rule eval-Is*)

apply (*erule-tac V* = *the* (*new-Addr* -) = *b* **in** *thin-rl*)
apply (*erule-tac V* = *atleast-free* - *three* **in** *thin-rl*)
apply (*erule-tac* [*2*] *V* = *atleast-free* - *four* **in** *thin-rl*)
apply (*erule-tac* [*2*] *V* = *new-Addr* - = *Some a* **in** *thin-rl*)
apply (*rule eval-Is*)
apply (*simp*)
apply (*rule conjI*)
prefer *2* **apply** (*rule conjI HOL.refl*)+
apply (*rule eval-Is*)
apply (*simp add*: *arr-viewed-from-def*)
apply (*rule conjI*)
apply (*rule eval-Is*)
apply (*simp*)
apply (*simp*)
apply (*rule conjI*, *rule-tac* [*2*] *HOL.refl*)
apply (*rule eval-Is*)
apply (*rule eval-Is*)
apply (*rule eval-Is*)
apply (*rule init-done*, *simp*)
apply (*rule eval-Is*)
apply (*simp*)
apply (*simp add*: *check-field-access-def Let-def*)
apply (*rule eval-Is*)
apply (*simp*)
apply (*rule eval-Is*)
apply (*simp*)
apply (*rule halloc.New*)
apply (*simp* (*no-asm-simp*))
apply (*drule atleast-free-weaken*,*drule atleast-free-weaken*)

```
apply  (simp (no-asm-simp))
apply (simp add: upd-gobj-def)

apply (rule halloc.New)
apply  (drule alloc-one)
prefer 2 apply fast
apply  (simp (no-asm-simp))
apply (drule atleast-free-weaken)
apply force
apply (simp)
apply (drule alloc-one)
apply (simp (no-asm-simp))
apply clarsimp
apply (erule-tac V = atleast-free - three in thin-rl)
apply (drule-tac x = a in new-AddrD2 [THEN spec])
apply (simp (no-asm-use))
apply (rule eval-Is )
apply  (rule eval-Is )

apply  (rule eval-Is )
apply     (rule eval-Is )
apply     (rule eval-Is )
apply    (rule eval-Is )
apply     (rule eval-Is )
apply    (rule eval-Is )
apply   (simp)
apply   (simp)
apply   (subgoal-tac
         tprg⊢(Ext,mdecl Ext-foo) in Ext dyn-accessible-from Main)
apply     (simp add: check-method-access-def Let-def
                 invocation-declclass-def dynlookup-def dynmethd-Ext-foo)
apply     (rule Ext-foo-dyn-accessible)
apply  (rule eval-Is )
apply  (simp add: body-def Let-def)
apply  (rule eval-Is )
apply    (rule init-done, simp)
apply      (simp add: invocation-declclass-def dynlookup-def dynmethd-Ext-foo)
apply   (simp add: invocation-declclass-def dynlookup-def dynmethd-Ext-foo)
apply   (rule eval-Is )
apply    (rule eval-Is )
apply    (rule eval-Is )
apply     (rule eval-Is )
apply        (rule init-done, simp)
apply       (rule eval-Is )
apply        (rule eval-Is )
apply        (rule eval-Is )
apply       (simp)
apply      (simp split del: if-split)
apply     (simp add: check-field-access-def Let-def)
apply    (rule eval-Is )
apply   (simp)
apply   (rule conjI)
apply    (simp)
apply   (rule eval-Is )
apply  (simp)

apply simp
apply (rule sxalloc.intros)
apply (rule halloc.New)
```

apply (*erule alloc-one* [*THEN conjunct1*])
apply (*simp* (*no-asm-simp*))
apply (*simp* (*no-asm-simp*))
apply (*simp add*: *gupd-def lupd-def obj-ty-def split del*: *if-split*)
apply (*drule alloc-one* [*THEN conjunct1*])
apply (*simp* (*no-asm-simp*))
apply (*erule-tac V* = *atleast-free - two* **in** *thin-rl*)
apply (*drule-tac x* = *a* **in** *new-AddrD2* [*THEN spec*])
apply *simp*
apply (*rule eval-Is*)
apply (*rule eval-Is*)
apply (*rule eval-Is*)
apply (*rule eval-Is*)
apply (*rule eval-Is*)
apply (*rule init-done*, *simp*)
apply (*rule eval-Is*)
apply (*simp*)
apply (*simp add*: *check-field-access-def Let-def*)
apply (*rule eval-Is*)
apply (*simp* (*no-asm-simp*))
apply (*auto simp add*: *in-bounds-def*)
done
declare *prod.inject* [*simp*]

end

Chapter 17

Conform

1 Conformance notions for the type soundness proof for Java

theory *Conform* **imports** *State* **begin**

design issues:

- lconf allows for (arbitrary) inaccessible values
- "conforms" does not directly imply that the dynamic types of all objects on the heap are indeed existing classes. Yet this can be inferred for all referenced objs.

type-synonym $env' = prog \times (lname,\ ty)\ table$

extension of global store

definition $gext :: st \Rightarrow st \Rightarrow bool$ ($\text{-}\leq|\text{-}$ $[71,71]$ 70) **where**
$s{\leq}|s' \equiv \forall r.\ \forall obj{\in}globs\ s\ r{:}\ \exists obj'{\in}globs\ s'\ r{:}\ tag\ obj'{=}\ tag\ obj$

For the the proof of type soundness we will need the property that during execution, objects are not lost and moreover retain the values of their tags. So the object store grows conservatively. Note that if we considered garbage collection, we would have to restrict this property to accessible objects.

lemma *gext-objD*:
$[\![s{\leq}|s';\ globs\ s\ r = Some\ obj]\!]$
$\Longrightarrow \exists obj'.\ globs\ s'\ r = Some\ obj' \wedge tag\ obj' = tag\ obj$
apply (*simp only*: *gext-def*)
by *force*

lemma *rev-gext-objD*:
$[\![globs\ s\ r = Some\ obj;\ s{\leq}|s']\!]$
$\Longrightarrow \exists obj'.\ globs\ s'\ r = Some\ obj' \wedge tag\ obj' = tag\ obj$
by (*auto elim*: *gext-objD*)

lemma *init-class-obj-inited*:
$init\text{-}class\text{-}obj\ G\ C\ s1{\leq}|s2 \Longrightarrow inited\ C\ (globs\ s2)$
apply (*unfold inited-def init-obj-def*)
apply (*auto dest*!: *gext-objD*)
done

lemma *gext-refl* [*intro*!, *simp*]: $s{\leq}|s$
apply (*unfold gext-def*)
apply (*fast del*: *fst-splitE*)

done

lemma *gext-gupd* [*simp*, *elim*!]: $\bigwedge s.$ *globs s r* = *None* $\Longrightarrow$ $s{\le}|$*gupd*$(r{\mapsto}x)s$
by (*auto simp*: *gext-def*)

lemma *gext-new* [*simp*, *elim*!]: $\bigwedge s.$ *globs s r* = *None* $\Longrightarrow$ $s{\le}|$*init-obj G oi r s*
apply (*simp only*: *init-obj-def*)
apply (*erule-tac gext-gupd*)
done

lemma *gext-trans* [*elim*]: $\bigwedge X.\ [\![s{\le}|s';\ s'{\le}|s'']\!] \Longrightarrow s{\le}|s''$
by (*force simp*: *gext-def*)

lemma *gext-upd-gobj* [*intro*!]: $s{\le}|$*upd-gobj r n v s*
apply (*simp only*: *gext-def*)
apply *auto*
apply (*case-tac ra* = *r*)
apply *auto*
apply (*case-tac globs s r* = *None*)
apply *auto*
done

lemma *gext-cong1* [*simp*]: *set-locals l s1*$\le|$*s2* = *s1*$\le|$*s2*
by (*auto simp*: *gext-def*)

lemma *gext-cong2* [*simp*]: *s1*$\le|$*set-locals l s2* = *s1*$\le|$*s2*
by (*auto simp*: *gext-def*)

lemma *gext-lupd1* [*simp*]: *lupd*$(vn{\mapsto}v)$*s1*$\le|$*s2* = *s1*$\le|$*s2*
by (*auto simp*: *gext-def*)

lemma *gext-lupd2* [*simp*]: *s1*$\le|$*lupd*$(vn{\mapsto}v)$*s2* = *s1*$\le|$*s2*
by (*auto simp*: *gext-def*)

lemma *inited-gext*: $[\![$*inited C* (*globs s*); $s{\le}|s']\!] \Longrightarrow$ *inited C* (*globs s'*)
apply (*unfold inited-def*)
apply (*auto dest*: *gext-objD*)
done

value conformance

definition *conf* :: *prog* ⇒ *st* ⇒ *val* ⇒ *ty* ⇒ *bool* (-,-⊢-::≼- [*71*,*71*,*71*,*71*] *70*)
where $G,s{\vdash}v{::}{\preceq}T$ = ($\exists\, T'{\in}$*typeof* ($\lambda a.$ *map-option obj-ty* (*heap s a*)) $v{:}G{\vdash}T'{\preceq}T$)

lemma *conf-cong* [*simp*]: $G,$*set-locals l s*$\vdash v{::}{\preceq}T$ = $G,s{\vdash}v{::}{\preceq}T$
by (*auto simp*: *conf-def*)

lemma *conf-lupd* [*simp*]: $G,lupd(vn \mapsto va)s \vdash v::\preceq T = G,s \vdash v::\preceq T$
by (*auto simp*: *conf-def*)

lemma *conf-PrimT* [*simp*]: $\forall dt.\ typeof\ dt\ v = Some\ (PrimT\ t) \Longrightarrow G,s \vdash v::\preceq PrimT\ t$
apply (*simp add*: *conf-def*)
done

lemma *conf-Boolean*: $G,s \vdash v::\preceq PrimT\ Boolean \Longrightarrow \exists\ b.\ v = Bool\ b$
by (*cases v*)
  (*auto simp*: *conf-def obj-ty-def*
    dest: *widen-Boolean2*
    split: *obj-tag.splits*)

lemma *conf-litval* [*rule-format* (*no-asm*)]:
  $typeof\ (\lambda a.\ None)\ v = Some\ T \longrightarrow G,s \vdash v::\preceq T$
apply (*unfold conf-def*)
apply (*rule val.induct*)
apply *auto*
done

lemma *conf-Null* [*simp*]: $G,s \vdash Null::\preceq T = G \vdash NT \preceq T$
by (*simp add*: *conf-def*)

lemma *conf-Addr*:
  $G,s \vdash Addr\ a::\preceq T = (\exists obj.\ heap\ s\ a = Some\ obj \wedge G \vdash obj\text{-}ty\ obj \preceq T)$
by (*auto simp*: *conf-def*)

lemma *conf-AddrI*: $[\![heap\ s\ a = Some\ obj;\ G \vdash obj\text{-}ty\ obj \preceq T]\!] \Longrightarrow G,s \vdash Addr\ a::\preceq T$
apply (*rule conf-Addr* [*THEN iffD2*])
by *fast*

lemma *defval-conf* [*rule-format* (*no-asm*), *elim*]:
  $is\text{-}type\ G\ T \longrightarrow G,s \vdash default\text{-}val\ T::\preceq T$
apply (*unfold conf-def*)
apply (*induct T*)
apply (*auto intro*: *prim-ty.induct*)
done

lemma *conf-widen* [*rule-format* (*no-asm*), *elim*]:
  $G \vdash T \preceq T' \Longrightarrow G,s \vdash x::\preceq T \longrightarrow ws\text{-}prog\ G \longrightarrow G,s \vdash x::\preceq T'$
apply (*unfold conf-def*)
apply (*rule val.induct*)
apply (*auto elim*: *ws-widen-trans*)
done

lemma *conf-gext* [*rule-format* (*no-asm*), *elim*]:
  $G,s \vdash v::\preceq T \longrightarrow s \le | s' \longrightarrow G,s' \vdash v::\preceq T$
apply (*unfold gext-def conf-def*)

apply (*rule val.induct*)
apply *force+*
done

lemma *conf-list-widen* [*rule-format* (*no-asm*)]:
ws-prog G $\Longrightarrow$
$\forall$ *Ts Ts'*. *list-all2* (*conf G s*) *vs Ts*
$\longrightarrow$ $G\vdash Ts[\preceq]$ *Ts'* $\longrightarrow$ *list-all2* (*conf G s*) *vs Ts'*
apply (*unfold widens-def*)
apply (*rule list-all2-trans*)
apply *auto*
done

lemma *conf-RefTD* [*rule-format* (*no-asm*)]:
$G,s\vdash a'::\preceq RefT\ T$
$\longrightarrow a' = Null \vee (\exists\, a\ obj\ T'.\ a' = Addr\ a \wedge heap\ s\ a = Some\ obj \wedge$
$obj\text{-}ty\ obj = T' \wedge G\vdash T'\preceq RefT\ T)$
apply (*unfold conf-def*)
apply (*induct-tac a'*)
apply (*auto dest*: *widen-PrimT*)
done

value list conformance

definition
lconf :: *prog* $\Rightarrow$ *st* $\Rightarrow$ (*'a*, *val*) *table* $\Rightarrow$ (*'a*, *ty*) *table* $\Rightarrow$ *bool* (-,-$\vdash$-[::$\preceq$]- [*71*,*71*,*71*,*71*] *70*)
where $G,s\vdash vs[::\preceq]Ts = (\forall\, n.\ \forall\, T\in Ts\ n:\ \exists\, v\in vs\ n:\ G,s\vdash v::\preceq T)$

lemma *lconfD*: $[\![G,s\vdash vs[::\preceq]Ts;\ Ts\ n = Some\ T]\!] \Longrightarrow G,s\vdash(the\ (vs\ n))::\preceq T$
by (*force simp*: *lconf-def*)

lemma *lconf-cong* [*simp*]: $\bigwedge s.\ G,set\text{-}locals\ x\ s\vdash l[::\preceq]L = G,s\vdash l[::\preceq]L$
by (*auto simp*: *lconf-def*)

lemma *lconf-lupd* [*simp*]: $G,lupd(vn\mapsto v)s\vdash l[::\preceq]L = G,s\vdash l[::\preceq]L$
by (*auto simp*: *lconf-def*)

lemma *lconf-new*: $[\![L\ vn = None;\ G,s\vdash l[::\preceq]L]\!] \Longrightarrow G,s\vdash l(vn\mapsto v)[::\preceq]L$
by (*auto simp*: *lconf-def*)

lemma *lconf-upd*: $[\![G,s\vdash l[::\preceq]L;\ G,s\vdash v::\preceq T;\ L\ vn = Some\ T]\!] \Longrightarrow$
$G,s\vdash l(vn\mapsto v)[::\preceq]L$
by (*auto simp*: *lconf-def*)

lemma *lconf-ext*: $[\![G,s\vdash l[::\preceq]L;\ G,s\vdash v::\preceq T]\!] \Longrightarrow G,s\vdash l(vn\mapsto v)[::\preceq]L(vn\mapsto T)$
by (*auto simp*: *lconf-def*)

lemma *lconf-map-sum* [*simp*]:
$G,s \vdash l1\ (+)\ l2[::\preceq]L1\ (+)\ L2 = (G,s \vdash l1[::\preceq]L1 \wedge G,s \vdash l2[::\preceq]L2)$
apply (*unfold lconf-def*)
apply *safe*
apply (*case-tac* [*3*] *n*)
apply (*force split*: *sum.split*)+
done

lemma *lconf-ext-list* [*rule-format* (*no-asm*)]:
$\bigwedge X.\ [\![G,s \vdash l[::\preceq]L]\!] \Longrightarrow$
$\forall vs\ Ts.\ distinct\ vns \longrightarrow length\ Ts = length\ vns$
$\longrightarrow list\text{-}all2\ (conf\ G\ s)\ vs\ Ts \longrightarrow G,s \vdash l(vns[\mapsto]vs)[::\preceq]L(vns[\mapsto]Ts)$
apply (*unfold lconf-def*)
apply (*induct-tac vns*)
apply *clarsimp*
apply *clarify*
apply (*frule list-all2-lengthD*)
apply (*clarsimp*)
done

lemma *lconf-deallocL*: $[\![G,s \vdash l[::\preceq]L(vn \mapsto T);\ L\ vn = None]\!] \Longrightarrow G,s \vdash l[::\preceq]L$
apply (*simp only*: *lconf-def*)
apply *safe*
apply (*drule spec*)
apply (*drule ospec*)
apply *auto*
done

lemma *lconf-gext* [*elim*]: $[\![G,s \vdash l[::\preceq]L;\ s \leq|s']\!] \Longrightarrow G,s' \vdash l[::\preceq]L$
apply (*simp only*: *lconf-def*)
apply *fast*
done

lemma *lconf-empty* [*simp*, *intro*!]: $G,s \vdash vs[::\preceq]Map.empty$
apply (*unfold lconf-def*)
apply *force*
done

lemma *lconf-init-vals* [*intro*!]:
$\forall n.\ \forall T \in fs\ n{:}is\text{-}type\ G\ T \Longrightarrow G,s \vdash init\text{-}vals\ fs[::\preceq]fs$
apply (*unfold lconf-def*)
apply *force*
done

weak value list conformance

Only if the value is defined it has to conform to its type. This is the contribution of the definite assignment analysis to the notion of conformance. The definite assignment analysis ensures that the program only attempts to access local variables that actually have a defined value in the state. So conformance must only ensure that the defined values are of the right type, and not also that the value is defined.

definition
wlconf :: *prog* ⇒ *st* ⇒ ('*a*, *val*) *table* ⇒ ('*a*, *ty*) *table* ⇒ *bool* (-,-⊢-[∼::≼]- [*71*,*71*,*71*,*71*] *70*)
where *G*,*s*⊢*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*
by (*auto simp*: *wlconf-def*)

lemma *wlconf-cong* [*simp*]: ⋀*s*. *G*,*set-locals x s*⊢*l*[∼::≼]*L* = *G*,*s*⊢*l*[∼::≼]*L*
by (*auto simp*: *wlconf-def*)

lemma *wlconf-lupd* [*simp*]: *G*,*lupd*(*vn*↦*v*)*s*⊢*l*[∼::≼]*L* = *G*,*s*⊢*l*[∼::≼]*L*
by (*auto simp*: *wlconf-def*)

lemma *wlconf-upd*: ⟦*G*,*s*⊢*l*[∼::≼]*L*; *G*,*s*⊢*v*::≼*T*; *L vn* = *Some T*⟧ ⟹
G,*s*⊢*l*(*vn*↦*v*)[∼::≼]*L*
by (*auto simp*: *wlconf-def*)

lemma *wlconf-ext*: ⟦*G*,*s*⊢*l*[∼::≼]*L*; *G*,*s*⊢*v*::≼*T*⟧ ⟹ *G*,*s*⊢*l*(*vn*↦*v*)[∼::≼]*L*(*vn*↦*T*)
by (*auto simp*: *wlconf-def*)

lemma *wlconf-map-sum* [*simp*]:
G,*s*⊢*l1* (+) *l2*[∼::≼]*L1* (+) *L2* = (*G*,*s*⊢*l1*[∼::≼]*L1* ∧ *G*,*s*⊢*l2*[∼::≼]*L2*)
apply (*unfold wlconf-def*)
apply *safe*
apply (*case-tac* [*3*] *n*)
apply (*force split*: *sum.split*)+
done

lemma *wlconf-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*)
apply (*unfold wlconf-def*)
apply (*induct-tac vns*)
apply *clarsimp*
apply *clarify*
apply (*frule list-all2-lengthD*)
apply *clarsimp*
done

lemma *wlconf-deallocL*: ⟦*G*,*s*⊢*l*[∼::≼]*L*(*vn*↦*T*); *L vn* = *None*⟧ ⟹ *G*,*s*⊢*l*[∼::≼]*L*
apply (*simp only*: *wlconf-def*)
apply *safe*
apply (*drule spec*)
apply (*drule ospec*)
defer
apply (*drule ospec*)
apply *auto*

done

lemma *wlconf-gext* [*elim*]: ⟦*G*,*s*⊢*l*[∼::≼]*L*; *s*≤|*s'*⟧ ⟹ *G*,*s'*⊢*l*[∼::≼]*L*
apply (*simp only*: *wlconf-def*)
apply *fast*
done

lemma *wlconf-empty* [*simp*, *intro*!]: *G*,*s*⊢*vs*[∼::≼]*Map.empty*
apply (*unfold wlconf-def*)
apply *force*
done

lemma *wlconf-empty-vals*: *G*,*s*⊢*Map.empty*[∼::≼]*ts*
by (*simp add*: *wlconf-def*)

lemma *wlconf-init-vals* [*intro*!]:
∀ *n*. ∀ *T*∈*fs n*:*is-type G T* ⟹ *G*,*s*⊢*init-vals fs*[∼::≼]*fs*
apply (*unfold wlconf-def*)
apply *force*
done

lemma *lconf-wlconf*:
G,*s*⊢*l*[::≼]*L* ⟹ *G*,*s*⊢*l*[∼::≼]*L*
by (*force simp add*: *lconf-def wlconf-def*)

object conformance

definition
oconf :: *prog* ⇒ *st* ⇒ *obj* ⇒ *oref* ⇒ *bool* (-,-⊢-::≼√- [*71*,*71*,*71*,*71*] *70*) **where**
(*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*)
by (*auto simp*: *oconf-def Let-def*)

lemma *oconf-lconf*: *G*,*s*⊢*obj*::≼√*r* ⟹ *G*,*s*⊢*values obj*[::≼]*var-tys G* (*tag obj*) *r*
by (*simp add*: *oconf-def*)

lemma *oconf-cong* [*simp*]: *G*,*set-locals l s*⊢*obj*::≼√*r* = *G*,*s*⊢*obj*::≼√*r*
by (*auto simp*: *oconf-def Let-def*)

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*
apply (*auto simp add*: *oconf-def*)
apply (*drule-tac var-tys-Some-eq* [*THEN iffD1*])
defer
apply (*subst obj-ty-cong*)
apply (*auto dest*!: *fields-table-SomeD split*: *sum.split-asm obj-tag.split-asm*)
done

state conformance

definition
  conforms :: *state* ⇒ *env'* ⇒ *bool* (-::≼- [*71*,*71*] *70*) **where**
  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*
by (*auto simp*: *conforms-def Let-def*)

lemma *conforms-localD*: (*x*, *s*)::≼(*G*, *L*) ⟹ *G*,*s*⊢*locals s*[∼::≼]*L*
by (*auto simp*: *conforms-def Let-def*)

lemma *conforms-XcptLocD*: ⟦(*x*, *s*)::≼(*G*, *L*); *x* = *Some* (*Xcpt* (*Loc a*))⟧ ⟹
        G,*s*⊢*Addr a*::≼ *Class* (*SXcpt Throwable*)
by (*auto simp*: *conforms-def Let-def*)

lemma *conforms-RetD*: ⟦(*x*, *s*)::≼(*G*, *L*); *x* = *Some* (*Jump Ret*)⟧ ⟹
        (*locals s*) *Result* ≠ *None*
by (*auto simp*: *conforms-def Let-def*)

lemma *conforms-RefTD*:
 ⟦*G*,*s*⊢*a'*::≼*RefT t*; *a'* ≠ *Null*; (*x*,*s*) ::≼(*G*, *L*)⟧ ⟹
  ∃ *a obj*. *a'* = *Addr a* ∧ *globs s* (*Inl a*) = *Some obj* ∧
  G⊢*obj-ty obj*≼*RefT t* ∧ *is-type G* (*obj-ty obj*)
apply (*drule-tac conf-RefTD*)
apply *clarsimp*
apply (*rule conforms-globsD* [*THEN oconf-is-type*])
apply *auto*
done

lemma *conforms-Jump* [*iff*]:
 j=*Ret* ⟶ *locals s Result* ≠ *None*
  ⟹ ((*Some* (*Jump j*), *s*)::≼(*G*, *L*)) = (*Norm s*::≼(*G*, *L*))
by (*auto simp*: *conforms-def Let-def*)

lemma *conforms-StdXcpt* [*iff*]:
((*Some* (*Xcpt* (*Std xn*)), *s*)::⪯(*G*, *L*)) = (*Norm s*::⪯(*G*, *L*))
by (*auto simp*: *conforms-def*)

lemma *conforms-Err* [*iff*]:
((*Some* (*Error e*), *s*)::⪯(*G*, *L*)) = (*Norm s*::⪯(*G*, *L*))
by (*auto simp*: *conforms-def*)

lemma *conforms-raise-if* [*iff*]:
((*raise-if c xn x*, *s*)::⪯(*G*, *L*)) = ((*x*, *s*)::⪯(*G*, *L*))
by (*auto simp*: *abrupt-if-def*)

lemma *conforms-error-if* [*iff*]:
((*error-if c err x*, *s*)::⪯(*G*, *L*)) = ((*x*, *s*)::⪯(*G*, *L*))
by (*auto simp*: *abrupt-if-def*)

lemma *conforms-NormI*: (*x*, *s*)::⪯(*G*, *L*) ⟹ *Norm s*::⪯(*G*, *L*)
by (*auto simp*: *conforms-def Let-def*)

lemma *conforms-absorb* [*rule-format*]:
(*a*, *b*)::⪯(*G*, *L*) ⟶ (*absorb j a*, *b*)::⪯(*G*, *L*)
apply (*rule impI*)
apply (*case-tac a*)
apply (*case-tac absorb j a*)
apply *auto*
apply (*rename-tac a′*)
apply (*case-tac absorb j* (*Some a′*),*auto*)
apply (*erule conforms-NormI*)
done

lemma *conformsI*: ⟦∀ *r*. ∀ *obj*∈*globs s r*: *G*,*s*⊢*obj*::⪯√*r*;
G,*s*⊢*locals s*[∼::⪯]*L*;
∀ *a*. *x* = *Some* (*Xcpt* (*Loc a*)) ⟶ *G*,*s*⊢*Addr a*::⪯ *Class* (*SXcpt Throwable*);
x = *Some* (*Jump Ret*)⟶ *locals s Result* ≠ *None*⟧ ⟹
(*x*, *s*)::⪯(*G*, *L*)
by (*auto simp*: *conforms-def Let-def*)

lemma *conforms-xconf*: ⟦(*x*, *s*)::⪯(*G*,*L*);
∀ *a*. *x′* = *Some* (*Xcpt* (*Loc a*)) ⟶ *G*,*s*⊢*Addr a*::⪯ *Class* (*SXcpt Throwable*);
x′ = *Some* (*Jump Ret*) ⟶ *locals s Result* ≠ *None*⟧ ⟹
(*x′*,*s*)::⪯(*G*,*L*)
by (*fast intro*: *conformsI elim*: *conforms-globsD conforms-localD*)

lemma *conforms-lupd*:
⟦(*x*, *s*)::⪯(*G*, *L*); *L vn* = *Some T*; *G*,*s*⊢*v*::⪯*T*⟧ ⟹ (*x*, *lupd*(*vn*↦*v*)*s*)::⪯(*G*, *L*)
by (*force intro*: *conformsI wlconf-upd dest*: *conforms-globsD conforms-localD*
conforms-XcptLocD conforms-RetD
simp: *oconf-def*)

lemmas *conforms-allocL-aux* = *conforms-localD* [*THEN wlconf-ext*]

lemma *conforms-allocL*:
$[\![(x,\ s)::\preceq(G,\ L);\ G,s\vdash v::\preceq T]\!] \Longrightarrow (x,\ lupd(vn\mapsto v)s)::\preceq(G,\ L(vn\mapsto T))$
by (*force intro*: *conformsI dest*: *conforms-globsD conforms-RetD*
elim: *conforms-XcptLocD conforms-allocL-aux*
simp: *oconf-def*)

lemmas *conforms-deallocL-aux* = *conforms-localD* [*THEN wlconf-deallocL*]

lemma *conforms-deallocL*: $\bigwedge s.[\![s::\preceq(G,\ L(vn\mapsto T));\ L\ vn = None]\!] \Longrightarrow s::\preceq(G,L)$
by (*fast intro*: *conformsI dest*: *conforms-globsD conforms-RetD*
elim: *conforms-XcptLocD conforms-deallocL-aux*)

lemma *conforms-gext*: $[\![(x,\ s)::\preceq(G,L);\ s\leq|s'$;
$\forall r.\ \forall obj\in globs\ s'\ r:\ G,s'\vdash obj::\preceq\surd r$;
$locals\ s'=locals\ s]\!] \Longrightarrow (x,s')::\preceq(G,L)$
apply (*rule conformsI*)
apply *assumption*
apply (*drule conforms-localD*) **apply** *force*
apply (*intro strip*)
apply (*drule* (*1*) *conforms-XcptLocD*) **apply** *force*
apply (*intro strip*)
apply (*drule* (*1*) *conforms-RetD*) **apply** *force*
done

lemma *conforms-xgext*:
$[\![(x\ ,s)::\preceq(G,L);\ (x',\ s')::\preceq(G,\ L);\ s'\leq|s;dom\ (locals\ s') \subseteq dom\ (locals\ s)]\!]$
$\Longrightarrow (x',s)::\preceq(G,L)$
apply (*erule-tac conforms-xconf*)
apply (*fast dest*: *conforms-XcptLocD*)
apply (*intro strip*)
apply (*drule* (*1*) *conforms-RetD*)
apply (*auto dest*: *domI*)
done

lemma *conforms-gupd*: $\bigwedge obj.\ [\![(x,\ s)::\preceq(G,\ L);\ G,s\vdash obj::\preceq\surd r;\ s\leq|gupd(r\mapsto obj)s]\!]$
$\Longrightarrow\ (x,\ gupd(r\mapsto obj)s)::\preceq(G,\ L)$
apply (*rule conforms-gext*)
apply *auto*
apply (*force dest*: *conforms-globsD simp add*: *oconf-def*)+
done

lemma *conforms-upd-gobj*: $[\![(x,s)::\preceq(G,\ L);\ globs\ s\ r = Some\ obj$;
$var\text{-}tys\ G\ (tag\ obj)\ r\ n = Some\ T;\ G,s\vdash v::\preceq T]\!] \Longrightarrow (x,upd\text{-}gobj\ r\ n\ v\ s)::\preceq(G,L)$
apply (*rule conforms-gext*)
apply *auto*
apply (*drule* (*1*) *conforms-globsD*)
apply (*simp add*: *oconf-def*)
apply *safe*
apply (*rule lconf-upd*)
apply *auto*

apply (*simp only*: *obj-ty-cong*)
apply (*force dest*: *conforms-globsD intro*!: *lconf-upd*
simp add: *oconf-def cong del*: *old.sum.case-cong-weak*)
done

lemma *conforms-set-locals*:
$[\![(x,s)::\preceq(G,\ L'); \ G,s\vdash l[\sim::\preceq]L; \ x=Some\ (Jump\ Ret) \longrightarrow l\ Result \neq None]\!]$
$\Longrightarrow (x,set\text{-}locals\ l\ s)::\preceq(G,L)$
apply (*rule conformsI*)
apply (*intro strip*)
apply *simp*
apply (*drule* (*2*) *conforms-globsD*)
apply *simp*
apply (*intro strip*)
apply (*drule* (*1*) *conforms-XcptLocD*)
apply *simp*
apply (*intro strip*)
apply (*drule* (*1*) *conforms-RetD*)
apply *simp*
done

lemma *conforms-locals*:
$[\![(a,b)::\preceq(G,\ L); \ L\ x = Some\ T; locals\ b\ x \neq None]\!]$
$\Longrightarrow G,b\vdash the\ (locals\ b\ x)::\preceq T$
apply (*force simp*: *conforms-def Let-def wlconf-def*)
done

lemma *conforms-return*:
$\bigwedge s'. \ [\![(x,s)::\preceq(G,\ L); \ (x',s')::\preceq(G,\ L'); \ s\leq|s'; x'\neq Some\ (Jump\ Ret)]\!] \Longrightarrow$
$(x',set\text{-}locals\ (locals\ s)\ s')::\preceq(G,\ L)$
apply (*rule conforms-xconf*)
prefer *2* **apply** (*force dest*: *conforms-XcptLocD*)
apply (*erule conforms-gext*)
apply (*force dest*: *conforms-globsD*)+
done

end

Chapter 18

DefiniteAssignmentCorrect

1 Correctness of Definite Assignment

theory *DefiniteAssignmentCorrect* **imports** *WellForm Eval* **begin**

declare [[*simproc del*: *wt-expr wt-var wt-exprs wt-stmt*]]

lemma *sxalloc-no-jump*:
  assumes *sxalloc*: $G \vdash s0$ $-sxalloc\rightarrow$ *s1* **and**
    no-jmp: *abrupt s0* $\neq$ *Some* (*Jump j*)
  shows *abrupt s1* $\neq$ *Some* (*Jump j*)
using *sxalloc no-jmp*
by *cases simp-all*

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*)
using *sxalloc jump*
by *cases simp-all*

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*)
using *halloc no-jmp*
by *cases simp-all*

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*)
using *halloc jump*
by *cases simp-all*

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 (*cases normal s0*)

```
  case True
  with eval obtain s0' where eval': G⊢Norm s0' −Body D c−≻v→s1 and
                             s0: s0 = Norm s0'
    by (cases s0) simp
  from eval' obtain s2 where
     s1: s1 = abupd (absorb Ret)
             (if ∃ l. abrupt s2 = Some (Jump (Break l)) ∨
                      abrupt s2 = Some (Jump (Cont l))
              then abupd (λx. Some (Error CrossMethodJump)) s2 else s2)
    by cases simp
  show ?thesis
  proof (cases ∃ l. abrupt s2 = Some (Jump (Break l)) ∨
                    abrupt s2 = Some (Jump (Cont l)))
    case True
    with s1 have abrupt s1 = Some (Error CrossMethodJump)
      by (cases s2) simp
    thus ?thesis by simp
  next
    case False
    with s1 have s1=abupd (absorb Ret) s2
      by simp
    with False show ?thesis
      by (cases s2,cases j) (auto simp add: absorb-def)
  qed
next
  case False
  with eval obtain s0' abr where G⊢(Some abr,s0') −Body D c−≻v→s1
                                 s0 = (Some abr, s0')
    by (cases s0) fastforce
  with this jump
  show ?thesis
    by (cases) (simp)
qed


lemma Methd-no-jump:
   assumes eval: G⊢s0 −Methd D sig−≻v→ s1 and
           jump: abrupt s0 ≠ Some (Jump j)
   shows abrupt s1 ≠ Some (Jump j)
proof (cases normal s0)
  case True
   with eval obtain s0' where G⊢Norm s0' −Methd D sig−≻v→s1
                              s0 = Norm s0'
    by (cases s0) simp
  then obtain D' body where G⊢s0 −Body D' body−≻v→s1
    by (cases) (simp add: body-def2)
  from this jump
  show ?thesis
    by (rule Body-no-jump)
next
  case False
  with eval obtain s0' abr where G⊢(Some abr,s0') −Methd D sig−≻v→s1
                                 s0 = (Some abr, s0')
    by (cases s0) fastforce
  with this jump
  show ?thesis
    by (cases) (simp)
qed
```

```
lemma jumpNestingOkS-mono:
  assumes jumpNestingOk-l': jumpNestingOkS jmps' c
      and       subset: jmps' ⊆ jmps
 shows jumpNestingOkS jmps c
proof −
    have True and True and
       ⋀ jmps' jmps. ⟦jumpNestingOkS jmps' c; jmps' ⊆ jmps⟧ ⟹ jumpNestingOkS jmps c
  proof (induct rule: var.induct expr.induct stmt.induct)
    case (Lab j c jmps' jmps)
    note jmpOk = ‹jumpNestingOkS jmps' (j· c)›
    note jmps = ‹jmps' ⊆ jmps›
    with jmpOk have jumpNestingOkS ({j} ∪ jmps') c by simp
    moreover from jmps have ({j} ∪ jmps') ⊆ ({j} ∪ jmps) by auto
    ultimately
    have jumpNestingOkS ({j} ∪ jmps) c
      by (rule Lab.hyps)
    thus ?case
      by simp
  next
    case (Jmp j jmps' jmps)
    thus ?case
      by (cases j) auto
  next
    case (Comp c1 c2 jmps' jmps)
    from Comp.prems
    have jumpNestingOkS jmps c1 by − (rule Comp.hyps,auto)
    moreover from Comp.prems
    have jumpNestingOkS jmps c2 by − (rule Comp.hyps,auto)
    ultimately show ?case
      by simp
  next
    case (If' e c1 c2 jmps' jmps)
    from If'.prems
    have jumpNestingOkS jmps c1 by − (rule If'.hyps,auto)
    moreover from If'.prems
    have jumpNestingOkS jmps c2 by − (rule If'.hyps,auto)
    ultimately show ?case
      by simp
  next
    case (Loop l e c jmps' jmps)
    from ‹jumpNestingOkS jmps' (l· While(e) c)›
    have jumpNestingOkS ({Cont l} ∪ jmps') c by simp
    moreover
    from ‹jmps' ⊆ jmps›
    have {Cont l} ∪ jmps'  ⊆ {Cont l} ∪ jmps by auto
    ultimately
    have jumpNestingOkS ({Cont l} ∪ jmps) c
      by (rule Loop.hyps)
    thus ?case by simp
  next
    case (TryC c1 C vn c2 jmps' jmps)
    from TryC.prems
    have jumpNestingOkS jmps c1 by − (rule TryC.hyps,auto)
    moreover from TryC.prems
    have jumpNestingOkS jmps c2 by − (rule TryC.hyps,auto)
    ultimately show ?case
      by simp
  next
```

```
    case (Fin c1 c2 jmps′ jmps)
    from Fin.prems
    have jumpNestingOkS jmps c1 by − (rule Fin.hyps,auto)
    moreover from Fin.prems
    have jumpNestingOkS jmps c2 by − (rule Fin.hyps,auto)
    ultimately show ?case
      by simp
  qed (simp-all)
  with jumpNestingOk-l′ subset
  show ?thesis
    by iprover
qed

corollary jumpNestingOk-mono:
  assumes jmpOk: jumpNestingOk jmps′ t
     and subset: jmps′ ⊆ jmps
  shows  jumpNestingOk jmps t
proof (cases t)
  case (In1 expr-stmt)
  show ?thesis
  proof (cases expr-stmt)
    case (Inl e)
    with In1 show ?thesis by simp
  next
    case (Inr s)
    with In1 jmpOk subset show ?thesis by (auto intro: jumpNestingOkS-mono)
  qed
qed (simp-all)


lemma assign-abrupt-propagation:
 assumes f-ok: abrupt (f n s) ≠ x
   and     ass: abrupt (assign f n s) = x
  shows abrupt s = x
proof (cases x)
  case None
  with ass show ?thesis
    by (cases s) (simp add: assign-def Let-def)
next
  case (Some xcpt)
  from f-ok
  obtain xf sf where f n s = (xf,sf)
    by (cases f n s)
  with Some ass f-ok show ?thesis
    by (cases s) (simp add: assign-def Let-def)
qed


lemma wt-init-comp-ty′:
is-acc-type (prg Env) (pid (cls Env)) T ⟹ Env⊢init-comp-ty T::√
apply (unfold init-comp-ty-def)
apply (clarsimp simp add: accessible-in-RefT-simp
                          is-acc-type-def is-acc-class-def)
done


lemma fvar-upd-no-jump:
      assumes upd: upd = snd (fst (fvar statDeclC stat fn a s′))
        and noJmp: abrupt s ≠ Some (Jump j)
```

```
      shows abrupt (upd val s) ≠ Some (Jump j)
proof (cases stat)
  case True
  with noJmp upd
  show ?thesis
    by (cases s) (simp add: fvar-def2)
next
  case False
  with noJmp upd
  show ?thesis
    by (cases s) (simp add: fvar-def2)
qed
```

```
lemma avar-state-no-jump:
  assumes jmp: abrupt (snd (avar G i a s)) = Some (Jump j)
  shows abrupt s = Some (Jump j)
proof (cases normal s)
  case True with jmp show ?thesis by (auto simp add: avar-def2 abrupt-if-def)
next
  case False with jmp show ?thesis by (auto simp add: avar-def2 abrupt-if-def)
qed
```

```
lemma avar-upd-no-jump:
     assumes upd: upd = snd (fst (avar G i a s'))
       and noJmp: abrupt s ≠ Some (Jump j)
       shows abrupt (upd val s) ≠ Some (Jump j)
using upd noJmp
by (cases s) (simp add: avar-def2 abrupt-if-def)
```

The next theorem expresses: If jumps (breaks, continues, returns) are nested correctly, we won't find an unexpected jump in the result state of the evaluation. For exeample, a break can't leave its enclosing loop, an return cant leave its enclosing method. To proove this, the method call is critical. Allthough the wellformedness of the whole program guarantees that the jumps (breaks,continues and returns) are nested correctly in all method bodies, the call rule alone does not guarantee that I will call a method or even a class that is part of the program due to dynamic binding! To be able to enshure this we need a kind of conformance of the state, like in the typesafety proof. But then we will redo the typesafty proof here. It would be nice if we could find an easy precondition that will guarantee that all calls will actually call classes and methods of the current program, which can be instantiated in the typesafty proof later on. To fix this problem, I have instrumented the semantic definition of a call to filter out any breaks in the state and to throw an error instead.

To get an induction hypothesis which is strong enough to perform the proof, we can't just assume *jumpNestingOk* for the empty set and conlcude, that no jump at all will be in the resulting state, because the set is altered by the statements *Lab* and *While*.

The wellformedness of the program is used to enshure that for all classinitialisations and methods the nesting of jumps is wellformed, too.

```
theorem jumpNestingOk-eval:
  assumes eval: G⊢ s0 −t≻→ (v,s1)
     and jmpOk: jumpNestingOk jmps t
     and wt: Env⊢t::T
     and wf: wf-prog G
     and  G: prg Env = G
     and no-jmp: ∀ j. abrupt s0 = Some (Jump j) ⟶ j ∈ jmps
               (is ?Jmp jmps s0)
```

```
  shows (∀j. fst s1 = Some (Jump j) ⟶ j ∈ jmps) ∧
          (normal s1 ⟶
           (∀ w upd. v=In2 (w,upd)
            ⟶   (∀ s j val.
                  abrupt s ≠ Some (Jump j) ⟶
                  abrupt (upd val s) ≠ Some (Jump j))))
      (is ?Jmp jmps s1 ∧ ?Upd v s1)
proof −
  let ?HypObj = λ t s0 s1 v.
      (∀ jmps T Env.
        ?Jmp jmps s0 ⟶ jumpNestingOk jmps t ⟶ Env⊢t::T ⟶ prg Env=G⟶
        ?Jmp jmps s1 ∧ ?Upd v s1)
```

— Variable *?HypObj* is the following goal spelled in terms of the object logic, instead of the meta logic. It is needed in some cases of the induction were, the atomize-rulify process of induct does not work fine, because the eval rules mix up object and meta logic. See for example the case for the loop.

```
  from eval
  have ⋀ jmps T Env. ⟦?Jmp jmps s0; jumpNestingOk jmps t; Env⊢t::T;prg Env=G⟧
         ⟹ ?Jmp jmps s1 ∧ ?Upd v s1
      (is PROP ?Hyp t s0 s1 v)
```

— We need to abstract over *jmps* since *jmps* are extended during analysis of *Lab*. Also we need to abstract over *T* and *Env* since they are altered in various typing judgements.

```
  proof (induct)
    case Abrupt thus ?case by simp
  next
    case Skip thus ?case by simp
  next
    case Expr thus ?case by (elim wt-elim-cases) simp
  next
    case (Lab s0 c s1 jmp jmps T Env)
    note jmpOK = ‹jumpNestingOk jmps (In1r (jmp• c))›
    note G = ‹prg Env = G›
    have wt-c: Env⊢c::√
      using Lab.prems by (elim wt-elim-cases)
    {
      fix j
      assume ab-s1: abrupt (abupd (absorb jmp) s1) = Some (Jump j)
      have j∈jmps
      proof −
        from ab-s1 have jmp-s1: abrupt s1 = Some (Jump j)
          by (cases s1) (simp add: absorb-def)
        note hyp-c = ‹PROP ?Hyp (In1r c) (Norm s0) s1 ◇›
        from ab-s1 have j ≠ jmp
          by (cases s1) (simp add: absorb-def)
        moreover have j ∈ {jmp} ∪ jmps
        proof −
          from jmpOK
          have jumpNestingOk ({jmp} ∪ jmps) (In1r c) by simp
          with wt-c jmp-s1 G hyp-c
          show ?thesis
            by − (rule hyp-c [THEN conjunct1,rule-format],simp)
        qed
        ultimately show ?thesis
          by simp
      qed
    }
    thus ?case by simp
  next
    case (Comp s0 c1 s1 c2 s2 jmps T Env)
    note jmpOk = ‹jumpNestingOk jmps (In1r (c1;; c2))›
```

```
    note G = ‹prg Env = G›
    from Comp.prems obtain
      wt-c1: Env⊢c1::√ and wt-c2: Env⊢c2::√
      by (elim wt-elim-cases)
    {
      fix j
      assume abr-s2: abrupt s2 = Some (Jump j)
      have j∈jmps
      proof -
        have jmp: ?Jmp jmps s1
        proof -
          note hyp-c1 = ‹PROP ?Hyp (In1r c1) (Norm s0) s1 ◇›
          with wt-c1 jmpOk G
          show ?thesis by simp
        qed
        moreover note hyp-c2 = ‹PROP ?Hyp (In1r c2) s1 s2 (◇::vals)›
        have jmpOk': jumpNestingOk jmps (In1r c2) using jmpOk by simp
        moreover note wt-c2 G abr-s2
        ultimately show j ∈ jmps
          by (rule hyp-c2 [THEN conjunct1,rule-format (no-asm)])
      qed
    } thus ?case by simp
  next
    case (If s0 e b s1 c1 c2 s2 jmps T Env)
    note jmpOk = ‹jumpNestingOk jmps (In1r (If(e) c1 Else c2))›
    note G = ‹prg Env = G›
    from If.prems obtain
              wt-e: Env⊢e::−PrimT Boolean and
      wt-then-else: Env⊢(if the-Bool b then c1 else c2)::√
      by (elim wt-elim-cases) simp
    {
      fix j
      assume jmp: abrupt s2 = Some (Jump j)
      have j∈jmps
      proof -
        note ‹PROP ?Hyp (In1l e) (Norm s0) s1 (In1 b)›
        with wt-e G have ?Jmp jmps s1
          by simp
        moreover note hyp-then-else =
          ‹PROP ?Hyp (In1r (if the-Bool b then c1 else c2)) s1 s2 ◇›
        have jumpNestingOk jmps (In1r (if the-Bool b then c1 else c2))
          using jmpOk by (cases the-Bool b) simp-all
        moreover note wt-then-else G jmp
        ultimately show j∈ jmps
          by (rule hyp-then-else [THEN conjunct1,rule-format (no-asm)])
      qed
    }
    thus ?case by simp
  next
    case (Loop s0 e b s1 c s2 l s3 jmps T Env)
    note jmpOk = ‹jumpNestingOk jmps (In1r (l• While(e) c))›
    note G = ‹prg Env = G›
    note wt = ‹Env⊢In1r (l• While(e) c)::T›
    then obtain
              wt-e: Env⊢e::−PrimT Boolean and
              wt-c: Env⊢c::√
      by (elim wt-elim-cases)
    {
      fix j
```

```
assume jmp: abrupt s3 = Some (Jump j)
have j∈jmps
proof –
  note ‹PROP ?Hyp (In1l e) (Norm s0) s1 (In1 b)›
  with wt-e G have jmp-s1: ?Jmp jmps s1
    by simp
  show ?thesis
  proof (cases the-Bool b)
    case False
    from Loop.hyps
    have s3=s1
      by (simp (no-asm-use) only: if-False False)
    with jmp-s1 jmp have j ∈ jmps by simp
    thus ?thesis by simp
  next
    case True
    from Loop.hyps

    have ?HypObj (In1r c) s1 s2 (◇::vals)
      apply (simp (no-asm-use) only: True if-True )
      apply (erule conjE)+
      apply assumption
      done
    note hyp-c = this [rule-format (no-asm)]
    moreover from jmpOk have jumpNestingOk ({Cont l} ∪ jmps) (In1r c)
      by simp
    moreover from jmp-s1 have ?Jmp ({Cont l} ∪ jmps) s1 by simp
    ultimately have jmp-s2: ?Jmp ({Cont l} ∪ jmps) s2
      using wt-c G by iprover
    have ?Jmp jmps (abupd (absorb (Cont l)) s2)
    proof –
      {
        fix j'
        assume abs: abrupt (abupd (absorb (Cont l)) s2)=Some (Jump j')
        have j' ∈ jmps
        proof (cases j' = Cont l)
          case True
          with abs show ?thesis
            by (cases s2) (simp add: absorb-def)
        next
          case False
          with abs have abrupt s2 = Some (Jump j')
            by (cases s2) (simp add: absorb-def)
          with jmp-s2 False show ?thesis
            by simp
        qed
      }
      thus ?thesis by simp
    qed
    moreover
    from Loop.hyps
    have ?HypObj (In1r (l· While(e) c))
                (abupd (absorb (Cont l)) s2) s3 (◇::vals)
      apply (simp (no-asm-use) only: True if-True)
      apply (erule conjE)+
      apply assumption
      done
    note hyp-w = this [rule-format (no-asm)]
    note jmpOk wt G jmp
```

```
        ultimately show j∈ jmps
          by (rule hyp-w [THEN conjunct1,rule-format (no-asm)])
      qed
    qed
  }
  thus ?case by simp
next
  case (Jmp s j jmps T Env) thus ?case by simp
next
  case (Throw s0 e a s1 jmps T Env)
  note jmpOk = ‹jumpNestingOk jmps (In1r (Throw e))›
  note G = ‹prg Env = G›
  from Throw.prems obtain Te where
    wt-e: Env⊢e::−Te
    by (elim wt-elim-cases)
  {
    fix j
    assume jmp: abrupt (abupd (throw a) s1) = Some (Jump j)
    have j∈jmps
    proof −
      from ‹PROP ?Hyp (In1l e) (Norm s0) s1 (In1 a)›
      have ?Jmp jmps s1 using wt-e G by simp
      moreover
      from jmp
      have abrupt s1 = Some (Jump j)
        by (cases s1) (simp add: throw-def abrupt-if-def)
      ultimately show j ∈ jmps by simp
    qed
  }
  thus ?case by simp
next
  case (Try s0 c1 s1 s2 C vn c2 s3 jmps T Env)
  note jmpOk = ‹jumpNestingOk jmps (In1r (Try c1 Catch(C vn) c2))›
  note G = ‹prg Env = G›
  from Try.prems obtain
    wt-c1: Env⊢c1::√ and
    wt-c2: Env(|lcl := (lcl Env)(VName vn↦Class C)|)⊢c2::√
    by (elim wt-elim-cases)
  {
    fix j
    assume jmp: abrupt s3 = Some (Jump j)
    have j∈jmps
    proof −
      note ‹PROP ?Hyp (In1r c1) (Norm s0) s1 (◇::vals)›
      with jmpOk wt-c1 G
      have jmp-s1: ?Jmp jmps s1 by simp
      note s2 = ‹G⊢s1 −sxalloc→ s2›
      show j ∈ jmps
      proof (cases G,s2⊢catch C)
        case False
        from Try.hyps have s3=s2
          by (simp (no-asm-use) only: False if-False)
        with jmp have abrupt s1 = Some (Jump j)
          using sxalloc-no-jump' [OF s2] by simp
        with jmp-s1
        show ?thesis by simp
      next
        case True
        with Try.hyps
```

```
        have ?HypObj (In1r c2) (new-xcpt-var vn s2) s3 (◇::vals)
          apply (simp (no-asm-use) only: True if-True simp-thms)
          apply (erule conjE)+
          apply assumption
          done
        note hyp-c2 = this [rule-format (no-asm)]
        from jmp-s1 sxalloc-no-jump' [OF s2]
        have ?Jmp jmps s2
          by simp
        hence ?Jmp jmps (new-xcpt-var vn s2)
          by (cases s2) simp
        moreover have jumpNestingOk jmps (In1r c2) using jmpOk by simp
        moreover note wt-c2
        moreover from G
        have prg (Env(|lcl := (lcl Env)(VName vn↦Class C)|)) = G
          by simp
        moreover note jmp
        ultimately show ?thesis
          by (rule hyp-c2 [THEN conjunct1,rule-format (no-asm)])
      qed
    qed
  }
  thus ?case by simp
next
  case (Fin s0 c1 x1 s1 c2 s2 s3 jmps T Env)
  note jmpOk = ‹jumpNestingOk jmps (In1r (c1 Finally c2))›
  note G = ‹prg Env = G›
  from Fin.prems obtain
    wt-c1: Env⊢c1::√ and wt-c2: Env⊢c2::√
    by (elim wt-elim-cases)
  {
    fix j
    assume jmp: abrupt s3 = Some (Jump j)
    have j ∈ jmps
    proof (cases x1=Some (Jump j))
      case True
      note hyp-c1 = ‹PROP ?Hyp (In1r c1) (Norm s0) (x1,s1) ◇›
      with True jmpOk wt-c1 G show ?thesis
        by – (rule hyp-c1 [THEN conjunct1,rule-format (no-asm)],simp-all)
    next
      case False
      note hyp-c2 = ‹PROP ?Hyp (In1r c2) (Norm s1) s2 ◇›
      note ‹s3 = (if ∃ err. x1 = Some (Error err) then (x1, s1)
                 else abupd (abrupt-if (x1 ≠ None) x1) s2)›
      with False jmp have abrupt s2 = Some (Jump j)
        by (cases s2) (simp add: abrupt-if-def)
      with jmpOk wt-c2 G show ?thesis
        by – (rule hyp-c2 [THEN conjunct1,rule-format (no-asm)],simp-all)
    qed
  }
  thus ?case by simp
next
  case (Init C c s0 s3 s1 s2 jmps T Env)
  note ‹jumpNestingOk jmps (In1r (Init C))›
  note G = ‹prg Env = G›
  note ‹the (class G C) = c›
  with Init.prems have c: class G C = Some c
    by (elim wt-elim-cases) auto
  {
```

```
      fix j
      assume jmp: abrupt s3 = (Some (Jump j))
      have j∈jmps
      proof (cases inited C (globs s0))
        case True
        with Init.hyps have s3=Norm s0
          by simp
        with jmp
        have False by simp thus ?thesis ..
      next
        case False
        from wf c G
        have wf-cdecl: wf-cdecl G (C,c)
          by (simp add: wf-prog-cdecl)
        from Init.hyps
        have ?HypObj (In1r (if C = Object then Skip else Init (super c)))
                     (Norm ((init-class-obj G C) s0)) s1 (◇::vals)
          apply (simp (no-asm-use) only: False if-False simp-thms)
          apply (erule conjE)+
          apply assumption
          done
        note hyp-s1 = this [rule-format (no-asm)]
        from wf-cdecl G have
          wt-super: Env⊢(if C = Object then Skip else Init (super c))::√
          by (cases C=Object)
             (auto dest: wf-cdecl-supD is-acc-classD)
        from hyp-s1 [OF - - wt-super G]
        have ?Jmp jmps s1
          by simp
        hence jmp-s1: ?Jmp jmps ((set-lvars Map.empty) s1) by (cases s1) simp
        from False Init.hyps
        have ?HypObj (In1r (init c)) ((set-lvars Map.empty) s1) s2 (◇::vals)
          apply (simp (no-asm-use) only: False if-False simp-thms)
          apply (erule conjE)+
          apply assumption
          done
        note hyp-init-c = this [rule-format (no-asm)]
        from wf-cdecl
        have wt-init-c: (|prg = G, cls = C, lcl = Map.empty|)⊢init c::√
          by (rule wf-cdecl-wt-init)
        from wf-cdecl have jumpNestingOkS {} (init c)
          by (cases rule: wf-cdeclE)
        hence jumpNestingOkS jmps (init c)
          by (rule jumpNestingOkS-mono) simp
        moreover
        have abrupt s2 = Some (Jump j)
        proof −
          from False Init.hyps
          have s3 = (set-lvars (locals (store s1))) s2  by simp
          with jmp show ?thesis by (cases s2) simp
        qed
        ultimately show ?thesis
          using hyp-init-c [OF jmp-s1 - wt-init-c]
          by simp
      qed
    }
    thus ?case by simp
  next
    case (NewC s0 C s1 a s2 jmps T Env)
```

```
    {
      fix j
      assume jmp: abrupt s2 = Some (Jump j)
      have j∈jmps
      proof −
        note ‹prg Env = G›
        moreover note hyp-init = ‹PROP ?Hyp (In1r (Init C)) (Norm s0) s1 ◇›
        moreover from wf NewC.prems
        have Env⊢(Init C)::√
          by (elim wt-elim-cases) (drule is-acc-classD,simp)
        moreover
        have abrupt s1 = Some (Jump j)
        proof −
          from ‹G⊢s1 −halloc CInst C≻a→ s2› and jmp show ?thesis
            by (rule halloc-no-jump')
        qed
        ultimately show j ∈ jmps
          by − (rule hyp-init [THEN conjunct1,rule-format (no-asm)],auto)
      qed
    }
    thus ?case by simp
  next
    case (NewA s0 elT s1 e i s2 a s3 jmps T Env)
    {
      fix j
      assume jmp: abrupt s3 = Some (Jump j)
      have j∈jmps
      proof −
        note G = ‹prg Env = G›
        from NewA.prems
        obtain wt-init: Env⊢init-comp-ty elT::√ and
               wt-size: Env⊢e::−PrimT Integer
          by (elim wt-elim-cases) (auto dest: wt-init-comp-ty')
        note ‹PROP ?Hyp (In1r (init-comp-ty elT)) (Norm s0) s1 ◇›
        with wt-init G
        have ?Jmp jmps s1
          by (simp add: init-comp-ty-def)
        moreover
        note hyp-e = ‹PROP ?Hyp (In1l e) s1 s2 (In1 i)›
        have abrupt s2 = Some (Jump j)
        proof −
          note ‹G⊢abupd (check-neg i) s2−halloc Arr elT (the-Intg i)≻a→ s3›
          moreover note jmp
          ultimately
          have abrupt (abupd (check-neg i) s2) = Some (Jump j)
            by (rule halloc-no-jump')
          thus ?thesis by (cases s2) auto
        qed
        ultimately show j∈jmps using wt-size G
          by − (rule hyp-e [THEN conjunct1,rule-format (no-asm)],simp-all)
      qed
    }
    thus ?case by simp
  next
    case (Cast s0 e v s1 s2 cT jmps T Env)
    {
      fix j
      assume jmp: abrupt s2 = Some (Jump j)
      have j∈jmps
```

```
      proof -
        note hyp-e = ‹PROP ?Hyp (In1l e) (Norm s0) s1 (In1 v)›
        note ‹prg Env = G›
        moreover from Cast.prems
        obtain eT where Env⊢e::−eT by (elim wt-elim-cases)
        moreover
        have abrupt s1 = Some (Jump j)
        proof -
          note ‹s2 = abupd (raise-if (¬ G,snd s1⊢v fits cT) ClassCast) s1›
          moreover note jmp
          ultimately show ?thesis by (cases s1) (simp add: abrupt-if-def)
        qed
        ultimately show ?thesis
          by - (rule hyp-e [THEN conjunct1,rule-format (no-asm)], simp-all)
      qed
    }
    thus ?case by simp
  next
    case (Inst s0 e v s1 b eT jmps T Env)
    {
      fix j
      assume jmp: abrupt s1 = Some (Jump j)
      have j∈jmps
      proof -
        note hyp-e = ‹PROP ?Hyp (In1l e) (Norm s0) s1 (In1 v)›
        note ‹prg Env = G›
        moreover from Inst.prems
        obtain eT where Env⊢e::−eT by (elim wt-elim-cases)
        moreover note jmp
        ultimately show j∈jmps
          by - (rule hyp-e [THEN conjunct1,rule-format (no-asm)], simp-all)
      qed
    }
    thus ?case by simp
  next
    case Lit thus ?case by simp
  next
    case (UnOp s0 e v s1 unop jmps T Env)
    {
      fix j
      assume jmp: abrupt s1 = Some (Jump j)
      have j∈jmps
      proof -
        note hyp-e = ‹PROP ?Hyp (In1l e) (Norm s0) s1 (In1 v)›
        note ‹prg Env = G›
        moreover from UnOp.prems
        obtain eT where Env⊢e::−eT by (elim wt-elim-cases)
        moreover note jmp
        ultimately show  j∈jmps
          by - (rule hyp-e [THEN conjunct1,rule-format (no-asm)], simp-all)
      qed
    }
    thus ?case by simp
  next
    case (BinOp s0 e1 v1 s1 binop e2 v2 s2 jmps T Env)
    {
      fix j
      assume jmp: abrupt s2 = Some (Jump j)
      have j∈jmps
```

```
    proof -
      note G = ‹prg Env = G›
      from BinOp.prems
      obtain e1T e2T where
        wt-e1: Env⊢e1::-e1T and
        wt-e2: Env⊢e2::-e2T
        by (elim wt-elim-cases)
      note ‹PROP ?Hyp (In1l e1) (Norm s0) s1 (In1 v1)›
      with G wt-e1 have jmp-s1: ?Jmp jmps s1 by simp
      note hyp-e2 =
        ‹PROP ?Hyp (if need-second-arg binop v1 then In1l e2 else In1r Skip)
                   s1 s2 (In1 v2)›
      show j∈jmps
      proof (cases need-second-arg binop v1)
        case True with jmp-s1 wt-e2 jmp G
        show ?thesis
          by - (rule hyp-e2 [THEN conjunct1,rule-format (no-asm)],simp-all)
      next
        case False with jmp-s1 jmp G
        show ?thesis
          by - (rule hyp-e2 [THEN conjunct1,rule-format (no-asm)],auto)
      qed
    qed
  }
  thus ?case by simp
next
  case Super thus ?case by simp
next
  case (Acc s0 va v f s1 jmps T Env)
  {
    fix j
    assume jmp: abrupt s1 = Some (Jump j)
    have j∈jmps
    proof -
      note hyp-va = ‹PROP ?Hyp (In2 va) (Norm s0) s1 (In2 (v,f))›
      note ‹prg Env = G›
      moreover from Acc.prems
      obtain vT where Env⊢va::=vT by (elim wt-elim-cases)
      moreover note jmp
      ultimately show j∈jmps
        by - (rule hyp-va [THEN conjunct1,rule-format (no-asm)], simp-all)
    qed
  }
  thus ?case by simp
next
  case (Ass s0 va w f s1 e v s2 jmps T Env)
  note G = ‹prg Env = G›
  from Ass.prems
  obtain vT eT where
    wt-va: Env⊢va::=vT and
     wt-e: Env⊢e::-eT
    by (elim wt-elim-cases)
  note hyp-v = ‹PROP ?Hyp (In2 va) (Norm s0) s1 (In2 (w,f))›
  note hyp-e = ‹PROP ?Hyp (In1l e) s1 s2 (In1 v)›
  {
    fix j
    assume jmp: abrupt (assign f v s2) = Some (Jump j)
    have j∈jmps
    proof -
```

```
      have abrupt s2 = Some (Jump j)
      proof (cases normal s2)
        case True
        from ‹G⊢s1 −e−≻v→ s2› and True have nrm-s1: normal s1
          by (rule eval-no-abrupt-lemma [rule-format])
        with nrm-s1 wt-va G True
        have abrupt (f v s2) ≠ Some (Jump j)
          using hyp-v [THEN conjunct2,rule-format (no-asm)]
          by simp
        from this jmp
        show ?thesis
          by (rule assign-abrupt-propagation)
      next
        case False with jmp
        show ?thesis by (cases s2) (simp add: assign-def Let-def)
      qed
      moreover from wt-va G
      have ?Jmp jmps s1
        by − (rule hyp-v [THEN conjunct1],simp-all)
      ultimately show ?thesis using G
        by − (rule hyp-e [THEN conjunct1,rule-format (no-asm)], simp-all, rule wt-e)
    qed
  }
  thus ?case by simp
next
  case (Cond s0 e0 b s1 e1 e2 v s2 jmps T Env)
  note G = ‹prg Env = G›
  note hyp-e0 = ‹PROP ?Hyp (In1l e0) (Norm s0) s1 (In1 b)›
  note hyp-e1-e2 = ‹PROP ?Hyp (In1l (if the-Bool b then e1 else e2)) s1 s2 (In1 v)›
  from Cond.prems
  obtain e1T e2T
    where wt-e0: Env⊢e0::−PrimT Boolean
     and  wt-e1: Env⊢e1::−e1T
     and  wt-e2: Env⊢e2::−e2T
    by (elim wt-elim-cases)
  {
    fix j
    assume jmp: abrupt s2 = Some (Jump j)
    have j∈jmps
    proof −
      from wt-e0 G
      have jmp-s1: ?Jmp jmps s1
        by − (rule hyp-e0 [THEN conjunct1],simp-all)
      show ?thesis
      proof (cases the-Bool b)
        case True
        with jmp-s1 wt-e1 G jmp
        show ?thesis
          by−(rule hyp-e1-e2 [THEN conjunct1,rule-format (no-asm)],simp-all)
      next
        case False
        with jmp-s1 wt-e2 G jmp
        show ?thesis
          by−(rule hyp-e1-e2 [THEN conjunct1,rule-format (no-asm)],simp-all)
      qed
    qed
  }
  thus ?case by simp
next
```

```
    case (Call s0 e a s1 args vs s2 D mode statT mn pTs s3 s3' accC v s4
               jmps T Env)
    note G = ‹prg Env = G›
    from Call.prems
    obtain eT argsT
      where wt-e: Env⊢e::−eT and wt-args: Env⊢args::≐argsT
      by (elim wt-elim-cases)
    {
      fix j
      assume jmp: abrupt ((set-lvars (locals (store s2))) s4)
                     = Some (Jump j)
      have j∈jmps
      proof −
        note hyp-e = ‹PROP ?Hyp (In1l e) (Norm s0) s1 (In1 a)›
        from wt-e G
        have jmp-s1: ?Jmp jmps s1
          by − (rule hyp-e [THEN conjunct1],simp-all)
        note hyp-args = ‹PROP ?Hyp (In3 args) s1 s2 (In3 vs)›
        have abrupt s2 = Some (Jump j)
        proof −
          note ‹G⊢s3' −Methd D (|name = mn, parTs = pTs|)−≻v→ s4›
          moreover
          from jmp have abrupt s4 = Some (Jump j)
            by (cases s4) simp
          ultimately have abrupt s3' = Some (Jump j)
            by − (rule ccontr,drule (1) Methd-no-jump,simp)
          moreover note ‹s3' = check-method-access G accC statT mode
                            (|name = mn, parTs = pTs|) a s3›
          ultimately have abrupt s3 = Some (Jump j)
            by (cases s3)
               (simp add: check-method-access-def abrupt-if-def Let-def)
          moreover
          note ‹s3 = init-lvars G D (|name=mn, parTs=pTs|) mode a vs s2›
          ultimately show ?thesis
            by (cases s2) (auto simp add: init-lvars-def2)
        qed
        with jmp-s1 wt-args G
        show ?thesis
          by − (rule hyp-args [THEN conjunct1,rule-format (no-asm)], simp-all)
      qed
    }
    thus ?case by simp
  next
    case (Methd s0 D sig v s1 jmps T Env)
    from ‹G⊢Norm s0 −body G D sig−≻v→ s1›
    have G⊢Norm s0 −Methd D sig−≻v→ s1
      by (rule eval.Methd)
    hence ⋀ j. abrupt s1 ≠ Some (Jump j)
      by (rule Methd-no-jump) simp
    thus ?case by simp
  next
    case (Body s0 D s1 c s2 s3 jmps T Env)
    have G⊢Norm s0 −Body D c−≻the (locals (store s2) Result)
           → abupd (absorb Ret) s3
      by (rule eval.Body) (rule Body)+
    hence ⋀ j. abrupt (abupd (absorb Ret) s3) ≠ Some (Jump j)
      by (rule Body-no-jump) simp
    thus ?case by simp
  next
```

```
    case LVar
    thus ?case by (simp add: lvar-def Let-def)
  next
    case (FVar s0 statDeclC s1 e a s2 v s2' stat fn s3 accC jmps T Env)
    note G = ‹prg Env = G›
    from wf FVar.prems
    obtain statC f where
      wt-e: Env⊢e::−Class statC and
      accfield: accfield (prg Env) accC statC fn = Some (statDeclC,f)
      by (elim wt-elim-cases) simp
    have wt-init: Env⊢Init statDeclC::√
    proof −
      from wf wt-e G
      have is-class (prg Env) statC
        by (auto dest: ty-expr-is-type type-is-class)
      with wf accfield G
      have is-class (prg Env) statDeclC
        by (auto dest!: accfield-fields dest: fields-declC)
      thus ?thesis
        by simp
    qed
    note fvar = ‹(v, s2') = fvar statDeclC stat fn a s2›
    {
      fix j
      assume jmp: abrupt s3 = Some (Jump j)
      have j∈jmps
      proof −
        note hyp-init = ‹PROP ?Hyp (In1r (Init statDeclC)) (Norm s0) s1 ◇›
        from G wt-init
        have ?Jmp jmps s1
          by − (rule hyp-init [THEN conjunct1],auto)
        moreover
        note hyp-e = ‹PROP ?Hyp (In1l e) s1 s2 (In1 a)›
        have abrupt s2 = Some (Jump j)
        proof −
          note ‹s3 = check-field-access G accC statDeclC fn stat a s2'›
          with jmp have abrupt s2' = Some (Jump j)
            by (cases s2')
               (simp add: check-field-access-def abrupt-if-def Let-def)
         with fvar show abrupt s2 =  Some (Jump j)
           by (cases s2) (simp add: fvar-def2 abrupt-if-def)
        qed
        ultimately show ?thesis
          using G wt-e
          by − (rule hyp-e [THEN conjunct1, rule-format (no-asm)],simp-all)
      qed
    }
    moreover
    from fvar obtain upd w
      where upd: upd = snd (fst (fvar statDeclC stat fn a s2)) and
              v: v=(w,upd)
      by (cases fvar statDeclC stat fn a s2)
        (simp, use surjective-pairing in blast)
    {
      fix j val fix s::state
      assume normal s3
      assume no-jmp: abrupt s ≠ Some (Jump j)
      with upd
      have abrupt (upd val s) ≠ Some (Jump j)
```

```
      by (rule fvar-upd-no-jump)
    }
    ultimately show ?case using v by simp
  next
    case (AVar s0 e1 a s1 e2 i s2 v s2' jmps T Env)
    note G = ‹prg Env = G›
    from AVar.prems
    obtain e1T e2T where
      wt-e1: Env⊢e1::−e1T and wt-e2: Env⊢e2::−e2T
      by (elim wt-elim-cases) simp
    note avar = ‹(v, s2') = avar G i a s2›
    {
      fix j
      assume jmp: abrupt s2' = Some (Jump j)
      have j∈jmps
      proof −
        note hyp-e1 = ‹PROP ?Hyp (In1l e1) (Norm s0) s1 (In1 a)›
        from G wt-e1
        have ?Jmp jmps s1
          by − (rule hyp-e1 [THEN conjunct1], auto)
        moreover
        note hyp-e2 = ‹PROP ?Hyp (In1l e2) s1 s2 (In1 i)›
        have abrupt s2 = Some (Jump j)
        proof −
          from avar have s2' = snd (avar G i a s2)
            by (cases avar G i a s2) simp
          with jmp show ?thesis by − (rule avar-state-no-jump,simp)
        qed
        ultimately show ?thesis
          using wt-e2 G
          by − (rule hyp-e2 [THEN conjunct1, rule-format (no-asm)],simp-all)
      qed
    }
    moreover
    from avar obtain upd w
      where upd: upd = snd (fst (avar G i a s2)) and
            v: v=(w,upd)
      by (cases avar G i a s2)
        (simp, use surjective-pairing in blast)
    {
      fix j val fix s::state
      assume normal s2'
      assume no-jmp: abrupt s ≠ Some (Jump j)
      with upd
      have abrupt (upd val s) ≠ Some (Jump j)
        by (rule avar-upd-no-jump)
    }
    ultimately show ?case using v by simp
  next
    case Nil thus ?case by simp
  next
    case (Cons s0 e v s1 es vs s2 jmps T Env)
    note G = ‹prg Env = G›
    from Cons.prems obtain eT esT
      where wt-e: Env⊢e::−eT and wt-e2: Env⊢es::≐esT
      by (elim wt-elim-cases) simp
    {
      fix j
      assume jmp: abrupt s2 = Some (Jump j)
```

```
      have j∈jmps
      proof −
        note hyp-e = ‹PROP ?Hyp (In1l e) (Norm s0) s1 (In1 v)›
        from G wt-e
        have ?Jmp jmps s1
          by − (rule hyp-e [THEN conjunct1],simp-all)
        moreover
        note hyp-es = ‹PROP ?Hyp (In3 es) s1 s2 (In3 vs)›
        ultimately show ?thesis
          using wt-e2 G jmp
          by − (rule hyp-es [THEN conjunct1, rule-format (no-asm)],
                (assumption|simp (no-asm-simp))+)
      qed
    }
    thus ?case by simp
  qed
  note generalized = this
  from no-jmp jmpOk wt G
  show ?thesis
    by (rule generalized)
qed

lemmas jumpNestingOk-evalE = jumpNestingOk-eval [THEN conjE,rule-format]



lemma jumpNestingOk-eval-no-jump:
 assumes    eval: prg Env⊢ s0 −t≻→ (v,s1) and
           jmpOk: jumpNestingOk {} t and
          no-jmp: abrupt s0 ≠ Some (Jump j) and
              wt: Env⊢t::T and
              wf: wf-prog (prg Env)
 shows abrupt s1 ≠ Some (Jump j) ∧
        (normal s1 ⟶ v=In2 (w,upd)
         ⟶ abrupt s ≠ Some (Jump j′)
         ⟶ abrupt (upd val s) ≠ Some (Jump j′))
proof (cases ∃ j′. abrupt s0 = Some (Jump j′))
  case True
  then obtain j′ where jmp: abrupt s0 = Some (Jump j′) ..
  with no-jmp have j′≠j by simp
  with eval jmp have s1=s0 by auto
  with no-jmp jmp show ?thesis by simp
next
  case False
  obtain G where G: prg Env = G
    by (cases Env) simp
  from G eval have G⊢ s0 −t≻→ (v,s1) by simp
  moreover note jmpOk wt
  moreover from G wf have wf-prog G by simp
  moreover note G
  moreover from False have ⋀ j. abrupt s0 = Some (Jump j) ⟹ j ∈ {}
    by simp
  ultimately show ?thesis
    apply (rule jumpNestingOk-evalE)
    apply assumption
    apply simp
    apply fastforce
    done
qed
```

lemmas *jumpNestingOk-eval-no-jumpE*
= *jumpNestingOk-eval-no-jump* [*THEN conjE*,*rule-format*]

corollary *eval-expression-no-jump*:
assumes *eval*: *prg Env*⊢*s0* −*e*−≻*v*→ *s1* **and**
no-jmp: *abrupt s0* ≠ *Some* (*Jump j*) **and**
wt: *Env*⊢*e*::−*T* **and**
wf: *wf-prog* (*prg Env*)
shows *abrupt s1* ≠ *Some* (*Jump j*)
using *eval* - *no-jmp wt wf*
by (*rule jumpNestingOk-eval-no-jumpE*, *simp-all*)

corollary *eval-var-no-jump*:
assumes *eval*: *prg Env*⊢*s0* −*var*=≻(*w*,*upd*)→ *s1* **and**
no-jmp: *abrupt s0* ≠ *Some* (*Jump j*) **and**
wt: *Env*⊢*var*::=*T* **and**
wf: *wf-prog* (*prg Env*)
shows *abrupt s1* ≠ *Some* (*Jump j*) ∧
(*normal s1* ⟶
(*abrupt s* ≠ *Some* (*Jump j*′)
⟶ *abrupt* (*upd val s*) ≠ *Some* (*Jump j*′)))
apply (*rule-tac upd*=*upd* **and** *val*=*val* **and** *s*=*s* **and** *w*=*w* **and** *j*′=*j*′
in *jumpNestingOk-eval-no-jumpE* [*OF eval* - *no-jmp wt wf*])
by *simp-all*

lemmas *eval-var-no-jumpE* = *eval-var-no-jump* [*THEN conjE*,*rule-format*]

corollary *eval-statement-no-jump*:
assumes *eval*: *prg Env*⊢*s0* −*c*→ *s1* **and**
jmpOk: *jumpNestingOkS* {} *c* **and**
no-jmp: *abrupt s0* ≠ *Some* (*Jump j*) **and**
wt: *Env*⊢*c*::√ **and**
wf: *wf-prog* (*prg Env*)
shows *abrupt s1* ≠ *Some* (*Jump j*)
using *eval* - *no-jmp wt wf*
by (*rule jumpNestingOk-eval-no-jumpE*) (*simp-all add*: *jmpOk*)

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*)
using *eval* - *no-jmp wt wf*
by (*rule jumpNestingOk-eval-no-jumpE*, *simp-all*)

lemma *union-subseteq-elim* [*elim*]: ⟦*A* ∪ *B* ⊆ *C*; ⟦*A* ⊆ *C*; *B* ⊆ *C*⟧ ⟹ *P*⟧ ⟹ *P*
by *blast*

lemma *dom-locals-halloc-mono*:
assumes *halloc*: *G*⊢*s0*−*halloc oi*≻*a*→*s1*
shows *dom* (*locals* (*store s0*)) ⊆ *dom* (*locals* (*store s1*))
proof −
from *halloc* **show** *?thesis*
by *cases simp-all*

```
qed

lemma dom-locals-sxalloc-mono:
  assumes sxalloc: G⊢s0 −sxalloc→ s1
  shows dom (locals (store s0)) ⊆ dom (locals (store s1))
proof −
  from sxalloc show ?thesis
  proof (cases)
    case Norm thus ?thesis by simp
  next
    case Jmp thus ?thesis by simp
  next
    case Error thus ?thesis by simp
  next
    case XcptL thus ?thesis by simp
  next
    case SXcpt thus ?thesis
      by − (drule dom-locals-halloc-mono,simp)
  qed
qed

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 (cases normal s)
  case False thus ?thesis
    by (cases s) (auto simp add: assign-def Let-def)
next
  case True
  then obtain s' where s': s = (None,s')
    by auto
  moreover
  obtain x1 s1 where f n s = (x1,s1)
    by (cases f n s)
  ultimately
  show ?thesis
    using f-ok
    by (simp add: assign-def Let-def)
qed

lemma dom-locals-lvar-mono:
 dom (locals (store s)) ⊆ dom (locals (store (snd (lvar vn s') val s)))
by (simp add: lvar-def) blast

lemma dom-locals-fvar-vvar-mono:
dom (locals (store s))
  ⊆ dom (locals (store (snd (fst (fvar statDeclC stat fn a s')) val s)))
proof (cases stat)
  case True
  thus ?thesis
    by (cases s) (simp add: fvar-def2)
```

next
  case *False*
  thus *?thesis*
    by (*cases s*) (*simp add*: *fvar-def2*)
qed

lemma *dom-locals-fvar-mono*:
dom (*locals* (*store s*))
  $\subseteq$ *dom* (*locals* (*store* (*snd* (*fvar statDeclC stat fn a s*))))
proof (*cases stat*)
  case *True*
  thus *?thesis*
    by (*cases s*) (*simp add*: *fvar-def2*)
next
  case *False*
  thus *?thesis*
    by (*cases s*) (*simp add*: *fvar-def2*)
qed

lemma *dom-locals-avar-vvar-mono*:
dom (*locals* (*store s*))
  $\subseteq$ *dom* (*locals* (*store* (*snd* (*fst* (*avar G i a s'*)) *val s*)))
by (*cases s*, *simp add*: *avar-def2*)

lemma *dom-locals-avar-mono*:
dom (*locals* (*store s*))
  $\subseteq$ *dom* (*locals* (*store* (*snd* (*avar G i a s*))))
by (*cases s*, *simp add*: *avar-def2*)

Since assignments are modelled as functions from states to states, we must take into account these functions. They appear only in the assignment rule and as result from evaluating a variable. Thats why we need the complicated second part of the conjunction in the goal. The reason for the very generic way to treat assignments was the aim to omit redundancy. There is only one evaluation rule for each kind of variable (locals, fields, arrays). These rules are used for both accessing variables and updating variables. Thats why the evaluation rules for variables result in a pair consisting of a value and an update function. Of course we could also think of a pair of a value and a reference in the store, instead of the generic update function. But as only array updates can cause a special exception (if the types mismatch) and not array reads we then have to introduce two different rules to handle array reads and updates

lemma *dom-locals-eval-mono*:
  assumes *eval*: $G\vdash s0 -t\succ\rightarrow (v,s1)$
  shows *dom* (*locals* (*store s0*)) $\subseteq$ *dom* (*locals* (*store s1*)) $\wedge$
      ($\forall$ *vv*. *v*=*In2 vv* $\wedge$ *normal s1*
        $\longrightarrow$ ($\forall$ *s val*. *dom* (*locals* (*store s*))
              $\subseteq$ *dom* (*locals* (*store* ((*snd vv*) *val s*)))))
proof −
  from *eval* **show** *?thesis*
  proof (*induct*)
    case *Abrupt* **thus** *?case* **by** *simp*
  next
    case *Skip* **thus** *?case* **by** *simp*
  next
    case *Expr* **thus** *?case* **by** *simp*
  next

```
    case Lab thus ?case by simp
  next
    case (Comp s0 c1 s1 c2 s2)
    from Comp.hyps
    have dom (locals (store ((Norm s0)::state))) ⊆ dom (locals (store s1))
      by simp
    also
    from Comp.hyps
    have ... ⊆ dom (locals (store s2))
      by simp
    finally show ?case by simp
  next
    case (If s0 e b s1 c1 c2 s2)
    from If.hyps
    have dom (locals (store ((Norm s0)::state))) ⊆ dom (locals (store s1))
      by simp
    also
    from If.hyps
    have ... ⊆ dom (locals (store s2))
      by simp
    finally show ?case by simp
  next
    case (Loop s0 e b s1 c s2 l s3)
    show ?case
    proof (cases the-Bool b)
      case True
      with Loop.hyps
      obtain
        s0-s1:
        dom (locals (store ((Norm s0)::state))) ⊆ dom (locals (store s1)) and
        s1-s2: dom (locals (store s1)) ⊆ dom (locals (store s2)) and
        s2-s3: dom (locals (store s2)) ⊆ dom (locals (store s3))
        by simp
      note s0-s1 also note s1-s2 also note s2-s3
      finally show ?thesis
        by simp
    next
      case False
      with Loop.hyps show ?thesis
        by simp
    qed
  next
    case Jmp thus ?case by simp
  next
    case Throw thus ?case by simp
  next
    case (Try s0 c1 s1 s2 C vn c2 s3)
    then
    have s0-s1: dom (locals (store ((Norm s0)::state)))
                  ⊆ dom (locals (store s1)) by simp
    from ‹G⊢s1 −sxalloc→ s2›
    have s1-s2: dom (locals (store s1)) ⊆ dom (locals (store s2))
      by (rule dom-locals-sxalloc-mono)
    thus ?case
    proof (cases G,s2⊢catch C)
      case True
      note s0-s1 also note s1-s2
      also
      from True Try.hyps
```

have *dom* (*locals* (*store* (*new-xcpt-var vn s2*)))
⊆ *dom* (*locals* (*store s3*))
by *simp*
hence *dom* (*locals* (*store s2*)) ⊆ *dom* (*locals* (*store s3*))
by (*cases s2*, *simp*)
finally show *?thesis* **by** *simp*
next
case *False*
note *s0-s1* **also note** *s1-s2*
finally
show *?thesis*
using *False Try.hyps* **by** *simp*
qed
next
case (*Fin s0 c1 x1 s1 c2 s2 s3*)
show *?case*
proof (*cases* ∃ *err*. *x1* = *Some* (*Error err*))
case *True*
with *Fin.hyps* **show** *?thesis*
by *simp*
next
case *False*
from *Fin.hyps*
have *dom* (*locals* (*store* ((*Norm s0*)::*state*)))
⊆ *dom* (*locals* (*store* (*x1*, *s1*)))
by *simp*
hence *dom* (*locals* (*store* ((*Norm s0*)::*state*)))
⊆ *dom* (*locals* (*store* ((*Norm s1*)::*state*)))
by *simp*
also
from *Fin.hyps*
have . . . ⊆ *dom* (*locals* (*store s2*))
by *simp*
finally show *?thesis*
using *Fin.hyps* **by** *simp*
qed
next
case (*Init C c s0 s3 s1 s2*)
show *?case*
proof (*cases inited C* (*globs s0*))
case *True*
with *Init.hyps* **show** *?thesis* **by** *simp*
next
case *False*
with *Init.hyps*
obtain *s0-s1*: *dom* (*locals* (*store* (*Norm* ((*init-class-obj G C*) *s0*))))
⊆ *dom* (*locals* (*store s1*)) **and**
s3: *s3* = (*set-lvars* (*locals* (*snd s1*))) *s2*
by *simp*
from *s0-s1*
have *dom* (*locals* (*store* (*Norm s0*))) ⊆ *dom* (*locals* (*store s1*))
by (*cases s0*) *simp*
with *s3*
have *dom* (*locals* (*store* (*Norm s0*))) ⊆ *dom* (*locals* (*store s3*))
by (*cases s2*) *simp*
thus *?thesis* **by** *simp*
qed
next
case (*NewC s0 C s1 a s2*)

```
    note halloc = ‹G⊢s1 −halloc CInst C≻a→ s2›
    from NewC.hyps
    have dom (locals (store ((Norm s0)::state))) ⊆ dom (locals (store s1))
      by simp
    also
    from halloc
    have ... ⊆ dom (locals (store s2)) by (rule dom-locals-halloc-mono)
    finally show ?case by simp
  next
    case (NewA s0 T s1 e i s2 a s3)
    note halloc = ‹G⊢abupd (check-neg i) s2 −halloc Arr T (the-Intg i)≻a→ s3›
    from NewA.hyps
    have dom (locals (store ((Norm s0)::state))) ⊆ dom (locals (store s1))
      by simp
    also
    from NewA.hyps
    have ... ⊆ dom (locals (store s2)) by simp
    also
    from halloc
    have ... ⊆ dom (locals (store s3))
      by (rule dom-locals-halloc-mono [elim-format]) simp
    finally show ?case by simp
  next
    case Cast thus ?case by simp
  next
    case Inst thus ?case by simp
  next
    case Lit thus ?case by simp
  next
    case UnOp thus ?case by simp
  next
    case (BinOp s0 e1 v1 s1 binop e2 v2 s2)
    from BinOp.hyps
    have dom (locals (store ((Norm s0)::state))) ⊆ dom (locals (store s1))
      by simp
    also
    from BinOp.hyps
    have ... ⊆ dom (locals (store s2)) by simp
    finally show ?case by simp
  next
    case Super thus ?case by simp
  next
    case Acc thus ?case by simp
  next
    case (Ass s0 va w f s1 e v s2)
    from Ass.hyps
    have s0-s1:
      dom (locals (store ((Norm s0)::state))) ⊆ dom (locals (store s1))
      by simp
    show ?case
    proof (cases normal s1)
      case True
      with Ass.hyps
      have ass-ok:
        ⋀ s val. dom (locals (store s)) ⊆ dom (locals (store (f val s)))
        by simp
      note s0-s1
      also
      from Ass.hyps
```

have *dom* (*locals* (*store s1*)) ⊆ *dom* (*locals* (*store s2*))
by *simp*
also
from *ass-ok*
have ... ⊆ *dom* (*locals* (*store* (*assign f v s2*)))
by (*rule dom-locals-assign-mono* [**where** *f* = *f*])
finally show *?thesis* **by** *simp*
next
case *False*
with ‹*G*⊢*s1* −*e*−≻*v*→ *s2*›
have *s2*=*s1*
by *auto*
with *s0-s1 False*
have *dom* (*locals* (*store* ((*Norm s0*)::*state*)))
⊆ *dom* (*locals* (*store* (*assign f v s2*)))
by *simp*
thus *?thesis*
by *simp*
qed
next
case (*Cond s0 e0 b s1 e1 e2 v s2*)
from *Cond.hyps*
have *dom* (*locals* (*store* ((*Norm s0*)::*state*))) ⊆ *dom* (*locals* (*store s1*))
by *simp*
also
from *Cond.hyps*
have ... ⊆ *dom* (*locals* (*store s2*))
by *simp*
finally show *?case* **by** *simp*
next
case (*Call s0 e a′ s1 args vs s2 D mode statT mn pTs s3 s3′ accC v s4*)
note *s3* = ‹*s3* = *init-lvars G D* (|*name* = *mn*, *parTs* = *pTs*|) *mode a′ vs s2*›
from *Call.hyps*
have *dom* (*locals* (*store* ((*Norm s0*)::*state*))) ⊆ *dom* (*locals* (*store s1*))
by *simp*
also
from *Call.hyps*
have ... ⊆ *dom* (*locals* (*store s2*))
by *simp*
also
have ... ⊆ *dom* (*locals* (*store* ((*set-lvars* (*locals* (*store s2*))) *s4*)))
by (*cases s4*) *simp*
finally show *?case* **by** *simp*
next
case *Methd* **thus** *?case* **by** *simp*
next
case (*Body s0 D s1 c s2 s3*)
from *Body.hyps*
have *dom* (*locals* (*store* ((*Norm s0*)::*state*))) ⊆ *dom* (*locals* (*store s1*))
by *simp*
also
from *Body.hyps*
have ... ⊆ *dom* (*locals* (*store s2*))
by *simp*
also
have ... ⊆ *dom* (*locals* (*store* (*abupd* (*absorb Ret*) *s2*)))
by *simp*
also
have ... ⊆ *dom* (*locals* (*store* (*abupd* (*absorb Ret*) *s3*)))

```
    proof -
      from ‹s3 =
         (if ∃ l. abrupt s2 = Some (Jump (Break l)) ∨
                 abrupt s2 = Some (Jump (Cont l))
             then abupd (λx. Some (Error CrossMethodJump)) s2 else s2)›
      show ?thesis
        by simp
    qed
    finally show ?case by simp
  next
    case LVar
    thus ?case
      using dom-locals-lvar-mono
      by simp
  next
    case (FVar s0 statDeclC s1 e a s2 v s2' stat fn s3 accC)
    from FVar.hyps
    obtain s2': s2' = snd (fvar statDeclC stat fn a s2) and
             v: v = fst (fvar statDeclC stat fn a s2)
      by (cases fvar statDeclC stat fn a s2 ) simp
    from v
    have ∀ s val. dom (locals (store s))
                          ⊆ dom (locals (store (snd v val s))) (is ?V-ok)
      by (simp add: dom-locals-fvar-vvar-mono)
    hence v-ok: (∀ vv. In2 v = In2 vv ∧ normal s3 ⟶ ?V-ok)
      by - (intro strip, simp)
    note s3 = ‹s3 = check-field-access G accC statDeclC fn stat a s2'›
    from FVar.hyps
    have dom (locals (store ((Norm s0)::state))) ⊆ dom (locals (store s1))
      by simp
    also
    from FVar.hyps
    have ... ⊆ dom (locals (store s2))
      by simp
    also
    from s2'
    have ... ⊆ dom (locals (store s2'))
      by (simp add: dom-locals-fvar-mono)
    also
    from s3
    have ... ⊆ dom (locals (store s3))
      by (simp add: check-field-access-def Let-def)
    finally
    show ?case
      using v-ok
      by simp
  next
    case (AVar s0 e1 a s1 e2 i s2 v s2')
    from AVar.hyps
    obtain s2': s2' = snd (avar G i a s2) and
             v: v   = fst (avar G i a s2)
      by (cases avar G i a s2) simp
    from v
    have ∀ s val. dom (locals (store s))
                          ⊆ dom (locals (store (snd v val s))) (is ?V-ok)
      by (simp add: dom-locals-avar-vvar-mono)
    hence v-ok: (∀ vv. In2 v = In2 vv ∧ normal s2' ⟶ ?V-ok)
      by - (intro strip, simp)
    from AVar.hyps
```

```
    have dom (locals (store ((Norm s0)::state))) ⊆ dom (locals (store s1))
      by simp
    also
    from AVar.hyps
    have ... ⊆ dom (locals (store s2))
      by simp
    also
    from s2'
    have ... ⊆ dom (locals (store s2'))
      by (simp add: dom-locals-avar-mono)
    finally
    show ?case using v-ok by simp
  next
    case Nil thus ?case by simp
  next
    case (Cons s0 e v s1 es vs s2)
    from Cons.hyps
    have dom (locals (store ((Norm s0)::state))) ⊆ dom (locals (store s1))
      by simp
    also
    from Cons.hyps
    have ... ⊆ dom (locals (store s2))
      by simp
    finally show ?case by simp
  qed
qed


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)))
  using eval by (rule dom-locals-eval-mono [THEN conjE]) (rule that, auto)


lemma halloc-no-abrupt:
  assumes halloc: G⊢s0−halloc oi≻a→s1 and
          normal: normal s1
  shows normal s0
proof −
  from halloc normal show ?thesis
    by cases simp-all
qed


lemma sxalloc-mono-no-abrupt:
  assumes sxalloc: G⊢s0−sxalloc→s1 and
           normal: normal s1
  shows normal s0
proof −
  from sxalloc normal show ?thesis
    by cases simp-all
qed


lemma union-subseteqI: ⟦A ∪ B ⊆ C; A' ⊆ A; B' ⊆ B⟧ ⟹ A' ∪ B' ⊆ C
  by blast
```

lemma *union-subseteqIl*: $[\![A \cup B \subseteq C;\ A' \subseteq A]\!] \implies A' \cup B \subseteq C$
by *blast*

lemma *union-subseteqIr*: $[\![A \cup B \subseteq C;\ B' \subseteq B]\!] \implies A \cup B' \subseteq C$
by *blast*

lemma *subseteq-union-transl* [*trans*]: $[\![A \subseteq B;\ B \cup C \subseteq D]\!] \implies A \cup C \subseteq D$
by *blast*

lemma *subseteq-union-transr* [*trans*]: $[\![A \subseteq B;\ C \cup B \subseteq D]\!] \implies A \cup C \subseteq D$
by *blast*

lemma *union-subseteq-weaken*: $[\![A \cup B \subseteq C;\ [\![A \subseteq C;\ B \subseteq C]\!] \implies P\ ]\!] \implies P$
by *blast*

```
lemma assigns-good-approx:
  assumes
      eval: G⊢ s0 −t≻→ (v,s1) and
    normal: normal s1
  shows assigns t ⊆ dom (locals (store s1))
proof −
  from eval normal show ?thesis
  proof (induct)
    case Abrupt thus ?case by simp
  next — For statements its trivial, since then assigns t = {}
    case Skip show ?case by simp
  next
    case Expr show ?case by simp
  next
    case Lab show ?case by simp
  next
    case Comp show ?case by simp
  next
    case If show ?case by simp
  next
    case Loop show ?case by simp
  next
    case Jmp show ?case by simp
  next
    case Throw show ?case by simp
  next
    case Try show ?case by simp
  next
    case Fin show ?case by simp
  next
    case Init show ?case by simp
  next
    case NewC show ?case by simp
  next
    case (NewA s0 T s1 e i s2 a s3)
    note halloc = ‹G⊢abupd (check-neg i) s2 −halloc Arr T (the-Intg i)≻a→ s3›
    have assigns (In1l e) ⊆ dom (locals (store s2))
```

```
    proof -
      from NewA
      have normal (abupd (check-neg i) s2)
        by - (erule halloc-no-abrupt [rule-format])
      hence normal s2 by (cases s2) simp
      with NewA.hyps
      show ?thesis by iprover
    qed
    also
    from halloc
    have ... ⊆  dom (locals (store s3))
      by (rule dom-locals-halloc-mono [elim-format]) simp
    finally show ?case by simp
  next
    case (Cast s0 e v s1 s2 T)
    hence normal s1 by (cases s1,simp)
    with Cast.hyps
    have assigns (In1l e) ⊆ dom (locals (store s1))
      by simp
    also
    from Cast.hyps
    have ... ⊆ dom (locals (store s2))
      by simp
    finally
    show ?case
      by simp
  next
    case Inst thus ?case by simp
  next
    case Lit thus ?case by simp
  next
    case UnOp thus ?case by simp
  next
    case (BinOp s0 e1 v1 s1 binop e2 v2 s2)
    hence normal s1 by - (erule eval-no-abrupt-lemma [rule-format])
    with BinOp.hyps
    have assigns (In1l e1) ⊆ dom (locals (store s1))
      by iprover
    also
    have ...  ⊆ dom (locals (store s2))
    proof -
      note ‹G⊢s1 −(if need-second-arg binop v1 then In1l e2
                    else In1r Skip)≻→ (In1 v2, s2)›
      thus ?thesis
        by (rule dom-locals-eval-mono-elim)
    qed
    finally have s2: assigns (In1l e1) ⊆ dom (locals (store s2)) .
    show ?case
    proof (cases binop=CondAnd ∨ binop=CondOr)
      case True
      with s2 show ?thesis by simp
    next
      case False
      with BinOp
      have assigns (In1l e2) ⊆ dom (locals (store s2))
        by (simp add: need-second-arg-def)
      with s2
      show ?thesis using False by simp
    qed
```

```
next
  case Super thus ?case by simp
next
  case Acc thus ?case by simp
next
  case (Ass s0 va w f s1 e v s2)
  note nrm-ass-s2 = ‹normal (assign f v s2)›
  hence nrm-s2: normal s2
    by (cases s2, simp add: assign-def Let-def)
  with Ass.hyps
  have nrm-s1: normal s1
    by − (erule eval-no-abrupt-lemma [rule-format])
  with Ass.hyps
  have assigns (In2 va) ⊆ dom (locals (store s1))
    by iprover
  also
  from Ass.hyps
  have ... ⊆ dom (locals (store s2))
    by − (erule dom-locals-eval-mono-elim)
  also
  from nrm-s2 Ass.hyps
  have assigns (In1l e) ⊆ dom (locals (store s2))
    by iprover
  ultimately
  have assigns (In2 va) ∪ assigns (In1l e) ⊆  dom (locals (store s2))
    by (rule Un-least)
  also
  from Ass.hyps nrm-s1
  have ...  ⊆ dom (locals (store (f v s2)))
    by − (erule dom-locals-eval-mono-elim, cases s2,simp)
  then
  have dom (locals (store s2))  ⊆ dom (locals (store (assign f v s2)))
    by (rule dom-locals-assign-mono)
  finally
  have va-e:  assigns (In2 va) ∪ assigns (In1l e)
              ⊆ dom (locals (snd (assign f v s2))) .
  show ?case
  proof (cases ∃ n. va = LVar n)
    case False
    with va-e show ?thesis
      by (simp add: Un-assoc)
  next
    case True
    then obtain n where va: va = LVar n
      by blast
    with Ass.hyps
    have G⊢Norm s0 −LVar n=≻(w,f)→ s1
      by simp
    hence (w,f) = lvar n s0
      by (rule eval-elim-cases) simp
    with nrm-ass-s2
    have n ∈ dom (locals (store (assign f v s2)))
      by (cases s2) (simp add: assign-def Let-def lvar-def)
    with va-e True va
    show ?thesis by (simp add: Un-assoc)
  qed
next
  case (Cond s0 e0 b s1 e1 e2 v s2)
  hence normal s1
```

by − (*erule eval-no-abrupt-lemma* [*rule-format*])
with *Cond.hyps*
have *assigns* (*In1l e0*) ⊆ *dom* (*locals* (*store s1*))
by *iprover*
also from *Cond.hyps*
have . . . ⊆ *dom* (*locals* (*store s2*))
by − (*erule dom-locals-eval-mono-elim*)
finally have *e0*: *assigns* (*In1l e0*) ⊆ *dom* (*locals* (*store s2*)) **.**
show *?case*
proof (*cases the-Bool b*)
case *True*
with *Cond*
have *assigns* (*In1l e1*) ⊆ *dom* (*locals* (*store s2*))
by *simp*
hence *assigns* (*In1l e1*) ∩ *assigns* (*In1l e2*) ⊆ . . .
by *blast*
with *e0*
have *assigns* (*In1l e0*) ∪ *assigns* (*In1l e1*) ∩ *assigns* (*In1l e2*)
⊆ *dom* (*locals* (*store s2*))
by (*rule Un-least*)
thus *?thesis* **using** *True* **by** *simp*
next
case *False*
with *Cond*
have *assigns* (*In1l e2*) ⊆ *dom* (*locals* (*store s2*))
by *simp*
hence *assigns* (*In1l e1*) ∩ *assigns* (*In1l e2*) ⊆ . . .
by *blast*
with *e0*
have *assigns* (*In1l e0*) ∪ *assigns* (*In1l e1*) ∩ *assigns* (*In1l e2*)
⊆ *dom* (*locals* (*store s2*))
by (*rule Un-least*)
thus *?thesis* **using** *False* **by** *simp*
qed
next
case (*Call s0 e a′ s1 args vs s2 D mode statT mn pTs s3 s3′ accC v s4*)
have *nrm-s2*: *normal s2*
proof −
from ‹*normal* ((*set-lvars* (*locals* (*snd s2*))) *s4*)›
have *normal-s4*: *normal s4* **by** *simp*
hence *normal s3′* **using** *Call.hyps*
by − (*erule eval-no-abrupt-lemma* [*rule-format*])
moreover note
‹*s3′* = *check-method-access G accC statT mode* (|*name*=*mn*, *parTs*=*pTs*|) *a′ s3*›
ultimately have *normal s3*
by (*cases s3*) (*simp add*: *check-method-access-def Let-def*)
moreover
note *s3* = ‹*s3* = *init-lvars G D* (|*name* = *mn*, *parTs* = *pTs*|) *mode a′ vs s2*›
ultimately show *normal s2*
by (*cases s2*) (*simp add*: *init-lvars-def2*)
qed
hence *normal s1* **using** *Call.hyps*
by − (*erule eval-no-abrupt-lemma* [*rule-format*])
with *Call.hyps*
have *assigns* (*In1l e*) ⊆ *dom* (*locals* (*store s1*))
by *iprover*
also from *Call.hyps*
have . . . ⊆ *dom* (*locals* (*store s2*))
by − (*erule dom-locals-eval-mono-elim*)

```
    also
    from nrm-s2 Call.hyps
    have assigns (In3 args) ⊆ dom (locals (store s2))
      by iprover
    ultimately have assigns (In1l e) ∪ assigns (In3 args) ⊆ ...
      by (rule Un-least)
    also
    have ... ⊆ dom (locals (store ((set-lvars (locals (store s2))) s4)))
      by (cases s4) simp
    finally show ?case
      by simp
  next
    case Methd thus ?case by simp
  next
    case Body thus ?case by simp
  next
    case LVar thus ?case by simp
  next
    case (FVar s0 statDeclC s1 e a s2 v s2' stat fn s3 accC)
    note s3 = ‹s3 = check-field-access G accC statDeclC fn stat a s2'›
    note avar = ‹(v, s2') = fvar statDeclC stat fn a s2›
    have nrm-s2: normal s2
    proof −
      note ‹normal s3›
      with s3 have normal s2'
        by (cases s2') (simp add: check-field-access-def Let-def)
      with avar show normal s2
        by (cases s2) (simp add: fvar-def2)
    qed
    with FVar.hyps
    have assigns (In1l e) ⊆ dom (locals (store s2))
      by iprover
    also
    have ... ⊆ dom (locals (store s2'))
    proof −
      from avar
      have s2' = snd (fvar statDeclC stat fn a s2)
        by (cases fvar statDeclC stat fn a s2) simp
      thus ?thesis
        by simp (rule dom-locals-fvar-mono)
    qed
    also from s3
    have ... ⊆ dom (locals (store s3))
      by (cases s2') (simp add: check-field-access-def Let-def)
    finally show ?case
      by simp
  next
    case (AVar s0 e1 a s1 e2 i s2 v s2')
    note avar = ‹(v, s2') = avar G i a s2›
    have nrm-s2: normal s2
    proof −
      from avar and ‹normal s2'›
      show ?thesis by (cases s2) (simp add: avar-def2)
    qed
    with AVar.hyps
    have normal s1
      by − (erule eval-no-abrupt-lemma [rule-format])
    with AVar.hyps
    have assigns (In1l e1) ⊆ dom (locals (store s1))
```

```
      by iprover
    also from AVar.hyps
    have ... ⊆  dom (locals (store s2))
      by − (erule dom-locals-eval-mono-elim)
    also
    from AVar.hyps nrm-s2
    have assigns (In1l e2) ⊆ dom (locals (store s2))
      by iprover
    ultimately
    have assigns (In1l e1) ∪ assigns (In1l e2) ⊆ ...
      by (rule Un-least)
    also
    have dom (locals (store s2)) ⊆  dom (locals (store s2'))
    proof −
      from avar have s2' = snd (avar G i a s2)
        by (cases avar G i a s2) simp
      thus ?thesis
        by simp (rule dom-locals-avar-mono)
    qed
    finally
    show ?case
      by simp
  next
    case Nil show ?case by simp
  next
    case (Cons s0 e v s1 es vs s2)
    have assigns (In1l e) ⊆ dom (locals (store s1))
    proof −
      from Cons
      have normal s1 by − (erule eval-no-abrupt-lemma [rule-format])
      with Cons.hyps show ?thesis by iprover
    qed
    also from Cons.hyps
    have ... ⊆  dom (locals (store s2))
      by − (erule dom-locals-eval-mono-elim)
    also from Cons
    have assigns (In3 es) ⊆ dom (locals (store s2))
      by iprover
    ultimately
    have assigns (In1l e) ∪ assigns (In3 es) ⊆ dom (locals (store s2))
      by (rule Un-least)
    thus ?case
      by simp
  qed
qed

corollary assignsE-good-approx:
  assumes
      eval: prg Env⊢ s0 −e−≻v→ s1 and
    normal: normal s1
  shows assignsE e ⊆ dom (locals (store s1))
proof −
from eval normal show ?thesis
  by (rule assigns-good-approx [elim-format]) simp
qed

corollary assignsV-good-approx:
  assumes
     eval: prg Env⊢ s0 −v=≻vf→ s1 and
```

```
    normal: normal s1
  shows assignsV v ⊆ dom (locals (store s1))
proof -
from eval normal show ?thesis
  by (rule assigns-good-approx [elim-format]) simp
qed

corollary assignsEs-good-approx:
  assumes
     eval: prg Env⊢ s0 −es≐≻vs→ s1 and
    normal: normal s1
  shows assignsEs es ⊆ dom (locals (store s1))
proof -
from eval normal show ?thesis
  by (rule assigns-good-approx [elim-format]) simp
qed


lemma constVal-eval:
 assumes const: constVal e = Some c and
         eval: G⊢Norm s0 −e−≻v→ s
  shows v = c ∧ normal s
proof -
  have  True and
       ⋀ c v s0 s. ⟦ constVal e = Some c; G⊢Norm s0 −e−≻v→ s⟧
                 ⟹ v = c ∧ normal s
       and True
  proof (induct rule: var.induct expr.induct stmt.induct)
    case NewC hence False by simp thus ?case ..
  next
    case NewA hence False by simp thus ?case ..
  next
    case Cast hence False by simp thus ?case ..
  next
    case Inst hence False by simp thus ?case ..
  next
    case (Lit val c v s0 s)
    note ‹constVal (Lit val) = Some c›
    moreover
    from ‹G⊢Norm s0 −Lit val−≻v→ s›
    obtain v=val and normal s
      by cases simp
    ultimately show v=c ∧ normal s by simp
  next
    case (UnOp unop e c v s0 s)
    note const = ‹constVal (UnOp unop e) = Some c›
    then obtain ce where ce: constVal e = Some ce by simp
    from ‹G⊢Norm s0 −UnOp unop e−≻v→ s›
    obtain ve where ve: G⊢Norm s0 −e−≻ve→ s and
                  v: v = eval-unop unop ve
      by cases simp
    from ce ve
    obtain eq-ve-ce: ve=ce and nrm-s: normal s
      by (rule UnOp.hyps [elim-format]) iprover
    from eq-ve-ce const ce v
    have v=c
      by simp
    from this nrm-s
    show ?case ..
```

```
next
  case (BinOp binop e1 e2 c v s0 s)
  note const = ‹constVal (BinOp binop e1 e2) = Some c›
  then obtain c1 c2 where c1: constVal e1 = Some c1 and
                          c2: constVal e2 = Some c2 and
                           c: c = eval-binop binop c1 c2
    by simp
  from ‹G⊢Norm s0 −BinOp binop e1 e2−≻v→ s›
  obtain v1 s1 v2
    where v1: G⊢Norm s0 −e1−≻v1→ s1 and
          v2: G⊢s1 −(if need-second-arg binop v1 then In1l e2
                       else In1r Skip)≻→ (In1 v2, s) and
           v: v = eval-binop binop v1 v2
    by cases simp
  from c1 v1
  obtain eq-v1-c1: v1 = c1 and
          nrm-s1: normal s1
    by (rule BinOp.hyps [elim-format]) iprover
  show ?case
  proof (cases need-second-arg binop v1)
    case True
    with v2 nrm-s1  obtain s1'
      where G⊢Norm s1' −e2−≻v2→ s
      by (cases s1) simp
    with c2 obtain v2 = c2 normal s
      by (rule BinOp.hyps [elim-format]) iprover
    with c c1 c2 eq-v1-c1 v
    show ?thesis by simp
  next
    case False
    with nrm-s1 v2
    have s=s1
      by (cases s1) (auto elim!: eval-elim-cases)
    moreover
    from False c v eq-v1-c1
    have v = c
      by (simp add: eval-binop-arg2-indep)
    ultimately
    show ?thesis
      using nrm-s1 by simp
  qed
next
  case Super hence False by simp thus ?case ..
next
  case Acc hence False by simp thus ?case ..
next
  case Ass hence False by simp thus ?case ..
next
  case (Cond b e1 e2 c v s0 s)
  note c = ‹constVal (b ? e1 : e2) = Some c›
  then obtain cb c1 c2 where
    cb: constVal b  = Some cb and
    c1: constVal e1 = Some c1 and
    c2: constVal e2 = Some c2
    by (auto split: bool.splits)
  from ‹G⊢Norm s0 −b ? e1 : e2−≻v→ s›
  obtain vb s1
    where     vb: G⊢Norm s0 −b−≻vb→ s1 and
          eval-v: G⊢s1 −(if the-Bool vb then e1 else e2)−≻v→ s
```

```
      by cases simp
    from cb vb
    obtain eq-vb-cb: vb = cb and nrm-s1: normal s1
      by (rule Cond.hyps [elim-format]) iprover
    show ?case
    proof (cases the-Bool vb)
      case True
      with c cb c1 eq-vb-cb
      have c = c1
        by simp
      moreover
      from True eval-v nrm-s1 obtain s1'
        where G⊢Norm s1' −e1−≻v→ s
        by (cases s1) simp
      with c1 obtain c1 = v normal s
        by (rule Cond.hyps [elim-format]) iprover
      ultimately show ?thesis by simp
    next
      case False
      with c cb c2 eq-vb-cb
      have c = c2
        by simp
      moreover
      from False eval-v nrm-s1 obtain s1'
        where G⊢Norm s1' −e2−≻v→ s
        by (cases s1) simp
      with c2 obtain c2 = v normal s
        by (rule Cond.hyps [elim-format]) iprover
      ultimately show ?thesis by simp
    qed
  next
    case Call hence False by simp thus ?case ..
  qed simp-all
  with const eval
  show ?thesis
    by iprover
qed

lemmas constVal-eval-elim = constVal-eval [THEN conjE]


lemma eval-unop-type:
  typeof dt (eval-unop unop v) = Some (PrimT (unop-type unop))
  by (cases unop) simp-all


lemma eval-binop-type:
  typeof dt (eval-binop binop v1 v2) = Some (PrimT (binop-type binop))
  by (cases binop) simp-all


lemma constVal-Boolean:
 assumes const: constVal e = Some c and
          wt: Env⊢e::−PrimT Boolean
  shows typeof empty-dt c = Some (PrimT Boolean)
proof −
  have True and
     ⋀ c. ⟦constVal e = Some c;Env⊢e::−PrimT Boolean⟧
           ⟹ typeof empty-dt c = Some (PrimT Boolean)
```

```
    and True
proof (induct rule: var.induct expr.induct stmt.induct)
  case NewC hence False by simp thus ?case ..
next
  case NewA hence False by simp thus ?case ..
next
  case Cast hence False by simp thus ?case ..
next
  case Inst hence False by simp thus ?case ..
next
  case (Lit v c)
  from ‹constVal (Lit v) = Some c›
  have c=v by simp
  moreover
  from ‹Env⊢Lit v::−PrimT Boolean›
  have typeof empty-dt v = Some (PrimT Boolean)
    by cases simp
  ultimately show ?case by simp
next
  case (UnOp unop e c)
  from ‹Env⊢UnOp unop e::−PrimT Boolean›
  have Boolean = unop-type unop by cases simp
  moreover
  from ‹constVal (UnOp unop e) = Some c›
  obtain ce where c = eval-unop unop ce by auto
  ultimately show ?case by (simp add: eval-unop-type)
next
  case (BinOp binop e1 e2 c)
  from ‹Env⊢BinOp binop e1 e2::−PrimT Boolean›
  have Boolean = binop-type binop by cases simp
  moreover
  from ‹constVal (BinOp binop e1 e2) = Some c›
  obtain c1 c2 where c = eval-binop binop c1 c2 by auto
  ultimately show ?case by (simp add: eval-binop-type)
next
  case Super hence False by simp thus ?case ..
next
  case Acc hence False by simp thus ?case ..
next
  case Ass hence False by simp thus ?case ..
next
  case (Cond b e1 e2 c)
  note c = ‹constVal (b ? e1 : e2) = Some c›
  then obtain cb c1 c2 where
    cb: constVal b  = Some cb and
    c1: constVal e1 = Some c1 and
    c2: constVal e2 = Some c2
    by (auto split: bool.splits)
  note wt = ‹Env⊢b ? e1 : e2::−PrimT Boolean›
  then
  obtain T1 T2
    where Env⊢b::−PrimT Boolean and
          wt-e1: Env⊢e1::−PrimT Boolean and
          wt-e2: Env⊢e2::−PrimT Boolean
    by cases (auto dest: widen-Boolean2)
  show ?case
  proof (cases the-Bool cb)
    case True
    from c1 wt-e1
```

have *typeof empty-dt c1* = *Some* (*PrimT Boolean*)
by (*rule Cond.hyps*)
with *True c cb c1* **show** *?thesis* **by** *simp*
next
case *False*
from *c2 wt-e2*
have *typeof empty-dt c2* = *Some* (*PrimT Boolean*)
by (*rule Cond.hyps*)
with *False c cb c2* **show** *?thesis* **by** *simp*
qed
next
case *Call* **hence** *False* **by** *simp* **thus** *?case* **..**
qed *simp-all*
with *const wt*
show *?thesis*
by *iprover*
qed

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 −
— To properly perform induction on the evaluation relation we have to generalize the lemma to terms not only expressions.
{ fix *t val*
assume *eval′*: *prg Env*⊢ *s0* −*t*≻→ (*val*,*s1*)
assume *bool′*: *Env*⊢ *t*::*Inl* (*PrimT Boolean*)
assume *expr*: ∃ *expr*. *t*=*In1l expr*
have *assigns-if* (*the-Bool* (*the-In1 val*)) (*the-In1l t*)
⊆ *dom* (*locals* (*store s1*))
using *eval′ normal bool′ expr*
proof (*induct*)
case *Abrupt* **thus** *?case* **by** *simp*
next
case (*NewC s0 C s1 a s2*)
from ‹*Env*⊢*NewC C*::−*PrimT Boolean*›
have *False*
by *cases simp*
thus *?case* **..**
next
case (*NewA s0 T s1 e i s2 a s3*)
from ‹*Env*⊢*New T*[*e*]::−*PrimT Boolean*›
have *False*
by *cases simp*
thus *?case* **..**
next
case (*Cast s0 e b s1 s2 T*)
note *s2* = ‹*s2* = *abupd* (*raise-if* (¬ *prg Env*,*snd s1*⊢*b fits T*) *ClassCast*) *s1*›
have *assigns-if* (*the-Bool b*) *e* ⊆ *dom* (*locals* (*store s1*))
proof −
from *s2* **and** ‹*normal s2*›
have *normal s1*
by (*cases s1*) *simp*
moreover
from ‹*Env*⊢*Cast T e*::−*PrimT Boolean*›

```
      have Env⊢e::− PrimT Boolean
        by cases (auto dest: cast-Boolean2)
      ultimately show ?thesis
        by (rule Cast.hyps [elim-format]) auto
    qed
    also from s2
    have ... ⊆ dom (locals (store s2))
      by simp
    finally show ?case by simp
  next
    case (Inst s0 e v s1 b T)
    from ‹prg Env⊢Norm s0 −e−≻v→ s1› and ‹normal s1›
    have assignsE e ⊆ dom (locals (store s1))
      by (rule assignsE-good-approx)
    thus ?case
      by simp
  next
    case (Lit s v)
    from ‹Env⊢Lit v::−PrimT Boolean›
    have typeof empty-dt v = Some (PrimT Boolean)
      by cases simp
    then obtain b where v=Bool b
      by (cases v) (simp-all add: empty-dt-def)
    thus ?case
      by simp
  next
    case (UnOp s0 e v s1 unop)
    note bool = ‹Env⊢UnOp unop e::−PrimT Boolean›
    hence bool-e: Env⊢e::−PrimT Boolean
      by cases (cases unop,simp-all)
    show ?case
    proof (cases constVal (UnOp unop e))
      case None
      note ‹normal s1›
      moreover note bool-e
      ultimately have assigns-if (the-Bool v) e ⊆ dom (locals (store s1))
        by (rule UnOp.hyps [elim-format]) auto
      moreover
      from bool have unop = UNot
        by cases (cases unop, simp-all)
      moreover note None
      ultimately
      have assigns-if (the-Bool (eval-unop unop v)) (UnOp unop e)
              ⊆ dom (locals (store s1))
        by simp
      thus ?thesis by simp
    next
      case (Some c)
      moreover
      from ‹prg Env⊢Norm s0 −e−≻v→ s1›
      have prg Env⊢Norm s0 −UnOp unop e−≻eval-unop unop v→ s1
        by (rule eval.UnOp)
      with Some
      have eval-unop unop v=c
        by (rule constVal-eval-elim) simp
      moreover
      from Some bool
      obtain b where c=Bool b
        by (rule constVal-Boolean [elim-format])
```

```
        (cases c, simp-all add: empty-dt-def)
    ultimately
    have assigns-if (the-Bool (eval-unop unop v)) (UnOp unop e) = {}
      by simp
    thus ?thesis by simp
  qed
next
  case (BinOp s0 e1 v1 s1 binop e2 v2 s2)
  note bool = ‹Env⊢BinOp binop e1 e2::−PrimT Boolean›
  show ?case
  proof (cases constVal (BinOp binop e1 e2))
    case (Some c)
    moreover
    from BinOp.hyps
    have
      prg Env⊢Norm s0 −BinOp binop e1 e2−≻eval-binop binop v1 v2→ s2
      by − (rule eval.BinOp)
    with Some
    have eval-binop binop v1 v2=c
      by (rule constVal-eval-elim) simp
    moreover
    from Some bool
    obtain b where c = Bool b
      by (rule constVal-Boolean [elim-format])
        (cases c, simp-all add: empty-dt-def)
    ultimately
    have assigns-if (the-Bool (eval-binop binop v1 v2)) (BinOp binop e1 e2)
          = {}
      by simp
    thus ?thesis by simp
  next
    case None
    show ?thesis
    proof (cases binop=CondAnd ∨ binop=CondOr)
      case True
      from bool obtain bool-e1: Env⊢e1::−PrimT Boolean and
                       bool-e2: Env⊢e2::−PrimT Boolean
        using True by cases auto
      have assigns-if (the-Bool v1) e1 ⊆ dom (locals (store s1))
      proof −
        from BinOp have normal s1
          by − (erule eval-no-abrupt-lemma [rule-format])
        from this bool-e1
        show ?thesis
          by (rule BinOp.hyps [elim-format]) auto
      qed
      also
      from BinOp.hyps
      have ... ⊆ dom (locals (store s2))
        by − (erule dom-locals-eval-mono-elim,simp)
      finally
      have e1-s2: assigns-if (the-Bool v1) e1 ⊆ dom (locals (store s2)).
      from True show ?thesis
      proof
        assume condAnd: binop = CondAnd
        show ?thesis
        proof (cases the-Bool (eval-binop binop v1 v2))
          case True
          with condAnd
```

```
      have need-second: need-second-arg binop v1
        by (simp add: need-second-arg-def)
      from ‹normal s2›
      have assigns-if (the-Bool v2) e2 ⊆ dom (locals (store s2))
        by (rule BinOp.hyps [elim-format])
           (simp add: need-second bool-e2)+
      with e1-s2
      have assigns-if (the-Bool v1) e1 ∪ assigns-if (the-Bool v2) e2
             ⊆ dom (locals (store s2))
        by (rule Un-least)
      with True condAnd None show ?thesis
        by simp
    next
      case False
      note binop-False = this
      show ?thesis
      proof (cases need-second-arg binop v1)
        case True
        with binop-False condAnd
        obtain the-Bool v1=True and the-Bool v2 = False
          by (simp add: need-second-arg-def)
        moreover
        from ‹normal s2›
        have assigns-if (the-Bool v2) e2 ⊆ dom (locals (store s2))
          by (rule BinOp.hyps [elim-format]) (simp add: True bool-e2)+
        with e1-s2
        have assigns-if (the-Bool v1) e1 ∪ assigns-if (the-Bool v2) e2
               ⊆ dom (locals (store s2))
          by (rule Un-least)
        moreover note binop-False condAnd None
        ultimately show ?thesis
          by auto
      next
        case False
        with binop-False condAnd
        have the-Bool v1=False
          by (simp add: need-second-arg-def)
        with e1-s2
        show ?thesis
          using binop-False condAnd None by auto
      qed
    qed
  next
    assume condOr: binop = CondOr
    show ?thesis
    proof (cases the-Bool (eval-binop binop v1 v2))
      case False
      with condOr
      have need-second: need-second-arg binop v1
        by (simp add: need-second-arg-def)
      from ‹normal s2›
      have assigns-if (the-Bool v2) e2 ⊆ dom (locals (store s2))
        by (rule BinOp.hyps [elim-format])
           (simp add: need-second bool-e2)+
      with e1-s2
      have assigns-if (the-Bool v1) e1 ∪ assigns-if (the-Bool v2) e2
             ⊆ dom (locals (store s2))
        by (rule Un-least)
      with False condOr None show ?thesis
```

```
          by simp
        next
          case True
          note binop-True = this
          show ?thesis
          proof (cases need-second-arg binop v1)
            case True
            with binop-True condOr
            obtain the-Bool v1=False and the-Bool v2 = True
              by (simp add: need-second-arg-def)
            moreover
            from ‹normal s2›
            have assigns-if (the-Bool v2) e2 ⊆ dom (locals (store s2))
              by (rule BinOp.hyps [elim-format]) (simp add: True bool-e2)+
            with e1-s2
            have assigns-if (the-Bool v1) e1 ∪ assigns-if (the-Bool v2) e2
                   ⊆ dom (locals (store s2))
              by (rule Un-least)
            moreover note binop-True condOr None
            ultimately show ?thesis
              by auto
          next
            case False
            with binop-True condOr
            have the-Bool v1=True
              by (simp add: need-second-arg-def)
            with e1-s2
            show ?thesis
              using binop-True condOr None by auto
          qed
        qed
      qed
    next
      case False
      note ‹¬ (binop = CondAnd ∨ binop = CondOr)›
      from BinOp.hyps
      have
        prg Env⊢Norm s0 −BinOp binop e1 e2−≻eval-binop binop v1 v2→ s2
        by − (rule eval.BinOp)
      moreover note ‹normal s2›
      ultimately
      have assignsE (BinOp binop e1 e2) ⊆ dom (locals (store s2))
        by (rule assignsE-good-approx)
      with False None
      show ?thesis
        by simp
    qed
  qed
next
  case Super
  note ‹Env⊢Super::−PrimT Boolean›
  hence False
    by cases simp
  thus ?case ..
next
  case (Acc s0 va v f s1)
  from ‹prg Env⊢Norm s0 −va=≻(v, f)→ s1› and ‹normal s1›
  have assignsV va ⊆ dom (locals (store s1))
    by (rule assignsV-good-approx)
```

```
    thus ?case by simp
  next
    case (Ass s0 va w f s1 e v s2)
    hence prg Env⊢Norm s0 −va := e−≻v→ assign f v s2
      by − (rule eval.Ass)
    moreover note ‹normal (assign f v s2)›
    ultimately
    have assignsE (va := e) ⊆ dom (locals (store (assign f v s2)))
      by (rule assignsE-good-approx)
    thus ?case by simp
  next
    case (Cond s0 e0 b s1 e1 e2 v s2)
    from ‹Env⊢e0 ? e1 : e2::−PrimT Boolean›
    obtain wt-e1: Env⊢e1::−PrimT Boolean and
           wt-e2: Env⊢e2::−PrimT Boolean
      by cases (auto dest: widen-Boolean2)
    note eval-e0 = ‹prg Env⊢Norm s0 −e0−≻b→ s1›
    have e0-s2: assignsE e0 ⊆ dom (locals (store s2))
    proof −
      note eval-e0
      moreover
      from Cond.hyps and ‹normal s2› have normal s1
        by − (erule eval-no-abrupt-lemma [rule-format],simp)
      ultimately
      have assignsE e0 ⊆ dom (locals (store s1))
        by (rule assignsE-good-approx)
      also
      from Cond
      have . . . ⊆ dom (locals (store s2))
        by − (erule dom-locals-eval-mono [elim-format],simp)
      finally show ?thesis .
    qed
    show ?case
    proof (cases constVal e0)
      case None
      have assigns-if (the-Bool v) e1 ∩  assigns-if (the-Bool v) e2
             ⊆ dom (locals (store s2))
      proof (cases the-Bool b)
        case True
        from ‹normal s2›
        have assigns-if (the-Bool v) e1 ⊆ dom (locals (store s2))
          by (rule Cond.hyps [elim-format]) (simp-all add: wt-e1 True)
        thus ?thesis
          by blast
      next
        case False
        from ‹normal s2›
        have assigns-if (the-Bool v) e2 ⊆ dom (locals (store s2))
          by (rule Cond.hyps [elim-format]) (simp-all add: wt-e2 False)
        thus ?thesis
          by blast
      qed
      with e0-s2
      have assignsE e0 ∪
            (assigns-if (the-Bool v) e1 ∩  assigns-if (the-Bool v) e2)
             ⊆ dom (locals (store s2))
        by (rule Un-least)
      with None show ?thesis
        by simp
```

```
    next
      case (Some c)
      from this eval-e0 have eq-b-c: b=c
        by (rule constVal-eval-elim)
      show ?thesis
      proof (cases the-Bool c)
        case True
        from ‹normal s2›
        have assigns-if (the-Bool v) e1 ⊆ dom (locals (store s2))
          by (rule Cond.hyps [elim-format]) (simp-all add: eq-b-c True wt-e1)
        with e0-s2
        have assignsE e0 ∪ assigns-if (the-Bool v) e1 ⊆ ...
          by (rule Un-least)
        with Some True show ?thesis
          by simp
      next
        case False
        from ‹normal s2›
        have assigns-if (the-Bool v) e2 ⊆ dom (locals (store s2))
          by (rule Cond.hyps [elim-format]) (simp-all add: eq-b-c False wt-e2)
        with e0-s2
        have assignsE e0 ∪ assigns-if (the-Bool v) e2 ⊆ ...
          by (rule Un-least)
        with Some False show ?thesis
          by simp
      qed
    qed
  next
    case (Call s0 e a s1 args vs s2 D mode statT mn pTs s3 s3' accC v s4)
    hence
    prg Env⊢Norm s0 −({accC,statT,mode}e·mn( {pTs}args))−≻v→
                  (set-lvars (locals (store s2)) s4)
      by − (rule eval.Call)
    hence assignsE ({accC,statT,mode}e·mn( {pTs}args))
            ⊆  dom (locals (store ((set-lvars (locals (store s2))) s4)))
      using ‹normal ((set-lvars (locals (snd s2))) s4)›
      by (rule assignsE-good-approx)
    thus ?case by simp
  next
    case Methd show ?case by simp
  next
    case Body show ?case by simp
  qed simp+ — all the statements and variables
 }
 note generalized = this
 from eval bool show ?thesis
   by (rule generalized [elim-format]) simp+
qed


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 −
  from eval have prg (|prg=G,cls=C,lcl=L|)⊢s0 −e−≻b→ s1 by simp
  from this normal bool show ?thesis
    by (rule assigns-if-good-approx)
```

qed

lemma *subset-Intl*: $A \subseteq C \Longrightarrow A \cap B \subseteq C$
  by *blast*

lemma *subset-Intr*: $B \subseteq C \Longrightarrow A \cap B \subseteq C$
  by *blast*

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 −
  note *inj-term-simps* [*simp*]
  obtain *G* **where** *G*: *prg Env* = *G* **by** (*cases Env*) *simp*
  with *eval* **have** *eval*: *G*⊢*s0* −*t*≻→ (*v*,*s1*) **by** *simp*
  from *G wf* **have** *wf*: *wf-prog G* **by** *simp*
  let *?HypObj* = λ *t s0 s1*.
    ∀ *Env T A*. *?Wt Env t T* ⟶ *?Da Env s0 t A* ⟶ *prg Env* = *G*
     ⟶ *?NormalAssigned s1 A* ∧ *?BreakAssigned s0 s1 A* ∧ *?ResAssigned s0 s1*
  — Goal in object logic variant
  let *?Hyp* = λ*t s0 s1*. (⋀ *Env T A*. ⟦*?Wt Env t T*; *?Da Env s0 t A*; *prg Env* = *G*⟧
     ⟹ *?NormalAssigned s1 A* ∧ *?BreakAssigned s0 s1 A* ∧ *?ResAssigned s0 s1*)
  from *eval* **and** *wt da G*
  show *?thesis*
  proof (*induct arbitrary*: *Env T A*)
   case (*Abrupt xc s t Env T A*)
   have *da*: *Env*⊢ *dom* (*locals s*) »*t*» *A* **using** *Abrupt.prems* **by** *simp*
   have *?NormalAssigned* (*Some xc*,*s*) *A*
    by *simp*
   moreover
   have *?BreakAssigned* (*Some xc*,*s*) (*Some xc*,*s*) *A*
    by *simp*
   moreover have *?ResAssigned* (*Some xc*,*s*) (*Some xc*,*s*)
    by *simp*
   ultimately show *?case* **by** (*intro conjI*)
  next
   case (*Skip s Env T A*)
   have *da*: *Env*⊢ *dom* (*locals* (*store* (*Norm s*))) »⟨*Skip*⟩» *A*
    using *Skip.prems* **by** *simp*
   hence *nrm A* = *dom* (*locals* (*store* (*Norm s*)))
    by (*rule da-elim-cases*) *simp*
   hence *?NormalAssigned* (*Norm s*) *A*
    by *auto*
   moreover
   have *?BreakAssigned* (*Norm s*) (*Norm s*) *A*
    by *simp*
   moreover have *?ResAssigned* (*Norm s*) (*Norm s*)

by *simp*
ultimately show *?case* **by** (*intro conjI*)
next
case (*Expr s0 e v s1 Env T A*)
from *Expr.prems*
show *?NormalAssigned s1 A* ∧ *?BreakAssigned* (*Norm s0*) *s1 A*
∧ *?ResAssigned* (*Norm s0*) *s1*
by (*elim wt-elim-cases da-elim-cases*)
(*rule Expr.hyps*, *auto*)
next
case (*Lab s0 c s1 j Env T A*)
note *G* = ‹*prg Env* = *G*›
from *Lab.prems*
obtain *C l* **where**
da-c: *Env*⊢ *dom* (*locals* (*snd* (*Norm s0*))) »⟨*c*⟩» *C* **and**
A: *nrm A* = *nrm C* ∩ (*brk C*) *l brk A* = *rmlab l* (*brk C*) **and**
j: *j* = *Break l*
by − (*erule da-elim-cases*, *simp*)
from *Lab.prems*
have *wt-c*: *Env*⊢*c*::√
by − (*erule wt-elim-cases*, *simp*)
from *wt-c da-c G* **and** *Lab.hyps*
have *norm-c*: *?NormalAssigned s1 C* **and**
brk-c: *?BreakAssigned* (*Norm s0*) *s1 C* **and**
res-c: *?ResAssigned* (*Norm s0*) *s1*
by *simp-all*
have *?NormalAssigned* (*abupd* (*absorb j*) *s1*) *A*
proof
assume *normal*: *normal* (*abupd* (*absorb j*) *s1*)
show *nrm A* ⊆ *dom* (*locals* (*store* (*abupd* (*absorb j*) *s1*)))
proof (*cases abrupt s1*)
case *None*
with *norm-c A*
show *?thesis*
by *auto*
next
case *Some*
with *normal j*
have *abrupt s1* = *Some* (*Jump* (*Break l*))
by (*auto dest*: *absorb-Some-NoneD*)
with *brk-c A*
show *?thesis*
by *auto*
qed
qed
moreover
have *?BreakAssigned* (*Norm s0*) (*abupd* (*absorb j*) *s1*) *A*
proof −
{
fix *l′*
assume *break*: *abrupt* (*abupd* (*absorb j*) *s1*) = *Some* (*Jump* (*Break l′*))
with *j*
have *l*≠*l′*
by (*cases s1*) (*auto dest*!: *absorb-Some-JumpD*)
hence (*rmlab l* (*brk C*)) *l′*= (*brk C*) *l′*
by (*simp*)
with *break brk-c A*
have
(*brk A l′* ⊆ *dom* (*locals* (*store* (*abupd* (*absorb j*) *s1*))))

```
        by (cases s1) auto
      }
      then show ?thesis
        by simp
    qed
    moreover
    from res-c have ?ResAssigned (Norm s0) (abupd (absorb j) s1)
      by (cases s1) (simp add: absorb-def)
    ultimately show ?case by (intro conjI)
  next
    case (Comp s0 c1 s1 c2 s2 Env T A)
    note G = ‹prg Env = G›
    from Comp.prems
    obtain C1 C2
      where da-c1: Env⊢ dom (locals (snd (Norm s0))) »⟨c1⟩» C1 and
            da-c2: Env⊢ nrm C1 »⟨c2⟩» C2 and
                A: nrm A = nrm C2 brk A = (brk C1) ⇒∩ (brk C2)
      by (elim da-elim-cases) simp
    from Comp.prems
    obtain wt-c1: Env⊢c1::√ and
           wt-c2: Env⊢c2::√
      by (elim wt-elim-cases) simp
    note ‹PROP ?Hyp (In1r c1) (Norm s0) s1›
    with wt-c1 da-c1 G
    obtain nrm-c1: ?NormalAssigned s1 C1 and
           brk-c1: ?BreakAssigned (Norm s0) s1 C1 and
           res-c1: ?ResAssigned (Norm s0) s1
      by simp
    show ?case
    proof (cases normal s1)
      case True
      with nrm-c1 have nrm C1 ⊆ dom (locals (snd s1)) by iprover
      with da-c2 obtain C2'
        where  da-c2': Env⊢ dom (locals (snd s1)) »⟨c2⟩» C2' and
               nrm-c2: nrm C2 ⊆ nrm C2'                  and
               brk-c2: ∀ l. brk C2 l ⊆ brk C2' l
        by (rule da-weakenE) iprover
      note ‹PROP ?Hyp (In1r c2) s1 s2›
      with wt-c2 da-c2' G
      obtain nrm-c2': ?NormalAssigned s2 C2' and
             brk-c2': ?BreakAssigned s1 s2 C2' and
             res-c2 : ?ResAssigned s1 s2
        by simp
      from nrm-c2' nrm-c2 A
      have ?NormalAssigned s2 A
        by blast
      moreover from brk-c2' brk-c2 A
      have ?BreakAssigned s1 s2 A
        by fastforce
      with True
      have ?BreakAssigned (Norm s0) s2 A by simp
      moreover from res-c2 True
      have ?ResAssigned (Norm s0) s2
        by simp
      ultimately show ?thesis by (intro conjI)
    next
      case False
      with ‹G⊢s1 −c2→ s2›
      have eq-s1-s2: s2=s1 by auto
```

```
      with False have ?NormalAssigned s2 A by blast
      moreover
      have ?BreakAssigned (Norm s0) s2 A
      proof (cases ∃ l. abrupt s1 = Some (Jump (Break l)))
        case True
        then obtain l where l: abrupt s1 = Some (Jump (Break l)) ..
        with brk-c1
        have brk C1 l ⊆ dom (locals (store s1))
          by simp
        with A eq-s1-s2
        have brk A l ⊆ dom (locals (store s2))
          by auto
        with l eq-s1-s2
        show ?thesis by simp
      next
        case False
        with eq-s1-s2 show ?thesis by simp
      qed
      moreover from False res-c1 eq-s1-s2
      have ?ResAssigned (Norm s0) s2
        by simp
      ultimately show ?thesis by (intro conjI)
    qed
  next
    case (If s0 e b s1 c1 c2 s2 Env T A)
    note G = ‹prg Env = G›
    with If.hyps have eval-e: prg Env ⊢Norm s0 −e−≻b→ s1 by simp
    from If.prems
    obtain E C1 C2 where
      da-e: Env⊢ dom (locals (store ((Norm s0)::state))) »⟨e⟩» E and
      da-c1: Env⊢ (dom (locals (store ((Norm s0)::state)))
                  ∪ assigns-if True e) »⟨c1⟩» C1 and
      da-c2: Env⊢ (dom (locals (store ((Norm s0)::state)))
                  ∪ assigns-if False e) »⟨c2⟩» C2 and
      A: nrm A = nrm C1 ∩ nrm C2  brk A = brk C1 ⇒∩ brk C2
      by (elim da-elim-cases)
    from If.prems
    obtain
       wt-e: Env⊢e::− PrimT Boolean and
       wt-c1: Env⊢c1::√ and
       wt-c2: Env⊢c2::√
      by (elim wt-elim-cases)
    from If.hyps have
     s0-s1:dom (locals (store ((Norm s0)::state))) ⊆ dom (locals (store s1))
      by (elim dom-locals-eval-mono-elim)
    show ?case
    proof (cases normal s1)
      case True
      note normal-s1 = this
      show ?thesis
      proof (cases the-Bool b)
        case True
        from eval-e normal-s1 wt-e
        have assigns-if True e ⊆ dom (locals (store s1))
          by (rule assigns-if-good-approx [elim-format]) (simp add: True)
        with s0-s1
        have dom (locals (store ((Norm s0)::state))) ∪ assigns-if True e ⊆ ...
          by (rule Un-least)
        with da-c1 obtain C1'
```

```
      where da-c1': Env⊢ dom (locals (store s1)) »⟨c1⟩» C1' and
            nrm-c1: nrm C1 ⊆ nrm C1'                  and
            brk-c1: ∀ l. brk C1 l ⊆ brk C1' l
      by (rule da-weakenE) iprover
    from If.hyps True have PROP ?Hyp (In1r c1) s1 s2 by simp
    with wt-c1 da-c1'
    obtain nrm-c1': ?NormalAssigned s2 C1' and
           brk-c1': ?BreakAssigned s1 s2 C1' and
           res-c1: ?ResAssigned s1 s2
      using G by simp
    from nrm-c1' nrm-c1 A
    have ?NormalAssigned s2 A
      by blast
    moreover from brk-c1' brk-c1 A
    have ?BreakAssigned s1 s2 A
      by fastforce
    with normal-s1
    have ?BreakAssigned (Norm s0) s2 A by simp
    moreover from res-c1 normal-s1 have ?ResAssigned (Norm s0) s2
      by simp
    ultimately show ?thesis by (intro conjI)
  next
    case False
    from eval-e normal-s1 wt-e
    have assigns-if False e ⊆ dom (locals (store s1))
      by (rule assigns-if-good-approx [elim-format]) (simp add: False)
    with s0-s1
    have dom (locals (store ((Norm s0)::state)))∪ assigns-if False e ⊆ ...
      by (rule Un-least)
    with da-c2 obtain C2'
      where da-c2': Env⊢ dom (locals (store s1)) »⟨c2⟩» C2' and
            nrm-c2: nrm C2 ⊆ nrm C2'                  and
            brk-c2: ∀ l. brk C2 l ⊆ brk C2' l
      by (rule da-weakenE) iprover
    from If.hyps False have PROP ?Hyp (In1r c2) s1 s2 by simp
    with wt-c2 da-c2'
    obtain nrm-c2': ?NormalAssigned s2 C2' and
           brk-c2': ?BreakAssigned s1 s2 C2' and
           res-c2: ?ResAssigned s1 s2
      using G by simp
    from nrm-c2' nrm-c2 A
    have ?NormalAssigned s2 A
      by blast
    moreover from brk-c2' brk-c2 A
    have ?BreakAssigned s1 s2 A
      by fastforce
    with normal-s1
    have ?BreakAssigned (Norm s0) s2 A by simp
    moreover from res-c2 normal-s1 have ?ResAssigned (Norm s0) s2
      by simp
    ultimately show ?thesis by (intro conjI)
  qed
next
  case False
  then obtain abr where abr: abrupt s1 = Some abr
    by (cases s1) auto
  moreover
  from eval-e - wt-e have ⋀ j. abrupt s1 ≠ Some (Jump j)
    by (rule eval-expression-no-jump) (simp-all add: G wf)
```

```
      moreover
      have s2 = s1
      proof -
        from abr and ‹G⊢s1 −(if the-Bool b then c1 else c2)→ s2›
        show ?thesis
          by (cases s1) simp
      qed
      ultimately show ?thesis by simp
    qed
  next
    case (Loop s0 e b s1 c s2 l s3 Env T A)
    note G = ‹prg Env = G›
    with Loop.hyps have eval-e: prg Env⊢Norm s0 −e−≻b→ s1
      by (simp (no-asm-simp))
    from Loop.prems
    obtain E C where
      da-e: Env⊢ dom (locals (store ((Norm s0)::state))) »⟨e⟩» E and
      da-c: Env⊢ (dom (locals (store ((Norm s0)::state)))
                  ∪ assigns-if True e) »⟨c⟩» C and
     A: nrm A = nrm C ∩
             (dom (locals (store ((Norm s0)::state))) ∪ assigns-if False e)
        brk A = brk C
      by (elim da-elim-cases)
    from Loop.prems
    obtain
       wt-e: Env⊢e::−PrimT Boolean and
       wt-c: Env⊢c::√
      by (elim wt-elim-cases)
    from wt-e da-e G
    obtain res-s1: ?ResAssigned (Norm s0) s1
      by (elim Loop.hyps [elim-format]) simp+
    from Loop.hyps have
      s0-s1:dom (locals (store ((Norm s0)::state))) ⊆ dom (locals (store s1))
      by (elim dom-locals-eval-mono-elim)
    show ?case
    proof (cases normal s1)
      case True
      note normal-s1 = this
      show ?thesis
      proof (cases the-Bool b)
        case True
        with Loop.hyps obtain
          eval-c: G⊢s1 −c→ s2 and
          eval-while: G⊢abupd (absorb (Cont l)) s2 −l· While(e) c→ s3
          by simp
        from Loop.hyps True
        have ?HypObj (In1r c) s1 s2 by simp
        note hyp-c = this [rule-format]
        from Loop.hyps True
        have ?HypObj (In1r (l· While(e) c)) (abupd (absorb (Cont l)) s2) s3
          by simp
        note hyp-while = this [rule-format]
        from eval-e normal-s1 wt-e
        have assigns-if True e ⊆ dom (locals (store s1))
          by (rule assigns-if-good-approx [elim-format]) (simp add: True)
        with s0-s1
        have dom (locals (store ((Norm s0)::state))) ∪ assigns-if True e ⊆ ...
          by (rule Un-least)
        with da-c obtain C'
```

```
  where da-c': Env⊢ dom (locals (store s1)) »⟨c⟩» C' and
     nrm-C-C': nrm C ⊆ nrm C'                     and
     brk-C-C': ∀ l. brk C l ⊆ brk C' l
  by (rule da-weakenE) iprover
from hyp-c wt-c da-c'
obtain nrm-C': ?NormalAssigned s2 C' and
  brk-C': ?BreakAssigned s1 s2 C' and
  res-s2: ?ResAssigned s1 s2
  using G by simp
show ?thesis
proof (cases normal s2 ∨ abrupt s2 = Some (Jump (Cont l)))
  case True
  from Loop.prems obtain
    wt-while: Env⊢In1r (l• While(e) c)::T and
    da-while: Env⊢ dom (locals (store ((Norm s0)::state)))
                 »⟨l• While(e) c⟩» A
    by simp
  have dom (locals (store ((Norm s0)::state)))
         ⊆ dom (locals (store (abupd (absorb (Cont l)) s2)))
  proof −
    note s0-s1
    also from eval-c
    have dom (locals (store s1)) ⊆ dom (locals (store s2))
      by (rule dom-locals-eval-mono-elim)
    also have ... ⊆ dom (locals (store (abupd (absorb (Cont l)) s2)))
      by simp
    finally show ?thesis .
  qed
  with da-while obtain A'
    where
    da-while': Env⊢ dom (locals (store (abupd (absorb (Cont l)) s2)))
               »⟨l• While(e) c⟩» A'
    and  nrm-A-A': nrm A ⊆ nrm A'
    and  brk-A-A': ∀ l. brk A l ⊆ brk A' l
    by (rule da-weakenE) simp
  with wt-while hyp-while
  obtain nrm-A': ?NormalAssigned s3 A' and
         brk-A': ?BreakAssigned (abupd (absorb (Cont l)) s2) s3 A' and
         res-s3: ?ResAssigned (abupd (absorb (Cont l)) s2) s3
    using G by simp
  from nrm-A-A' nrm-A'
  have ?NormalAssigned s3 A
    by blast
  moreover
  have ?BreakAssigned (Norm s0) s3 A
  proof −
    from brk-A-A' brk-A'
    have ?BreakAssigned (abupd (absorb (Cont l)) s2) s3 A
      by fastforce
    moreover
    from True have normal (abupd (absorb (Cont l)) s2)
      by (cases s2) auto
    ultimately show ?thesis
      by simp
  qed
  moreover from res-s3 True have ?ResAssigned (Norm s0) s3
    by auto
  ultimately show ?thesis by (intro conjI)
next
```

```
        case False
        then obtain abr where
          abrupt s2 = Some abr and
          abrupt (abupd (absorb (Cont l)) s2) = Some abr
          by auto
        with eval-while
        have eq-s3-s2: s3=s2
          by auto
        with nrm-C-C′ nrm-C′ A
        have ?NormalAssigned s3 A
          by auto
        moreover
        from eq-s3-s2 brk-C-C′ brk-C′ normal-s1 A
        have ?BreakAssigned (Norm s0) s3 A
          by fastforce
        moreover
        from eq-s3-s2 res-s2 normal-s1 have ?ResAssigned (Norm s0) s3
          by simp
        ultimately show ?thesis by (intro conjI)
      qed
    next
      case False
      with Loop.hyps have eq-s3-s1: s3=s1
        by simp
      from eq-s3-s1 res-s1
      have res-s3: ?ResAssigned (Norm s0) s3
        by simp
      from eval-e True wt-e
      have assigns-if False e ⊆ dom (locals (store s1))
        by (rule assigns-if-good-approx [elim-format]) (simp add: False)
      with s0-s1
      have dom (locals (store ((Norm s0)::state)))∪assigns-if False e ⊆ . . .
        by (rule Un-least)
      hence nrm C ∩
            (dom (locals (store ((Norm s0)::state))) ∪ assigns-if False e)
            ⊆ dom (locals (store s1))
        by (rule subset-Intr)
      with normal-s1 A eq-s3-s1
      have ?NormalAssigned s3 A
        by simp
      moreover
      from normal-s1 eq-s3-s1
      have ?BreakAssigned (Norm s0) s3 A
        by simp
      moreover note res-s3
      ultimately show ?thesis by (intro conjI)
    qed
  next
    case False
    then obtain abr where abr: abrupt s1 = Some abr
      by (cases s1) auto
    moreover
    from eval-e - wt-e have no-jmp: ⋀ j. abrupt s1 ≠ Some (Jump j)
      by (rule eval-expression-no-jump) (simp-all add: wf G)
    moreover
    have eq-s3-s1: s3=s1
    proof (cases the-Bool b)
      case True
      with Loop.hyps obtain
```

```
        eval-c: G⊢s1 −c→ s2 and
        eval-while: G⊢abupd (absorb (Cont l)) s2 −l• While(e) c→ s3
        by simp
      from eval-c abr have s2=s1 by auto
      moreover from calculation no-jmp have abupd (absorb (Cont l)) s2=s2
        by (cases s1) (simp add: absorb-def)
      ultimately show ?thesis
        using eval-while abr
        by auto
    next
      case False
      with Loop.hyps show ?thesis by simp
    qed
    moreover
    from eq-s3-s1 res-s1
    have res-s3: ?ResAssigned (Norm s0) s3
      by simp
    ultimately show ?thesis
      by simp
  qed
next
  case (Jmp s j Env T A)
  have ?NormalAssigned (Some (Jump j),s) A by simp
  moreover
  from Jmp.prems
  obtain ret: j = Ret ⟶ Result ∈ dom (locals (store (Norm s))) and
         brk: brk A = (case j of
                         Break l ⇒ λ k. if k=l
                                   then dom (locals (store ((Norm s)::state)))
                                   else UNIV
                       | Cont l  ⇒ λ k. UNIV
                       | Ret     ⇒ λ k. UNIV)
    by (elim da-elim-cases) simp
  from brk have ?BreakAssigned (Norm s) (Some (Jump j),s) A
    by simp
  moreover from ret have ?ResAssigned (Norm s) (Some (Jump j),s)
    by simp
  ultimately show ?case by (intro conjI)
next
  case (Throw s0 e a s1 Env T A)
  note G = ‹prg Env = G›
  from Throw.prems obtain E where
    da-e: Env⊢ dom (locals (store ((Norm s0)::state))) »⟨e⟩» E
    by (elim da-elim-cases)
  from Throw.prems
    obtain eT where wt-e: Env⊢e::−eT
      by (elim wt-elim-cases)
  have ?NormalAssigned (abupd (throw a) s1) A
    by (cases s1) (simp add: throw-def)
  moreover
  have ?BreakAssigned (Norm s0) (abupd (throw a) s1) A
  proof −
    from G Throw.hyps have eval-e: prg Env⊢Norm s0 −e−≻a→ s1
      by (simp (no-asm-simp))
    from eval-e - wt-e
    have ⋀ l. abrupt s1 ≠ Some (Jump (Break l))
      by (rule eval-expression-no-jump) (simp-all add: wf G)
    hence ⋀ l. abrupt (abupd (throw a) s1) ≠ Some (Jump (Break l))
      by (cases s1) (simp add: throw-def abrupt-if-def)
```

```
      thus ?thesis
        by simp
    qed
    moreover
    from wt-e da-e G have ?ResAssigned (Norm s0) s1
      by (elim Throw.hyps [elim-format]) simp+
    hence ?ResAssigned (Norm s0) (abupd (throw a) s1)
      by (cases s1) (simp add: throw-def abrupt-if-def)
    ultimately show ?case by (intro conjI)
  next
    case (Try s0 c1 s1 s2 C vn c2 s3 Env T A)
    note G = ‹prg Env = G›
    from Try.prems obtain C1 C2 where
      da-c1: Env⊢ dom (locals (store ((Norm s0)::state))) »⟨c1⟩» C1  and
      da-c2:
       Env(|lcl := (lcl Env)(VName vn↦Class C)|)
       ⊢ (dom (locals (store ((Norm s0)::state))) ∪ {VName vn}) »⟨c2⟩» C2 and
      A: nrm A = nrm C1 ∩ nrm C2 brk A = brk C1 ⇒∩ brk C2
      by (elim da-elim-cases) simp
    from Try.prems obtain
      wt-c1: Env⊢c1::√ and
      wt-c2: Env(|lcl := (lcl Env)(VName vn↦Class C)|)⊢c2::√
      by (elim wt-elim-cases)
    have sxalloc: prg Env⊢s1 −sxalloc→ s2 using Try.hyps G
      by (simp (no-asm-simp))
    note ‹PROP ?Hyp (In1r c1) (Norm s0) s1›
    with wt-c1 da-c1 G
    obtain nrm-C1: ?NormalAssigned s1 C1 and
           brk-C1: ?BreakAssigned (Norm s0) s1 C1 and
           res-s1: ?ResAssigned (Norm s0) s1
      by simp
    show ?case
    proof (cases normal s1)
      case True
      with nrm-C1 have nrm C1 ∩ nrm C2 ⊆ dom (locals (store s1))
        by auto
      moreover
      have s3=s1
      proof −
        from sxalloc True have eq-s2-s1: s2=s1
          by (cases s1) (auto elim: sxalloc-elim-cases)
        with True have ¬  G,s2⊢catch C
          by (simp add: catch-def)
        with Try.hyps have s3=s2
          by simp
        with eq-s2-s1 show ?thesis by simp
      qed
      ultimately show ?thesis
        using True A res-s1 by simp
    next
      case False
      note not-normal-s1 = this
      show ?thesis
      proof (cases ∃ l. abrupt s1 = Some (Jump (Break l)))
        case True
        then obtain l where l: abrupt s1 = Some (Jump (Break l))
          by auto
        with brk-C1 have (brk C1 ⇒∩ brk C2) l ⊆ dom (locals (store s1))
          by auto
```

moreover have *s3=s1*
proof −
from *sxalloc l* **have** *eq-s2-s1*: *s2=s1*
by (*cases s1*) (*auto elim*: *sxalloc-elim-cases*)
with *l* **have** ¬ *G*,*s2*⊢*catch C*
by (*simp add*: *catch-def*)
with *Try.hyps* **have** *s3=s2*
by *simp*
with *eq-s2-s1* **show** *?thesis* **by** *simp*
qed
ultimately show *?thesis*
using *l A res-s1* **by** *simp*
next
case *False*
note *abrupt-no-break* = *this*
show *?thesis*
proof (*cases G*,*s2*⊢*catch C*)
case *True*
with *Try.hyps* **have** *?HypObj* (*In1r c2*) (*new-xcpt-var vn s2*) *s3*
by *simp*
note *hyp-c2* = *this* [*rule-format*]
have (*dom* (*locals* (*store* ((*Norm s0*)::*state*))) ∪ {*VName vn*})
⊆ *dom* (*locals* (*store* (*new-xcpt-var vn s2*)))
proof −
from ‹*G*⊢*Norm s0* −*c1*→ *s1*›
have *dom* (*locals* (*store* ((*Norm s0*)::*state*)))
⊆ *dom* (*locals* (*store s1*))
by (*rule dom-locals-eval-mono-elim*)
also
from *sxalloc*
have . . . ⊆ *dom* (*locals* (*store s2*))
by (*rule dom-locals-sxalloc-mono*)
also
have . . . ⊆ *dom* (*locals* (*store* (*new-xcpt-var vn s2*)))
by (*cases s2*) (*simp add*: *new-xcpt-var-def*, *blast*)
also
have {*VName vn*} ⊆ . . .
by (*cases s2*) *simp*
ultimately show *?thesis*
by (*rule Un-least*)
qed
with *da-c2*
obtain *C2′* **where**
da-C2′: *Env*(|*lcl* := (*lcl Env*)(*VName vn*↦*Class C*)|)
⊢ *dom* (*locals* (*store* (*new-xcpt-var vn s2*))) »⟨*c2*⟩» *C2′*
and *nrm-C2′*: *nrm C2* ⊆ *nrm C2′*
and *brk-C2′*: ∀ *l*. *brk C2 l* ⊆ *brk C2′ l*
by (*rule da-weakenE*) *simp*
from *wt-c2 da-C2′ G* **and** *hyp-c2*
obtain *nrmAss-C2*: *?NormalAssigned s3 C2′* **and**
brkAss-C2: *?BreakAssigned* (*new-xcpt-var vn s2*) *s3 C2′* **and**
resAss-s3: *?ResAssigned* (*new-xcpt-var vn s2*) *s3*
by *simp*
from *nrmAss-C2 nrm-C2′ A*
have *?NormalAssigned s3 A*
by *auto*
moreover
have *?BreakAssigned* (*Norm s0*) *s3 A*
proof −

```
          from brkAss-C2 have ?BreakAssigned (Norm s0) s3 C2'
            by (cases s2) (auto simp add: new-xcpt-var-def)
          with brk-C2' A show ?thesis
            by fastforce
        qed
        moreover
        from resAss-s3 have ?ResAssigned (Norm s0) s3
          by (cases s2) ( simp add: new-xcpt-var-def)
        ultimately show ?thesis by (intro conjI)
      next
        case False
        with Try.hyps
        have eq-s3-s2: s3=s2 by simp
        moreover from sxalloc not-normal-s1 abrupt-no-break
        obtain ¬ normal s2
               ∀ l. abrupt s2 ≠ Some (Jump (Break l))
          by − (rule sxalloc-cases,auto)
        ultimately obtain
          ?NormalAssigned s3 A and ?BreakAssigned (Norm s0) s3 A
          by (cases s2) auto
        moreover have ?ResAssigned (Norm s0) s3
        proof (cases abrupt s1 = Some (Jump Ret))
          case True
          with sxalloc have s2=s1
            by (elim sxalloc-cases) auto
          with res-s1 eq-s3-s2 show ?thesis by simp
        next
          case False
          with sxalloc
          have abrupt s2 ≠ Some (Jump Ret)
            by (rule sxalloc-no-jump)
          with eq-s3-s2 show ?thesis
            by simp
        qed
        ultimately show ?thesis by (intro conjI)
      qed
    qed
  qed
next
  case (Fin s0 c1 x1 s1 c2 s2 s3 Env T A)
  note G = ‹prg Env = G›
  from Fin.prems obtain C1 C2 where
    da-C1: Env⊢ dom (locals (store ((Norm s0)::state))) »⟨c1⟩» C1 and
    da-C2: Env⊢ dom (locals (store ((Norm s0)::state))) »⟨c2⟩» C2 and
    nrm-A: nrm A = nrm C1 ∪ nrm C2 and
    brk-A: brk A = ((brk C1) ⇒∪∀ (nrm C2)) ⇒∩ (brk C2)
    by (elim da-elim-cases) simp
  from Fin.prems obtain
    wt-c1: Env⊢c1::√ and
    wt-c2: Env⊢c2::√
    by (elim wt-elim-cases)
  note ‹PROP ?Hyp (In1r c1) (Norm s0) (x1,s1)›
  with wt-c1 da-C1 G
  obtain nrmAss-C1: ?NormalAssigned (x1,s1) C1 and
         brkAss-C1: ?BreakAssigned (Norm s0) (x1,s1) C1 and
         resAss-s1: ?ResAssigned (Norm s0) (x1,s1)
    by simp
  obtain nrmAss-C2: ?NormalAssigned s2 C2 and
         brkAss-C2: ?BreakAssigned (Norm s1) s2 C2 and
```

resAss-s2: *?ResAssigned* (*Norm s1*) *s2*
proof −
from *Fin.hyps*
have *dom* (*locals* (*store* ((*Norm s0*)::*state*)))
⊆ *dom* (*locals* (*store* (*x1*,*s1*)))
by − (*rule dom-locals-eval-mono-elim*)
with *da-C2* **obtain** *C2′*
where
da-C2′: *Env*⊢ *dom* (*locals* (*store* (*x1*,*s1*))) »⟨*c2*⟩» *C2′* **and**
nrm-C2′: *nrm C2* ⊆ *nrm C2′* **and**
brk-C2′: ∀ *l*. *brk C2 l* ⊆ *brk C2′ l*
by (*rule da-weakenE*) *simp*
note ‹*PROP ?Hyp* (*In1r c2*) (*Norm s1*) *s2*›
with *wt-c2 da-C2′ G*
obtain *nrmAss-C2′*: *?NormalAssigned s2 C2′* **and**
brkAss-C2′: *?BreakAssigned* (*Norm s1*) *s2 C2′* **and**
resAss-s2′: *?ResAssigned* (*Norm s1*) *s2*
by *simp*
from *nrmAss-C2′ nrm-C2′* **have** *?NormalAssigned s2 C2*
by *blast*
moreover
from *brkAss-C2′ brk-C2′* **have** *?BreakAssigned* (*Norm s1*) *s2 C2*
by *fastforce*
ultimately
show *?thesis*
using *that resAss-s2′* **by** *simp*
qed
note *s3* = ‹*s3* = (*if* ∃ *err*. *x1* = *Some* (*Error err*) *then* (*x1*, *s1*)
else abupd (*abrupt-if* (*x1* ≠ *None*) *x1*) *s2*)›
have *s1-s2*: *dom* (*locals s1*) ⊆ *dom* (*locals* (*store s2*))
proof −
from ‹*G*⊢*Norm s1* −*c2*→ *s2*›
show *?thesis*
by (*rule dom-locals-eval-mono-elim*) *simp*
qed

have *?NormalAssigned s3 A*
proof
assume *normal-s3*: *normal s3*
show *nrm A* ⊆ *dom* (*locals* (*snd s3*))
proof −
have *nrm C1* ⊆ *dom* (*locals* (*snd s3*))
proof −
from *normal-s3 s3*
have *normal* (*x1*,*s1*)
by (*cases s2*) (*simp add*: *abrupt-if-def*)
with *normal-s3 nrmAss-C1 s3 s1-s2*
show *?thesis*
by *fastforce*
qed
moreover
have *nrm C2* ⊆ *dom* (*locals* (*snd s3*))
proof −
from *normal-s3 s3*
have *normal s2*
by (*cases s2*) (*simp add*: *abrupt-if-def*)
with *normal-s3 nrmAss-C2 s3 s1-s2*
show *?thesis*
by *fastforce*

```
        qed
        ultimately have nrm C1 ∪ nrm C2 ⊆ ...
          by (rule Un-least)
        with nrm-A show ?thesis
          by simp
      qed
    qed
    moreover
    {
      fix l assume brk-s3: abrupt s3 = Some (Jump (Break l))
      have brk A l ⊆ dom (locals (store s3))
      proof (cases normal s2)
        case True
        with brk-s3 s3
        have s2-s3: dom (locals (store s2)) ⊆ dom (locals (store s3))
          by simp
        have brk C1 l ⊆ dom (locals (store s3))
        proof -
          from True brk-s3 s3 have x1=Some (Jump (Break l))
            by (cases s2) (simp add: abrupt-if-def)
          with brkAss-C1 s1-s2 s2-s3
          show ?thesis
            by simp
        qed
        moreover from True nrmAss-C2 s2-s3
        have nrm C2 ⊆ dom (locals (store s3))
          by - (rule subset-trans, simp-all)
        ultimately
        have ((brk C1) ⇒∪∀ (nrm C2)) l ⊆ ...
          by blast
        with brk-A show ?thesis
          by simp blast
      next
        case False
        note not-normal-s2 = this
        have s3=s2
        proof (cases normal (x1,s1))
          case True with not-normal-s2 s3 show ?thesis
            by (cases s2) (simp add: abrupt-if-def)
        next
          case False with not-normal-s2 s3 brk-s3 show ?thesis
            by (cases s2) (simp add: abrupt-if-def)
        qed
        with brkAss-C2 brk-s3
        have brk C2 l ⊆ dom (locals (store s3))
          by simp
        with brk-A show ?thesis
          by simp blast
      qed
    }
    hence ?BreakAssigned (Norm s0) s3 A
      by simp
    moreover
    {
      assume abr-s3: abrupt s3 = Some (Jump Ret)
      have Result ∈ dom (locals (store s3))
      proof (cases x1 = Some (Jump Ret))
        case True
        note ret-x1 = this
```

```
      with resAss-s1 have res-s1: Result ∈ dom (locals s1)
        by simp
      moreover have dom (locals (store ((Norm s1)::state)))
                       ⊆ dom (locals (store s2))
        by (rule dom-locals-eval-mono-elim) (rule Fin.hyps)
      ultimately have Result ∈ dom (locals (store s2))
        by − (rule subsetD,auto)
      with res-s1 s3 show ?thesis
        by simp
    next
      case False
      with s3 abr-s3 obtain  abrupt s2 = Some (Jump Ret) and s3=s2
        by (cases s2) (simp add: abrupt-if-def)
      with resAss-s2 show ?thesis
        by simp
    qed
  }
  hence ?ResAssigned (Norm s0) s3
    by simp
  ultimately show ?case by (intro conjI)
next
  case (Init C c s0 s3 s1 s2 Env T A)
  note G = ‹prg Env = G›
  from Init.hyps
  have eval: prg Env⊢ Norm s0 −Init C→ s3
    apply (simp only: G)
    apply (rule eval.Init, assumption)
    apply (cases inited C (globs s0) )
    apply   simp
    apply (simp only: if-False )
    apply (elim conjE,intro conjI,assumption+,simp)
    done
  from Init.prems and ‹the (class G C) = c›
  have c: class G C = Some c
    by (elim wt-elim-cases) auto
  from Init.prems obtain
     nrm-A: nrm A = dom (locals (store ((Norm s0)::state)))
    by (elim da-elim-cases) simp
  show ?case
  proof (cases inited C (globs s0))
    case True
    with Init.hyps have s3=Norm s0 by simp
    thus ?thesis
      using nrm-A by simp
  next
    case False
    from Init.hyps False G
    obtain eval-initC:
            prg Env⊢Norm ((init-class-obj G C) s0)
                     −(if C = Object then Skip else Init (super c))→ s1 and
          eval-init: prg Env⊢(set-lvars Map.empty) s1 −init c→ s2 and
          s3: s3=(set-lvars (locals (store s1))) s2
      by simp
    have ?NormalAssigned s3 A
    proof
      show nrm A ⊆ dom (locals (store s3))
      proof −
        from nrm-A have nrm A ⊆ dom (locals (init-class-obj G C s0))
          by simp
```

```
        also from eval-initC have ... ⊆ dom (locals (store s1))
          by (rule dom-locals-eval-mono-elim) simp
        also from s3 have ... ⊆ dom (locals (store s3))
          by (cases s1) (cases s2, simp add: init-lvars-def2)
        finally show ?thesis .
      qed
    qed
    moreover
    from eval
    have ⋀ j. abrupt s3 ≠ Some (Jump j)
      by (rule eval-statement-no-jump) (auto simp add: wf c G)
    then obtain ?BreakAssigned (Norm s0) s3 A
          and ?ResAssigned (Norm s0) s3
      by simp
    ultimately show ?thesis by (intro conjI)
  qed
next
  case (NewC s0 C s1 a s2 Env T A)
  note G = ‹prg Env = G›
  from NewC.prems
  obtain A: nrm A = dom (locals (store ((Norm s0)::state)))
            brk A = (λ l. UNIV)
    by (elim da-elim-cases) simp
  from wf NewC.prems
  have wt-init: Env⊢(Init C)::√
    by  (elim wt-elim-cases) (drule is-acc-classD,simp)
  have dom (locals (store ((Norm s0)::state))) ⊆ dom (locals (store s2))
  proof −
    have dom (locals (store ((Norm s0)::state))) ⊆ dom (locals (store s1))
      by (rule dom-locals-eval-mono-elim) (rule NewC.hyps)
    also
    have ... ⊆ dom (locals (store s2))
      by (rule dom-locals-halloc-mono) (rule NewC.hyps)
    finally show ?thesis .
  qed
  with A have ?NormalAssigned s2 A
    by simp
  moreover
  {
    fix j have abrupt s2 ≠ Some (Jump j)
    proof −
      have eval: prg Env⊢ Norm s0 −NewC C−≻Addr a→ s2
        unfolding G by (rule eval.NewC NewC.hyps)+
      from NewC.prems
      obtain T' where  T=Inl T'
        by (elim wt-elim-cases) simp
      with NewC.prems have Env⊢NewC C::−T'
        by simp
      from eval - this
      show ?thesis
        by (rule eval-expression-no-jump) (simp-all add: G wf)
    qed
  }
  hence ?BreakAssigned (Norm s0) s2 A and ?ResAssigned (Norm s0) s2
    by simp-all
  ultimately show ?case by (intro conjI)
next
  case (NewA s0 elT s1 e i s2 a s3 Env T A)
  note G = ‹prg Env = G›
```

```
from NewA.prems obtain
  da-e: Env⊢ dom (locals (store ((Norm s0)::state))) »⟨e⟩» A
  by (elim da-elim-cases)
from NewA.prems obtain
  wt-init: Env⊢init-comp-ty elT::√ and
  wt-size: Env⊢e::−PrimT Integer
  by (elim wt-elim-cases) (auto dest:  wt-init-comp-ty')
note halloc = ‹G⊢abupd (check-neg i) s2−halloc Arr elT (the-Intg i)≻a→s3›
have dom (locals (store ((Norm s0)::state))) ⊆ dom (locals (store s1))
  by (rule dom-locals-eval-mono-elim) (rule NewA.hyps)
with da-e obtain A' where
          da-e': Env⊢ dom (locals (store s1)) »⟨e⟩» A'
    and  nrm-A-A': nrm A ⊆ nrm A'
    and  brk-A-A': ∀ l. brk A l ⊆ brk A' l
  by (rule da-weakenE) simp
note ‹PROP ?Hyp (In1l e) s1 s2›
with wt-size da-e' G obtain
  nrmAss-A': ?NormalAssigned s2 A' and
  brkAss-A': ?BreakAssigned s1 s2 A'
  by simp
have s2-s3: dom (locals (store s2)) ⊆ dom (locals (store s3))
proof −
  have dom (locals (store s2))
        ⊆ dom (locals (store (abupd (check-neg i) s2)))
    by (simp)
  also have ... ⊆ dom (locals (store s3))
    by (rule dom-locals-halloc-mono) (rule NewA.hyps)
  finally show ?thesis .
qed
have ?NormalAssigned s3 A
proof
  assume normal-s3: normal s3
  show nrm A ⊆ dom (locals (store s3))
  proof −
    from halloc normal-s3
    have normal (abupd (check-neg i) s2)
      by cases simp-all
    hence normal s2
      by (cases s2) simp
    with nrmAss-A' nrm-A-A' s2-s3 show ?thesis
      by blast
  qed
qed
moreover
{
  fix j have abrupt s3 ≠ Some (Jump j)
  proof −
    have eval: prg Env⊢ Norm s0 −New elT[e]−≻Addr a→ s3
      unfolding G by (rule eval.NewA NewA.hyps)+
    from NewA.prems
    obtain T' where  T=Inl T'
      by (elim wt-elim-cases) simp
    with NewA.prems have Env⊢New elT[e]::−T'
      by simp
    from eval - this
    show ?thesis
      by (rule eval-expression-no-jump) (simp-all add: G wf)
  qed
}
```

```
    hence ?BreakAssigned (Norm s0) s3 A and ?ResAssigned (Norm s0) s3
      by simp-all
    ultimately show ?case by (intro conjI)
  next
    case (Cast s0 e v s1 s2 cT Env T A)
    note G = ‹prg Env = G›
    from Cast.prems obtain
      da-e: Env⊢ dom (locals (store ((Norm s0)::state))) »⟨e⟩» A
      by (elim da-elim-cases)
    from Cast.prems obtain eT where
      wt-e: Env⊢e::−eT
      by (elim wt-elim-cases)
    note ‹PROP ?Hyp (In1l e) (Norm s0) s1›
    with wt-e da-e G obtain
      nrmAss-A: ?NormalAssigned s1 A and
      brkAss-A: ?BreakAssigned (Norm s0) s1 A
      by simp
    note s2 = ‹s2 = abupd (raise-if (¬ G,snd s1⊢v fits cT) ClassCast) s1›
    hence s1-s2: dom (locals (store s1)) ⊆ dom (locals (store s2))
      by simp
    have ?NormalAssigned s2 A
    proof
      assume normal s2
      with s2 have normal s1
        by (cases s1) simp
      with nrmAss-A s1-s2
      show nrm A ⊆ dom (locals (store s2))
        by blast
    qed
    moreover
    {
      fix j have abrupt s2 ≠ Some (Jump j)
      proof −
        have eval: prg Env⊢ Norm s0 −Cast cT e−≻v→ s2
          unfolding G by (rule eval.Cast Cast.hyps)+
        from Cast.prems
        obtain T' where  T=Inl T'
          by (elim wt-elim-cases) simp
        with Cast.prems have Env⊢Cast cT e::−T'
          by simp
        from eval - this
        show ?thesis
          by (rule eval-expression-no-jump) (simp-all add: G wf)
      qed
    }
    hence ?BreakAssigned (Norm s0) s2 A and ?ResAssigned (Norm s0) s2
      by simp-all
    ultimately show ?case by (intro conjI)
  next
    case (Inst s0 e v s1 b iT Env T A)
    note G = ‹prg Env = G›
    from Inst.prems obtain
      da-e: Env⊢ dom (locals (store ((Norm s0)::state))) »⟨e⟩» A
      by (elim da-elim-cases)
    from Inst.prems obtain eT where
      wt-e: Env⊢e::−eT
      by (elim wt-elim-cases)
    note ‹PROP ?Hyp (In1l e) (Norm s0) s1›
    with wt-e da-e G obtain
```

```
    ?NormalAssigned s1 A and
    ?BreakAssigned (Norm s0) s1 A and
    ?ResAssigned (Norm s0) s1
    by simp
  thus ?case by (intro conjI)
next
  case (Lit s v Env T A)
  from Lit.prems
  have nrm A = dom (locals (store ((Norm s)::state)))
    by (elim da-elim-cases) simp
  thus ?case by simp
next
  case (UnOp s0 e v s1 unop Env T A)
  note G = ‹prg Env = G›
  from UnOp.prems obtain
    da-e: Env⊢ dom (locals (store ((Norm s0)::state))) »⟨e⟩» A
    by (elim da-elim-cases)
  from UnOp.prems obtain eT where
    wt-e: Env⊢e::−eT
    by (elim wt-elim-cases)
  note ‹PROP ?Hyp (In1l e) (Norm s0) s1›
  with wt-e da-e G obtain
    ?NormalAssigned s1 A and
    ?BreakAssigned (Norm s0) s1 A and
    ?ResAssigned (Norm s0) s1
    by simp
  thus ?case by (intro conjI)
next
  case (BinOp s0 e1 v1 s1 binop e2 v2 s2 Env T A)
  note G = ‹prg Env = G›
  from BinOp.hyps
  have
    eval: prg Env⊢Norm s0 −BinOp binop e1 e2−≻(eval-binop binop v1 v2)→ s2
    by (simp only: G) (rule eval.BinOp)
  have s0-s1: dom (locals (store ((Norm s0)::state)))
              ⊆ dom (locals (store s1))
    by (rule dom-locals-eval-mono-elim) (rule BinOp)
  also have s1-s2: dom (locals (store s1)) ⊆  dom (locals (store s2))
    by (rule dom-locals-eval-mono-elim) (rule BinOp)
  finally
  have s0-s2: dom (locals (store ((Norm s0)::state)))
              ⊆ dom (locals (store s2)) .
  from BinOp.prems obtain e1T e2T
    where wt-e1: Env⊢e1::−e1T
     and  wt-e2: Env⊢e2::−e2T
     and  wt-binop: wt-binop (prg Env) binop e1T e2T
     and  T: T=Inl (PrimT (binop-type binop))
    by (elim wt-elim-cases) simp
  have ?NormalAssigned s2 A
  proof
    assume normal-s2: normal s2
    have   normal-s1: normal s1
      by (rule eval-no-abrupt-lemma [rule-format]) (rule BinOp.hyps, rule normal-s2)
    show nrm A ⊆ dom (locals (store s2))
    proof (cases binop=CondAnd)
      case True
      note CondAnd = this
      from BinOp.prems obtain
         nrm-A: nrm A = dom (locals (store ((Norm s0)::state)))
```

```
                    ∪ (assigns-if True (BinOp CondAnd e1 e2) ∩
                        assigns-if False (BinOp CondAnd e1 e2))
    by (elim da-elim-cases) (simp-all add: CondAnd)
  from T BinOp.prems CondAnd
  have Env⊢BinOp binop e1 e2::−PrimT Boolean
    by (simp)
  with eval normal-s2
  have ass-if: assigns-if (the-Bool (eval-binop binop v1 v2))
                      (BinOp binop e1 e2)
                ⊆ dom (locals (store s2))
    by (rule assigns-if-good-approx)
  have (assigns-if True (BinOp CondAnd e1 e2) ∩
                assigns-if False (BinOp CondAnd e1 e2)) ⊆ ...
  proof (cases the-Bool (eval-binop binop v1 v2))
    case True
    with ass-if CondAnd
    have assigns-if True (BinOp CondAnd e1 e2)
           ⊆ dom (locals (store s2))
      by simp
    thus ?thesis by blast
  next
    case False
    with ass-if CondAnd
    have assigns-if False (BinOp CondAnd e1 e2)
           ⊆ dom (locals (store s2))
      by (simp only: False)
    thus ?thesis by blast
  qed
  with s0-s2
  have dom (locals (store ((Norm s0)::state)))
          ∪ (assigns-if True (BinOp CondAnd e1 e2) ∩
             assigns-if False (BinOp CondAnd e1 e2)) ⊆ ...
    by (rule Un-least)
  thus ?thesis by (simp only: nrm-A)
next
  case False
  note notCondAnd = this
  show ?thesis
  proof (cases binop=CondOr)
    case True
    note CondOr = this
    from BinOp.prems obtain
      nrm-A: nrm A = dom (locals (store ((Norm s0)::state)))
                    ∪ (assigns-if True (BinOp CondOr e1 e2) ∩
                        assigns-if False (BinOp CondOr e1 e2))
      by (elim da-elim-cases) (simp-all add: CondOr)
    from T BinOp.prems CondOr
    have Env⊢BinOp binop e1 e2::−PrimT Boolean
      by (simp)
    with eval normal-s2
    have ass-if: assigns-if (the-Bool (eval-binop binop v1 v2))
                        (BinOp binop e1 e2)
                  ⊆ dom (locals (store s2))
      by (rule assigns-if-good-approx)
    have (assigns-if True (BinOp CondOr e1 e2) ∩
                  assigns-if False (BinOp CondOr e1 e2)) ⊆ ...
    proof (cases the-Bool (eval-binop binop v1 v2))
      case True
      with ass-if CondOr
```

```
        have assigns-if True (BinOp CondOr e1 e2)
                ⊆ dom (locals (store s2))
          by (simp)
        thus ?thesis by blast
      next
        case False
        with ass-if CondOr
        have assigns-if False (BinOp CondOr e1 e2)
                ⊆ dom (locals (store s2))
          by (simp)
        thus ?thesis by blast
      qed
      with s0-s2
      have dom (locals (store ((Norm s0)::state)))
              ∪ (assigns-if True (BinOp CondOr e1 e2) ∩
                 assigns-if False (BinOp CondOr e1 e2)) ⊆ . . .
        by (rule Un-least)
      thus ?thesis by (simp only: nrm-A)
    next
      case False
      with notCondAnd obtain notAndOr: binop≠CondAnd binop≠CondOr
        by simp
      from BinOp.prems obtain E1
        where da-e1: Env⊢ dom (locals (snd (Norm s0))) »⟨e1⟩» E1
         and  da-e2: Env⊢ nrm E1 »⟨e2⟩» A
        by (elim da-elim-cases) (simp-all add: notAndOr)
      note ‹PROP ?Hyp (In1l e1) (Norm s0) s1›
      with wt-e1 da-e1 G normal-s1
      obtain ?NormalAssigned s1 E1
        by simp
      with normal-s1 have nrm E1 ⊆ dom (locals (store s1)) by iprover
      with da-e2 obtain A'
        where  da-e2': Env⊢ dom (locals (store s1)) »⟨e2⟩» A' and
             nrm-A-A': nrm A ⊆ nrm A'
        by (rule da-weakenE) iprover
      from notAndOr have need-second-arg binop v1 by simp
      with BinOp.hyps
      have PROP ?Hyp (In1l e2) s1 s2 by simp
      with wt-e2 da-e2' G
      obtain ?NormalAssigned s2 A'
        by simp
      with nrm-A-A' normal-s2
      show nrm A ⊆ dom (locals (store s2))
        by blast
    qed
  qed
qed
moreover
{
  fix j have abrupt s2 ≠ Some (Jump j)
  proof −
    from BinOp.prems T
    have Env⊢In1l (BinOp binop e1 e2)::Inl (PrimT (binop-type binop))
      by simp
    from eval - this
    show ?thesis
      by (rule eval-expression-no-jump) (simp-all add: G wf)
  qed
}
```

```
    hence ?BreakAssigned (Norm s0) s2 A and ?ResAssigned (Norm s0) s2
      by simp-all
    ultimately show ?case by (intro conjI)
  next
    case (Super s Env T A)
    from Super.prems
    have nrm A = dom (locals (store ((Norm s)::state)))
      by (elim da-elim-cases) simp
    thus ?case by simp
  next
    case (Acc s0 v w upd s1 Env T A)
    show ?case
    proof (cases ∃ vn. v = LVar vn)
      case True
      then obtain vn where vn: v=LVar vn..
      from Acc.prems
      have nrm A = dom (locals (store ((Norm s0)::state)))
        by (simp only: vn) (elim da-elim-cases,simp-all)
      moreover
      from ‹G⊢Norm s0 −v=≻(w, upd)→ s1›
      have s1=Norm s0
        by (simp only: vn) (elim eval-elim-cases,simp)
      ultimately show ?thesis by simp
    next
      case False
      note G = ‹prg Env = G›
      from False Acc.prems
      have da-v: Env⊢ dom (locals (store ((Norm s0)::state))) »⟨v⟩» A
        by (elim da-elim-cases) simp-all
      from Acc.prems obtain vT where
        wt-v: Env⊢v::=vT
        by (elim wt-elim-cases)
      note ‹PROP ?Hyp (In2 v) (Norm s0) s1›
      with wt-v da-v G obtain
        ?NormalAssigned s1 A and
        ?BreakAssigned (Norm s0) s1 A and
        ?ResAssigned (Norm s0) s1
        by simp
      thus ?thesis by (intro conjI)
    qed
  next
    case (Ass s0 var w upd s1 e v s2 Env T A)
    note G = ‹prg Env = G›
    from Ass.prems obtain varT eT where
      wt-var: Env⊢var::=varT and
      wt-e:   Env⊢e::−eT
      by (elim wt-elim-cases) simp
    have eval-var: prg Env⊢Norm s0 −var=≻(w, upd)→ s1
      using Ass.hyps by (simp only: G)
    have ?NormalAssigned (assign upd v s2) A
    proof
      assume normal-ass-s2: normal (assign upd v s2)
      from normal-ass-s2
      have normal-s2: normal s2
        by (cases s2) (simp add: assign-def Let-def)
      hence normal-s1: normal s1
        by − (rule eval-no-abrupt-lemma [rule-format], rule Ass.hyps)
      note hyp-var = ‹PROP ?Hyp (In2 var) (Norm s0) s1›
      note hyp-e = ‹PROP ?Hyp (In1l e) s1 s2›
```

```
show nrm A ⊆ dom (locals (store (assign upd v s2)))
proof (cases ∃ vn. var = LVar vn)
  case True
  then obtain vn where vn: var=LVar vn..
  from Ass.prems obtain E where
    da-e: Env⊢ dom (locals (store ((Norm s0)::state))) »⟨e⟩» E and
    nrm-A: nrm A = nrm E ∪ {vn}
    by (elim da-elim-cases) (insert vn,auto)
  obtain E' where
    da-e': Env⊢ dom (locals (store s1)) »⟨e⟩» E' and
    E-E': nrm E ⊆ nrm E'
  proof −
    have dom (locals (store ((Norm s0)::state)))
            ⊆ dom (locals (store s1))
      by (rule dom-locals-eval-mono-elim) (rule Ass.hyps)
    with da-e show thesis
      by (rule da-weakenE) (rule that)
  qed
  from G eval-var vn
  have eval-lvar: G⊢Norm s0 −LVar vn=≻(w, upd)→ s1
    by simp
  then have upd: upd = snd (lvar vn (store s1))
    by cases (cases lvar vn (store s1),simp)
  have nrm E ⊆ dom (locals (store (assign upd v s2)))
  proof −
    from hyp-e wt-e da-e' G normal-s2
    have nrm E' ⊆ dom (locals (store s2))
      by simp
    also
    from upd
    have dom (locals (store s2))  ⊆ dom (locals (store (upd v s2)))
      by (simp add: lvar-def) blast
    hence dom (locals (store s2))
             ⊆ dom (locals (store (assign upd v s2)))
      by (rule dom-locals-assign-mono)
    finally
    show ?thesis using E-E'
      by blast
  qed
  moreover
  from upd normal-s2
  have {vn} ⊆ dom (locals (store (assign upd v s2)))
    by (auto simp add: assign-def Let-def lvar-def upd split: prod.split)
  ultimately
  show nrm A ⊆ . . .
    by (rule Un-least [elim-format]) (simp add: nrm-A)
next
  case False
  from Ass.prems obtain V where
    da-var: Env⊢ dom (locals (store ((Norm s0)::state))) »⟨var⟩» V and
    da-e:   Env⊢ nrm V »⟨e⟩» A
    by (elim da-elim-cases) (insert False,simp+)
  from hyp-var wt-var da-var G normal-s1
  have nrm V ⊆ dom (locals (store s1))
    by simp
  with da-e obtain A'
    where  da-e': Env⊢ dom (locals (store s1)) »⟨e⟩» A' and
           nrm-A-A': nrm A ⊆ nrm A'
    by (rule da-weakenE) iprover
```

```
      from hyp-e wt-e da-e' G normal-s2
      obtain nrm A' ⊆ dom (locals (store s2))
        by simp
      with nrm-A-A' have nrm A ⊆ . . .
        by blast
      also have . . . ⊆ dom (locals (store (assign upd v s2)))
      proof −
        from eval-var normal-s1
        have dom (locals (store s2)) ⊆ dom (locals (store (upd v s2)))
          by (cases rule: dom-locals-eval-mono-elim)
             (cases s2, simp)
        thus ?thesis
          by (rule dom-locals-assign-mono)
      qed
      finally show ?thesis .
    qed
  qed
  moreover
  {
    fix j have abrupt (assign upd v s2) ≠ Some (Jump j)
    proof −
      have eval: prg Env⊢Norm s0 −var:=e−≻v→ (assign upd v s2)
        by (simp only: G) (rule eval.Ass Ass.hyps)+
      from Ass.prems
      obtain T' where T=Inl T'
        by (elim wt-elim-cases) simp
      with Ass.prems have Env⊢var:=e::−T' by simp
      from eval - this
      show ?thesis
        by (rule eval-expression-no-jump) (simp-all add: G wf)
    qed
  }
  hence ?BreakAssigned (Norm s0) (assign upd v s2) A
    and ?ResAssigned (Norm s0) (assign upd v s2)
    by simp-all
  ultimately show ?case by (intro conjI)
next
  case (Cond s0 e0 b s1 e1 e2 v s2 Env T A)
  note G = ‹prg Env = G›
  have ?NormalAssigned s2 A
  proof
    assume normal-s2: normal s2
    show nrm A ⊆ dom (locals (store s2))
    proof (cases Env⊢(e0 ? e1 : e2)::−(PrimT Boolean))
      case True
      with Cond.prems
      have nrm-A: nrm A = dom (locals (store ((Norm s0)::state)))
                          ∪ (assigns-if True (e0 ? e1 : e2) ∩
                              assigns-if False (e0 ? e1 : e2))
        by (elim da-elim-cases) simp-all
      have eval: prg Env⊢Norm s0 −(e0 ? e1 : e2)−≻v→ s2
        unfolding G by (rule eval.Cond Cond.hyps)+
      from eval
      have dom (locals (store ((Norm s0)::state)))⊆dom (locals (store s2))
        by (rule dom-locals-eval-mono-elim)
      moreover
      from eval normal-s2 True
      have ass-if: assigns-if (the-Bool v) (e0 ? e1 : e2)
                    ⊆ dom (locals (store s2))
```

```
      by (rule assigns-if-good-approx)
    have assigns-if True  (e0 ? e1:e2) ∩ assigns-if False (e0 ? e1:e2)
           ⊆ dom (locals (store s2))
    proof (cases the-Bool v)
      case True
      from ass-if
      have assigns-if True  (e0 ? e1:e2) ⊆ dom (locals (store s2))
        by (simp only: True)
      thus ?thesis by blast
    next
      case False
      from ass-if
      have assigns-if False  (e0 ? e1:e2) ⊆ dom (locals (store s2))
        by (simp only: False)
      thus ?thesis by blast
    qed
    ultimately show nrm A ⊆ dom (locals (store s2))
      by (simp only: nrm-A) (rule Un-least)
  next
    case False
    with Cond.prems obtain E1 E2 where
     da-e1: Env⊢ (dom (locals (store ((Norm s0)::state)))
                    ∪ assigns-if True e0) »⟨e1⟩» E1 and
     da-e2: Env⊢ (dom (locals (store ((Norm s0)::state)))
                   ∪ assigns-if False e0) »⟨e2⟩» E2 and
     nrm-A: nrm A = nrm E1 ∩ nrm E2
      by (elim da-elim-cases) simp-all
    from Cond.prems obtain e1T e2T where
      wt-e0: Env⊢e0::− PrimT Boolean and
      wt-e1: Env⊢e1::−e1T and
      wt-e2: Env⊢e2::−e2T
      by (elim wt-elim-cases)
    have s0-s1: dom (locals (store ((Norm s0)::state)))
                  ⊆ dom (locals (store s1))
      by (rule dom-locals-eval-mono-elim) (rule Cond.hyps)
    have eval-e0: prg Env⊢Norm s0 −e0−≻b→ s1
      unfolding G by (rule Cond.hyps)
    have normal-s1: normal s1
      by (rule eval-no-abrupt-lemma [rule-format]) (rule Cond.hyps, rule normal-s2)
    show ?thesis
    proof (cases the-Bool b)
      case True
      from True Cond.hyps have PROP ?Hyp (In1l e1) s1 s2 by simp
      moreover
      from eval-e0 normal-s1 wt-e0
      have assigns-if True e0 ⊆ dom (locals (store s1))
        by (rule assigns-if-good-approx [elim-format]) (simp only: True)
      with s0-s1
      have dom (locals (store ((Norm s0)::state)))
             ∪ assigns-if True e0 ⊆ . . .
        by (rule Un-least)
      with da-e1 obtain E1' where
             da-e1': Env⊢ dom (locals (store s1)) »⟨e1⟩» E1' and
          nrm-E1-E1': nrm E1 ⊆ nrm E1'
        by (rule da-weakenE) iprover
      ultimately have nrm E1' ⊆ dom (locals (store s2))
        using wt-e1 G normal-s2 by simp
      with nrm-E1-E1' show ?thesis
        by (simp only: nrm-A) blast
```

```
    next
      case False
      from False Cond.hyps have PROP ?Hyp (In1l e2) s1 s2 by simp
      moreover
      from eval-e0 normal-s1 wt-e0
      have assigns-if False e0 ⊆ dom (locals (store s1))
        by (rule assigns-if-good-approx [elim-format]) (simp only: False)
      with s0-s1
      have dom (locals (store ((Norm s0)::state)))
             ∪ assigns-if False e0 ⊆ . . .
        by (rule Un-least)
      with da-e2 obtain E2' where
             da-e2': Env⊢ dom (locals (store s1)) »⟨e2⟩» E2' and
          nrm-E2-E2': nrm E2 ⊆ nrm E2'
        by (rule da-weakenE) iprover
      ultimately have nrm E2' ⊆ dom (locals (store s2))
        using wt-e2 G normal-s2 by simp
      with nrm-E2-E2' show ?thesis
        by (simp only: nrm-A) blast
    qed
  qed
qed
moreover
{
  fix j have abrupt s2 ≠ Some (Jump j)
  proof −
    have eval: prg Env⊢Norm s0 −e0 ? e1 : e2−≻v→ s2
      unfolding G by (rule eval.Cond Cond.hyps)+
    from Cond.prems
    obtain T' where  T=Inl T'
      by (elim wt-elim-cases) simp
    with Cond.prems have Env⊢e0 ? e1 : e2::−T' by simp
    from eval - this
    show ?thesis
      by (rule eval-expression-no-jump) (simp-all add: G wf)
  qed
}
hence ?BreakAssigned (Norm s0) s2 A and ?ResAssigned (Norm s0) s2
  by simp-all
ultimately show ?case by (intro conjI)
next
case (Call s0 e a s1 args vs s2 D mode statT mn pTs s3 s3' accC v s4
       Env T A)
note G = ‹prg Env = G›
have ?NormalAssigned (restore-lvars s2 s4) A
proof
  assume normal-restore-lvars: normal (restore-lvars s2 s4)
  show nrm A ⊆ dom (locals (store (restore-lvars s2 s4)))
  proof −
    from Call.prems obtain E where
         da-e: Env⊢ (dom (locals (store ((Norm s0)::state))))»⟨e⟩» E and
      da-args: Env⊢ nrm E »⟨args⟩» A
      by (elim da-elim-cases)
    from Call.prems obtain eT argsT where
         wt-e: Env⊢e::−eT and
      wt-args: Env⊢args::≐argsT
      by (elim wt-elim-cases)
    note s3 = ‹s3 = init-lvars G D (|name = mn, parTs = pTs|) mode a vs s2›
    note s3' = ‹s3' = check-method-access G accC statT mode
```

```
                                    (|name=mn,parTs=pTs|) a s3›
      have normal-s2: normal s2
      proof –
        from normal-restore-lvars have normal s4
          by simp
        then have normal s3′
          by – (rule eval-no-abrupt-lemma [rule-format], rule Call.hyps)
        with s3′ have normal s3
          by (cases s3) (simp add: check-method-access-def Let-def)
        with s3 show normal s2
          by (cases s2) (simp add: init-lvars-def Let-def)
      qed
      then have normal-s1: normal s1
        by  – (rule eval-no-abrupt-lemma [rule-format], rule Call.hyps)
      note ‹PROP ?Hyp (In1l e) (Norm s0) s1›
      with da-e wt-e G normal-s1
      have nrm E ⊆ dom (locals (store s1))
        by simp
      with da-args obtain A′ where
        da-args′: Env⊢ dom (locals (store s1)) »⟨args⟩» A′ and
        nrm-A-A′: nrm A ⊆ nrm A′
        by (rule da-weakenE) iprover
      note ‹PROP ?Hyp (In3 args) s1 s2›
      with da-args′ wt-args G normal-s2
      have nrm A′ ⊆ dom (locals (store s2))
        by simp
      with nrm-A-A′ have nrm A ⊆ dom (locals (store s2))
        by blast
      also have . . . ⊆ dom (locals (store (restore-lvars s2 s4)))
        by (cases s4) simp
      finally show ?thesis .
    qed
  qed
  moreover
  {
    fix j have abrupt (restore-lvars s2 s4) ≠ Some (Jump j)
    proof –
      have eval: prg Env⊢Norm s0 –({accC,statT,mode}e·mn( {pTs}args))–≻v
                          → (restore-lvars s2 s4)
        unfolding G by (rule eval.Call Call)+
      from Call.prems
      obtain T′ where  T=Inl T′
        by (elim wt-elim-cases) simp
      with Call.prems have Env⊢({accC,statT,mode}e·mn( {pTs}args))::–T′
        by simp
      from eval - this
      show ?thesis
        by (rule eval-expression-no-jump) (simp-all add: G wf)
    qed
  }
  hence ?BreakAssigned (Norm s0) (restore-lvars s2 s4) A
    and ?ResAssigned (Norm s0) (restore-lvars s2 s4)
    by simp-all
  ultimately show ?case by (intro conjI)
next
  case (Methd s0 D sig v s1 Env T A)
  note G = ‹prg Env = G›
  from Methd.prems obtain m where
     m:       methd (prg Env) D sig = Some m and
```

```
      da-body: Env⊢(dom (locals (store ((Norm s0)::state))))
                 »⟨Body (declclass m) (stmt (mbody (mthd m)))⟩» A
    by − (erule da-elim-cases)
  from Methd.prems m obtain
    isCls: is-class (prg Env) D and
    wt-body: Env ⊢In1l (Body (declclass m) (stmt (mbody (mthd m)))):: T
    by − (erule wt-elim-cases,simp)
  note ‹PROP ?Hyp (In1l (body G D sig)) (Norm s0) s1›
  moreover
  from wt-body have Env⊢In1l (body G D sig):: T
    using isCls m G by (simp add: body-def2)
  moreover
  from da-body have Env⊢(dom (locals (store ((Norm s0)::state))))
                      »⟨body G D sig⟩» A
    using isCls m G by (simp add: body-def2)
  ultimately show ?case
    using G by simp
next
  case (Body s0 D s1 c s2 s3 Env T A)
  note G = ‹prg Env = G›
  from Body.prems
  have nrm-A: nrm A = dom (locals (store ((Norm s0)::state)))
    by (elim da-elim-cases) simp
  have eval: prg Env⊢Norm s0 −Body D c−≻the (locals (store s2) Result)
                     →abupd (absorb Ret) s3
    unfolding G by (rule eval.Body Body.hyps)+
  hence nrm A ⊆ dom (locals (store (abupd (absorb Ret) s3)))
    by  (simp only: nrm-A) (rule dom-locals-eval-mono-elim)
  hence ?NormalAssigned (abupd (absorb Ret) s3) A
    by simp
  moreover
  from eval have ⋀ j. abrupt (abupd (absorb Ret) s3) ≠ Some (Jump j)
    by (rule Body-no-jump) simp
  hence ?BreakAssigned (Norm s0) (abupd (absorb Ret) s3) A and
        ?ResAssigned (Norm s0) (abupd (absorb Ret) s3)
    by simp-all
  ultimately show ?case by (intro conjI)
next
  case (LVar s vn Env T A)
  from LVar.prems
  have nrm A = dom (locals (store ((Norm s)::state)))
    by (elim da-elim-cases) simp
  thus ?case by simp
next
  case (FVar s0 statDeclC s1 e a s2 v s2' stat fn s3 accC Env T A)
  note G = ‹prg Env = G›
  have ?NormalAssigned s3 A
  proof
    assume normal-s3: normal s3
    show nrm A ⊆ dom (locals (store s3))
    proof −
      note fvar = ‹(v, s2') = fvar statDeclC stat fn a s2› and
        s3 = ‹s3 = check-field-access G accC statDeclC fn stat a s2'›
      from FVar.prems
      have da-e: Env⊢ (dom (locals (store ((Norm s0)::state))))»⟨e⟩» A
        by (elim da-elim-cases)
      from FVar.prems obtain eT where
        wt-e: Env⊢e::−eT
        by (elim wt-elim-cases)
```

```
      have (dom (locals (store ((Norm s0)::state))))
             ⊆ dom (locals (store s1))
        by (rule dom-locals-eval-mono-elim) (rule FVar.hyps)
      with da-e obtain A' where
        da-e': Env⊢ dom (locals (store s1)) »⟨e⟩» A' and
        nrm-A-A': nrm A ⊆ nrm A'
        by (rule da-weakenE) iprover
      have normal-s2: normal s2
      proof −
        from normal-s3 s3
        have normal s2'
          by (cases s2') (simp add: check-field-access-def Let-def)
        with fvar
        show normal s2
          by (cases s2) (simp add: fvar-def2)
      qed
      note ‹PROP ?Hyp (In1l e) s1 s2›
      with da-e' wt-e G normal-s2
      have nrm A' ⊆ dom (locals (store s2))
        by simp
      with nrm-A-A' have nrm A ⊆ dom (locals (store s2))
        by blast
      also have ... ⊆ dom (locals (store s3))
      proof −
        from fvar have s2' = snd (fvar statDeclC stat fn a s2)
          by (cases fvar statDeclC stat fn a s2) simp
        hence dom (locals (store s2)) ⊆  dom (locals (store s2'))
          by (simp) (rule dom-locals-fvar-mono)
        also from s3 have ... ⊆ dom (locals (store s3))
          by (cases s2') (simp add: check-field-access-def Let-def)
        finally show ?thesis .
      qed
      finally show ?thesis .
    qed
  qed
  moreover
  {
    fix j have abrupt s3 ≠ Some (Jump j)
    proof −
      obtain w upd where v: (w,upd)=v
        by (cases v) auto
      have eval: prg Env⊢Norm s0−({accC,statDeclC,stat}e..fn)=≻(w,upd)→s3
        by (simp only: G v) (rule eval.FVar FVar.hyps)+
      from FVar.prems
      obtain T' where  T=Inl T'
        by (elim wt-elim-cases) simp
      with FVar.prems have Env⊢({accC,statDeclC,stat}e..fn)::=T'
        by simp
      from eval - this
      show ?thesis
        by (rule eval-var-no-jump [THEN conjunct1]) (simp-all add: G wf)
    qed
  }
  hence ?BreakAssigned (Norm s0) s3 A and ?ResAssigned (Norm s0) s3
    by simp-all
  ultimately show ?case by (intro conjI)
next
  case (AVar s0 e1 a s1 e2 i s2 v s2' Env T A)
  note G = ‹prg Env = G›
```

```
have ?NormalAssigned s2' A
proof
  assume normal-s2': normal s2'
  show nrm A ⊆ dom (locals (store s2'))
  proof −
    note avar = ‹(v, s2') = avar G i a s2›
    from AVar.prems obtain E1 where
      da-e1: Env⊢ (dom (locals (store ((Norm s0)::state))))»⟨e1⟩» E1 and
      da-e2: Env⊢ nrm E1 »⟨e2⟩» A
      by (elim da-elim-cases)
    from AVar.prems obtain e1T e2T where
        wt-e1: Env⊢e1::−e1T and
        wt-e2: Env⊢e2::−e2T
      by (elim wt-elim-cases)
    from avar normal-s2'
    have normal-s2: normal s2
      by (cases s2) (simp add: avar-def2)
    hence normal s1
      by − (rule eval-no-abrupt-lemma [rule-format], rule AVar, rule normal-s2)
    moreover note ‹PROP ?Hyp (In1l e1) (Norm s0) s1›
    ultimately have nrm E1 ⊆ dom (locals (store s1))
      using da-e1 wt-e1 G by simp
    with da-e2 obtain A' where
      da-e2': Env⊢ dom (locals (store s1)) »⟨e2⟩» A' and
      nrm-A-A': nrm A ⊆ nrm A'
      by (rule da-weakenE) iprover
    note ‹PROP ?Hyp (In1l e2) s1 s2›
    with da-e2' wt-e2 G normal-s2
    have nrm A' ⊆ dom (locals (store s2))
      by simp
    with nrm-A-A' have nrm A ⊆ dom (locals (store s2))
      by blast
    also have . . . ⊆ dom (locals (store s2'))
    proof −
       from avar have s2' = snd (avar G i a s2)
        by (cases (avar G i a s2)) simp
      thus dom (locals (store s2)) ⊆  dom (locals (store s2'))
        by (simp) (rule dom-locals-avar-mono)
    qed
    finally show ?thesis .
  qed
qed
moreover
{
  fix j have abrupt s2' ≠ Some (Jump j)
  proof −
    obtain w upd where v: (w,upd)=v
      by (cases v) auto
    have eval: prg Env⊢Norm s0−(e1.[e2])=≻(w,upd)→s2'
      unfolding G v by (rule eval.AVar AVar.hyps)+
    from AVar.prems
    obtain T' where  T=Inl T'
      by (elim wt-elim-cases) simp
    with AVar.prems have Env⊢(e1.[e2])::=T'
      by simp
    from eval - this
    show ?thesis
      by (rule eval-var-no-jump [THEN conjunct1]) (simp-all add: G wf)
  qed
```

```
    }
    hence ?BreakAssigned (Norm s0) s2′ A and ?ResAssigned (Norm s0) s2′
      by simp-all
    ultimately show ?case by (intro conjI)
  next
    case (Nil s0 Env T A)
    from Nil.prems
    have nrm A = dom (locals (store ((Norm s0)::state)))
      by (elim da-elim-cases) simp
    thus ?case by simp
  next
    case (Cons s0 e v s1 es vs s2 Env T A)
    note G = ‹prg Env = G›
    have ?NormalAssigned s2 A
    proof
      assume normal-s2: normal s2
      show nrm A ⊆ dom (locals (store s2))
      proof −
        from Cons.prems obtain E where
          da-e: Env⊢ (dom (locals (store ((Norm s0)::state))))»⟨e⟩» E and
          da-es: Env⊢ nrm E »⟨es⟩» A
          by (elim da-elim-cases)
        from Cons.prems obtain eT esT where
            wt-e: Env⊢e::−eT and
            wt-es: Env⊢es::≐esT
          by (elim wt-elim-cases)
        have normal s1
          by − (rule eval-no-abrupt-lemma [rule-format], rule Cons.hyps, rule normal-s2)
        moreover note ‹PROP ?Hyp (In1l e) (Norm s0) s1›
        ultimately have nrm E ⊆ dom (locals (store s1))
          using da-e wt-e G by simp
        with da-es obtain A′ where
          da-es′: Env⊢ dom (locals (store s1)) »⟨es⟩» A′ and
          nrm-A-A′: nrm A ⊆ nrm A′
          by (rule da-weakenE) iprover
        note ‹PROP ?Hyp (In3 es) s1 s2›
        with da-es′ wt-es G normal-s2
        have nrm A′ ⊆ dom (locals (store s2))
          by simp
        with nrm-A-A′ show nrm A ⊆ dom (locals (store s2))
          by blast
      qed
    qed
    moreover
    {
      fix j have abrupt s2 ≠ Some (Jump j)
      proof −
        have eval: prg Env⊢Norm s0−(e # es)≐≻v#vs→s2
          unfolding G by (rule eval.Cons Cons.hyps)+
        from Cons.prems
        obtain T′ where  T=Inr T′
          by (elim wt-elim-cases) simp
        with Cons.prems have Env⊢(e # es)::≐T′
          by simp
        from eval - this
        show ?thesis
          by (rule eval-expression-list-no-jump) (simp-all add: G wf)
      qed
    }
```

    hence *?BreakAssigned* (*Norm s0*) *s2 A* **and** *?ResAssigned* (*Norm s0*) *s2*
      by *simp-all*
    ultimately show *?case* **by** (*intro conjI*)
  qed
qed

lemma *da-good-approxE*:
  assumes
    prg Env⊢*s0* −*t*≻→ (*v*, *s1*) **and** *Env*⊢*t*::*T* **and**
    Env⊢ *dom* (*locals* (*store s0*)) »*t*» *A* **and** *wf-prog* (*prg Env*)
  obtains
    normal s1 ⟹ *nrm A* ⊆ *dom* (*locals* (*store s1*)) **and**
    ⋀ *l*. ⟦*abrupt s1* = *Some* (*Jump* (*Break l*)); *normal s0*⟧
        ⟹ *brk A l* ⊆ *dom* (*locals* (*store s1*)) **and**
    ⟦*abrupt s1* = *Some* (*Jump Ret*);*normal s0*⟧⟹*Result* ∈ *dom* (*locals* (*store s1*))
  using *assms* **by** − (*drule* (*3*) *da-good-approx*, *simp*)

lemma *da-good-approxE′*:
  assumes *eval*: *G*⊢*s0* −*t*≻→ (*v*, *s1*)
    and *wt*: (|*prg*=*G*,*cls*=*C*,*lcl*=*L*|)⊢*t*::*T*
    and *da*: (|*prg*=*G*,*cls*=*C*,*lcl*=*L*|)⊢ *dom* (*locals* (*store s0*)) »*t*» *A*
    and *wf*: *wf-prog G*
  obtains *normal s1* ⟹ *nrm A* ⊆ *dom* (*locals* (*store s1*)) **and**
    ⋀ *l*. ⟦*abrupt s1* = *Some* (*Jump* (*Break l*)); *normal s0*⟧
              ⟹ *brk A l* ⊆ *dom* (*locals* (*store s1*)) **and**
    ⟦*abrupt s1* = *Some* (*Jump Ret*);*normal s0*⟧
      ⟹ *Result* ∈ *dom* (*locals* (*store s1*))
proof −
  from *eval* **have** *prg* (|*prg*=*G*,*cls*=*C*,*lcl*=*L*|)⊢*s0* −*t*≻→ (*v*, *s1*) **by** *simp*
  moreover note *wt da*
  moreover from *wf* **have** *wf-prog* (*prg* (|*prg*=*G*,*cls*=*C*,*lcl*=*L*|)) **by** *simp*
  ultimately show *thesis*
    using *that* **by** (*rule da-good-approxE*) *iprover*+
qed

declare [[*simproc add*: *wt-expr wt-var wt-exprs wt-stmt*]]

end

Chapter 19

TypeSafe

1 The type soundness proof for Java

theory *TypeSafe*
imports *DefiniteAssignmentCorrect Conform*
begin

error free

lemma *error-free-halloc*:
  assumes *halloc*: $G\vdash s0\ -halloc\ oi\succ a\rightarrow\ s1$ **and**
        error-free-s0: *error-free s0*
  shows *error-free s1*
proof −
  from *halloc error-free-s0*
  obtain *abrupt0 store0 abrupt1 store1*
    where *eqs*: $s0=(abrupt0,store0)$ $s1=(abrupt1,store1)$ **and**
        halloc': $G\vdash(abrupt0,store0)\ -halloc\ oi\succ a\rightarrow\ (abrupt1,store1)$ **and**
        error-free-s0': *error-free* $(abrupt0,store0)$
    by (*cases s0*,*cases s1*) *auto*
  from *halloc' error-free-s0'*
  have *error-free* $(abrupt1,store1)$
  proof (*induct*)
    case *Abrupt*
    then show *?case* **.**
  next
    case *New*
    then show *?case*
      by *auto*
  qed
  with *eqs*
  show *?thesis*
    by *simp*
qed

lemma *error-free-sxalloc*:
  assumes *sxalloc*: $G\vdash s0\ -sxalloc\rightarrow\ s1$ **and** *error-free-s0*: *error-free s0*
  shows *error-free s1*
proof −
  from *sxalloc error-free-s0*
  obtain *abrupt0 store0 abrupt1 store1*
    where *eqs*: $s0=(abrupt0,store0)$ $s1=(abrupt1,store1)$ **and**
        sxalloc': $G\vdash(abrupt0,store0)\ -sxalloc\rightarrow\ (abrupt1,store1)$ **and**
        error-free-s0': *error-free* $(abrupt0,store0)$

by (*cases s0*,*cases s1*) *auto*
from *sxalloc′ error-free-s0′*
have *error-free* (*abrupt1*,*store1*)
proof (*induct*)
qed (*auto*)
with *eqs*
show *?thesis*
by *simp*
qed

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*
apply (*cases s*)
apply (*auto simp add*: *check-field-access-def Let-def error-free-def*
abrupt-if-def
split: *if-split-asm*)
done

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*
apply (*cases s*)
apply (*auto simp add*: *check-method-access-def Let-def error-free-def*
abrupt-if-def)
done

lemma *error-free-FVar-lemma*:
error-free s
$\Longrightarrow$ *error-free* (*abupd* (*if stat then id else np a*) *s*)
by (*case-tac s*) *auto*

lemma *error-free-init-lvars* [*simp*,*intro*]:
error-free s $\Longrightarrow$
error-free (*init-lvars G C sig mode a pvs s*)
by (*cases s*) (*auto simp add*: *init-lvars-def Let-def*)

lemma *error-free-LVar-lemma*:
error-free s $\Longrightarrow$ *error-free* (*assign* (λv. *supd lupd*(*vn*$\mapsto$*v*)) *w s*)
by (*cases s*) *simp*

lemma *error-free-throw* [*simp*,*intro*]:
error-free s $\Longrightarrow$ *error-free* (*abupd* (*throw x*) *s*)
by (*cases s*) (*simp add*: *throw-def*)

result conformance

definition
assign-conforms :: *st* $\Rightarrow$ (*val* $\Rightarrow$ *state* $\Rightarrow$ *state*) $\Rightarrow$ *ty* $\Rightarrow$ *env′* $\Rightarrow$ *bool* (-$\leq$|-$\preceq$-::$\preceq$- [*71*,*71*,*71*,*71*] *70*)
where
s$\leq$|*f*$\preceq$*T*::$\preceq$*E* =
(($\forall$ *s′ w*. *Norm s′*::$\preceq$*E* $\longrightarrow$ *fst E*,*s′*$\vdash$*w*::$\preceq$*T* $\longrightarrow$ *s*$\leq$|*s′* $\longrightarrow$ *assign f w* (*Norm s′*)::$\preceq$*E*) $\wedge$
($\forall$ *s′ w*. *error-free s′* $\longrightarrow$ (*error-free* (*assign f w s′*))))

definition
rconf :: *prog* ⇒ *lenv* ⇒ *st* ⇒ *term* ⇒ *vals* ⇒ *tys* ⇒ *bool* (-,-,-⊢-≻-::≼- [*71*,*71*,*71*,*71*,*71*,*71*] *70*)
where
G,*L*,*s*⊢*t*≻*v*::≼*T* =
(*case T of*
Inl T ⇒ *if* (∃ *var*. *t*=*In2 var*)
then (∀ *n*. (*the-In2 t*) = *LVar n*
⟶ (*fst* (*the-In2 v*) = *the* (*locals s n*)) ∧
(*locals s n* ≠ *None* ⟶ *G*,*s*⊢*fst* (*the-In2 v*)::≼*T*)) ∧
(¬ (∃ *n*. *the-In2 t*=*LVar n*) ⟶ (*G*,*s*⊢*fst* (*the-In2 v*)::≼*T*))∧
(*s*≤|*snd* (*the-In2 v*)≼*T*::≼(*G*,*L*))
else G,*s*⊢*the-In1 v*::≼*T*
| *Inr Ts* ⇒ *list-all2* (*conf G s*) (*the-In3 v*) *Ts*)

With *rconf* we describe the conformance of the result value of a term. This definition gets rather complicated because of the relations between the injections of the different terms, types and values. The main case distinction is between single values and value lists. In case of value lists, every value has to conform to its type. For single values we have to do a further case distinction, between values of variables ∃ *var*. *t* = *In2 var* and ordinary values. Values of variables are modelled as pairs consisting of the current value and an update function which will perform an assignment to the variable. This stems form the decision, that we only have one evaluation rule for each kind of variable. The decision if we read or write to the variable is made by syntactic enclosing rules. So conformance of variable-values must ensure that both the current value and an update will conform to the type. With the introduction of definite assignment of local variables we have to do another case distinction. For the notion of conformance local variables are allowed to be *None*, since the definedness is not ensured by conformance but by definite assignment. Field and array variables must contain a value.

lemma *rconf-In1* [*simp*]:
G,*L*,*s*⊢*In1 ec*≻*In1 v* ::≼*Inl T* = *G*,*s*⊢*v*::≼*T*
apply (*unfold rconf-def*)
apply (*simp* (*no-asm*))
done

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*))
apply (*unfold rconf-def*)
apply *auto*
done

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*))
apply (*unfold rconf-def*)
by *simp*

lemma *rconf-In3* [*simp*]:
G,*L*,*s*⊢*In3 es*≻*In3 vs*::≼*Inr Ts* = *list-all2* (λ*v T*. *G*,*s*⊢*v*::≼*T*) *vs Ts*
apply (*unfold rconf-def*)
apply (*simp* (*no-asm*))

done

fits and conf

lemma *conf-fits*: $G,s \vdash v::\preceq T \Longrightarrow G,s \vdash v$ *fits* T
apply (*unfold fits-def*)
apply *clarify*
apply (*erule contrapos-np*, *simp* (*no-asm-use*))
apply (*drule conf-RefTD*)
apply *auto*
done

lemma *fits-conf*:
$[\![G,s \vdash v::\preceq T;\ G \vdash T \preceq ?\ T';\ G,s \vdash v\ \mathit{fits}\ T';\ \mathit{ws\text{-}prog}\ G]\!] \Longrightarrow G,s \vdash v::\preceq T'$
apply (*auto dest*!: *fitsD cast-PrimT2 cast-RefT2*)
apply (*force dest*: *conf-RefTD intro*: *conf-AddrI*)
done

lemma *fits-Array*:
$[\![G,s \vdash v::\preceq T;\ G \vdash T'.[] \preceq T.[];\ G,s \vdash v\ \mathit{fits}\ T';\ \mathit{ws\text{-}prog}\ G]\!] \Longrightarrow G,s \vdash v::\preceq T'$
apply (*auto dest*!: *fitsD widen-ArrayPrimT widen-ArrayRefT*)
apply (*force dest*: *conf-RefTD intro*: *conf-AddrI*)
done

gext

lemma *halloc-gext*: $\bigwedge s1\ s2.\ G \vdash s1\ -\mathit{halloc}\ oi \succ a \rightarrow\ s2 \Longrightarrow \mathit{snd}\ s1 \leq| \mathit{snd}\ s2$
apply (*simp* (*no-asm-simp*) *only*: *split-tupled-all*)
apply (*erule halloc.induct*)
apply (*auto dest*!: *new-AddrD*)
done

lemma *sxalloc-gext*: $\bigwedge s1\ s2.\ G \vdash s1\ -\mathit{sxalloc} \rightarrow\ s2 \Longrightarrow \mathit{snd}\ s1 \leq| \mathit{snd}\ s2$
apply (*simp* (*no-asm-simp*) *only*: *split-tupled-all*)
apply (*erule sxalloc.induct*)
apply (*auto dest*!: *halloc-gext*)
done

lemma *eval-gext-lemma* [*rule-format* (*no-asm*)]:
$G \vdash s\ -t \succ \rightarrow\ (w,s') \Longrightarrow \mathit{snd}\ s \leq| \mathit{snd}\ s' \wedge (\mathit{case}\ w\ \mathit{of}$
$\quad In1\ v \Rightarrow True$
$\ \ |\ In2\ vf \Rightarrow \mathit{normal}\ s \longrightarrow (\forall\ v\ x\ s.\ s \leq| \mathit{snd}\ (\mathit{assign}\ (\mathit{snd}\ vf)\ v\ (x,s)))$
$\ \ |\ In3\ vs \Rightarrow True)$
apply (*erule eval-induct*)
prefer *26*
apply (*case-tac inited C* (*globs s0*), *clarsimp*, *erule thin-rl*)
apply (*auto del*: *conjI dest*!: *not-initedD gext-new sxalloc-gext halloc-gext*
simp add: *lvar-def fvar-def2 avar-def2 init-lvars-def2*
check-field-access-def check-method-access-def Let-def
split del: *if-split-asm split*: *sum3.split*)

apply *force+*
done

lemma *evar-gext-f*:
$G \vdash Norm\ s1\ -e{=}{\succ}vf \rightarrow s2 \Longrightarrow s{\leq}|snd\ (assign\ (snd\ vf)\ v\ (x,s))$
apply (*drule eval-gext-lemma* [*THEN conjunct2*])
apply *auto*
done

lemmas *eval-gext* = *eval-gext-lemma* [*THEN conjunct1*]

lemma *eval-gext'*: $G \vdash (x1,s1)\ -t{\succ}{\rightarrow}\ (w,(x2,s2)) \Longrightarrow s1{\leq}|s2$
apply (*drule eval-gext*)
apply *auto*
done

lemma *init-yields-initd*: $G \vdash Norm\ s1\ -Init\ C{\rightarrow}\ s2 \Longrightarrow initd\ C\ s2$
apply (*erule eval-cases* , *auto split del*: *if-split-asm*)
apply (*case-tac inited C* (*globs s1*))
apply (*clarsimp split del*: *if-split-asm*)+
apply (*drule eval-gext'*)+
apply (*drule init-class-obj-inited*)
apply (*erule inited-gext*)
apply (*simp* (*no-asm-use*))
done

Lemmas

lemma *obj-ty-obj-class1*:
$[\![wf\text{-}prog\ G;\ is\text{-}type\ G\ (obj\text{-}ty\ obj)]\!] \Longrightarrow is\text{-}class\ G\ (obj\text{-}class\ obj)$
apply (*case-tac tag obj*)
apply (*auto simp add*: *obj-ty-def obj-class-def*)
done

lemma *oconf-init-obj*:
$[\![wf\text{-}prog\ G;$
$(case\ r\ of\ Heap\ a \Rightarrow is\text{-}type\ G\ (obj\text{-}ty\ obj)\ |\ Stat\ C \Rightarrow is\text{-}class\ G\ C)$
$]\!] \Longrightarrow G,s \vdash obj\ (\!|values{:=}init\text{-}vals\ (var\text{-}tys\ G\ (tag\ obj)\ r)|\!){::}{\preceq}\sqrt{r}$
apply (*auto intro*!: *oconf-init-obj-lemma unique-fields*)
done

lemma *conforms-newG*: $[\![globs\ s\ oref = None;\ (x,\ s){::}{\preceq}(G,L);$
$wf\text{-}prog\ G;\ case\ oref\ of\ Heap\ a \Rightarrow is\text{-}type\ G\ (obj\text{-}ty\ (\!|tag{=}oi,values{=}vs|\!))$
$|\ Stat\ C \Rightarrow is\text{-}class\ G\ C]\!] \Longrightarrow$
$(x,\ init\text{-}obj\ G\ oi\ oref\ s){::}{\preceq}(G,\ L)$
apply (*unfold init-obj-def*)
apply (*auto elim*!: *conforms-gupd dest*!: *oconf-init-obj*
)
done

lemma *conforms-init-class-obj*:
$[\![(x,s){::}{\preceq}(G,\ L);\ wf\text{-}prog\ G;\ class\ G\ C{=}Some\ y;\ \neg\ inited\ C\ (globs\ s)]\!] \Longrightarrow$
$(x,init\text{-}class\text{-}obj\ G\ C\ s){::}{\preceq}(G,\ L)$
apply (*rule not-initedD* [*THEN conforms-newG*])
apply (*auto*)
done

lemma *fst-init-lvars*[*simp*]:
fst (*init-lvars G C sig* (*invmode m e*) *a′ pvs* (*x*,*s*)) =
(*if is-static m then x else* (*np a′*) *x*)
apply (*simp* (*no-asm*) *add*: *init-lvars-def2*)
done

lemma *halloc-conforms*: $\bigwedge$*s1*. ⟦*G*⊢*s1* −*halloc oi*≻*a*→ *s2*; *wf-prog G*; *s1*::⪯(*G*, *L*);
is-type G (*obj-ty* (|*tag*=*oi*,*values*=*fs*|))⟧ ⟹ *s2*::⪯(*G*, *L*)
apply (*simp* (*no-asm-simp*) *only*: *split-tupled-all*)
apply (*case-tac aa*)
apply (*auto elim*!: *halloc-elim-cases dest*!: *new-AddrD*
intro!: *conforms-newG* [*THEN conforms-xconf*] *conf-AddrI*)
done

lemma *halloc-type-sound*:
$\bigwedge$*s1*. ⟦*G*⊢*s1* −*halloc oi*≻*a*→ (*x*,*s*); *wf-prog G*; *s1*::⪯(*G*, *L*);
T = *obj-ty* (|*tag*=*oi*,*values*=*fs*|); *is-type G T*⟧ ⟹
(*x*,*s*)::⪯(*G*, *L*) ∧ (*x* = *None* ⟶ *G*,*s*⊢*Addr a*::⪯*T*)
apply (*auto elim*!: *halloc-conforms*)
apply (*case-tac aa*)
apply (*subst obj-ty-eq*)
apply (*auto elim*!: *halloc-elim-cases dest*!: *new-AddrD intro*!: *conf-AddrI*)
done

lemma *sxalloc-type-sound*:
$\bigwedge$*s1 s2*. ⟦*G*⊢*s1* −*sxalloc*→ *s2*; *wf-prog G*⟧ ⟹
case fst s1 of
None ⇒ *s2* = *s1*
| *Some abr* ⇒ (*case abr of*
Xcpt x ⇒ (∃ *a*. *fst s2* = *Some*(*Xcpt* (*Loc a*)) ∧
(∀ *L*. *s1*::⪯(*G*,*L*) ⟶ *s2*::⪯(*G*,*L*)))
| *Jump j* ⇒ *s2* = *s1*
| *Error e* ⇒ *s2* = *s1*)
apply (*simp* (*no-asm-simp*) *only*: *split-tupled-all*)
apply (*erule sxalloc.induct*)
apply *auto*
apply (*rule halloc-conforms* [*THEN conforms-xconf*])
apply (*auto elim*!: *halloc-elim-cases dest*!: *new-AddrD intro*!: *conf-AddrI*)
done

lemma *wt-init-comp-ty*:
is-acc-type G (*pid C*) *T* ⟹ (|*prg*=*G*,*cls*=*C*,*lcl*=*L*|)⊢*init-comp-ty T*::√
apply (*unfold init-comp-ty-def*)
apply (*clarsimp simp add*: *accessible-in-RefT-simp*
is-acc-type-def is-acc-class-def)
done

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

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

```
definition
  DynT-prop :: [prog,inv-mode,qtname,ref-ty] ⇒ bool (-⊢-→-≼-[71,71,71,71]70)
where
  G⊢mode→D≼t = (mode = IntVir ⟶ is-class G D ∧
                      (if (∃ T. t=ArrayT T) then D=Object else G⊢Class D≼RefT t))


lemma DynT-propI:
⟦(x,s)::≼(G, L); G,s⊢a'::≼RefT statT; wf-prog G; mode = IntVir ⟶ a' ≠ Null⟧
 ⟹ G⊢mode→invocation-class mode s a' statT≼statT
proof (unfold DynT-prop-def)
  assume state-conform: (x,s)::≼(G, L)
     and       statT-a': G,s⊢a'::≼RefT statT
     and             wf: wf-prog G
     and           mode: mode = IntVir ⟶ a' ≠ Null
  let ?invCls = (invocation-class mode s a' statT)
  let ?IntVir = mode = IntVir
  let ?Concl  = λinvCls. is-class G invCls ∧
                          (if ∃ T. statT = ArrayT T
                                  then invCls = Object
                                  else G⊢Class invCls≼RefT statT)
  show ?IntVir ⟶ ?Concl ?invCls
  proof
    assume modeIntVir: ?IntVir
    with mode have not-Null: a' ≠ Null ..
    from statT-a' not-Null state-conform
    obtain a obj
      where obj-props:  a' = Addr a globs s (Inl a) = Some obj
                    G⊢obj-ty obj≼RefT statT is-type G (obj-ty obj)
      by (blast dest: conforms-RefTD)
    show ?Concl ?invCls
    proof (cases tag obj)
      case CInst
      with modeIntVir obj-props
      show ?thesis
        by (auto dest!: widen-Array2)
    next
      case Arr
      from Arr obtain T where obj-ty obj = T.[] by blast
      moreover from Arr have obj-class obj = Object
        by blast
      moreover note modeIntVir obj-props wf
      ultimately show ?thesis by (auto dest!: widen-Array )
    qed
  qed
qed


lemma invocation-methd:
⟦wf-prog G; statT ≠ NullT;
 (∀ statC. statT = ClassT statC ⟶ is-class G statC);
 (∀      I. statT = IfaceT I ⟶ is-iface G I ∧ mode ≠ SuperM);
 (∀      T. statT = ArrayT T ⟶ mode ≠ SuperM);
 G⊢mode→invocation-class mode s a' statT≼statT;
 dynlookup G statT (invocation-class mode s a' statT) sig = Some m ⟧
⟹ methd G (invocation-declclass G mode s a' statT sig) sig = Some m
proof −
  assume          wf: wf-prog G
     and  not-NullT: statT ≠ NullT
```

and *statC-prop*: (∀ *statC*. *statT* = *ClassT statC* ⟶ *is-class G statC*)
and *statI-prop*: (∀ *I*. *statT* = *IfaceT I* ⟶ *is-iface G I* ∧ *mode* ≠ *SuperM*)
and *statA-prop*: (∀ *T*. *statT* = *ArrayT T* ⟶ *mode* ≠ *SuperM*)
and *invC-prop*: *G*⊢*mode*→*invocation-class mode s a′ statT*≼*statT*
and *dynlookup*: *dynlookup G statT* (*invocation-class mode s a′ statT*) *sig*
= *Some m*
show *?thesis*
proof (*cases statT*)
case *NullT*
with *not-NullT* **show** *?thesis* **by** *simp*
next
case *IfaceT*
with *statI-prop* **obtain** *I*
where *statI*: *statT* = *IfaceT I* **and**
is-iface: *is-iface G I* **and**
not-SuperM: *mode* ≠ *SuperM* **by** *blast*

show *?thesis*
proof (*cases mode*)
case *Static*
with *wf dynlookup statI is-iface*
show *?thesis*
by (*auto simp add*: *invocation-declclass-def dynlookup-def*
dynimethd-def dynmethd-C-C
intro: *dynmethd-declclass*
dest!: *wf-imethdsD*
dest: *table-of-map-SomeI*)
next
case *SuperM*
with *not-SuperM* **show** *?thesis* **..**
next
case *IntVir*
with *wf dynlookup IfaceT invC-prop* **show** *?thesis*
by (*auto simp add*: *invocation-declclass-def dynlookup-def dynimethd-def*
DynT-prop-def
intro: *methd-declclass dynmethd-declclass*)
qed
next
case *ClassT*
show *?thesis*
proof (*cases mode*)
case *Static*
with *wf ClassT dynlookup statC-prop*
show *?thesis* **by** (*auto simp add*: *invocation-declclass-def dynlookup-def*
intro: *dynmethd-declclass*)
next
case *SuperM*
with *wf ClassT dynlookup statC-prop*
show *?thesis* **by** (*auto simp add*: *invocation-declclass-def dynlookup-def*
intro: *dynmethd-declclass*)
next
case *IntVir*
with *wf ClassT dynlookup statC-prop invC-prop*
show *?thesis*
by (*auto simp add*: *invocation-declclass-def dynlookup-def dynimethd-def*
DynT-prop-def
intro: *dynmethd-declclass*)
qed
next

```
    case ArrayT
    show ?thesis
    proof (cases mode)
      case Static
      with wf ArrayT dynlookup show ?thesis
        by (auto simp add: invocation-declclass-def dynlookup-def
                           dynimethd-def dynmethd-C-C
                     intro: dynmethd-declclass
                      dest: table-of-map-SomeI)
    next
      case SuperM
      with ArrayT statA-prop show ?thesis by blast
    next
      case IntVir
      with wf ArrayT dynlookup invC-prop show ?thesis
        by (auto simp add: invocation-declclass-def dynlookup-def dynimethd-def
                           DynT-prop-def dynmethd-C-C
                     intro: dynmethd-declclass
                      dest: table-of-map-SomeI)
    qed
  qed
qed


lemma DynT-mheadsD:
⟦G⊢invmode sm e→invC≼statT;
  wf-prog G; (|prg=G,cls=C,lcl=L|)⊢e::−RefT statT;
  (statDeclT,sm) ∈ mheads G C statT sig;
  invC = invocation-class (invmode sm e) s a′ statT;
  declC =invocation-declclass G (invmode sm e) s a′ statT sig
 ⟧ ⟹
  ∃ dm.
  methd G declC sig = Some dm ∧ dynlookup G statT invC sig = Some dm  ∧
  G⊢resTy (mthd dm)≼resTy sm ∧
  wf-mdecl G declC (sig, mthd dm) ∧
  declC = declclass dm ∧
  is-static dm = is-static sm ∧
  is-class G invC ∧ is-class G declC  ∧ G⊢invC≼_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))
proof −
  assume invC-prop: G⊢invmode sm e→invC≼statT
     and         wf: wf-prog G
     and       wt-e: (|prg=G,cls=C,lcl=L|)⊢e::−RefT statT
     and         sm: (statDeclT,sm) ∈ mheads G C statT sig
     and       invC: invC = invocation-class (invmode sm e) s a′ statT
     and      declC: declC =
                  invocation-declclass G (invmode sm e) s a′ statT sig
  from wt-e wf have type-statT: is-type G (RefT statT)
    by (auto dest: ty-expr-is-type)
  from sm have not-Null: statT ≠ NullT by auto
  from type-statT
  have wf-C: (∀ statC. statT = ClassT statC ⟶ is-class G statC)
    by (auto)
  from type-statT wt-e
  have wf-I: (∀I. statT = IfaceT I ⟶ is-iface G I ∧
```

invmode sm e ≠ SuperM)
by (*auto dest*: *invocationTypeExpr-noClassD*)
from *wt-e*
have *wf-A*: (∀ *T*. *statT* = *ArrayT T* ⟶ *invmode sm e* ≠ *SuperM*)
by (*auto dest*: *invocationTypeExpr-noClassD*)
show *?thesis*
proof (*cases invmode sm e* = *IntVir*)
case *True*
with *invC-prop not-Null*
have *invC-prop′*: *is-class G invC* ∧
(*if* (∃ *T*. *statT*=*ArrayT T*) *then invC*=*Object*
else G⊢*Class invC*≼*RefT statT*)
by (*auto simp add*: *DynT-prop-def*)
from *True*
have ¬ *is-static sm*
by (*simp add*: *invmode-IntVir-eq member-is-static-simp*)
with *invC-prop′ not-Null*
have *G*,*statT* ⊢ *invC valid-lookup-cls-for* (*is-static sm*)
by (*cases statT*) *auto*
with *sm wf type-statT* **obtain** *dm* **where**
dm: *dynlookup G statT invC sig* = *Some dm* **and**
resT-dm: *G*⊢*resTy* (*mthd dm*)≼*resTy sm* **and**
static: *is-static dm* = *is-static sm*
by − (*drule dynamic-mheadsD*,*force*+)
with *declC invC not-Null*
have *declC′*: *declC* = (*declclass dm*)
by (*auto simp add*: *invocation-declclass-def*)
with *wf invC declC not-Null wf-C wf-I wf-A invC-prop dm*
have *dm′*: *methd G declC sig* = *Some dm*
by − (*drule invocation-methd*,*auto*)
from *wf dm invC-prop′ declC′ type-statT*
have *declC-prop*: *G*⊢*invC* $\preceq_C$ *declC* ∧ *is-class G declC*
by (*auto dest*: *dynlookup-declC*)
from *wf dm′ declC-prop declC′*
have *wf-dm*: *wf-mdecl G declC* (*sig*,(*mthd dm*))
by (*auto dest*: *methd-wf-mdecl*)
from *invC-prop′*
have *statC-prop*: (∀ *statC*. *statT*=*ClassT statC* ⟶ *G*⊢*invC* $\preceq_C$ *statC*)
by *auto*
from *True dm′ resT-dm wf-dm invC-prop′ declC-prop statC-prop declC′ static dm*
show *?thesis* **by** *auto*
next
case *False*
with *type-statT wf invC not-Null wf-I wf-A*
have *invC-prop′*: *is-class G invC* ∧
((∃ *statC*. *statT*=*ClassT statC* ∧ *invC*=*statC*) ∨
(∀ *statC*. *statT*≠*ClassT statC* ∧ *invC*=*Object*))
by (*case-tac statT*) (*auto simp add*: *invocation-class-def*
split: *inv-mode.splits*)
with *not-Null wf*
have *dynlookup-static*: *dynlookup G statT invC sig* = *methd G invC sig*
by (*case-tac statT*) (*auto simp add*: *dynlookup-def dynmethd-C-C*
dynimethd-def)
from *sm wf wt-e not-Null False invC-prop′* **obtain** *dm* **where**
dm: *methd G invC sig* = *Some dm* **and**
eq-declC-sm-dm:*statDeclT* = *ClassT* (*declclass dm*) **and**
eq-mheads:*sm*=*mhead* (*mthd dm*)
by − (*drule static-mheadsD*, (*force dest*: *accmethd-SomeD*)+)

```
    then have static: is-static dm = is-static sm by − (auto)
    with declC invC dynlookup-static dm
    have declC': declC = (declclass dm)
      by (auto simp add: invocation-declclass-def)
    from invC-prop' wf declC' dm
    have dm': methd G declC sig = Some dm
      by (auto intro: methd-declclass)
    from dynlookup-static dm
    have dm'': dynlookup G statT invC sig = Some dm
      by simp
    from wf dm invC-prop' declC' type-statT
    have declC-prop: G⊢invC ≼C declC ∧ is-class G declC
      by (auto dest: methd-declC )
    then have declC-prop1: invC=Object ⟶ declC=Object  by auto
    from wf dm' declC-prop declC'
    have wf-dm: wf-mdecl G declC (sig,(mthd dm))
      by (auto dest: methd-wf-mdecl)
    from invC-prop' declC-prop declC-prop1
    have statC-prop: (   (∃ statC. statT=ClassT statC ∧ G⊢statC≼C declC)
                     ∨  (∀ statC. statT≠ClassT statC ∧ declC=Object))
      by auto
    from False dm' dm'' wf-dm invC-prop' declC-prop statC-prop declC'
         eq-declC-sm-dm eq-mheads static
    show ?thesis by auto
  qed
qed

corollary DynT-mheadsE [consumes 7]:
— Same as DynT-mheadsD but better suited for application in typesafety proof
 assumes invC-compatible: G⊢mode→invC≼statT
     and wf: wf-prog G
     and wt-e: (|prg=G,cls=C,lcl=L|)⊢e::−RefT statT
     and mheads: (statDeclT,sm) ∈ mheads G C statT sig
     and mode: mode=invmode sm e
     and invC: invC = invocation-class mode s a' statT
     and declC: declC =invocation-declclass G mode s a' statT sig
     and dm: ⋀ dm. ⟦methd G declC sig = Some dm;
                    dynlookup G statT invC sig = Some dm;
                    G⊢resTy (mthd dm)≼resTy sm;
                    wf-mdecl G declC (sig, mthd dm);
                    declC = declclass dm;
                    is-static dm = is-static sm;
                    is-class G invC; is-class G declC; G⊢invC≼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 −
    from invC-compatible mode have G⊢invmode sm e→invC≼statT by simp
    moreover note wf wt-e mheads
    moreover from invC mode
    have invC = invocation-class (invmode sm e) s a' statT by simp
    moreover from declC mode
    have declC =invocation-declclass G (invmode sm e) s a' statT sig by simp
    ultimately show ?thesis
      by (rule DynT-mheadsD [THEN exE,rule-format])
         (elim conjE,rule dm)
```

qed

lemma *DynT-conf*: ⟦*G*⊢*invocation-class mode s a′ statT* $\preceq_C$ *declC*; *wf-prog G*;
isrtype G (*statT*);
G,*s*⊢*a′*::≼*RefT statT*; *mode* = *IntVir* ⟶ *a′* ≠ *Null*;
mode ≠ *IntVir* ⟶ (∃ *statC*. *statT*=*ClassT statC* ∧ *G*⊢*statC*$\preceq_C$ *declC*)
∨ (∀ *statC*. *statT*≠*ClassT statC* ∧ *declC*=*Object*)⟧
⟹*G*,*s*⊢*a′*::≼ *Class declC*
apply (*case-tac mode* = *IntVir*)
apply (*drule conf-RefTD*)
apply (*force intro*!: *conf-AddrI*
simp add: *obj-class-def split*: *obj-tag.split-asm*)
apply *clarsimp*
apply *safe*
apply (*erule* (*1*) *widen.subcls* [*THEN conf-widen*])
apply (*erule wf-ws-prog*)

apply (*frule widen-Object*) **apply** (*erule wf-ws-prog*)
apply (*erule* (*1*) *conf-widen*) **apply** (*erule wf-ws-prog*)
done

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*)
apply (*drule-tac x* = *None* **and** *s* = *s2* **and** *v* = *v* **in** *evar-gext-f*)
apply (*drule eval-gext′*, *clarsimp*)
apply (*erule conf-gext*)
apply *simp*
done

lemma *Throw-lemma*: ⟦*G*⊢*tn*$\preceq_C$ *SXcpt Throwable*; *wf-prog G*; (*x1*,*s1*)::≼(*G*, *L*);
x1 = *None* ⟶ *G*,*s1*⊢*a′*::≼ *Class tn*⟧ ⟹ (*throw a′ x1*, *s1*)::≼(*G*, *L*)
apply (*auto split*: *split-abrupt-if simp add*: *throw-def2*)
apply (*erule conforms-xconf*)
apply (*frule conf-RefTD*)
apply (*auto elim*: *widen.subcls* [*THEN conf-widen*])
done

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*))
apply (*rule conforms-allocL*)
apply (*erule conforms-NormI*)
apply (*drule conforms-XcptLocD* [*THEN conf-RefTD*],*rule HOL.refl*)
apply (*auto intro*!: *conf-AddrI*)
done

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*)

apply (*auto elim*: *eval-gext′ conforms-xgext split*: *split-abrupt-if*)
done

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*$\preceq_C$ *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*)
apply (*drule initedD*)
apply (*frule subcls-is-class2*, *simp* (*no-asm-simp*))
apply (*case-tac static f*)
apply *clarsimp*
apply (*drule* (*1*) *rev-gext-objD*, *clarsimp*)
apply (*frule fields-declC*, *erule wf-ws-prog*, *simp* (*no-asm-simp*))
apply (*rule var-tys-Some-eq* [*THEN iffD2*])
apply *clarsimp*
apply (*erule fields-table-SomeI*, *simp* (*no-asm*))
apply *clarsimp*
apply (*drule conf-RefTD*, *clarsimp*, *rule var-tys-Some-eq* [*THEN iffD2*])
apply (*auto dest*!: *widen-Array split*: *obj-tag.split*)
apply (*rule fields-table-SomeI*)
apply (*auto elim*!: *fields-mono subcls-is-class2*)
done

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 −
assume *error-free*: *error-free state*
obtain *a s* **where** *state*=(*a*,*s*)
by (*cases state*)
with *error-free*
show *?thesis*
by (*cases a*) *auto*
qed

declare *split-paired-All* [*simp del*] *split-paired-Ex* [*simp del*]
declare *if-split* [*split del*] *if-split-asm* [*split del*]
option.split [*split del*] *option.split-asm* [*split del*]
setup ‹*map-theory-simpset* (*fn ctxt* => *ctxt delloop split-all-tac*)›
setup ‹*map-theory-claset* (*fn ctxt* => *ctxt delSWrapper split-all-tac*)›

lemma *FVar-lemma*:

⟦((v, f), *Norm s2′*) = *fvar statDeclC* (*static field*) *fn a* (*x2*, *s2*);
G⊢*statC*$\preceq_C$ *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*)
apply (*unfold assign-conforms-def*)
apply (*drule sym*)
apply (*clarsimp simp add*: *fvar-def2*)
apply (*drule* (*9*) *FVar-lemma1*)
apply (*clarsimp*)
apply (*drule* (*2*) *conforms-globsD* [*THEN oconf-lconf*, *THEN lconfD*])
apply *clarsimp*
apply (*rule conjI*)
apply *clarsimp*
apply (*drule* (*1*) *rev-gext-objD*)
apply (*force elim*!: *conforms-upd-gobj*)

apply (*blast intro*: *FVar-lemma2*)
done
declare *split-paired-All* [*simp*] *split-paired-Ex* [*simp*]
declare *if-split* [*split*] *if-split-asm* [*split*]
option.split [*split*] *option.split-asm* [*split*]
setup ‹*map-theory-claset* (*fn ctxt* => *ctxt addSbefore* (*split-all-tac*, *split-all-tac*))›
setup ‹*map-theory-simpset* (*fn ctxt* => *ctxt addloop* (*split-all-tac*, *split-all-tac*))›

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*
apply (*erule widen-Array-Array* [*THEN conf-widen*])
apply (*erule-tac* [*2*] *wf-ws-prog*)
apply (*drule* (*1*) *conforms-globsD* [*THEN oconf-lconf*, *THEN lconfD*])
defer apply *assumption*
apply (*force intro*: *var-tys-Some-eq* [*THEN iffD2*])
done

lemma *obj-split*: ∃ *t vs*. *obj* = (|*tag*=*t*,*values*=*vs*|)
by (*cases obj*) *auto*

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 −
assume *error-free*: *error-free state*
obtain *a s* **where** *state*=(*a*,*s*)
by (*cases state*)
with *error-free*
show *?thesis*
by (*cases a*) *auto*

qed

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*)
apply (*unfold assign-conforms-def*)
apply (*drule sym*)
apply (*clarsimp simp add*: *avar-def2*)
apply (*drule* (*1*) *conf-gext*)
apply (*drule conf-RefTD*, *clarsimp*)
apply (*subgoal-tac* ∃ *t vs*. *obj* = (|*tag*=*t*,*values*=*vs*|))
defer
apply (*rule obj-split*)
apply *clarify*
apply (*frule obj-ty-widenD*)
apply (*auto dest*!: *widen-Class*)
apply (*force dest*: *AVar-lemma1*)

apply (*force elim*!: *fits-Array dest*: *gext-objD*
intro: *var-tys-Some-eq* [*THEN iffD2*] *conforms-upd-gobj*)
done

Call

lemma *conforms-init-lvars-lemma*: ⟦*wf-prog G*;
wf-mhead G P sig mh;
list-all2 (*conf G s*) *pvs pTsa*; *G*⊢*pTsa*[≼](*parTs sig*)⟧ ⟹
G,*s*⊢*Map.empty* (*pars mh*[↦]*pvs*)
[∼::≼](*table-of lvars*)(*pars mh*[↦]*parTs sig*)
apply (*unfold wf-mhead-def*)
apply *clarify*
apply (*erule* (*1*) *wlconf-empty-vals* [*THEN wlconf-ext-list*])
apply (*drule wf-ws-prog*)
apply (*erule* (*2*) *conf-list-widen*)
done

lemma *lconf-map-lname* [*simp*]:
G,*s*⊢(*case-lname l1 l2*)[::≼](*case-lname L1 L2*)
=
(*G*,*s*⊢*l1*[::≼]*L1* ∧ *G*,*s*⊢(λ*x*::*unit* . *l2*)[::≼](λ*x*::*unit*. *L2*))
apply (*unfold lconf-def*)
apply (*auto split*: *lname.splits*)
done

lemma *wlconf-map-lname* [*simp*]:
G,*s*⊢(*case-lname l1 l2*)[∼::≼](*case-lname L1 L2*)
=
(*G*,*s*⊢*l1*[∼::≼]*L1* ∧ *G*,*s*⊢(λ*x*::*unit* . *l2*)[∼::≼](λ*x*::*unit*. *L2*))
apply (*unfold wlconf-def*)
apply (*auto split*: *lname.splits*)
done

lemma *lconf-map-ename* [*simp*]:
G,*s*⊢(*case-ename l1 l2*)[::≼](*case-ename L1 L2*)

=
(G,s⊢l1[::≼]L1 ∧ G,s⊢(λx::unit. l2)[::≼](λx::unit. L2))
apply (*unfold lconf-def*)
apply (*auto split*: *ename.splits*)
done

lemma *wlconf-map-ename* [*simp*]:
G,s⊢(case-ename l1 l2)[∼::≼](case-ename L1 L2)
=
(G,s⊢l1[∼::≼]L1 ∧ G,s⊢(λx::unit. l2)[∼::≼](λx::unit. L2))
apply (*unfold wlconf-def*)
apply (*auto split*: *ename.splits*)
done

lemma *defval-conf1* [*rule-format* (*no-asm*), *elim*]:
is-type G T ⟶ (∃ v∈Some (default-val T): G,s⊢v::≼T)
apply (*unfold conf-def*)
apply (*induct T*)
apply (*auto intro*: *prim-ty.induct*)
done

lemma *np-no-jump*: x≠Some (Jump j) ⟹ (np a′) x ≠ Some (Jump j)
by (*auto simp add*: *abrupt-if-def*)

declare *split-paired-All* [*simp del*] *split-paired-Ex* [*simp del*]
declare *if-split* [*split del*] *if-split-asm* [*split del*]
option.split [*split del*] *option.split-asm* [*split del*]
setup ‹*map-theory-simpset* (*fn ctxt* => *ctxt delloop split-all-tac*)›
setup ‹*map-theory-claset* (*fn ctxt* => *ctxt delSWrapper split-all-tac*)›

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*)))
apply (*simp add*: *init-lvars-def2*)
apply (*rule conforms-set-locals*)
apply (*simp* (*no-asm-simp*) *split*: *if-split*)
apply (*drule* (*4*) *DynT-conf*)
apply *clarsimp*

apply (*drule* (*3*) *conforms-init-lvars-lemma*
[**where** *?lvars*=(*lcls* (*mbody* (*mthd dm*)))])
apply (*case-tac dm*,*simp*)
apply (*rule conjI*)
apply (*unfold wlconf-def*, *clarify*)
apply (*clarsimp simp add*: *wf-mhead-def is-acc-type-def*)
apply (*case-tac is-static sm*)
apply *simp*
apply *simp*

apply *simp*
apply (*case-tac is-static sm*)
apply *simp*
apply (*simp add*: *np-no-jump*)
done
declare *split-paired-All* [*simp*] *split-paired-Ex* [*simp*]
declare *if-split* [*split*] *if-split-asm* [*split*]
option.split [*split*] *option.split-asm* [*split*]
setup ‹*map-theory-claset* (*fn ctxt* => *ctxt addSbefore* (*split-all-tac*, *split-all-tac*))›
setup ‹*map-theory-simpset* (*fn ctxt* => *ctxt addloop* (*split-all-tac*, *split-all-tac*))›

2 accessibility

theorem *dynamic-field-access-ok*:
assumes *wf*: *wf-prog G* **and**
not-Null: ¬ *stat* ⟶ *a*≠*Null* **and**
conform-a: *G*,(*store s*)⊢*a*::≼ *Class statC* **and**
conform-s: *s*::≼(*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 (*cases stat*)
case *True*
with *stat* **have** *static*: (*is-static f*) **by** *simp*
from *True dynC*
have *dynC′*: *dynC*=*declclass f* **by** *simp*
with *f*
have *table-of* (*DeclConcepts.fields G statC*) (*fn*,*declclass f*) = *Some* (*fld f*)
by (*auto simp add*: *accfield-def Let-def intro*!: *table-of-remap-SomeD*)
moreover
from *wt-e wf* **have** *is-class G statC*
by (*auto dest*!: *ty-expr-is-type*)
moreover note *wf dynC′*
ultimately have
table-of (*DeclConcepts.fields G dynC*) (*fn*,*declclass f*) = *Some* (*fld f*)

by (*auto dest*: *fields-declC*)
with *dynC′ f static wf*
show *?thesis*
by (*auto dest*: *static-to-dynamic-accessible-from-static*
dest!: *accfield-accessibleD*)
next
case *False*
with *wf conform-a not-Null conform-s dynC*
obtain *subclseq*: $G \vdash dynC \preceq_C statC$ **and**
is-class G dynC
by (*auto dest*!: *conforms-RefTD* [*of* - - - - (*fst s*) *L*]
dest: *obj-ty-obj-class1*
simp add: *obj-ty-obj-class*)
with *wf f*
have *table-of* (*DeclConcepts.fields G dynC*) (*fn*,*declclass f*) = *Some* (*fld f*)
by (*auto simp add*: *accfield-def Let-def*
dest: *fields-mono*
dest!: *table-of-remap-SomeD*)
moreover
from *f subclseq*
have *G⊢Field fn f in dynC dyn-accessible-from accC*
by (*auto intro*!: *static-to-dynamic-accessible-from wf*
dest: *accfield-accessibleD*)
ultimately show *?thesis*
by *blast*
qed

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 −
from *fvar*
have *store-s2′*: *store s2′*=*store s2*
by (*cases s2*) (*simp add*: *fvar-def2*)
with *fvar conf-s2*
have *conf-s2′*: *s2′*::≼(*G*, *L*)
by (*cases s2*,*cases is-static f*) (*auto simp add*: *fvar-def2*)
from *eval-init*
have *initd-statDeclC-s1*: *initd statDeclC s1*
by (*rule init-yields-initd*)
with *eval-e store-s2′*
have *initd-statDeclC-s2′*: *initd statDeclC s2′*
by (*auto dest*: *eval-gext intro*: *inited-gext*)
show *?thesis*
proof (*cases normal s2′*)
case *False*
then show *?thesis*
by (*auto simp add*: *check-field-access-def Let-def*)
next
case *True*
with *fvar store-s2′*

```
    have not-Null: ¬ (is-static f) ⟶ a≠Null
      by (cases s2) (auto simp add: fvar-def2)
    from True fvar store-s2'
    have normal s2
      by (cases s2,cases is-static f) (auto simp add: fvar-def2)
    with conf-a store-s2'
    have conf-a': G,store s2'⊢a::≼Class statC
      by simp
    from conf-a' conf-s2' True initd-statDeclC-s2'
      dynamic-field-access-ok [OF wf not-Null conf-a' conf-s2'
                                 True wt-e accfield ]
    show ?thesis
      by (cases is-static f)
          (auto dest!: initedD
           simp add: check-field-access-def Let-def)
  qed
qed


lemma call-access-ok:
  assumes invC-prop: G⊢invmode statM e→invC≼statT
      and        wf: wf-prog G
      and      wt-e: (|prg=G,cls=C,lcl=L|)⊢e::−RefT statT
      and     statM: (statDeclT,statM) ∈ mheads G accC statT sig
      and      invC: invC = invocation-class (invmode statM e) s a statT
  shows ∃ dynM. dynlookup G statT invC sig = Some dynM ∧
  G⊢Methd sig dynM in invC dyn-accessible-from accC
proof −
  from wt-e wf have type-statT: is-type G (RefT statT)
    by (auto dest: ty-expr-is-type)
  from statM have not-Null: statT ≠ NullT by auto
  from type-statT wt-e
  have wf-I: (∀ I. statT = IfaceT I ⟶ is-iface G I ∧
                                        invmode statM e ≠ SuperM)
    by (auto dest: invocationTypeExpr-noClassD)
  from wt-e
  have wf-A: (∀      T. statT = ArrayT T ⟶ invmode statM e ≠ SuperM)
    by (auto dest: invocationTypeExpr-noClassD)
  show ?thesis
  proof (cases invmode statM e = IntVir)
    case True
    with invC-prop not-Null
    have invC-prop': is-class G invC ∧
                    (if (∃ T. statT=ArrayT T) then invC=Object
                                              else G⊢Class invC≼RefT statT)
      by (auto simp add: DynT-prop-def)
    with True not-Null
    have G,statT ⊢ invC valid-lookup-cls-for is-static statM
     by (cases statT) (auto simp add: invmode-def)
    with statM type-statT wf
    show ?thesis
      by − (rule dynlookup-access,auto)
  next
    case False
    with type-statT wf invC not-Null wf-I wf-A
    have invC-prop': is-class G invC ∧
                    ((∃ statC. statT=ClassT statC ∧ invC=statC) ∨
                    (∀ statC. statT≠ClassT statC ∧ invC=Object))
        by (case-tac statT) (auto simp add: invocation-class-def
```

split: *inv-mode.splits*)
with *not-Null wf*
have *dynlookup-static*: *dynlookup G statT invC sig* = *methd G invC sig*
by (*case-tac statT*) (*auto simp add*: *dynlookup-def dynmethd-C-C*
dynimethd-def)
from *statM wf wt-e not-Null False invC-prop′* **obtain** *dynM* **where**
accmethd G accC invC sig = *Some dynM*
by (*auto dest*!: *static-mheadsD*)
from *invC-prop′ False not-Null wf-I*
have *G*,*statT* ⊢ *invC valid-lookup-cls-for is-static statM*
by (*cases statT*) (*auto simp add*: *invmode-def*)
with *statM type-statT wf*
show *?thesis*
by − (*rule dynlookup-access*,*auto*)
qed
qed

lemma *error-free-call-access*:
assumes
eval-args: *G*⊢*s1* −*args*≐≻*vs*→ *s2* **and**
wt-e: (|*prg* = *G*, *cls* = *accC*, *lcl* = *L*|)⊢*e*::−(*RefT statT*) **and**
statM: *max-spec G accC statT* (|*name* = *mn*, *parTs* = *pTs*|)
= {((*statDeclT*, *statM*), *pTs′*)} **and**
conf-s2: *s2*::⪯(*G*, *L*) **and**
conf-a: *normal s1* ⟹ *G*, *store s1*⊢*a*::⪯*RefT statT* **and**
invProp: *normal s3* ⟹
G⊢*invmode statM e*→*invC*⪯*statT* **and**
s3: *s3*=*init-lvars G invDeclC* (|*name* = *mn*, *parTs* = *pTs′*|)
(*invmode statM e*) *a vs s2* **and**
invC: *invC* = *invocation-class* (*invmode statM e*) (*store s2*) *a statT***and**
invDeclC: *invDeclC* = *invocation-declclass G* (*invmode statM e*) (*store s2*)
a statT (|*name* = *mn*, *parTs* = *pTs′*|) **and**
wf: *wf-prog G*
shows *check-method-access G accC statT* (*invmode statM e*) (|*name*=*mn*,*parTs*=*pTs′*|) *a s3*
= *s3*
proof (*cases normal s2*)
case *False*
with *s3*
have *abrupt s3* = *abrupt s2*
by (*auto simp add*: *init-lvars-def2*)
with *False*
show *?thesis*
by (*auto simp add*: *check-method-access-def Let-def*)
next
case *True*
note *normal-s2* = *True*
with *eval-args*
have *normal-s1*: *normal s1*
by (*cases normal s1*) *auto*
with *conf-a eval-args*
have *conf-a-s2*: *G*, *store s2*⊢*a*::⪯*RefT statT*
by (*auto dest*: *eval-gext intro*: *conf-gext*)
show *?thesis*
proof (*cases a*=*Null* ⟶ (*is-static statM*))
case *False*
then obtain ¬ *is-static statM a*=*Null*
by *blast*
with *normal-s2 s3*

have *abrupt s3 = Some (Xcpt (Std NullPointer))*
by (*auto simp add*: *init-lvars-def2*)
then show *?thesis*
by (*auto simp add*: *check-method-access-def Let-def*)
next
case *True*
from *statM*
obtain
statM': (*statDeclT*,*statM*)∈*mheads G accC statT* (|*name*=*mn*,*parTs*=*pTs'*|)
by (*blast dest*: *max-spec2mheads*)
from *True normal-s2 s3*
have *normal s3*
by (*auto simp add*: *init-lvars-def2*)
then have *G*⊢*invmode statM e*→*invC*≼*statT*
by (*rule invProp*)
with *wt-e statM' wf invC*
obtain *dynM* **where**
dynM: *dynlookup G statT invC* (|*name*=*mn*,*parTs*=*pTs'*|) = *Some dynM* **and**
acc-dynM: *G* ⊢*Methd* (|*name*=*mn*,*parTs*=*pTs'*|) *dynM*
in invC dyn-accessible-from accC
by (*force dest*!: *call-access-ok*)
moreover
from *s3 invC*
have *invC'*: *invC*=(*invocation-class* (*invmode statM e*) (*store s3*) *a statT*)
by (*cases s2*,*cases invmode statM e*)
(*simp add*: *init-lvars-def2 del*: *invmode-Static-eq*)+
ultimately
show *?thesis*
by (*auto simp add*: *check-method-access-def Let-def*)
qed
qed

lemma *map-upds-eq-length-append-simp*:
⋀ *tab qs*. *length ps* = *length qs* ⟹ *tab*(*ps*[↦]*qs*@*zs*) = *tab*(*ps*[↦]*qs*)
proof (*induct ps*)
case *Nil* **thus** *?case* **by** *simp*
next
case (*Cons p ps tab qs*)
from ‹*length* (*p*#*ps*) = *length qs*›
obtain *q qs'* **where** *qs*: *qs*=*q*#*qs'* **and** *eq-length*: *length ps*=*length qs'*
by (*cases qs*) *auto*
from *eq-length* **have** (*tab*(*p*↦*q*))(*ps*[↦]*qs'*@*zs*)=(*tab*(*p*↦*q*))(*ps*[↦]*qs'*)
by (*rule Cons.hyps*)
with *qs* **show** *?case*
by *simp*
qed

lemma *map-upds-upd-eq-length-simp*:
⋀ *tab qs x y*. *length ps* = *length qs*
⟹ *tab*(*ps*[↦]*qs*, *x*↦*y*) = *tab*(*ps*@[*x*][↦]*qs*@[*y*])
proof (*induct ps*)
case *Nil* **thus** *?case* **by** *simp*
next
case (*Cons p ps tab qs x y*)
from ‹*length* (*p*#*ps*) = *length qs*›
obtain *q qs'* **where** *qs*: *qs*=*q*#*qs'* **and** *eq-length*: *length ps*=*length qs'*
by (*cases qs*) *auto*

```
  from eq-length
  have (tab(p↦q))(ps[↦]qs', x↦y) = (tab(p↦q))(ps@[x][↦]qs'@[y])
    by (rule Cons.hyps)
  with qs show ?case
    by simp
qed


lemma map-upd-cong: tab=tab'⟹ tab(x↦y) = tab'(x↦y)
by simp


lemma map-upd-cong-ext: tab z=tab' z⟹ (tab(x↦y)) z = (tab'(x↦y)) z
by (simp add: fun-upd-def)


lemma map-upds-cong: tab=tab'⟹ tab(xs[↦]ys) = tab'(xs[↦]ys)
by (cases xs) simp+


lemma map-upds-cong-ext:
 ⋀ tab tab' ys. tab z=tab' z ⟹ (tab(xs[↦]ys)) z = (tab'(xs[↦]ys)) z
proof (induct xs)
  case Nil thus ?case by simp
next
  case (Cons x xs tab tab' ys)
  note Hyps = this
  show ?case
  proof (cases ys)
    case Nil
    with Hyps
    show ?thesis by simp
  next
    case (Cons y ys')
    have (tab(x↦y, xs[↦]ys')) z = (tab'(x↦y, xs[↦]ys')) z
      by (iprover intro: Hyps map-upd-cong-ext)
    with Cons show ?thesis
      by simp
  qed
qed


lemma map-upd-override: (tab(x↦y)) x = (tab'(x↦y)) x
  by simp


lemma map-upds-eq-length-suffix: ⋀ tab qs.
        length ps = length qs ⟹ tab(ps@xs[↦]qs) = tab(ps[↦]qs, xs[↦][])
proof (induct ps)
  case Nil thus ?case by simp
next
  case (Cons p ps tab qs)
  then obtain q qs' where qs: qs=q#qs' and eq-length: length ps=length qs'
    by (cases qs) auto
  from eq-length
  have tab(p↦q, ps @ xs[↦]qs') = tab(p↦q, ps[↦]qs', xs[↦][])
    by (rule Cons.hyps)
  with qs show ?case
```

by *simp*
qed

lemma *map-upds-upds-eq-length-prefix-simp*:
$\bigwedge$ *tab qs. length ps = length qs*
$\Longrightarrow$ *tab(ps[↦]qs, xs[↦]ys) = tab(ps@xs[↦]qs@ys)*
proof (*induct ps*)
case *Nil* **thus** *?case* **by** *simp*
next
case (*Cons p ps tab qs*)
then obtain *q qs′* **where** *qs*: *qs=q#qs′* **and** *eq-length*: *length ps=length qs′*
by (*cases qs*) *auto*
from *eq-length*
have *tab(p↦q, ps[↦]qs′, xs[↦]ys) = tab(p↦q, ps @ xs[↦](qs′ @ ys))*
by (*rule Cons.hyps*)
with *qs*
show *?case* **by** *simp*
qed

lemma *map-upd-cut-irrelevant*:
⟦(*tab(x↦y)) vn = Some el*; (*tab′(x↦y)) vn = None*⟧
$\Longrightarrow$ *tab vn = Some el*
by (*cases tab′ vn = None*) (*simp add*: *fun-upd-def*)+

lemma *map-upd-Some-expand*:
⟦*tab vn = Some z*⟧
$\Longrightarrow$ ∃ *z*. (*tab(x↦y)) vn = Some z*
by (*simp add*: *fun-upd-def*)

lemma *map-upds-Some-expand*:
$\bigwedge$ *tab ys z*. ⟦*tab vn = Some z*⟧
$\Longrightarrow$ ∃ *z*. (*tab(xs[↦]ys)) vn = Some z*
proof (*induct xs*)
case *Nil* **thus** *?case* **by** *simp*
next
case (*Cons x xs tab ys z*)
note *z* = ‹*tab vn = Some z*›
show *?case*
proof (*cases ys*)
case *Nil*
with *z* **show** *?thesis* **by** *simp*
next
case (*Cons y ys′*)
note *ys* = ‹*ys = y#ys′*›
from *z* **obtain** *z′* **where** (*tab(x↦y)) vn = Some z′*
by (*rule map-upd-Some-expand* [*of tab,elim-format*]) *blast*
hence ∃ *z*. ((*tab(x↦y))(xs[↦]ys′)) vn = Some z*
by (*rule Cons.hyps*)
with *ys* **show** *?thesis*
by *simp*
qed
qed

```
lemma map-upd-Some-swap:
 (tab(r↦w, u↦v)) vn = Some z ⟹ ∃ z. (tab(u↦v, r↦w)) vn = Some z
by (simp add: fun-upd-def)


lemma map-upd-None-swap:
 (tab(r↦w, u↦v)) vn = None ⟹ (tab(u↦v, r↦w)) vn = None
by (simp add: fun-upd-def)


lemma map-eq-upd-eq: tab vn = tab′ vn ⟹ (tab(x↦y)) vn = (tab′(x↦y)) vn
by (simp add: fun-upd-def)


lemma map-upd-in-expansion-map-swap:
⟦(tab(x↦y)) vn = Some z;tab vn ≠ Some z⟧
                 ⟹  (tab′(x↦y)) vn = Some z
by (simp add: fun-upd-def)


lemma map-upds-in-expansion-map-swap:
 ⋀tab tab′ ys z. ⟦(tab(xs[↦]ys)) vn = Some z;tab vn ≠ Some z⟧
                 ⟹  (tab′(xs[↦]ys)) vn = Some z
proof (induct xs)
  case Nil thus ?case by simp
next
  case (Cons x xs tab tab′ ys z)
  note some = ‹(tab(x # xs[↦]ys)) vn = Some z›
  note tab-not-z = ‹tab vn ≠ Some z›
  show ?case
  proof (cases ys)
    case Nil with some tab-not-z show ?thesis by simp
  next
    case (Cons y tl)
    note ys = ‹ys = y#tl›
    show ?thesis
    proof (cases (tab(x↦y)) vn ≠ Some z)
      case True
      with some ys have (tab′(x↦y, xs[↦]tl)) vn = Some z
        by (fastforce intro: Cons.hyps)
      with ys show ?thesis
        by simp
    next
      case False
      hence tabx-z: (tab(x↦y)) vn = Some z by blast
      moreover
      from tabx-z tab-not-z
      have (tab′(x↦y)) vn = Some z
        by (rule map-upd-in-expansion-map-swap)
      ultimately
      have (tab(x↦y)) vn =(tab′(x↦y)) vn
        by simp
      hence (tab(x↦y, xs[↦]tl)) vn = (tab′(x↦y, xs[↦]tl)) vn
        by (rule map-upds-cong-ext)
      with some ys
      show ?thesis
        by simp
```

```
    qed
  qed
qed


lemma map-upds-Some-swap:
 assumes r-u: (tab(r↦w, u↦v, xs[↦]ys)) vn = Some z
    shows ∃ z. (tab(u↦v, r↦w, xs[↦]ys)) vn = Some z
proof (cases (tab(r↦w, u↦v)) vn = Some z)
  case True
  then obtain z' where (tab(u↦v, r↦w)) vn = Some z'
    by (rule map-upd-Some-swap [elim-format]) blast
  thus ∃ z. (tab(u↦v, r↦w, xs[↦]ys)) vn = Some z
    by (rule map-upds-Some-expand)
next
  case False
  with r-u
  have (tab(u↦v, r↦w, xs[↦]ys)) vn = Some z
    by (rule map-upds-in-expansion-map-swap)
  thus ?thesis
    by simp
qed


lemma map-upds-Some-insert:
  assumes z: (tab(xs[↦]ys)) vn = Some z
    shows ∃ z. (tab(u↦v, xs[↦]ys)) vn = Some z
proof (cases ∃ z. tab vn = Some z)
  case True
  then obtain z' where tab vn = Some z' by blast
  then obtain z'' where (tab(u↦v)) vn = Some z''
    by (rule map-upd-Some-expand [elim-format]) blast
  thus ?thesis
    by (rule map-upds-Some-expand)
next
  case False
  hence tab vn ≠ Some z by simp
  with z
  have (tab(u↦v, xs[↦]ys)) vn = Some z
    by (rule map-upds-in-expansion-map-swap)
  thus ?thesis ..
qed


lemma map-upds-None-cut:
assumes expand-None: (tab(xs[↦]ys)) vn = None
  shows tab vn = None
proof (cases tab vn = None)
  case True thus ?thesis by simp
next
  case False then obtain z where tab vn = Some z by blast
  then obtain z' where (tab(xs[↦]ys)) vn = Some z'
    by (rule map-upds-Some-expand [where ?tab=tab,elim-format]) blast
  with expand-None show ?thesis
    by simp
qed
```

```
lemma map-upds-cut-irrelevant:
⋀ tab tab' ys. ⟦(tab(xs[↦]ys)) vn = Some el; (tab'(xs[↦]ys)) vn = None⟧
                  ⟹ tab vn = Some el
proof (induct xs)
  case Nil thus ?case by simp
next
  case (Cons x xs tab tab' ys)
  note tab-vn = ‹(tab(x # xs[↦]ys)) vn = Some el›
  note tab'-vn = ‹(tab'(x # xs[↦]ys)) vn = None›
  show ?case
  proof (cases ys)
    case Nil
    with tab-vn show ?thesis by simp
  next
    case (Cons y tl)
    note ys = ‹ys=y#tl›
    with tab-vn tab'-vn
    have (tab(x↦y)) vn = Some el
      by − (rule Cons.hyps,auto)
    moreover from tab'-vn ys
    have (tab'(x↦y, xs[↦]tl)) vn = None
      by simp
    hence (tab'(x↦y)) vn = None
      by (rule map-upds-None-cut)
    ultimately show tab vn = Some el
      by (rule map-upd-cut-irrelevant)
  qed
qed


lemma dom-vname-split:
 dom (case-lname (case-ename (tab(x↦y, xs[↦]ys)) a) b)
   = dom (case-lname (case-ename (tab(x↦y)) a) b) ∪
     dom (case-lname (case-ename (tab(xs[↦]ys)) a) b)
  (is ?List x xs y ys = ?Hd x y ∪ ?Tl xs ys)
proof
  show ?List x xs y ys ⊆ ?Hd x y ∪ ?Tl xs ys
  proof
    fix el
    assume el-in-list: el ∈ ?List x xs y ys
    show el ∈  ?Hd x y ∪ ?Tl xs ys
    proof (cases el)
      case This
      with el-in-list show ?thesis by (simp add: dom-def)
    next
      case (EName en)
      show ?thesis
      proof (cases en)
        case Res
        with EName el-in-list show ?thesis by (simp add: dom-def)
      next
        case (VNam vn)
        with EName el-in-list show ?thesis
          by (auto simp add: dom-def dest: map-upds-cut-irrelevant)
      qed
    qed
  qed
next
```

```
  show ?Hd x y ∪ ?Tl xs ys  ⊆ ?List x xs y ys
  proof (rule subsetI)
    fix el
    assume  el-in-hd-tl: el ∈  ?Hd x y ∪ ?Tl xs ys
    show el ∈ ?List x xs y ys
    proof (cases el)
      case This
      with el-in-hd-tl show ?thesis by (simp add: dom-def)
    next
      case (EName en)
      show ?thesis
      proof (cases en)
        case Res
        with EName el-in-hd-tl show ?thesis by (simp add: dom-def)
      next
        case (VNam vn)
        with EName el-in-hd-tl show ?thesis
          by (auto simp add: dom-def intro: map-upds-Some-expand
                                   map-upds-Some-insert)
      qed
    qed
  qed
qed


lemma dom-map-upd: ⋀ tab. dom (tab(x↦y)) = dom tab ∪ {x}
by (auto simp add: dom-def fun-upd-def)


lemma dom-map-upds: ⋀ tab ys. length xs = length ys
  ⟹ dom (tab(xs[↦]ys)) = dom tab ∪ set xs
proof (induct xs)
  case Nil thus ?case by (simp add: dom-def)
next
  case (Cons x xs tab ys)
  note Hyp = Cons.hyps
  note len = ‹length (x#xs)=length ys›
  show ?case
  proof (cases ys)
    case Nil with len show ?thesis by simp
  next
    case (Cons y tl)
    with len have dom (tab(x↦y, xs[↦]tl)) = dom (tab(x↦y)) ∪ set xs
      by − (rule Hyp,simp)
    moreover
    have dom (tab(x↦hd ys)) = dom tab ∪ {x}
      by (rule dom-map-upd)
    ultimately
    show ?thesis using Cons
      by simp
  qed
qed


lemma dom-case-ename-None-simp:
 dom (case-ename vname-tab None) = VNam ‘ (dom vname-tab)
  apply (auto simp add: dom-def image-def )
  apply (case-tac x)
  apply auto
```

done

lemma *dom-case-ename-Some-simp*:
dom (*case-ename vname-tab* (*Some a*)) = *VNam* ‘ (*dom vname-tab*) ∪ {*Res*}
apply (*auto simp add*: *dom-def image-def*)
apply (*case-tac x*)
apply *auto*
done

lemma *dom-case-lname-None-simp*:
dom (*case-lname ename-tab None*) = *EName* ‘ (*dom ename-tab*)
apply (*auto simp add*: *dom-def image-def*)
apply (*case-tac x*)
apply *auto*
done

lemma *dom-case-lname-Some-simp*:
dom (*case-lname ename-tab* (*Some a*)) = *EName* ‘ (*dom ename-tab*) ∪ {*This*}
apply (*auto simp add*: *dom-def image-def*)
apply (*case-tac x*)
apply *auto*
done

lemmas *dom-lname-case-ename-simps* =
dom-case-ename-None-simp dom-case-ename-Some-simp
dom-case-lname-None-simp dom-case-lname-Some-simp

lemma *image-comp*:
f ‘ *g* ‘ *A* = (*f* ∘ *g*) ‘ *A*
by (*auto simp add*: *image-def*)

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 −
from *m*
have *static-m'*: *is-static m* = *static m*
by *simp*
from *len*
have *dom-vnames*: *dom* (*Map.empty*(*pars m*[↦]*pvs*))=*set* (*pars m*)
by (*simp add*: *dom-map-upds*)
show *?thesis*
proof (*cases static m*)
case *True*
with *static-m' dom-vnames m*
show *?thesis*
by (*cases s*) (*simp add*: *init-lvars-def Let-def parameters-def*
dom-lname-case-ename-simps image-comp)
next
case *False*
with *static-m' dom-vnames m*

```
    show ?thesis
      by (cases s) (simp add: init-lvars-def Let-def parameters-def
                              dom-lname-case-ename-simps image-comp)
  qed
qed


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 −
  note inj-term-simps [simp]
  from da obtain E1 where
    da-e1: (|prg=G,cls=accC,lcl=L|) ⊢ dom (locals (store s0)) »⟨e1⟩e» E1
    by cases simp+
  obtain E2 where
    (|prg=G,cls=accC,lcl=L|)⊢ dom (locals (store s1))
      »(if need-second-arg binop v1 then ⟨e2⟩e else ⟨Skip⟩s)» E2
  proof (cases need-second-arg binop v1)
    case False
    obtain S where
      daSkip: (|prg=G,cls=accC,lcl=L|)
                 ⊢ dom (locals (store s1)) »⟨Skip⟩s» S
      by (auto intro: da-Skip [simplified] assigned.select-convs)
    thus ?thesis
      using that by (simp add: False)
  next
    case True
    from eval-e1 have
      s0-s1:dom (locals (store s0)) ⊆ dom (locals (store s1))
      by (rule dom-locals-eval-mono-elim)
    {
      assume condAnd: binop=CondAnd
      have ?thesis
      proof −
        from da obtain E2' where
          (|prg=G,cls=accC,lcl=L|)
             ⊢ dom (locals (store s0)) ∪ assigns-if True e1 »⟨e2⟩e» E2'
          by cases (simp add: condAnd)+
        moreover
        have dom (locals (store s0))
          ∪ assigns-if True e1 ⊆ dom (locals (store s1))
        proof −
          from condAnd wt-binop have e1T: e1T=PrimT Boolean
            by simp
          with normal-s1 conf-v1 obtain b where v1=Bool b
            by (auto dest: conf-Boolean)
          with True condAnd
```

```
      have v1: v1=Bool True
        by simp
      from eval-e1 normal-s1
      have assigns-if True e1 ⊆ dom (locals (store s1))
        by (rule assigns-if-good-approx' [elim-format])
           (insert wt-e1, simp-all add: e1T v1)
      with s0-s1 show ?thesis by (rule Un-least)
    qed
    ultimately
    show ?thesis
      using that by (cases rule: da-weakenE) (simp add: True)
  qed
}
moreover
{
  assume condOr: binop=CondOr
  have ?thesis

  proof −
    from da obtain E2' where
      (|prg=G,cls=accC,lcl=L|)
         ⊢ dom (locals (store s0)) ∪ assigns-if False e1 »⟨e2⟩e» E2'
      by cases (simp add: condOr)+
    moreover
    have dom (locals (store s0))
                 ∪ assigns-if False e1 ⊆ dom (locals (store s1))
    proof −
      from condOr wt-binop have e1T: e1T=PrimT Boolean
        by simp
      with normal-s1 conf-v1 obtain b where v1=Bool b
        by (auto dest: conf-Boolean)
      with True condOr
      have v1: v1=Bool False
        by simp
      from eval-e1 normal-s1
      have assigns-if False e1 ⊆ dom (locals (store s1))
        by (rule assigns-if-good-approx' [elim-format])
           (insert wt-e1, simp-all add: e1T v1)
      with s0-s1 show ?thesis by (rule Un-least)
    qed
    ultimately
    show ?thesis
      using that by (rule da-weakenE) (simp add: True)
  qed
}
moreover
{
  assume notAndOr: binop≠CondAnd binop≠CondOr
  have ?thesis
  proof −
    from da notAndOr obtain E1' where
      da-e1: (|prg=G,cls=accC,lcl=L|)
             ⊢ dom (locals (store s0)) »⟨e1⟩e» E1'
      and da-e2: (|prg=G,cls=accC,lcl=L|)⊢ nrm E1' »In1l e2» A
      by cases simp+
    from eval-e1 wt-e1 da-e1 wf normal-s1
    have nrm E1' ⊆ dom (locals (store s1))
      by (cases rule: da-good-approxE') iprover
    with da-e2 show ?thesis
```

```
        using that by (rule da-weakenE) (simp add: True)
      qed
    }
    ultimately show ?thesis
      by (cases binop) auto
  qed
  thus ?thesis ..
qed
```

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 −
  note inj-term-simps [simp]
  let ?TypeSafeObj = λ s0 s1 t v.
        ∀ L accC T A. s0::≼(G,L) ⟶ (|prg=G,cls=accC,lcl=L|)⊢t::T
                    ⟶ (|prg=G,cls=accC,lcl=L|)⊢dom (locals (store s0))»t»A
                    ⟶ s1::≼(G,L) ∧ (normal s1 ⟶ G,L,store s1⊢t≻v::≼T)
                       ∧ (error-free s0 = error-free s1)
  from eval
  have ⋀ L accC T A. [[s0::≼(G,L);(|prg=G,cls=accC,lcl=L|)⊢t::T;
                    (|prg=G,cls=accC,lcl=L|)⊢dom (locals (store s0))»t»A]]
       ⟹ s1::≼(G,L) ∧ (normal s1 ⟶ G,L,store s1⊢t≻v::≼T)
          ∧ (error-free s0 = error-free s1)
   (is PROP ?TypeSafe s0 s1 t v
    is ⋀ L accC T A. ?Conform L s0 ⟹ ?WellTyped L accC T t
               ⟹ ?DefAss L accC s0 t A
               ⟹ ?Conform L s1 ∧ ?ValueTyped L T s1 t v ∧
                  ?ErrorFree s0 s1)
  proof (induct)
    case (Abrupt xc s t L accC T A)
    from ‹(Some xc, s)::≼(G,L)›
    show (Some xc, s)::≼(G,L) ∧
      (normal (Some xc, s)
      ⟶ G,L,store (Some xc,s)⊢t≻undefined3 t::≼T) ∧
      (error-free (Some xc, s) = error-free (Some xc, s))
      by simp
  next
    case (Skip s L accC T A)
    from ‹Norm s::≼(G, L)› and
      ‹(|prg = G, cls = accC, lcl = L|)⊢In1r Skip::T›
    show Norm s::≼(G, L) ∧
              (normal (Norm s) ⟶ G,L,store (Norm s)⊢In1r Skip≻◇::≼T) ∧
              (error-free (Norm s) = error-free (Norm s))
      by simp
  next
    case (Expr s0 e v s1 L accC T A)
    note ‹G⊢Norm s0 −e−≻v→ s1›
    note hyp = ‹PROP ?TypeSafe (Norm s0) s1 (In1l e) (In1 v)›
    note conf-s0 = ‹Norm s0::≼(G, L)›
    moreover
    note wt = ‹(|prg = G, cls = accC, lcl = L|)⊢In1r (Expr e)::T›
```

then obtain *eT*
  where (|*prg* = *G*, *cls* = *accC*, *lcl* = *L*|)⊢*In1l e*::*eT*
  by (*rule wt-elim-cases*) *blast*
moreover
from *Expr.prems* **obtain** *E* **where**
  (|*prg*=*G*,*cls*=*accC*, *lcl*=*L*|)⊢*dom* (*locals* (*store* ((*Norm s0*)::*state*)))»*In1l e*»*E*
  by (*elim da-elim-cases*) *simp*
ultimately
obtain *s1*::≼(*G*, *L*) **and** *error-free s1*
  by (*rule hyp* [*elim-format*]) *simp*
with *wt*
show *s1*::≼(*G*, *L*) ∧
    (*normal s1* ⟶ *G*,*L*,*store s1*⊢*In1r* (*Expr e*)≻◇::≼*T*) ∧
    (*error-free* (*Norm s0*) = *error-free s1*)
  by (*simp*)
next
case (*Lab s0 c s1 l L accC T A*)
note *hyp* = ‹*PROP ?TypeSafe* (*Norm s0*) *s1* (*In1r c*) ◇›
note *conf-s0* = ‹*Norm s0*::≼(*G*, *L*)›
moreover
note *wt* = ‹(|*prg* = *G*, *cls* = *accC*, *lcl* = *L*|)⊢*In1r* (*l*• *c*)::*T*›
then have (|*prg* = *G*, *cls* = *accC*, *lcl* = *L*|)⊢*c*::√
  by (*rule wt-elim-cases*) *blast*
moreover from *Lab.prems* **obtain** *C* **where**
  (|*prg*=*G*,*cls*=*accC*, *lcl*=*L*|)⊢*dom* (*locals* (*store* ((*Norm s0*)::*state*)))»*In1r c*»*C*
  by (*elim da-elim-cases*) *simp*
ultimately
obtain *conf-s1*: *s1*::≼(*G*, *L*) **and**
    error-free-s1: *error-free s1*
  by (*rule hyp* [*elim-format*]) *simp*
from *conf-s1* **have** *abupd* (*absorb l*) *s1*::≼(*G*, *L*)
  by (*cases s1*) (*auto intro*: *conforms-absorb*)
with *wt error-free-s1*
show *abupd* (*absorb l*) *s1*::≼(*G*, *L*) ∧
    (*normal* (*abupd* (*absorb l*) *s1*)
    ⟶ *G*,*L*,*store* (*abupd* (*absorb l*) *s1*)⊢*In1r* (*l*• *c*)≻◇::≼*T*) ∧
    (*error-free* (*Norm s0*) = *error-free* (*abupd* (*absorb l*) *s1*))
  by (*simp*)
next
case (*Comp s0 c1 s1 c2 s2 L accC T A*)
note *eval-c1* = ‹*G*⊢*Norm s0* −*c1*→ *s1*›
note *eval-c2* = ‹*G*⊢*s1* −*c2*→ *s2*›
note *hyp-c1* = ‹*PROP ?TypeSafe* (*Norm s0*) *s1* (*In1r c1*) ◇›
note *hyp-c2* = ‹*PROP ?TypeSafe s1* *s2* (*In1r c2*) ◇›
note *conf-s0* = ‹*Norm s0*::≼(*G*, *L*)›
note *wt* = ‹(|*prg* = *G*, *cls* = *accC*, *lcl* = *L*|)⊢*In1r* (*c1*;; *c2*)::*T*›
then obtain *wt-c1*: (|*prg* = *G*, *cls* = *accC*, *lcl* = *L*|)⊢*c1*::√ **and**
    wt-c2: (|*prg* = *G*, *cls* = *accC*, *lcl* = *L*|)⊢*c2*::√
  by (*rule wt-elim-cases*) *blast*
from *Comp.prems*
obtain *C1 C2*
  where *da-c1*: (|*prg*=*G*, *cls*=*accC*, *lcl*=*L*|)⊢
      dom (*locals* (*store* ((*Norm s0*)::*state*))) »*In1r c1*» *C1* **and**
    da-c2: (|*prg*=*G*, *cls*=*accC*, *lcl*=*L*|)⊢ *nrm C1* »*In1r c2*» *C2*
  by (*elim da-elim-cases*) *simp*
from *conf-s0 wt-c1 da-c1*
obtain *conf-s1*: *s1*::≼(*G*, *L*) **and**
    error-free-s1: *error-free s1*
  by (*rule hyp-c1* [*elim-format*]) *simp*

```
    show s2::≼(G, L) ∧
          (normal s2 ⟶ G,L,store s2⊢In1r (c1;; c2)≻◇::≼T) ∧
          (error-free (Norm s0) = error-free s2)
    proof (cases normal s1)
      case False
      with eval-c2 have s2=s1 by auto
      with conf-s1 error-free-s1 False wt show ?thesis
        by simp
    next
      case True
      obtain C2' where
        ⦇prg=G, cls=accC, lcl=L⦈⊢ dom (locals (store s1)) »In1r c2» C2'
      proof -
        from eval-c1 wt-c1 da-c1 wf True
        have nrm C1 ⊆ dom (locals (store s1))
          by (cases rule: da-good-approxE') iprover
        with da-c2 show thesis
          by (rule da-weakenE) (rule that)
      qed
      with conf-s1 wt-c2
      obtain s2::≼(G, L) and error-free s2
        by (rule hyp-c2 [elim-format]) (simp add: error-free-s1)
      thus ?thesis
        using wt by simp
    qed
  next
    case (If s0 e b s1 c1 c2 s2 L accC T A)
    note eval-e = ‹G⊢Norm s0 −e−≻b→ s1›
    note eval-then-else = ‹G⊢s1 −(if the-Bool b then c1 else c2)→ s2›
    note hyp-e = ‹PROP ?TypeSafe (Norm s0) s1 (In1l e) (In1 b)›
    note hyp-then-else =
      ‹PROP ?TypeSafe s1 s2 (In1r (if the-Bool b then c1 else c2)) ◇›
    note conf-s0 = ‹Norm s0::≼(G, L)›
    note wt = ‹⦇prg = G, cls = accC, lcl = L⦈⊢In1r (If(e) c1 Else c2)::T›
    then obtain
              wt-e: ⦇prg=G, cls=accC, lcl=L⦈⊢e::−PrimT Boolean and
      wt-then-else: ⦇prg=G, cls=accC, lcl=L⦈⊢(if the-Bool b then c1 else c2)::√

      by (rule wt-elim-cases) auto
    from If.prems obtain E C where
      da-e: ⦇prg=G,cls=accC,lcl=L⦈⊢ dom (locals (store ((Norm s0)::state)))
                                       »In1l e» E and
      da-then-else:
      ⦇prg=G,cls=accC,lcl=L⦈⊢
         (dom (locals (store ((Norm s0)::state))) ∪ assigns-if (the-Bool b) e)
          »In1r (if the-Bool b then c1 else c2)» C

      by (elim da-elim-cases) (cases the-Bool b,auto)
    from conf-s0 wt-e da-e
    obtain conf-s1: s1::≼(G, L) and error-free-s1: error-free s1
      by (rule hyp-e [elim-format]) simp
    show s2::≼(G, L) ∧
           (normal s2 ⟶ G,L,store s2⊢In1r (If(e) c1 Else c2)≻◇::≼T) ∧
           (error-free (Norm s0) = error-free s2)
    proof (cases normal s1)
      case False
      with eval-then-else have s2=s1 by auto
      with conf-s1 error-free-s1 False wt show ?thesis
        by simp
```

```
    next
      case True
      obtain C' where
        (|prg=G,cls=accC,lcl=L|)⊢
          (dom (locals (store s1)))»In1r (if the-Bool b then c1 else c2)» C'
      proof -
        from eval-e have
          dom (locals (store ((Norm s0)::state))) ⊆ dom (locals (store s1))
          by (rule dom-locals-eval-mono-elim)
        moreover
        from eval-e True wt-e
        have assigns-if (the-Bool b) e ⊆ dom (locals (store s1))
          by (rule assigns-if-good-approx')
        ultimately
        have dom (locals (store ((Norm s0)::state)))
                ∪ assigns-if (the-Bool b) e ⊆ dom (locals (store s1))
          by (rule Un-least)
        with da-then-else show thesis
          by (rule da-weakenE) (rule that)
      qed
      with conf-s1 wt-then-else
      obtain s2::≼(G, L) and error-free s2
        by (rule hyp-then-else [elim-format]) (simp add: error-free-s1)
      with wt show ?thesis
        by simp
    qed
```

— Note that we don't have to show that *b* really is a boolean value. With *the-Bool* we enforce to get a value of boolean type. So execution will be type safe, even if b would be a string, for example. We might not expect such a behaviour to be called type safe. To remedy the situation we would have to change the evaulation rule, so that it only has a type safe evaluation if we actually get a boolean value for the condition. That b is actually a boolean value is part of *hyp-e*. See also Loop

```
  next
    case (Loop s0 e b s1 c s2 l s3 L accC T A)
    note eval-e = ‹G⊢Norm s0 −e−≻b→ s1›
    note hyp-e = ‹PROP ?TypeSafe (Norm s0) s1 (In1l e) (In1 b)›
    note conf-s0 = ‹Norm s0::≼(G, L)›
    note wt = ‹(|prg = G, cls = accC, lcl = L|)⊢In1r (l· While(e) c)::T›
    then obtain wt-e: (|prg = G, cls = accC, lcl = L|)⊢e::−PrimT Boolean and
                wt-c: (|prg = G, cls = accC, lcl = L|)⊢c::√
      by (rule wt-elim-cases) blast
    note da = ‹(|prg=G, cls=accC, lcl=L|)
            ⊢ dom (locals(store ((Norm s0)::state))) »In1r (l· While(e) c)» A›
    then
    obtain E C where
      da-e: (|prg=G, cls=accC, lcl=L|)
              ⊢ dom (locals (store ((Norm s0)::state))) »In1l e» E and
      da-c: (|prg=G, cls=accC, lcl=L|)
              ⊢ (dom (locals (store ((Norm s0)::state)))
                  ∪ assigns-if True e) »In1r c» C
      by (rule da-elim-cases) simp
    from conf-s0 wt-e da-e
    obtain conf-s1: s1::≼(G, L) and error-free-s1: error-free s1
      by (rule hyp-e [elim-format]) simp
    show s3::≼(G, L) ∧
          (normal s3 ⟶ G,L,store s3⊢In1r (l· While(e) c)≻◇::≼T) ∧
          (error-free (Norm s0) = error-free s3)
    proof (cases normal s1)
      case True
      note normal-s1 = this
```

```
    show ?thesis
    proof (cases the-Bool b)
      case True
      with Loop.hyps obtain
        eval-c: G⊢s1 −c→ s2 and
        eval-while: G⊢abupd (absorb (Cont l)) s2 −l· While(e) c→ s3
        by simp
      have ?TypeSafeObj s1 s2 (In1r c) ◊
        using Loop.hyps True by simp
      note hyp-c = this [rule-format]
      have ?TypeSafeObj (abupd (absorb (Cont l)) s2)
        s3 (In1r (l· While(e) c)) ◊
        using Loop.hyps True by simp
      note hyp-w = this [rule-format]
      from eval-e have
        s0-s1: dom (locals (store ((Norm s0)::state)))
                 ⊆ dom (locals (store s1))
        by (rule dom-locals-eval-mono-elim)
      obtain C' where
        (|prg=G, cls=accC, lcl=L|)⊢(dom (locals (store s1)))»In1r c» C'
      proof −
        note s0-s1
        moreover
        from eval-e normal-s1 wt-e
        have assigns-if True e ⊆ dom (locals (store s1))
          by (rule assigns-if-good-approx' [elim-format]) (simp add: True)
        ultimately
        have dom (locals (store ((Norm s0)::state)))
               ∪ assigns-if True e ⊆ dom (locals (store s1))
          by (rule Un-least)
        with da-c show thesis
          by (rule da-weakenE) (rule that)
      qed
      with conf-s1 wt-c
      obtain conf-s2: s2::≼(G, L) and error-free-s2: error-free s2
        by (rule hyp-c [elim-format]) (simp add: error-free-s1)
      from error-free-s2
      have error-free-ab-s2: error-free (abupd (absorb (Cont l)) s2)
        by simp
      from conf-s2 have abupd (absorb (Cont l)) s2 ::≼(G, L)
        by (cases s2) (auto intro: conforms-absorb)
      moreover note wt
      moreover
      obtain A' where
        (|prg=G,cls=accC,lcl=L|)⊢
            dom (locals(store (abupd (absorb (Cont l)) s2)))
              »In1r (l· While(e) c)» A'
      proof −
        note s0-s1
        also from eval-c
        have dom (locals (store s1)) ⊆ dom (locals (store s2))
          by (rule dom-locals-eval-mono-elim)
        also have . . . ⊆ dom (locals (store (abupd (absorb (Cont l)) s2)))
          by simp
        finally
        have dom (locals (store ((Norm s0)::state))) ⊆ . . . .
        with da show thesis
          by (rule da-weakenE) (rule that)
      qed
```

```
      ultimately obtain s3::≼(G, L) and error-free s3
        by (rule hyp-w [elim-format]) (simp add: error-free-ab-s2)
      with wt show ?thesis
        by simp
    next
      case False
      with Loop.hyps have s3=s1 by simp
      with conf-s1 error-free-s1 wt
      show ?thesis
        by simp
    qed
  next
    case False
    have s3=s1
    proof −
      from False obtain abr where abr: abrupt s1 = Some abr
        by (cases s1) auto
      from eval-e - wt-e have no-jmp: ⋀ j. abrupt s1 ≠ Some (Jump j)
        by (rule eval-expression-no-jump
             [where ?Env=(|prg=G,cls=accC,lcl=L|),simplified])
           (simp-all add: wf)

      show ?thesis
      proof (cases the-Bool b)
        case True
        with Loop.hyps obtain
          eval-c: G⊢s1 −c→ s2 and
          eval-while: G⊢abupd (absorb (Cont l)) s2 −l• While(e) c→ s3
          by simp
        from eval-c abr have s2=s1 by auto
        moreover from calculation no-jmp have abupd (absorb (Cont l)) s2=s2
          by (cases s1) (simp add: absorb-def)
        ultimately show ?thesis
          using eval-while abr
          by auto
      next
        case False
        with Loop.hyps show ?thesis by simp
      qed
    qed
    with conf-s1 error-free-s1 wt
    show ?thesis
      by simp
  qed
next
  case (Jmp s j L accC T A)
  note ‹Norm s::≼(G, L)›
  moreover
  from Jmp.prems
  have j=Ret ⟶ Result ∈ dom (locals (store ((Norm s)::state)))
    by (elim da-elim-cases)
  ultimately have (Some (Jump j), s)::≼(G, L) by auto
  then
  show (Some (Jump j), s)::≼(G, L) ∧
         (normal (Some (Jump j), s)
         ⟶ G,L,store (Some (Jump j), s)⊢In1r (Jmp j)≻◇::≼T) ∧
         (error-free (Norm s) = error-free (Some (Jump j), s))
    by simp
next
```

case (*Throw s0 e a s1 L accC T A*)
note ‹*G*⊢*Norm s0* −*e*−≻*a*→ *s1*›
note *hyp* = ‹*PROP ?TypeSafe* (*Norm s0*) *s1* (*In1l e*) (*In1 a*)›
note *conf-s0* = ‹*Norm s0*::≼(*G*, *L*)›
note *wt* = ‹⦇*prg* = *G*, *cls* = *accC*, *lcl* = *L*⦈⊢*In1r* (*Throw e*)::*T*›
then obtain *tn*
where *wt-e*: ⦇*prg* = *G*, *cls* = *accC*, *lcl* = *L*⦈⊢*e*::−*Class tn* **and**
throwable: *G*⊢*tn*$\preceq_C$ *SXcpt Throwable*
by (*rule wt-elim-cases*) (*auto*)
from *Throw.prems* **obtain** *E* **where**
da-e: ⦇*prg*=*G*,*cls*=*accC*,*lcl*=*L*⦈
⊢ *dom* (*locals* (*store* ((*Norm s0*)::*state*))) »*In1l e*» *E*
by (*elim da-elim-cases*) *simp*
from *conf-s0 wt-e da-e* **obtain**
s1::≼(*G*, *L*) **and**
(*normal s1* ⟶ *G*,*store s1*⊢*a*::≼*Class tn*) **and**
error-free-s1: *error-free s1*
by (*rule hyp* [*elim-format*]) *simp*
with *wf throwable*
have *abupd* (*throw a*) *s1*::≼(*G*, *L*)
by (*cases s1*) (*auto dest*: *Throw-lemma*)
with *wt error-free-s1*
show *abupd* (*throw a*) *s1*::≼(*G*, *L*) ∧
(*normal* (*abupd* (*throw a*) *s1*) ⟶
G,*L*,*store* (*abupd* (*throw a*) *s1*)⊢*In1r* (*Throw e*)≻◇::≼*T*) ∧
(*error-free* (*Norm s0*) = *error-free* (*abupd* (*throw a*) *s1*))
by *simp*
next
case (*Try s0 c1 s1 s2 catchC vn c2 s3 L accC T A*)
note *eval-c1* = ‹*G*⊢*Norm s0* −*c1*→ *s1*›
note *sx-alloc* = ‹*G*⊢*s1* −*sxalloc*→ *s2*›
note *hyp-c1* = ‹*PROP ?TypeSafe* (*Norm s0*) *s1* (*In1r c1*) ◇›
note *conf-s0* = ‹*Norm s0*::≼(*G*, *L*)›
note *wt* = ‹⦇*prg*=*G*,*cls*=*accC*,*lcl*=*L*⦈⊢*In1r* (*Try c1 Catch*(*catchC vn*) *c2*)::*T*›
then obtain
wt-c1: ⦇*prg*=*G*,*cls*=*accC*,*lcl*=*L*⦈⊢*c1*::√ **and**
wt-c2: ⦇*prg*=*G*,*cls*=*accC*,*lcl*=*L*(*VName vn*↦*Class catchC*)⦈⊢*c2*::√ **and**
fresh-vn: *L*(*VName vn*)=*None*
by (*rule wt-elim-cases*) *simp*
from *Try.prems* **obtain** *C1 C2* **where**
da-c1: ⦇*prg*=*G*,*cls*=*accC*,*lcl*=*L*⦈
⊢ *dom* (*locals* (*store* ((*Norm s0*)::*state*))) »*In1r c1*» *C1* **and**
da-c2:
⦇*prg*=*G*,*cls*=*accC*,*lcl*=*L*(*VName vn*↦*Class catchC*)⦈
⊢ (*dom* (*locals* (*store* ((*Norm s0*)::*state*))) ∪ {*VName vn*}) »*In1r c2*» *C2*
by (*elim da-elim-cases*) *simp*
from *conf-s0 wt-c1 da-c1*
obtain *conf-s1*: *s1*::≼(*G*, *L*) **and** *error-free-s1*: *error-free s1*
by (*rule hyp-c1* [*elim-format*]) *simp*
from *conf-s1 sx-alloc wf*
have *conf-s2*: *s2*::≼(*G*, *L*)
by (*auto dest*: *sxalloc-type-sound split*: *option.splits abrupt.splits*)
from *sx-alloc error-free-s1*
have *error-free-s2*: *error-free s2*
by (*rule error-free-sxalloc*)
show *s3*::≼(*G*, *L*) ∧
(*normal s3* ⟶ *G*,*L*,*store s3*⊢*In1r* (*Try c1 Catch*(*catchC vn*) *c2*)≻◇::≼*T*)∧
(*error-free* (*Norm s0*) = *error-free s3*)
proof (*cases* ∃ *x*. *abrupt s1* = *Some* (*Xcpt x*))

```
    case False
    from sx-alloc wf
    have eq-s2-s1: s2=s1
      by (rule sxalloc-type-sound [elim-format])
         (insert False, auto split: option.splits abrupt.splits )
    with False
    have ¬ G,s2⊢catch catchC
      by (simp add: catch-def)
    with Try
    have s3=s2
      by simp
    with wt conf-s1 error-free-s1 eq-s2-s1
    show ?thesis
      by simp
  next
    case True
    note exception-s1 = this
    show ?thesis
    proof (cases G,s2⊢catch catchC)
      case False
      with Try
      have s3=s2
        by simp
      with wt conf-s2 error-free-s2
      show ?thesis
        by simp
    next
      case True
      with Try have G⊢new-xcpt-var vn s2 −c2→ s3 by simp
      from True Try.hyps
      have ?TypeSafeObj (new-xcpt-var vn s2) s3 (In1r c2) ♢
        by simp
      note hyp-c2 = this [rule-format]
      from exception-s1 sx-alloc wf
      obtain a
        where xcpt-s2: abrupt s2 = Some (Xcpt (Loc a))
        by (auto dest!: sxalloc-type-sound split: option.splits abrupt.splits)
      with True
      have G⊢obj-ty (the (globs (store s2) (Heap a)))≼Class catchC
        by (cases s2) simp
      with xcpt-s2 conf-s2 wf
      have new-xcpt-var vn s2 ::≼(G, L(VName vn↦Class catchC))
        by (auto dest: Try-lemma)
      moreover note wt-c2
      moreover
      obtain C2' where
        ⦇prg=G,cls=accC,lcl=L(VName vn↦Class catchC)⦈
        ⊢ (dom (locals (store (new-xcpt-var vn s2)))) »In1r c2» C2'
      proof −
        have (dom (locals (store ((Norm s0)::state))) ∪ {VName vn})
                ⊆ dom (locals (store (new-xcpt-var vn s2)))
        proof −
          from ‹G⊢Norm s0 −c1→ s1›
          have dom (locals (store ((Norm s0)::state)))
                  ⊆ dom (locals (store s1))
            by (rule dom-locals-eval-mono-elim)
          also
          from sx-alloc
          have ... ⊆ dom (locals (store s2))
```

```
          by (rule dom-locals-sxalloc-mono)
        also
        have ... ⊆ dom (locals (store (new-xcpt-var vn s2)))
          by (cases s2) (simp add: new-xcpt-var-def, blast)
        also
        have {VName vn} ⊆ ...
          by (cases s2) simp
        ultimately show ?thesis
          by (rule Un-least)
      qed
      with da-c2 show thesis
        by (rule da-weakenE) (rule that)
    qed
    ultimately
    obtain       conf-s3: s3::≼(G, L(VName vn↦Class catchC)) and
           error-free-s3: error-free s3
      by (rule hyp-c2 [elim-format])
         (cases s2, simp add: xcpt-s2 error-free-s2)
    from conf-s3 fresh-vn
    have s3::≼(G,L)
      by (blast intro: conforms-deallocL)
    with wt error-free-s3
    show ?thesis
      by simp
  qed
 qed
next
 case (Fin s0 c1 x1 s1 c2 s2 s3 L accC T A)
 note eval-c1 = ‹G⊢Norm s0 −c1→ (x1, s1)›
 note eval-c2 = ‹G⊢Norm s1 −c2→ s2›
 note s3 = ‹s3 = (if ∃ err. x1 = Some (Error err)
                  then (x1, s1)
                  else abupd (abrupt-if (x1 ≠ None) x1) s2)›
 note hyp-c1 = ‹PROP ?TypeSafe (Norm s0) (x1,s1) (In1r c1) ♢›
 note hyp-c2 = ‹PROP ?TypeSafe (Norm s1) s2      (In1r c2) ♢›
 note conf-s0 = ‹Norm s0::≼(G, L)›
 note wt = ‹(|prg = G, cls = accC, lcl = L|)⊢In1r (c1 Finally c2)::T›
 then obtain
   wt-c1: (|prg=G,cls=accC,lcl=L|)⊢c1::√ and
   wt-c2: (|prg=G,cls=accC,lcl=L|)⊢c2::√
   by (rule wt-elim-cases) blast
 from Fin.prems obtain C1 C2 where
   da-c1: (|prg=G,cls=accC,lcl=L|)
          ⊢ dom (locals (store ((Norm s0)::state))) »In1r c1» C1 and
   da-c2: (|prg=G,cls=accC,lcl=L|)
          ⊢ dom (locals (store ((Norm s0)::state))) »In1r c2» C2
   by (elim da-elim-cases) simp
 from conf-s0 wt-c1 da-c1
 obtain conf-s1: (x1,s1)::≼(G, L) and error-free-s1: error-free (x1,s1)
   by (rule hyp-c1 [elim-format]) simp
 from conf-s1 have Norm s1::≼(G, L)
   by (rule conforms-NormI)
 moreover note wt-c2
 moreover obtain C2'
   where (|prg=G,cls=accC,lcl=L|)
          ⊢ dom (locals (store ((Norm s1)::state))) »In1r c2» C2'
 proof −
   from eval-c1
   have dom (locals (store ((Norm s0)::state)))
```

```
           ⊆ dom (locals (store (x1,s1)))
      by (rule dom-locals-eval-mono-elim)
    hence dom (locals (store ((Norm s0)::state)))
            ⊆ dom (locals (store ((Norm s1)::state)))
      by simp
    with da-c2 show thesis
      by (rule da-weakenE) (rule that)
  qed
  ultimately
  obtain conf-s2: s2::≼(G, L) and error-free-s2: error-free s2
    by (rule hyp-c2 [elim-format]) simp
  from error-free-s1 s3
  have s3': s3=abupd (abrupt-if (x1 ≠ None) x1) s2
    by simp
  show s3::≼(G, L) ∧
       (normal s3 ⟶ G,L,store s3 ⊢In1r (c1 Finally c2)≻◇::≼T) ∧
       (error-free (Norm s0) = error-free s3)
  proof (cases x1)
    case None with conf-s2 s3' wt error-free-s2
    show ?thesis by auto
  next
    case (Some x)
    from eval-c2 have
      dom (locals (store ((Norm s1)::state))) ⊆ dom (locals (store s2))
      by (rule dom-locals-eval-mono-elim)
    with Some eval-c2 wf conf-s1 conf-s2
    have conf: (abrupt-if True (Some x) (abrupt s2), store s2)::≼(G, L)
      by (cases s2) (auto dest: Fin-lemma)
    from Some error-free-s1
    have ¬ (∃ err. x=Error err)
      by (simp add: error-free-def)
    with error-free-s2
    have error-free (abrupt-if True (Some x) (abrupt s2), store s2)
      by (cases s2) simp
    with Some wt conf s3' show ?thesis
      by (cases s2) auto
  qed
next
  case (Init C c s0 s3 s1 s2 L accC T A)
  note cls = ‹the (class G C) = c›
  note conf-s0 = ‹Norm s0::≼(G, L)›
  note wt = ‹⦇prg = G, cls = accC, lcl = L⦈⊢In1r (Init C)::T›
  with cls
  have cls-C: class G C = Some c
    by − (erule wt-elim-cases, auto)
  show s3::≼(G, L) ∧ (normal s3 ⟶ G,L,store s3⊢In1r (Init C)≻◇::≼T) ∧
       (error-free (Norm s0) = error-free s3)
  proof (cases inited C (globs s0))
    case True
    with Init.hyps have s3 = Norm s0
      by simp
    with conf-s0 wt show ?thesis
      by simp
  next
    case False
    with Init.hyps obtain
         eval-init-super:
         G⊢Norm ((init-class-obj G C) s0)
           −(if C = Object then Skip else Init (super c))→ s1 and
```

eval-init: *G*⊢(*set-lvars Map.empty*) *s1* −*init c*→ *s2* **and**
s3: *s3* = (*set-lvars* (*locals* (*store s1*))) *s2*
by *simp*
have *?TypeSafeObj* (*Norm* ((*init-class-obj G C*) *s0*)) *s1*
(*In1r* (*if C* = *Object then Skip else Init* (*super c*))) ♢
using *False Init.hyps* **by** *simp*
note *hyp-init-super* = *this* [*rule-format*]
have *?TypeSafeObj* ((*set-lvars Map.empty*) *s1*) *s2* (*In1r* (*init c*)) ♢
using *False Init.hyps* **by** *simp*
note *hyp-init-c* = *this* [*rule-format*]
from *conf-s0 wf cls-C False*
have (*Norm* ((*init-class-obj G C*) *s0*))::⪯(*G*, *L*)
by (*auto dest*: *conforms-init-class-obj*)
moreover from *wf cls-C* **have**
wt-init-super: ⦇*prg* = *G*, *cls* = *accC*, *lcl* = *L*⦈
⊢(*if C* = *Object then Skip else Init* (*super c*))::√
by (*cases C*=*Object*)
(*auto dest*: *wf-prog-cdecl wf-cdecl-supD is-acc-classD*)
moreover
obtain *S* **where**
da-init-super:
⦇*prg*=*G*,*cls*=*accC*,*lcl*=*L*⦈
⊢ *dom* (*locals* (*store* ((*Norm* ((*init-class-obj G C*) *s0*))::*state*)))
»*In1r* (*if C* = *Object then Skip else Init* (*super c*))» *S*
proof (*cases C*=*Object*)
case *True*
with *da-Skip* **show** *?thesis*
using *that* **by** (*auto intro*: *assigned.select-convs*)
next
case *False*
with *da-Init* **show** *?thesis*
by − (*rule that*, *auto intro*: *assigned.select-convs*)
qed
ultimately
obtain *conf-s1*: *s1*::⪯(*G*, *L*) **and** *error-free-s1*: *error-free s1*
by (*rule hyp-init-super* [*elim-format*]) *simp*
from *eval-init-super wt-init-super wf*
have *s1-no-ret*: ⋀ *j*. *abrupt s1* ≠ *Some* (*Jump j*)
by − (*rule eval-statement-no-jump* [**where** *?Env*=⦇*prg*=*G*,*cls*=*accC*,*lcl*=*L*⦈],
auto)
with *conf-s1*
have (*set-lvars Map.empty*) *s1*::⪯(*G*, *Map.empty*)
by (*cases s1*) (*auto intro*: *conforms-set-locals*)
moreover
from *error-free-s1*
have *error-free-empty*: *error-free* ((*set-lvars Map.empty*) *s1*)
by *simp*
from *cls-C wf* **have** *wt-init-c*: ⦇*prg*=*G*, *cls*=*C*,*lcl*=*Map.empty*⦈⊢(*init c*)::√
by (*rule wf-prog-cdecl* [*THEN wf-cdecl-wt-init*])
moreover from *cls-C wf* **obtain** *I*
where ⦇*prg*=*G*,*cls*=*C*,*lcl*=*Map.empty*⦈⊢ {} »*In1r* (*init c*)» *I*
by (*rule wf-prog-cdecl* [*THEN wf-cdeclE*,*simplified*]) *blast*

then obtain *I′* **where**
⦇*prg*=*G*,*cls*=*C*,*lcl*=*Map.empty*⦈⊢*dom* (*locals* (*store* ((*set-lvars Map.empty*) *s1*)))
»*In1r* (*init c*)» *I′*
by (*rule da-weakenE*) *simp*
ultimately
obtain *conf-s2*: *s2*::⪯(*G*, *Map.empty*) **and** *error-free-s2*: *error-free s2*

```
      by (rule hyp-init-c [elim-format]) (simp add: error-free-empty)
    have abrupt s2 ≠ Some (Jump Ret)
    proof −
      from s1-no-ret
      have ⋀ j. abrupt ((set-lvars Map.empty) s1) ≠ Some (Jump j)
        by simp
      moreover
      from cls-C wf have jumpNestingOkS {} (init c)
        by (rule wf-prog-cdecl [THEN wf-cdeclE])
      ultimately
      show ?thesis
        using eval-init wt-init-c wf
        by − (rule eval-statement-no-jump
                [where ?Env=(|prg=G,cls=C,lcl=Map.empty|)],simp+)
    qed
    with conf-s2 s3 conf-s1 eval-init
    have s3::≼(G, L)
      by (cases s2,cases s1) (force dest: conforms-return eval-gext′)
    moreover from error-free-s2 s3
    have error-free s3
      by simp
    moreover note wt
    ultimately show ?thesis
      by simp
  qed
next
  case (NewC s0 C s1 a s2 L accC T A)
  note ‹G⊢Norm s0 −Init C→ s1›
  note halloc = ‹G⊢s1 −halloc CInst C≻a→ s2›
  note hyp = ‹PROP ?TypeSafe (Norm s0) s1 (In1r (Init C)) ◇›
  note conf-s0 = ‹Norm s0::≼(G, L)›
  moreover
  note wt = ‹(|prg=G, cls=accC, lcl=L|)⊢In1l (NewC C)::T›
  then obtain is-cls-C: is-class G C and
                T: T=Inl (Class C)
    by (rule wt-elim-cases) (auto dest: is-acc-classD)
  hence (|prg=G, cls=accC, lcl=L|)⊢Init C::√ by auto
  moreover obtain I where
    (|prg=G,cls=accC,lcl=L|)
       ⊢ dom (locals (store ((Norm s0)::state))) »In1r (Init C)» I
    by (auto intro: da-Init [simplified] assigned.select-convs)

  ultimately
  obtain conf-s1: s1::≼(G, L) and error-free-s1: error-free s1
    by (rule hyp [elim-format]) simp
  from conf-s1 halloc wf is-cls-C
  obtain halloc-type-safe: s2::≼(G, L)
                         (normal s2 ⟶ G,store s2⊢Addr a::≼Class C)
    by (cases s2) (auto dest!: halloc-type-sound)
  from halloc error-free-s1
  have error-free s2
    by (rule error-free-halloc)
  with halloc-type-safe T
  show s2::≼(G, L) ∧
        (normal s2 ⟶ G,L,store s2⊢In1l (NewC C)≻In1 (Addr a)::≼T) ∧
        (error-free (Norm s0) = error-free s2)
    by auto
next
  case (NewA s0 elT s1 e i s2 a s3 L accC T A)
```

```
note eval-init = ‹G⊢Norm s0 −init-comp-ty elT→ s1›
note eval-e = ‹G⊢s1 −e−≻i→ s2›
note halloc = ‹G⊢abupd (check-neg i) s2−halloc Arr elT (the-Intg i)≻a→ s3›
note hyp-init = ‹PROP ?TypeSafe (Norm s0) s1 (In1r (init-comp-ty elT)) ◇›
note hyp-size = ‹PROP ?TypeSafe s1 s2 (In1l e) (In1 i)›
note conf-s0 = ‹Norm s0::≼(G, L)›
note wt = ‹(|prg = G, cls = accC, lcl = L|)⊢In1l (New elT[e])::T›
then obtain
  wt-init: (|prg = G, cls = accC, lcl = L|)⊢init-comp-ty elT::√ and
  wt-size: (|prg = G, cls = accC, lcl = L|)⊢e::−PrimT Integer and
       elT: is-type G elT and
      T: T=Inl (elT.[])
  by (rule wt-elim-cases) (auto intro: wt-init-comp-ty dest: is-acc-typeD)
from NewA.prems
have da-e:(|prg=G,cls=accC,lcl=L|)
           ⊢ dom (locals (store ((Norm s0)::state))) »In1l e» A
  by (elim da-elim-cases) simp
obtain conf-s1: s1::≼(G, L) and error-free-s1: error-free s1
proof −
  note conf-s0 wt-init
  moreover obtain I where
    (|prg=G,cls=accC,lcl=L|)
     ⊢ dom (locals (store ((Norm s0)::state))) »In1r (init-comp-ty elT)» I
  proof (cases ∃ C. elT = Class C)
    case True
    thus ?thesis
      by − (rule that, (auto intro: da-Init [simplified]
                                   assigned.select-convs
                        simp add: init-comp-ty-def))

  next
    case False
    thus ?thesis
    by − (rule that, (auto intro: da-Skip [simplified]
                                  assigned.select-convs
                     simp add: init-comp-ty-def))

  qed
  ultimately show thesis
    by (rule hyp-init [elim-format]) (auto intro: that)
qed
obtain conf-s2: s2::≼(G, L) and error-free-s2: error-free s2
proof −
  from eval-init
  have dom (locals (store ((Norm s0)::state))) ⊆ dom (locals (store s1))
    by (rule dom-locals-eval-mono-elim)
  with da-e
  obtain A' where
   (|prg=G,cls=accC,lcl=L|)
        ⊢ dom (locals (store s1)) »In1l e» A'
    by (rule da-weakenE)
  with conf-s1 wt-size
  show ?thesis
    by (rule hyp-size [elim-format]) (simp add: that error-free-s1)
qed
from conf-s2 have abupd (check-neg i) s2::≼(G, L)
  by (cases s2) auto
with halloc wf elT
have halloc-type-safe:
```

```
        s3::≼(G, L) ∧ (normal s3 ⟶ G,store s3⊢Addr a::≼elT.[])
      by (cases s3) (auto dest!: halloc-type-sound)
    from halloc error-free-s2
    have error-free s3
      by (auto dest: error-free-halloc)
    with halloc-type-safe T
    show s3::≼(G, L) ∧
          (normal s3 ⟶ G,L,store s3⊢In1l (New elT[e])≻In1 (Addr a)::≼T) ∧
          (error-free (Norm s0) = error-free s3)
      by simp
  next
    case (Cast s0 e v s1 s2 castT L accC T A)
    note ‹G⊢Norm s0 −e−≻v→ s1›
    note s2 = ‹s2 = abupd (raise-if (¬ G,store s1⊢v fits castT) ClassCast) s1›
    note hyp = ‹PROP ?TypeSafe (Norm s0) s1 (In1l e) (In1 v)›
    note conf-s0 = ‹Norm s0::≼(G, L)›
    note wt = ‹(|prg = G, cls = accC, lcl = L|)⊢In1l (Cast castT e)::T›
    then obtain eT
      where wt-e: (|prg = G, cls = accC, lcl = L|)⊢e::−eT and
            eT: G⊢eT≼? castT and
             T: T=Inl castT
      by (rule wt-elim-cases) auto
    from Cast.prems
    have (|prg=G,cls=accC,lcl=L|)
              ⊢ dom (locals (store ((Norm s0)::state))) »In1l e» A
      by (elim da-elim-cases) simp
    with conf-s0 wt-e
    obtain conf-s1: s1::≼(G, L) and
           v-ok: normal s1 ⟶ G,store s1⊢v::≼eT and
      error-free-s1: error-free s1
      by (rule hyp [elim-format]) simp
    from conf-s1 s2
    have conf-s2: s2::≼(G, L)
      by (cases s1) simp
    from error-free-s1 s2
    have error-free-s2: error-free s2
      by simp
    {
      assume norm-s2: normal s2
      have G,L,store s2⊢In1l (Cast castT e)≻In1 v::≼T
      proof −
        from s2 norm-s2 have normal s1
          by (cases s1) simp
        with v-ok
        have G,store s1⊢v::≼eT
          by simp
        with eT wf s2 T norm-s2
        show ?thesis
          by (cases s1) (auto dest: fits-conf)
      qed
    }
    with conf-s2 error-free-s2
    show s2::≼(G, L) ∧
           (normal s2 ⟶ G,L,store s2⊢In1l (Cast castT e)≻In1 v::≼T)  ∧
           (error-free (Norm s0) = error-free s2)
      by blast
  next
    case (Inst s0 e v s1 b instT L accC T A)
    note hyp = ‹PROP ?TypeSafe (Norm s0) s1 (In1l e) (In1 v)›
```

```
    note conf-s0 = ‹Norm s0::≼(G, L)›
    from Inst.prems obtain eT
    where wt-e: ⦇prg = G, cls = accC, lcl = L⦈⊢e::−RefT eT  and
             T: T=Inl (PrimT Boolean)
      by (elim wt-elim-cases) simp
    from Inst.prems
    have da-e: ⦇prg=G,cls=accC,lcl=L⦈
                 ⊢ dom (locals (store ((Norm s0)::state))) »In1l e» A
      by (elim da-elim-cases) simp
    from conf-s0 wt-e da-e
    obtain conf-s1: s1::≼(G, L) and
              v-ok: normal s1 ⟶ G,store s1⊢v::≼RefT eT and
      error-free-s1: error-free s1
      by (rule hyp [elim-format]) simp
    with T show ?case
      by simp
  next
    case (Lit s v L accC T A)
    then show ?case
      by (auto elim!: wt-elim-cases
              intro: conf-litval simp add: empty-dt-def)
  next
    case (UnOp s0 e v s1 unop L accC T A)
    note hyp = ‹PROP ?TypeSafe (Norm s0) s1 (In1l e) (In1 v)›
    note conf-s0 = ‹Norm s0::≼(G, L)›
    note wt = ‹⦇prg = G, cls = accC, lcl = L⦈⊢In1l (UnOp unop e)::T›
    then obtain eT
      where     wt-e: ⦇prg = G, cls = accC, lcl = L⦈⊢e::−eT and
            wt-unop: wt-unop unop eT and
                  T: T=Inl (PrimT (unop-type unop))
      by (auto elim!: wt-elim-cases)
    from UnOp.prems obtain A where
       da-e: ⦇prg=G,cls=accC,lcl=L⦈
                 ⊢ dom (locals (store ((Norm s0)::state))) »In1l e» A
      by (elim da-elim-cases) simp
    from conf-s0 wt-e da-e
    obtain      conf-s1: s1::≼(G, L)  and
                wt-v: normal s1 ⟶ G,store s1⊢v::≼eT and
         error-free-s1: error-free s1
      by (rule hyp [elim-format]) simp
    from wt-v T wt-unop
    have normal s1⟶G,L,snd s1⊢In1l (UnOp unop e)≻In1 (eval-unop unop v)::≼T
      by (cases unop) auto
    with conf-s1 error-free-s1
    show s1::≼(G, L) ∧
     (normal s1 ⟶ G,L,snd s1⊢In1l (UnOp unop e)≻In1 (eval-unop unop v)::≼T) ∧
     error-free (Norm s0) = error-free s1
      by simp
  next
    case (BinOp s0 e1 v1 s1 binop e2 v2 s2 L accC T A)
    note eval-e1 = ‹G⊢Norm s0 −e1−≻v1→ s1›
    note eval-e2 = ‹G⊢s1 −(if need-second-arg binop v1 then In1l e2
                          else In1r Skip)≻→ (In1 v2, s2)›
    note hyp-e1 = ‹PROP ?TypeSafe (Norm s0) s1 (In1l e1) (In1 v1)›
    note hyp-e2 = ‹PROP ?TypeSafe        s1  s2
                  (if need-second-arg binop v1 then In1l e2 else In1r Skip)
                  (In1 v2)›
    note conf-s0 = ‹Norm s0::≼(G, L)›
    note wt = ‹⦇prg = G, cls = accC, lcl = L⦈⊢In1l (BinOp binop e1 e2)::T›
```

then obtain *e1T e2T* **where**
wt-e1: $(\|prg = G, cls = accC, lcl = L\|)\vdash e1::-e1T$ **and**
wt-e2: $(\|prg = G, cls = accC, lcl = L\|)\vdash e2::-e2T$ **and**
wt-binop: *wt-binop G binop e1T e2T* **and**
T: *T=Inl (PrimT (binop-type binop))*
by (*elim wt-elim-cases*) *simp*
have *wt-Skip*: $(\|prg = G, cls = accC, lcl = L\|)\vdash Skip::\surd$
by *simp*
obtain *S* **where**
daSkip: $(\|prg=G,cls=accC,lcl=L\|)$
$\vdash$ *dom (locals (store s1))* »*In1r Skip*» *S*
by (*auto intro*: *da-Skip* [*simplified*] *assigned.select-convs*)
note *da* = ‹$(\|prg=G,cls=accC,lcl=L\|)\vdash$ *dom (locals (store ((Norm s0::state))))*
»$\langle$*BinOp binop e1 e2*$\rangle_e$» *A*›
then obtain *E1* **where**
da-e1: $(\|prg=G,cls=accC,lcl=L\|)$
$\vdash$ *dom (locals (store ((Norm s0)::state)))* »*In1l e1*» *E1*
by (*elim da-elim-cases*) *simp+*
from *conf-s0 wt-e1 da-e1*
obtain *conf-s1*: *s1*$::\preceq(G, L)$ **and**
wt-v1: *normal s1* $\longrightarrow$ *G,store s1*$\vdash$*v1*$::\preceq$*e1T* **and**
error-free-s1: *error-free s1*
by (*rule hyp-e1* [*elim-format*]) *simp*
from *wt-binop T*
have *conf-v*:
G,L,snd s2$\vdash$*In1l (BinOp binop e1 e2)*$\succ$*In1 (eval-binop binop v1 v2)*$::\preceq$*T*
by (*cases binop*) *auto*

— Note that we don't use the information that v1 really is compatible with the expected type e1T and v2 is compatible with e2T, because *eval-binop* will anyway produce an output of the right type. So evaluating the addition of an integer with a string is type safe. This is a little bit annoying since we may regard such a behaviour as not type safe. If we want to avoid this we can redefine *eval-binop* so that it only produces a output of proper type if it is assigned to values of the expected types, and arbitrary if the inputs have unexpected types. The proof can easily be adapted since we have the hypothesis that the values have a proper type. This also applies to unary operations.

from *eval-e1* **have**
s0-s1:*dom (locals (store ((Norm s0)::state)))* $\subseteq$ *dom (locals (store s1))*
by (*rule dom-locals-eval-mono-elim*)
show *s2*$::\preceq(G, L) \wedge$
(*normal s2* $\longrightarrow$
G,L,snd s2$\vdash$*In1l (BinOp binop e1 e2)*$\succ$*In1 (eval-binop binop v1 v2)*$::\preceq$*T*) $\wedge$
error-free (Norm s0) = error-free s2
proof (*cases normal s1*)
case *False*
with *eval-e2* **have** *s2=s1* **by** *auto*
with *conf-s1 error-free-s1 False* **show** *?thesis*
by *auto*
next
case *True*
note *normal-s1 = this*
show *?thesis*
proof (*cases need-second-arg binop v1*)
case *False*
with *normal-s1 eval-e2* **have** *s2=s1*
by (*cases s1*) (*simp*, *elim eval-elim-cases*,*simp*)
with *conf-s1 conf-v error-free-s1*
show *?thesis* **by** *simp*
next
case *True*
note *need-second-arg = this*

```
      with hyp-e2
      have hyp-e2': PROP ?TypeSafe s1 s2 (In1l e2) (In1 v2) by simp
      from da wt-e1 wt-e2 wt-binop conf-s0 normal-s1 eval-e1
        wt-v1 [rule-format,OF normal-s1] wf
      obtain E2 where
        (|prg=G,cls=accC,lcl=L|)⊢ dom (locals (store s1)) »In1l e2» E2
        by (rule da-e2-BinOp [elim-format])
           (auto simp add: need-second-arg )
      with conf-s1 wt-e2
      obtain s2::≼(G, L) and error-free s2
        by (rule hyp-e2' [elim-format]) (simp add: error-free-s1)
      with conf-v show ?thesis by simp
    qed
  qed
next
  case (Super s L accC T A)
  note conf-s = ‹Norm s::≼(G, L)›
  note wt = ‹(|prg = G, cls = accC, lcl = L|)⊢In1l Super::T›
  then obtain C c where
           C: L This = Some (Class C) and
      neq-Obj: C≠Object and
        cls-C: class G C = Some c and
           T: T=Inl (Class (super c))
    by (rule wt-elim-cases) auto
  from Super.prems
  obtain This ∈ dom (locals s)
    by (elim da-elim-cases) simp
  with conf-s C  have G,s⊢val-this s::≼Class C
    by (auto dest: conforms-localD [THEN wlconfD])
  with neq-Obj cls-C wf
  have G,s⊢val-this s::≼Class (super c)
    by (auto intro: conf-widen
               dest: subcls-direct[THEN widen.subcls])
  with T conf-s
  show Norm s::≼(G, L) ∧
         (normal (Norm s) ⟶
            G,L,store (Norm s)⊢In1l Super≻In1 (val-this s)::≼T) ∧
         (error-free (Norm s) = error-free (Norm s))
    by simp
next
  case (Acc s0 v w upd s1 L accC T A)
  note hyp = ‹PROP ?TypeSafe (Norm s0) s1 (In2 v) (In2 (w,upd))›
  note conf-s0 = ‹Norm s0::≼(G, L)›
  from Acc.prems obtain vT where
    wt-v: (|prg = G, cls = accC, lcl = L|)⊢v::=vT and
       T: T=Inl vT
    by (elim wt-elim-cases) simp
  from Acc.prems obtain V where
    da-v: (|prg=G,cls=accC,lcl=L|)
                ⊢ dom (locals (store ((Norm s0)::state))) »In2 v» V
    by (cases ∃ n. v=LVar n) (insert da.LVar, auto elim!: da-elim-cases)
  {
    fix n assume lvar: v=LVar n
    have locals (store s1) n ≠ None
    proof −
      from Acc.prems lvar have
        n ∈ dom (locals s0)
        by (cases ∃ n. v=LVar n) (auto elim!: da-elim-cases)
      also
```

```
        have dom (locals s0) ⊆ dom (locals (store s1))
        proof -
          from ‹G⊢Norm s0 -v=≻(w, upd)→ s1›
          show ?thesis
            by (rule dom-locals-eval-mono-elim) simp
        qed
        finally show ?thesis
          by blast
      qed
    } note lvar-in-locals = this
    from conf-s0 wt-v da-v
    obtain conf-s1: s1::≼(G, L)
      and conf-var: (normal s1 ⟶ G,L,store s1⊢In2 v≻In2 (w, upd)::≼Inl vT)
      and error-free-s1: error-free s1
      by (rule hyp [elim-format]) simp
    from lvar-in-locals conf-var T
    have (normal s1 ⟶ G,L,store s1⊢In1l (Acc v)≻In1 w::≼T)
      by (cases ∃ n. v=LVar n) auto
    with conf-s1 error-free-s1 show ?case
      by simp
  next
    case (Ass s0 var w upd s1 e v s2 L accC T A)
    note eval-var = ‹G⊢Norm s0 -var=≻(w, upd)→ s1›
    note eval-e = ‹G⊢s1 -e-≻v→ s2›
    note hyp-var = ‹PROP ?TypeSafe (Norm s0) s1 (In2 var) (In2 (w,upd))›
    note hyp-e = ‹PROP ?TypeSafe s1 s2 (In1l e) (In1 v)›
    note conf-s0 = ‹Norm s0::≼(G, L)›
    note wt = ‹(|prg = G, cls = accC, lcl = L|)⊢In1l (var:=e)::T›
    then obtain varT eT where
         wt-var: (|prg = G, cls = accC, lcl = L|)⊢var::=varT and
           wt-e: (|prg = G, cls = accC, lcl = L|)⊢e::-eT and
          widen: G⊢eT≼varT and
              T: T=Inl eT
      by (rule wt-elim-cases) auto
    show assign upd v s2::≼(G, L) ∧
           (normal (assign upd v s2) ⟶
            G,L,store (assign upd v s2)⊢In1l (var:=e)≻In1 v::≼T) ∧
      (error-free (Norm s0) = error-free (assign upd v s2))
    proof (cases ∃ vn. var=LVar vn)
      case False
      with Ass.prems
      obtain V E where
        da-var: (|prg=G,cls=accC,lcl=L|)
                   ⊢ dom (locals (store ((Norm s0)::state))) »In2 var» V and
        da-e:   (|prg=G,cls=accC,lcl=L|) ⊢ nrm V »In1l e» E
        by (elim da-elim-cases) simp+
      from conf-s0 wt-var da-var
      obtain conf-s1: s1::≼(G, L)
        and conf-var: normal s1
                        ⟶ G,L,store s1⊢In2 var≻In2 (w, upd)::≼Inl varT
        and error-free-s1: error-free s1
        by (rule hyp-var [elim-format]) simp
      show ?thesis
      proof (cases normal s1)
        case False
        with eval-e have s2=s1 by auto
        with False have assign upd v s2=s1
          by simp
        with conf-s1 error-free-s1 False show ?thesis
```

```
        by auto
    next
      case True
      note normal-s1=this
      obtain A' where (|prg=G,cls=accC,lcl=L|)
                   ⊢ dom (locals (store s1)) »In1l e» A'
      proof −
        from eval-var wt-var da-var wf normal-s1
        have nrm V ⊆  dom (locals (store s1))
          by (cases rule: da-good-approxE') iprover
        with da-e show thesis
          by (rule da-weakenE) (rule that)
      qed
      with conf-s1 wt-e
      obtain conf-s2: s2::≼(G, L) and
        conf-v: normal s2 ⟶ G,store s2⊢v::≼eT and
        error-free-s2: error-free s2
        by (rule hyp-e [elim-format]) (simp add: error-free-s1)
      show ?thesis
      proof (cases normal s2)
        case False
        with conf-s2 error-free-s2
        show ?thesis
          by auto
      next
        case True
        from True conf-v
        have conf-v-eT: G,store s2⊢v::≼eT
          by simp
        with widen wf
        have conf-v-varT: G,store s2⊢v::≼varT
          by (auto intro: conf-widen)
        from normal-s1 conf-var
        have G,L,store s1⊢In2 var≻In2 (w, upd)::≼Inl varT
          by simp
        then
        have conf-assign:  store s1≤|upd≼varT::≼(G, L)
          by (simp add: rconf-def)
        from conf-v-eT conf-v-varT conf-assign normal-s1 True wf eval-var
          eval-e T conf-s2 error-free-s2
        show ?thesis
          by (cases s1, cases s2)
             (auto dest!: Ass-lemma simp add: assign-conforms-def)
      qed
    qed
  next
    case True
    then obtain vn where vn: var=LVar vn
      by blast
    with Ass.prems
    obtain E where
      da-e: (|prg=G,cls=accC,lcl=L|)
               ⊢ dom (locals (store ((Norm s0)::state))) »In1l e» E
      by (elim da-elim-cases) simp+
    from da.LVar vn obtain V where
      da-var: (|prg=G,cls=accC,lcl=L|)
               ⊢ dom (locals (store ((Norm s0)::state))) »In2 var» V
      by auto
    obtain E' where
```

```
  da-e': (|prg=G,cls=accC,lcl=L|)
          ⊢ dom (locals (store s1)) »In1l e» E'
proof −
  have dom (locals (store ((Norm s0)::state)))
        ⊆ dom (locals (store (s1)))
    by (rule dom-locals-eval-mono-elim) (rule Ass.hyps)
  with da-e show thesis
    by (rule da-weakenE) (rule that)
qed
from conf-s0 wt-var da-var
obtain conf-s1: s1::≼(G, L)
  and  conf-var: normal s1
                 ⟶ G,L,store s1⊢In2 var≻In2 (w, upd)::≼Inl varT
  and  error-free-s1: error-free s1
  by (rule hyp-var [elim-format]) simp
show ?thesis
proof (cases normal s1)
  case False
  with eval-e have s2=s1 by auto
  with False have assign upd v s2=s1
    by simp
  with conf-s1 error-free-s1 False show ?thesis
    by auto
next
  case True
  note normal-s1 = this
  from conf-s1 wt-e da-e'
  obtain conf-s2: s2::≼(G, L) and
    conf-v: normal s2 ⟶ G,store s2⊢v::≼eT and
    error-free-s2: error-free s2
    by (rule hyp-e [elim-format]) (simp add: error-free-s1)
  show ?thesis
  proof (cases normal s2)
    case False
    with conf-s2 error-free-s2
    show ?thesis
      by auto
  next
    case True
    from True conf-v
    have conf-v-eT: G,store s2⊢v::≼eT
      by simp
    with widen wf
    have conf-v-varT: G,store s2⊢v::≼varT
      by (auto intro: conf-widen)
    from normal-s1 conf-var
    have G,L,store s1⊢In2 var≻In2 (w, upd)::≼Inl varT
      by simp
    then
    have conf-assign:  store s1≤|upd≼varT::≼(G, L)
      by (simp add: rconf-def)
    from conf-v-eT conf-v-varT conf-assign normal-s1 True wf eval-var
      eval-e T conf-s2 error-free-s2
    show ?thesis
      by (cases s1, cases s2)
         (auto dest!: Ass-lemma simp add: assign-conforms-def)
  qed
qed
qed
```

```
next
  case (Cond s0 e0 b s1 e1 e2 v s2 L accC T A)
  note eval-e0 = ‹G⊢Norm s0 −e0−≻b→ s1›
  note eval-e1-e2 = ‹G⊢s1 −(if the-Bool b then e1 else e2)−≻v→ s2›
  note hyp-e0 = ‹PROP ?TypeSafe (Norm s0) s1 (In1l e0) (In1 b)›
  note hyp-if = ‹PROP ?TypeSafe s1 s2
                   (In1l (if the-Bool b then e1 else e2)) (In1 v)›
  note conf-s0 = ‹Norm s0::≼(G, L)›
  note wt = ‹⦇prg = G, cls = accC, lcl = L⦈⊢In1l (e0 ? e1 : e2)::T›
  then obtain T1 T2 statT where
    wt-e0: ⦇prg = G, cls = accC, lcl = L⦈⊢e0::−PrimT Boolean and
    wt-e1: ⦇prg = G, cls = accC, lcl = L⦈⊢e1::−T1 and
    wt-e2: ⦇prg = G, cls = accC, lcl = L⦈⊢e2::−T2 and
    statT: G⊢T1≼T2 ∧ statT = T2  ∨  G⊢T2≼T1 ∧ statT =  T1 and
    T    : T=Inl statT
    by (rule wt-elim-cases) auto
  with Cond.prems obtain E0 E1 E2 where
       da-e0: ⦇prg=G,cls=accC,lcl=L⦈
               ⊢ dom (locals (store ((Norm s0)::state)))
                   »In1l e0» E0 and
       da-e1: ⦇prg=G,cls=accC,lcl=L⦈
               ⊢ (dom (locals (store ((Norm s0)::state)))
                     ∪ assigns-if True e0) »In1l e1» E1 and
       da-e2: ⦇prg=G,cls=accC,lcl=L⦈
               ⊢ (dom (locals (store ((Norm s0)::state)))
                    ∪ assigns-if False e0) »In1l e2» E2
     by (elim da-elim-cases) simp+
  from conf-s0 wt-e0 da-e0
  obtain conf-s1: s1::≼(G, L) and error-free-s1: error-free s1
    by (rule hyp-e0 [elim-format]) simp
  show s2::≼(G, L) ∧
        (normal s2 ⟶ G,L,store s2⊢In1l (e0 ? e1 : e2)≻In1 v::≼T) ∧
        (error-free (Norm s0) = error-free s2)
  proof (cases normal s1)
    case False
    with eval-e1-e2 have s2=s1 by auto
    with conf-s1 error-free-s1 False show ?thesis
      by auto
  next
    case True
    have s0-s1: dom (locals (store ((Norm s0)::state)))
                 ∪ assigns-if (the-Bool b) e0 ⊆ dom (locals (store s1))
    proof −
      from eval-e0 have
        dom (locals (store ((Norm s0)::state))) ⊆ dom (locals (store s1))
        by (rule dom-locals-eval-mono-elim)
      moreover
      from eval-e0 True wt-e0
      have assigns-if (the-Bool b) e0 ⊆ dom (locals (store s1))
        by (rule assigns-if-good-approx')
      ultimately show ?thesis by (rule Un-least)
    qed
    show ?thesis
    proof (cases the-Bool b)
      case True
      with hyp-if have hyp-e1: PROP ?TypeSafe s1 s2 (In1l e1) (In1 v)
        by simp
      from da-e1 s0-s1 True obtain E1' where
        ⦇prg=G,cls=accC,lcl=L⦈⊢ (dom (locals (store s1)))»In1l e1» E1'
```

by − (*rule da-weakenE*, *auto iff del*: *Un-subset-iff sup.bounded-iff*)
with *conf-s1 wt-e1*
obtain
s2::⪯(*G*, *L*)
(*normal s2* ⟶ *G*,*L*,*store s2*⊢*In1l e1*≻*In1 v*::⪯*Inl T1*)
error-free s2
by (*rule hyp-e1* [*elim-format*]) (*simp add*: *error-free-s1*)
moreover
from *statT*
have *G*⊢*T1*⪯*statT*
by *auto*
ultimately show *?thesis*
using *T wf* **by** *auto*
next
case *False*
with *hyp-if* **have** *hyp-e2*: *PROP ?TypeSafe s1 s2* (*In1l e2*) (*In1 v*)
by *simp*
from *da-e2 s0-s1 False* **obtain** *E2′* **where**
⦇*prg*=*G*,*cls*=*accC*,*lcl*=*L*⦈⊢ (*dom* (*locals* (*store s1*)))»*In1l e2*» *E2′*
by − (*rule da-weakenE*, *auto iff del*: *Un-subset-iff sup.bounded-iff*)
with *conf-s1 wt-e2*
obtain
s2::⪯(*G*, *L*)
(*normal s2* ⟶ *G*,*L*,*store s2*⊢*In1l e2*≻*In1 v*::⪯*Inl T2*)
error-free s2
by (*rule hyp-e2* [*elim-format*]) (*simp add*: *error-free-s1*)
moreover
from *statT*
have *G*⊢*T2*⪯*statT*
by *auto*
ultimately show *?thesis*
using *T wf* **by** *auto*
qed
qed
next
case (*Call s0 e a s1 args vs s2 invDeclC mode statT mn pTs′ s3 s3′ accC′*
v s4 L accC T A)
note *eval-e* = ‹*G*⊢*Norm s0* −*e*−≻*a*→ *s1*›
note *eval-args* = ‹*G*⊢*s1* −*args*≐≻*vs*→ *s2*›
note *invDeclC* = ‹*invDeclC*
= *invocation-declclass G mode* (*store s2*) *a statT*
⦇*name* = *mn*, *parTs* = *pTs′*⦈›
note *init-lvars* =
‹*s3* = *init-lvars G invDeclC* ⦇*name* = *mn*, *parTs* = *pTs′*⦈ *mode a vs s2*›
note *check* = ‹*s3′* =
check-method-access G accC′ statT mode ⦇*name* = *mn*, *parTs* = *pTs′*⦈ *a s3*›
note *eval-methd* =
‹*G*⊢*s3′* −*Methd invDeclC* ⦇*name* = *mn*, *parTs* = *pTs′*⦈−≻*v*→ *s4*›
note *hyp-e* = ‹*PROP ?TypeSafe* (*Norm s0*) *s1* (*In1l e*) (*In1 a*)›
note *hyp-args* = ‹*PROP ?TypeSafe s1 s2* (*In3 args*) (*In3 vs*)›
note *hyp-methd* = ‹*PROP ?TypeSafe s3′ s4*
(*In1l* (*Methd invDeclC* ⦇*name* = *mn*, *parTs* = *pTs′*⦈)) (*In1 v*)›
note *conf-s0* = ‹*Norm s0*::⪯(*G*, *L*)›
note *wt* = ‹⦇*prg*=*G*, *cls*=*accC*, *lcl*=*L*⦈
⊢*In1l* ({*accC′*,*statT*,*mode*}*e*·*mn*({*pTs′*}*args*))::*T*›
from *wt* **obtain** *pTs statDeclT statM* **where**
wt-e: ⦇*prg*=*G*, *cls*=*accC*, *lcl*=*L*⦈⊢*e*::−*RefT statT* **and**
wt-args: ⦇*prg*=*G*, *cls*=*accC*, *lcl*=*L*⦈⊢*args*::≐*pTs* **and**
statM: *max-spec G accC statT* ⦇*name*=*mn*,*parTs*=*pTs*⦈

```
                   = {((statDeclT,statM),pTs')} and
          mode: mode = invmode statM e and
             T: T =Inl (resTy statM) and
    eq-accC-accC': accC=accC'
  by (rule wt-elim-cases) fastforce+
from Call.prems obtain E where
  da-e: (|prg=G,cls=accC,lcl=L|)
         ⊢ (dom (locals (store ((Norm s0)::state))))»In1l e» E and
  da-args: (|prg=G,cls=accC,lcl=L|)⊢ nrm E »In3 args» A
  by (elim da-elim-cases) simp
from conf-s0 wt-e da-e
obtain conf-s1: s1::≼(G, L) and
       conf-a: normal s1 ⟹ G, store s1⊢a::≼RefT statT and
       error-free-s1: error-free s1
  by (rule hyp-e [elim-format]) simp
{
  assume abnormal-s2: ¬ normal s2
  have set-lvars (locals (store s2)) s4 = s2
  proof −
    from abnormal-s2 init-lvars
    obtain keep-abrupt: abrupt s3 = abrupt s2 and
           store s3 = store (init-lvars G invDeclC (|name = mn, parTs = pTs'|)
                                         mode a vs s2)
      by (auto simp add: init-lvars-def2)
    moreover
    from keep-abrupt abnormal-s2 check
    have eq-s3'-s3: s3'=s3
      by (auto simp add: check-method-access-def Let-def)
    moreover
    from eq-s3'-s3 abnormal-s2 keep-abrupt eval-methd
    have s4=s3'
      by auto
    ultimately show
      set-lvars (locals (store s2)) s4 = s2
      by (cases s2,cases s3) (simp add: init-lvars-def2)
  qed
} note propagate-abnormal-s2 = this
show (set-lvars (locals (store s2))) s4::≼(G, L) ∧
       (normal ((set-lvars (locals (store s2))) s4) ⟶
         G,L,store ((set-lvars (locals (store s2))) s4)
           ⊢In1l ({accC',statT,mode}e·mn( {pTs'}args))≻In1 v::≼T) ∧
       (error-free (Norm s0) =
            error-free ((set-lvars (locals (store s2))) s4))
proof (cases normal s1)
  case False
  with eval-args have s2=s1 by auto
  with False propagate-abnormal-s2 conf-s1 error-free-s1
  show ?thesis
    by auto
next
  case True
  note normal-s1 = this
  obtain A' where
    (|prg=G,cls=accC,lcl=L|)⊢ dom (locals (store s1)) »In3 args» A'
  proof −
    from eval-e wt-e da-e wf normal-s1
    have nrm E ⊆ dom (locals (store s1))
      by (cases rule: da-good-approxE') iprover
    with da-args show thesis
```

```
      by (rule da-weakenE) (rule that)
  qed
  with conf-s1 wt-args
  obtain    conf-s2: s2::≼(G, L) and
        conf-args: normal s2
                 ⟹ list-all2 (conf G (store s2)) vs pTs and
      error-free-s2: error-free s2
    by (rule hyp-args [elim-format]) (simp add: error-free-s1)
  from error-free-s2 init-lvars
  have error-free-s3: error-free s3
    by (auto simp add: init-lvars-def2)
  from statM
  obtain
    statM': (statDeclT,statM)∈mheads G accC statT (|name=mn,parTs=pTs'|) and
    pTs-widen: G⊢pTs[≼]pTs'
    by (blast dest: max-spec2mheads)
  from check
  have eq-store-s3'-s3: store s3'=store s3
    by (cases s3) (simp add: check-method-access-def Let-def)
  obtain invC
    where invC: invC = invocation-class mode (store s2) a statT
    by simp
  with init-lvars
  have invC': invC = (invocation-class mode (store s3) a statT)
    by (cases s2,cases mode) (auto simp add: init-lvars-def2 )
  show ?thesis
  proof (cases normal s2)
    case False
    with propagate-abnormal-s2 conf-s2 error-free-s2
    show ?thesis
      by auto
  next
    case True
    note normal-s2 = True
    with normal-s1 conf-a eval-args
    have conf-a-s2: G, store s2⊢a::≼RefT statT
      by (auto dest: eval-gext intro: conf-gext)
    show ?thesis
    proof (cases a=Null ⟶ is-static statM)
      case False
      then obtain not-static: ¬ is-static statM and Null: a=Null
        by blast
      with normal-s2 init-lvars mode
      obtain np: abrupt s3 = Some (Xcpt (Std NullPointer)) and
                 store s3 = store (init-lvars G invDeclC
                                 (|name = mn, parTs = pTs'|) mode a vs s2)
        by (auto simp add: init-lvars-def2)
      moreover
      from np check
      have eq-s3'-s3: s3'=s3
        by (auto simp add: check-method-access-def Let-def)
      moreover
      from eq-s3'-s3 np eval-methd
      have s4=s3'
        by auto
      ultimately have
        set-lvars (locals (store s2)) s4
        = (Some (Xcpt (Std NullPointer)),store s2)
        by (cases s2,cases s3) (simp add: init-lvars-def2)
```

```
      with conf-s2 error-free-s2
      show ?thesis
        by (cases s2) (auto dest: conforms-NormI)
    next
      case True
      with mode have notNull: mode = IntVir ⟶ a ≠ Null
        by (auto dest!: Null-staticD)
      with conf-s2 conf-a-s2 wf invC
      have dynT-prop: G⊢mode→invC≼statT
        by (cases s2) (auto intro: DynT-propI)
      with wt-e statM' invC mode wf
      obtain dynM where
        dynM: dynlookup G statT invC  (|name=mn,parTs=pTs'|) = Some dynM and
        acc-dynM: G ⊢Methd  (|name=mn,parTs=pTs'|) dynM
                       in invC dyn-accessible-from accC
        by (force dest!: call-access-ok)
      with invC' check eq-accC-accC'
      have eq-s3'-s3: s3'=s3
        by (auto simp add: check-method-access-def Let-def)
      from dynT-prop wf wt-e statM' mode invC invDeclC dynM
      obtain
        wf-dynM: wf-mdecl G invDeclC ((|name=mn,parTs=pTs'|),mthd dynM) and
          dynM': methd G invDeclC (|name=mn,parTs=pTs'|) = Some dynM and
        iscls-invDeclC: is-class G invDeclC and
             invDeclC': invDeclC = declclass dynM and
          invC-widen: G⊢invC≼C invDeclC and
         resTy-widen: G⊢resTy dynM≼resTy statM and
        is-static-eq: is-static dynM = is-static statM and
        involved-classes-prop:
         (if invmode statM e = IntVir
           then ∀ statC. statT = ClassT statC ⟶ G⊢invC≼C statC
           else ((∃ statC. statT = ClassT statC ∧ G⊢statC≼C invDeclC) ∨
                 (∀ statC. statT ≠ ClassT statC ∧ invDeclC = Object)) ∧
                  statDeclT = ClassT invDeclC)
        by (cases rule: DynT-mheadsE) simp
      obtain L' where
       L':L'=(λ k.
             (case k of
                EName e
                ⇒ (case e of
                       VNam v
                       ⇒((table-of (lcls (mbody (mthd dynM))))
                          (pars (mthd dynM)[↦]pTs')) v
                     | Res ⇒ Some (resTy dynM))
              | This ⇒ if is-static statM
                          then None else Some (Class invDeclC)))
        by simp
      from wf-dynM [THEN wf-mdeclD1, THEN conjunct1] normal-s2 conf-s2 wt-e
        wf eval-args conf-a mode notNull wf-dynM involved-classes-prop
      have conf-s3: s3::≼(G,L')
        apply −

        apply (drule conforms-init-lvars [of G invDeclC
                (|name=mn,parTs=pTs'|) dynM store s2 vs pTs abrupt s2
                L statT invC a (statDeclT,statM) e])
        apply (rule wf)
        apply (rule conf-args,assumption)
        apply (simp add: pTs-widen)
        apply (cases s2,simp)
```

```
  apply (rule dynM')
  apply (force dest: ty-expr-is-type)
  apply (rule invC-widen)
  apply (force intro: conf-gext dest: eval-gext)
  apply simp
  apply simp
  apply (simp add: invC)
  apply (simp add: invDeclC)
  apply (simp add: normal-s2)
  apply (cases s2, simp add: L' init-lvars
                  cong add: lname.case-cong ename.case-cong)
  done
with eq-s3'-s3
have conf-s3': s3'::≼(G,L') by simp
moreover
from  is-static-eq wf-dynM L'
obtain mthdT where
  (|prg=G,cls=invDeclC,lcl=L'|)
    ⊢Body invDeclC (stmt (mbody (mthd dynM)))::−mthdT and
  mthdT-widen: G⊢mthdT≼resTy dynM
  by − (drule wf-mdecl-bodyD,
       auto simp add: callee-lcl-def
            cong add: lname.case-cong ename.case-cong)
with dynM' iscls-invDeclC invDeclC'
have
  (|prg=G,cls=invDeclC,lcl=L'|)
    ⊢(Methd invDeclC (|name = mn, parTs = pTs'|))::−mthdT
  by (auto intro: wt.Methd)
moreover
obtain M where
  (|prg=G,cls=invDeclC,lcl=L'|)
    ⊢ dom (locals (store s3'))
       »In1l (Methd invDeclC (|name = mn, parTs = pTs'|))» M
proof −
  from wf-dynM
  obtain M' where
    da-body:
    (|prg=G, cls=invDeclC
     ,lcl=callee-lcl invDeclC (|name = mn, parTs = pTs'|) (mthd dynM)
     |) ⊢ parameters (mthd dynM) »⟨stmt (mbody (mthd dynM))⟩» M' and
    res: Result ∈ nrm M'
    by (rule wf-mdeclE) iprover
  from da-body is-static-eq L' have
    (|prg=G, cls=invDeclC,lcl=L'|)
      ⊢ parameters (mthd dynM) »⟨stmt (mbody (mthd dynM))⟩» M'
    by (simp add: callee-lcl-def
        cong add: lname.case-cong ename.case-cong)
  moreover have parameters (mthd dynM) ⊆  dom (locals (store s3'))
  proof −
    from is-static-eq
    have (invmode (mthd dynM) e) = (invmode statM e)
      by (simp add: invmode-def)
    moreover
    have length (pars (mthd dynM)) = length vs
    proof −
      from normal-s2 conf-args
      have length vs = length pTs
        by (simp add: list-all2-iff)
      also from pTs-widen
```

```
      have ... = length pTs'
        by (simp add: widens-def list-all2-iff)
      also from wf-dynM
      have ... = length (pars (mthd dynM))
        by (simp add: wf-mdecl-def wf-mhead-def)
      finally show ?thesis ..
    qed
    moreover note init-lvars dynM' is-static-eq normal-s2 mode
    ultimately
    have parameters (mthd dynM) = dom (locals (store s3))
      using dom-locals-init-lvars
        [of mthd dynM G invDeclC (|name=mn,parTs=pTs'|) vs e a s2]
      by simp
    also from check
    have dom (locals (store s3)) ⊆ dom (locals (store s3'))
      by (simp add: eq-s3'-s3)
    finally show ?thesis .
  qed
  ultimately obtain M2 where
    da:
    (|prg=G, cls=invDeclC,lcl=L'|)
      ⊢ dom (locals (store s3')) »⟨stmt (mbody (mthd dynM))⟩» M2 and
    M2: nrm M' ⊆ nrm M2
    by (rule da-weakenE)
  from res M2 have Result ∈ nrm M2
    by blast
  moreover from wf-dynM
  have jumpNestingOkS {Ret} (stmt (mbody (mthd dynM)))
    by (rule wf-mdeclE)
  ultimately
  obtain M3 where
    (|prg=G, cls=invDeclC,lcl=L'|) ⊢ dom (locals (store s3'))
          »⟨Body (declclass dynM) (stmt (mbody (mthd dynM)))⟩» M3
    using da
    by (iprover intro: da.Body assigned.select-convs)
  from - this [simplified]
  show ?thesis
    by (rule da.Methd [simplified,elim-format]) (auto intro: dynM' that)
qed
ultimately obtain
  conf-s4: s4::≼(G, L') and
  conf-Res: normal s4 ⟶ G,store s4⊢v::≼mthdT and
  error-free-s4: error-free s4
  by (rule hyp-methd [elim-format])
    (simp add: error-free-s3 eq-s3'-s3)
from init-lvars eval-methd eq-s3'-s3
have store s2≤|store s4
  by (cases s2) (auto dest!: eval-gext simp add: init-lvars-def2 )
moreover
have abrupt s4 ≠ Some (Jump Ret)
proof -
  from normal-s2 init-lvars
  have abrupt s3 ≠ Some (Jump Ret)
    by (cases s2) (simp add: init-lvars-def2 abrupt-if-def)
  with check
  have abrupt s3' ≠ Some (Jump Ret)
    by (cases s3) (auto simp add: check-method-access-def Let-def)
  with eval-methd
  show ?thesis
```

```
        by (rule Methd-no-jump)
      qed
      ultimately
      have (set-lvars (locals (store s2))) s4::≼(G, L)
        using conf-s2 conf-s4
        by (cases s2,cases s4) (auto intro: conforms-return)
      moreover
      from conf-Res mthdT-widen resTy-widen wf
      have normal s4
              ⟶ G,store s4⊢v::≼(resTy statM)
        by (auto dest: widen-trans)
      then
      have normal ((set-lvars (locals (store s2))) s4)
           ⟶ G,store((set-lvars (locals (store s2))) s4) ⊢v::≼(resTy statM)
        by (cases s4) auto
      moreover note error-free-s4 T
      ultimately
      show ?thesis
        by simp
    qed
  qed
 qed
next
 case (Methd s0 D sig v s1 L accC T A)
 note ‹G⊢Norm s0 −body G D sig−≻v→ s1›
 note hyp = ‹PROP ?TypeSafe (Norm s0) s1 (In1l (body G D sig)) (In1 v)›
 note conf-s0 = ‹Norm s0::≼(G, L)›
 note wt = ‹⦇prg = G, cls = accC, lcl = L⦈⊢In1l (Methd D sig)::T›
 then obtain m bodyT where
   D: is-class G D and
   m: methd G D sig = Some m and
   wt-body: ⦇prg = G, cls = accC, lcl = L⦈
             ⊢Body (declclass m) (stmt (mbody (mthd m)))::−bodyT and
   T: T=Inl bodyT
   by (rule wt-elim-cases) auto
 moreover
 from Methd.prems m have
    da-body: ⦇prg=G,cls=accC,lcl=L⦈
              ⊢ (dom (locals (store ((Norm s0)::state))))
                 »In1l (Body (declclass m) (stmt (mbody (mthd m))))» A
   by − (erule da-elim-cases,simp)
 ultimately
 show s1::≼(G, L) ∧
       (normal s1 ⟶ G,L,snd s1⊢In1l (Methd D sig)≻In1 v::≼T) ∧
       (error-free (Norm s0) = error-free s1)
   using hyp [of - - (Inl bodyT)] conf-s0
   by (auto simp add: Let-def body-def)
next
 case (Body s0 D s1 c s2 s3 L accC T A)
 note eval-init = ‹G⊢Norm s0 −Init D→ s1›
 note eval-c = ‹G⊢s1 −c→ s2›
 note hyp-init = ‹PROP ?TypeSafe (Norm s0) s1 (In1r (Init D)) ♢›
 note hyp-c = ‹PROP ?TypeSafe s1 s2 (In1r c) ♢›
 note conf-s0 = ‹Norm s0::≼(G, L)›
 note wt = ‹⦇prg = G, cls = accC, lcl = L⦈⊢In1l (Body D c)::T›
 then obtain bodyT where
     iscls-D: is-class G D and
       wt-c: ⦇prg = G, cls = accC, lcl = L⦈⊢c::√ and
     resultT: L Result = Some bodyT and
```

```
    isty-bodyT: is-type G bodyT and
           T: T=Inl bodyT
    by (rule wt-elim-cases) auto
  from Body.prems obtain C where
    da-c: (|prg=G,cls=accC,lcl=L|)
                ⊢ (dom (locals (store ((Norm s0)::state))))»In1r c» C and
    jmpOk: jumpNestingOkS {Ret} c and
    res: Result ∈ nrm C
    by (elim da-elim-cases) simp
  note conf-s0
  moreover from iscls-D
  have (|prg=G, cls=accC, lcl=L|)⊢Init D::√ by auto
  moreover obtain I where
    (|prg=G,cls=accC,lcl=L|)
       ⊢ dom (locals (store ((Norm s0)::state))) »In1r (Init D)» I
    by (auto intro: da-Init [simplified] assigned.select-convs)
  ultimately obtain
    conf-s1: s1::≼(G, L) and error-free-s1:  error-free s1
     by (rule hyp-init [elim-format]) simp
  obtain C' where da-C': (|prg=G,cls=accC,lcl=L|)
                          ⊢ (dom (locals (store s1)))»In1r c» C'
           and nrm-C': nrm C ⊆ nrm C'
  proof -
    from eval-init
    have (dom (locals (store ((Norm s0)::state))))
                 ⊆ (dom (locals (store s1)))
      by (rule dom-locals-eval-mono-elim)
    with da-c show thesis by (rule da-weakenE) (rule that)
  qed
  from conf-s1 wt-c da-C'
  obtain conf-s2: s2::≼(G, L) and error-free-s2: error-free s2
    by (rule hyp-c [elim-format]) (simp add: error-free-s1)
  from conf-s2
  have abupd (absorb Ret) s2::≼(G, L)
    by (cases s2) (auto intro: conforms-absorb)
  moreover
  from error-free-s2
  have error-free (abupd (absorb Ret) s2)
    by simp
  moreover have abrupt (abupd (absorb Ret) s3) ≠ Some (Jump Ret)
    by (cases s3) (simp add: absorb-def)
  moreover have s3=s2
  proof -
    from iscls-D
    have wt-init: (|prg=G, cls=accC, lcl=L|)⊢(Init D)::√
      by auto
    from eval-init wf
    have s1-no-jmp: ⋀ j. abrupt s1 ≠ Some (Jump j)
      by - (rule eval-statement-no-jump [OF - - - wt-init],auto)
    from eval-c - wt-c wf
    have ⋀ j. abrupt s2 = Some (Jump j) ⟹ j=Ret
      by (rule jumpNestingOk-evalE) (auto intro: jmpOk simp add: s1-no-jmp)
    moreover
    note ‹s3 =
           (if ∃ l. abrupt s2 = Some (Jump (Break l)) ∨
                  abrupt s2 = Some (Jump (Cont l))
             then abupd (λx. Some (Error CrossMethodJump)) s2 else s2)›
    ultimately show ?thesis
      by force
```

```
  qed
  moreover
  {
    assume normal-upd-s2: normal (abupd (absorb Ret) s2)
    have Result ∈ dom (locals (store s2))
    proof −
      from normal-upd-s2
      have normal s2 ∨ abrupt s2 = Some (Jump Ret)
        by (cases s2) (simp add: absorb-def)
      thus ?thesis
      proof
        assume normal s2
        with eval-c wt-c da-C' wf res nrm-C'
        show ?thesis
          by (cases rule: da-good-approxE') blast
      next
        assume abrupt s2 = Some (Jump Ret)
        with conf-s2 show ?thesis
          by (cases s2) (auto dest: conforms-RetD simp add: dom-def)
      qed
    qed
  }
  moreover note T resultT
  ultimately
  show abupd (absorb Ret) s3::≼(G, L) ∧
         (normal (abupd (absorb Ret) s3) ⟶
           G,L,store (abupd (absorb Ret) s3)
           ⊢In1l (Body D c)≻In1 (the (locals (store s2) Result))::≼T) ∧
        (error-free (Norm s0) = error-free (abupd (absorb Ret) s3))
    by (cases s2) (auto intro: conforms-locals)
next
  case (LVar s vn L accC T)
  note conf-s = ‹Norm s::≼(G, L)› and
    wt = ‹(|prg = G, cls = accC, lcl = L|)⊢In2 (LVar vn)::T›
  then obtain vnT where
    vnT: L vn = Some vnT and
      T: T=Inl vnT
    by (auto elim!: wt-elim-cases)
  from conf-s vnT
  have conf-fst: locals s vn ≠ None ⟶ G,s⊢fst (lvar vn s)::≼vnT
   by (auto elim: conforms-localD [THEN wlconfD]
            simp add: lvar-def)
  moreover
  from conf-s conf-fst vnT
  have s≤|snd (lvar vn s)≼vnT::≼(G, L)
    by (auto elim: conforms-lupd simp add: assign-conforms-def lvar-def)
  moreover note conf-s T
  ultimately
  show Norm s::≼(G, L) ∧
              (normal (Norm s) ⟶
                 G,L,store (Norm s)⊢In2 (LVar vn)≻In2 (lvar vn s)::≼T) ∧
              (error-free (Norm s) = error-free (Norm s))
    by (simp add: lvar-def)
next
  case (FVar s0 statDeclC s1 e a s2 v s2' stat fn s3 accC L accC' T A)
  note eval-init = ‹G⊢Norm s0 −Init statDeclC→ s1›
  note eval-e = ‹G⊢s1 −e−≻a→ s2›
  note fvar = ‹(v, s2') = fvar statDeclC stat fn a s2›
  note check = ‹s3 = check-field-access G accC statDeclC fn stat a s2'›
```

note *hyp-init* = ‹*PROP ?TypeSafe* (*Norm s0*) *s1* (*In1r* (*Init statDeclC*)) ◇›
note *hyp-e* = ‹*PROP ?TypeSafe s1 s2* (*In1l e*) (*In1 a*)›
note *conf-s0* = ‹*Norm s0*::≼(*G*, *L*)›
note *wt* = ‹(|*prg*=*G*, *cls*=*accC*′, *lcl*=*L*|)⊢*In2* ({*accC*,*statDeclC*,*stat*}*e*..*fn*)::*T*›
then obtain *statC f* **where**
wt-e: (|*prg*=*G*, *cls*=*accC*, *lcl*=*L*|)⊢*e*::−*Class statC* **and**
accfield: *accfield G accC statC fn* = *Some* (*statDeclC*,*f*) **and**
eq-accC-accC′: *accC*=*accC*′ **and**
stat: *stat*=*is-static f* **and**
T: *T*=(*Inl* (*type f*))
by (*rule wt-elim-cases*) (*auto simp add*: *member-is-static-simp*)
from *FVar.prems eq-accC-accC*′
have *da-e*: (|*prg*=*G*, *cls*=*accC*, *lcl*=*L*|)
⊢ (*dom* (*locals* (*store* ((*Norm s0*)::*state*))))»*In1l e*» *A*
by (*elim da-elim-cases*) *simp*
note *conf-s0*
moreover
from *wf wt-e*
have *iscls-statC*: *is-class G statC*
by (*auto dest*: *ty-expr-is-type type-is-class*)
with *wf accfield*
have *iscls-statDeclC*: *is-class G statDeclC*
by (*auto dest*!: *accfield-fields dest*: *fields-declC*)
hence (|*prg*=*G*, *cls*=*accC*, *lcl*=*L*|)⊢(*Init statDeclC*)::√
by *simp*
moreover obtain *I* **where**
(|*prg*=*G*,*cls*=*accC*,*lcl*=*L*|)
⊢ *dom* (*locals* (*store* ((*Norm s0*)::*state*))) »*In1r* (*Init statDeclC*)» *I*
by (*auto intro*: *da-Init* [*simplified*] *assigned.select-convs*)
ultimately
obtain *conf-s1*: *s1*::≼(*G*, *L*) **and** *error-free-s1*: *error-free s1*
by (*rule hyp-init* [*elim-format*]) *simp*
obtain *A*′ **where**
(|*prg*=*G*, *cls*=*accC*, *lcl*=*L*|) ⊢ (*dom* (*locals* (*store s1*)))»*In1l e*» *A*′
proof −
from *eval-init*
have (*dom* (*locals* (*store* ((*Norm s0*)::*state*))))
⊆ (*dom* (*locals* (*store s1*)))
by (*rule dom-locals-eval-mono-elim*)
with *da-e* **show** *thesis*
by (*rule da-weakenE*) (*rule that*)
qed
with *conf-s1 wt-e*
obtain *conf-s2*: *s2*::≼(*G*, *L*) **and**
conf-a: *normal s2* ⟶ *G*,*store s2*⊢*a*::≼*Class statC* **and**
error-free-s2: *error-free s2*
by (*rule hyp-e* [*elim-format*]) (*simp add*: *error-free-s1*)
from *fvar*
have *store-s2*′: *store s2*′=*store s2*
by (*cases s2*) (*simp add*: *fvar-def2*)
with *fvar conf-s2*
have *conf-s2*′: *s2*′::≼(*G*, *L*)
by (*cases s2*,*cases stat*) (*auto simp add*: *fvar-def2*)
from *eval-init*
have *initd-statDeclC-s1*: *initd statDeclC s1*
by (*rule init-yields-initd*)
from *accfield wt-e eval-init eval-e conf-s2 conf-a fvar stat check wf*
have *eq-s3-s2*′: *s3*=*s2*′
by (*auto dest*!: *error-free-field-access*)

```
  have conf-v: normal s2' ⟹
        G,store s2'⊢fst v::≼type f ∧ store s2'≤|snd v≼type f::≼(G, L)
  proof −
    assume normal: normal s2'
    obtain vv vf x2 store2 store2'
      where  v: v=(vv,vf) and
            s2: s2=(x2,store2) and
       store2': store s2' = store2'
      by (cases v,cases s2,cases s2') blast
    from iscls-statDeclC obtain c
      where c: class G statDeclC = Some c
      by auto
    have G,store2'⊢vv::≼type f ∧ store2'≤|vf≼type f::≼(G, L)
    proof (rule FVar-lemma [of vv vf store2' statDeclC f fn a x2 store2
                           statC G c L store s1])
      from v normal s2 fvar stat store2'
      show ((vv, vf), Norm store2') =
             fvar statDeclC (static f) fn a (x2, store2)
        by (auto simp add: member-is-static-simp)
      from accfield iscls-statC wf
      show G⊢statC≼C statDeclC
        by (auto dest!: accfield-fields dest: fields-declC)
      from accfield
      show fld: table-of (DeclConcepts.fields G statC) (fn, statDeclC) = Some f
        by (auto dest!: accfield-fields)
      from wf show wf-prog G .
      from conf-a s2 show x2 = None ⟶ G,store2⊢a::≼Class statC
        by auto
      from fld wf iscls-statC
      show statDeclC ≠ Object
        by (cases statDeclC=Object) (drule fields-declC,simp+)+
      from c show class G statDeclC = Some c .
      from conf-s2 s2 show (x2, store2)::≼(G, L) by simp
      from eval-e s2 show snd s1≤|store2 by (auto dest: eval-gext)
      from initd-statDeclC-s1 show inited statDeclC (globs (snd s1))
        by simp
    qed
    with v s2 store2'
    show ?thesis
      by simp
  qed
  from fvar error-free-s2
  have error-free s2'
    by (cases s2)
       (auto simp add: fvar-def2 intro!: error-free-FVar-lemma)
  with conf-v T conf-s2' eq-s3-s2'
  show s3::≼(G, L) ∧
       (normal s3
        ⟶ G,L,store s3⊢In2 ({accC,statDeclC,stat}e..fn)≻In2 v::≼T) ∧
       (error-free (Norm s0) = error-free s3)
    by auto
next
  case (AVar s0 e1 a s1 e2 i s2 v s2' L accC T A)
  note eval-e1 = ‹G⊢Norm s0 −e1−≻a→ s1›
  note eval-e2 = ‹G⊢s1 −e2−≻i→ s2›
  note hyp-e1 = ‹PROP ?TypeSafe (Norm s0) s1 (In1l e1) (In1 a)›
  note hyp-e2 = ‹PROP ?TypeSafe s1 s2 (In1l e2) (In1 i)›
  note avar = ‹(v, s2') = avar G i a s2›
  note conf-s0 = ‹Norm s0::≼(G, L)›
```

note *wt* = ‹⦇*prg* = *G*, *cls* = *accC*, *lcl* = *L*⦈⊢*In2* (*e1*.[*e2*])::*T*›
then obtain *elemT*
  where *wt-e1*: ⦇*prg*=*G*,*cls*=*accC*,*lcl*=*L*⦈⊢*e1*::−*elemT*.[] **and**
    wt-e2: ⦇*prg*=*G*,*cls*=*accC*,*lcl*=*L*⦈⊢*e2*::−*PrimT Integer* **and**
      T: *T*= *Inl elemT*
 by (*rule wt-elim-cases*) *auto*
from *AVar.prems* **obtain** *E1* **where**
 da-e1: ⦇*prg*=*G*,*cls*=*accC*,*lcl*=*L*⦈
    ⊢ (*dom* (*locals* (*store* ((*Norm s0*)::*state*))))»*In1l e1*» *E1* **and**
 da-e2: ⦇*prg*=*G*,*cls*=*accC*,*lcl*=*L*⦈⊢ *nrm E1* »*In1l e2*» *A*
 by (*elim da-elim-cases*) *simp*
from *conf-s0 wt-e1 da-e1*
obtain *conf-s1*: *s1*::≼(*G*, *L*) **and**
   conf-a: (*normal s1* ⟶ *G*,*store s1*⊢*a*::≼*elemT*.[]) **and**
   error-free-s1: *error-free s1*
 by (*rule hyp-e1* [*elim-format*]) *simp*
show *s2′*::≼(*G*, *L*) ∧
   (*normal s2′* ⟶ *G*,*L*,*store s2′*⊢*In2* (*e1*.[*e2*])≻*In2 v*::≼*T*) ∧
   (*error-free* (*Norm s0*) = *error-free s2′*)
proof (*cases normal s1*)
 case *False*
 moreover
 from *False eval-e2* **have** *eq-s2-s1*: *s2*=*s1* **by** *auto*
 moreover
 from *eq-s2-s1 False* **have** ¬ *normal s2* **by** *simp*
 then have *snd* (*avar G i a s2*) = *s2*
  by (*cases s2*) (*simp add*: *avar-def2*)
 with *avar* **have** *s2′*=*s2*
  by (*cases* (*avar G i a s2*)) *simp*
 ultimately show *?thesis*
  using *conf-s1 error-free-s1*
  by *auto*
next
 case *True*
 obtain *A′* **where**
  ⦇*prg*=*G*,*cls*=*accC*,*lcl*=*L*⦈⊢ *dom* (*locals* (*store s1*)) »*In1l e2*» *A′*
 proof −
  from *eval-e1 wt-e1 da-e1 wf True*
  have *nrm E1* ⊆ *dom* (*locals* (*store s1*))
   by (*cases rule*: *da-good-approxE′*) *iprover*
  with *da-e2* **show** *thesis*
   by (*rule da-weakenE*) (*rule that*)
 qed
 with *conf-s1 wt-e2*
 obtain *conf-s2*: *s2*::≼(*G*, *L*) **and** *error-free-s2*: *error-free s2*
  by (*rule hyp-e2* [*elim-format*]) (*simp add*: *error-free-s1*)
 from *avar*
 have *store s2′*=*store s2*
  by (*cases s2*) (*simp add*: *avar-def2*)
 with *avar conf-s2*
 have *conf-s2′*: *s2′*::≼(*G*, *L*)
  by (*cases s2*) (*auto simp add*: *avar-def2*)
 from *avar error-free-s2*
 have *error-free-s2′*: *error-free s2′*
  by (*cases s2*) (*auto simp add*: *avar-def2*)
 have *normal s2′* ⟹
  G,*store s2′*⊢*fst v*::≼*elemT* ∧ *store s2′*≤|*snd v*≼*elemT*::≼(*G*, *L*)
 proof −
  assume *normal*: *normal s2′*

```
      show ?thesis
      proof –
        obtain vv vf x1 store1 x2 store2 store2'
          where  v: v=(vv,vf) and
                s1: s1=(x1,store1) and
                s2: s2=(x2,store2) and
            store2': store2'=store s2'
          by (cases v,cases s1, cases s2, cases s2') blast
        have G,store2'⊢vv::≼elemT ∧ store2'≤|vf≼elemT::≼(G, L)
        proof (rule AVar-lemma [of G x1 store1 e2 i x2 store2 vv vf store2' a,
                            OF wf])
          from s1 s2 eval-e2 show G⊢(x1, store1) −e2−≻i→ (x2, store2)
            by simp
          from v normal s2 store2' avar
          show ((vv, vf), Norm store2') = avar G i a (x2, store2)
            by auto
          from s2 conf-s2 show (x2, store2)::≼(G, L) by simp
          from s1 conf-a show  x1 = None ⟶ G,store1⊢a::≼elemT.[] by simp
          from eval-e2 s1 s2 show store1≤|store2 by (auto dest: eval-gext)
        qed
        with v s1 s2 store2'
        show ?thesis
          by simp
      qed
    qed
    with conf-s2' error-free-s2' T
    show ?thesis
      by auto
  qed
next
  case (Nil s0 L accC T)
  then show ?case
    by (auto elim!: wt-elim-cases)
next
  case (Cons s0 e v s1 es vs s2 L accC T A)
  note eval-e = ‹G⊢Norm s0 −e−≻v→ s1›
  note eval-es = ‹G⊢s1 −es≐≻vs→ s2›
  note hyp-e = ‹PROP ?TypeSafe (Norm s0) s1 (In1l e) (In1 v)›
  note hyp-es = ‹PROP ?TypeSafe s1 s2 (In3 es) (In3 vs)›
  note conf-s0 = ‹Norm s0::≼(G, L)›
  note wt = ‹(|prg = G, cls = accC, lcl = L|)⊢In3 (e # es)::T›
  then obtain eT esT where
     wt-e: (|prg = G, cls = accC, lcl = L|)⊢e::−eT and
     wt-es: (|prg = G, cls = accC, lcl = L|)⊢es::≐esT and
     T: T=Inr (eT#esT)
    by (rule wt-elim-cases) blast
  from Cons.prems obtain E where
    da-e: (|prg=G,cls=accC,lcl=L|)
             ⊢ (dom (locals (store ((Norm s0)::state))))»In1l e» E and
    da-es: (|prg=G,cls=accC,lcl=L|)⊢ nrm E »In3 es» A
    by (elim da-elim-cases) simp
  from conf-s0 wt-e da-e
  obtain conf-s1: s1::≼(G, L) and error-free-s1: error-free s1 and
    conf-v: normal s1 ⟶ G,store s1⊢v::≼eT
    by (rule hyp-e [elim-format]) simp
  show
    s2::≼(G, L) ∧
    (normal s2 ⟶ G,L,store s2⊢In3 (e # es)≻In3 (v # vs)::≼T) ∧
    (error-free (Norm s0) = error-free s2)
```

proof (*cases normal s1*)
case *False*
with *eval-es* **have** *s2=s1* **by** *auto*
with *False conf-s1 error-free-s1*
show *?thesis*
by *auto*
next
case *True*
obtain A' **where**
$(\|prg=G,cls=accC,lcl=L\|)\vdash$ *dom* (*locals* (*store s1*)) »*In3 es*» A'
proof −
from *eval-e wt-e da-e wf True*
have *nrm E* ⊆ *dom* (*locals* (*store s1*))
by (*cases rule*: *da-good-approxE′*) *iprover*
with *da-es* **show** *thesis*
by (*rule da-weakenE*) (*rule that*)
qed
with *conf-s1 wt-es*
obtain *conf-s2*: *s2*::≼(*G*, *L*) **and**
error-free-s2: *error-free s2* **and**
conf-vs: *normal s2* ⟶ *list-all2* (*conf G* (*store s2*)) *vs esT*
by (*rule hyp-es* [*elim-format*]) (*simp add*: *error-free-s1*)
moreover
from *True eval-es conf-v*
have *conf-v′*: *G*,*store s2*⊢*v*::≼*eT*
apply *clarify*
apply (*rule conf-gext*)
apply (*auto dest*: *eval-gext*)
done
ultimately show *?thesis* **using** *T* **by** *simp*
qed
qed
from *this* **and** *conf-s0 wt da* **show** *?thesis* **.**
qed

corollary *eval-type-soundE* [*consumes 5*]:
assumes *eval*: *G*⊢*s0* −*t*≻→ (*v*, *s1*)
and *conf*: *s0*::≼(*G*, *L*)
and *wt*: (|*prg* = *G*, *cls* = *accC*, *lcl* = *L*|)⊢*t*::*T*
and *da*: (|*prg* = *G*, *cls* = *accC*, *lcl* = *L*|)⊢ *dom* (*locals* (*snd s0*)) »*t*» *A*
and *wf*: *wf-prog G*
and *elim*: ⟦*s1*::≼(*G*, *L*); *normal s1* ⟹ *G*,*L*,*snd s1*⊢*t*≻*v*::≼*T*;
error-free s0 = *error-free s1*⟧ ⟹ *P*
shows *P*
using *eval wt da wf conf*
by (*rule eval-type-sound* [*elim-format*]) (*iprover intro*: *elim*)

corollary *eval-ts*:
⟦*G*⊢*s* −*e*−≻*v* → *s′*; *wf-prog G*; *s*::≼(*G*,*L*); (|*prg*=*G*,*cls*=*C*,*lcl*=*L*|)⊢*e*::−*T*;
(|*prg*=*G*,*cls*=*C*,*lcl*=*L*|)⊢*dom* (*locals* (*store s*))»*In1l e*»*A*⟧
⟹ *s′*::≼(*G*,*L*) ∧ (*normal s′* ⟶ *G*,*store s′*⊢*v*::≼*T*) ∧
(*error-free s* = *error-free s′*)
apply (*drule* (*4*) *eval-type-sound*)
apply *clarsimp*
done

corollary *evals-ts*:
⟦*G*⊢*s* −*es*≐≻*vs*→ *s′*; *wf-prog G*; *s*::≼(*G*,*L*); (|*prg*=*G*,*cls*=*C*,*lcl*=*L*|)⊢*es*::≐*Ts*;

```
  (|prg=G,cls=C,lcl=L|)⊢dom (locals (store s))»In3 es»A⟧
⟹  s'::≼(G,L) ∧ (normal s' ⟶ list-all2 (conf G (store s')) vs Ts) ∧
     (error-free s = error-free s')
apply (drule (4) eval-type-sound)
apply clarsimp
done

corollary evar-ts:
⟦G⊢s −v=≻vf→ s'; wf-prog G; s::≼(G,L); (|prg=G,cls=C,lcl=L|)⊢v::=T;
 (|prg=G,cls=C,lcl=L|)⊢dom (locals (store s))»In2 v»A⟧ ⟹
  s'::≼(G,L) ∧ (normal s' ⟶ G,L,(store s')⊢In2 v≻In2 vf::≼Inl T) ∧
  (error-free s = error-free s')
apply (drule (4) eval-type-sound)
apply clarsimp
done

theorem exec-ts:
⟦G⊢s −c→ s'; wf-prog G; s::≼(G,L); (|prg=G,cls=C,lcl=L|)⊢c::√;
 (|prg=G,cls=C,lcl=L|)⊢dom (locals (store s))»In1r c»A⟧
 ⟹ s'::≼(G,L) ∧ (error-free s ⟶ error-free s')
apply (drule (4) eval-type-sound)
apply clarsimp
done


lemma wf-eval-Fin:
  assumes wf:    wf-prog G
    and   wt-c1: (|prg = G, cls = C, lcl = L|)⊢In1r c1::Inl (PrimT Void)
    and   da-c1: (|prg=G,cls=C,lcl=L|)⊢dom (locals (store (Norm s0)))»In1r c1»A
    and conf-s0: Norm s0::≼(G, L)
    and eval-c1: G⊢Norm s0 −c1→ (x1,s1)
    and eval-c2: G⊢Norm s1 −c2→ s2
    and      s3: s3=abupd (abrupt-if (x1≠None) x1) s2
  shows G⊢Norm s0 −c1 Finally c2→ s3
proof −
  from eval-c1 wt-c1 da-c1 wf conf-s0
  have error-free (x1,s1)
    by (auto dest: eval-type-sound)
  with eval-c1 eval-c2 s3
  show ?thesis
    by − (rule eval.Fin, auto simp add: error-free-def)
qed
```

3 Ideas for the future

In the type soundness proof and the correctness proof of definite assignment we perform induction on the evaluation relation with the further preconditions that the term is welltyped and definitely assigned. During the proofs we have to establish the welltypedness and definite assignment of the subterms to be able to apply the induction hypothesis. So large parts of both proofs are the same work in propagating welltypedness and definite assignment. So we can derive a new induction rule for induction on the evaluation of a wellformed term, were these propagations is already done, once and forever. Then we can do the proofs with this rule and can enjoy the time we have saved. Here is a first and incomplete sketch of such a rule.

```
theorem wellformed-eval-induct [consumes 4, case-names Abrupt Skip Expr Lab
                                Comp If]:
  assumes  eval: G⊢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  abrupt: ⋀ s t abr L accC T A.
                [[(|prg=G,cls=accC,lcl=L|)⊢t::T;
                  (|prg=G,cls=accC,lcl=L|)⊢dom (locals (store (Some abr,s)))»t»A
                ]] ⟹ P L accC (Some abr, s) t (undefined3 t) (Some abr, s)
  and    skip: ⋀ s L accC. P L accC (Norm s) ⟨Skip⟩s ◇ (Norm s)
  and    expr: ⋀ e s0 s1 v L accC eT E.
              [[(|prg=G,cls=accC,lcl=L|)⊢e::−eT;
                (|prg=G,cls=accC,lcl=L|)
                  ⊢dom (locals (store ((Norm s0)::state)))»⟨e⟩e»E;
                P L accC (Norm s0) ⟨e⟩e ⌊v⌋e s1]]
              ⟹  P L accC (Norm s0) ⟨Expr e⟩s ◇ s1
  and     lab: ⋀ c l s0 s1 L accC C.
              [[(|prg=G,cls=accC,lcl=L|)⊢c::√;
                (|prg=G,cls=accC, lcl=L|)
                  ⊢dom (locals (store ((Norm s0)::state)))»⟨c⟩s»C;
                P L accC (Norm s0) ⟨c⟩s ◇ s1]]
                ⟹ P L accC (Norm s0) ⟨l· c⟩s ◇ (abupd (absorb l) s1)
  and    comp: ⋀ c1 c2 s0 s1 s2 L accC C1.
              [[G⊢Norm s0 −c1 → s1;G⊢s1 −c2 → s2;
                (|prg=G,cls=accC,lcl=L|)⊢c1::√;
                (|prg=G,cls=accC,lcl=L|)⊢c2::√;
                (|prg=G,cls=accC,lcl=L|)⊢
                   dom (locals (store ((Norm s0)::state))) »⟨c1⟩s» C1;
                P L accC (Norm s0) ⟨c1⟩s ◇ s1;
                ⋀ Q. [[normal s1;
                       ⋀ C2.[[(|prg=G,cls=accC,lcl=L|)
                                ⊢dom (locals (store s1)) »⟨c2⟩s» C2;
                             P L accC s1 ⟨c2⟩s ◇ s2]] ⟹ Q
                     ]] ⟹ Q
                ]]⟹ P L accC (Norm s0) ⟨c1;; c2⟩s ◇ s2
   and  if: ⋀ b c1 c2 e s0 s1 s2 L accC E.
              [[G⊢Norm s0 −e−≻b→ s1;
               G⊢s1 −(if the-Bool b then c1 else c2)→ s2;
               (|prg=G,cls=accC,lcl=L|)⊢e::−PrimT Boolean;
               (|prg=G, cls=accC, lcl=L|)⊢(if the-Bool b then c1 else c2)::√;
               (|prg=G,cls=accC,lcl=L|)⊢
                  dom (locals (store ((Norm s0)::state))) »⟨e⟩e» E;
               P L accC (Norm s0) ⟨e⟩e ⌊b⌋e s1;
               ⋀ Q. [[normal s1;
                      ⋀ C. [[(|prg=G,cls=accC,lcl=L|)⊢ (dom (locals (store s1)))
                                 »⟨if the-Bool b then c1 else c2⟩s» C;
                            P L accC s1 ⟨if the-Bool b then c1 else c2⟩s ◇ s2
                            ]] ⟹ Q
                     ]] ⟹ Q
              ]] ⟹ P L accC (Norm s0) ⟨If(e) c1 Else c2⟩s ◇ s2
  shows P L accC s0 t v s1
proof −
 note inj-term-simps [simp]
 from eval
 have ⋀L accC T A. [[(|prg=G,cls=accC,lcl=L|)⊢t::T;
                    (|prg=G,cls=accC,lcl=L|)⊢dom (locals (store s0))»t»A]]
       ⟹ P L accC s0 t v s1 (is PROP ?Hyp s0 t v s1)
 proof (induct)
   case Abrupt with abrupt show ?case .
 next
   case Skip from skip show ?case by simp
 next
```

case (*Expr s0 e v s1 L accC T A*)
from *Expr.prems* **obtain** *eT* **where**
⦇*prg* = *G*, *cls* = *accC*, *lcl* = *L*⦈⊢*e*::−*eT*
by (*elim wt-elim-cases*)
moreover
from *Expr.prems* **obtain** *E* **where**
⦇*prg*=*G*,*cls*=*accC*, *lcl*=*L*⦈⊢*dom* (*locals* (*store* ((*Norm s0*)::*state*)))»⟨*e*⟩$_e$»*E*
by (*elim da-elim-cases*) *simp*
moreover from *calculation*
have *P L accC* (*Norm s0*) ⟨*e*⟩$_e$ ⌊*v*⌋$_e$ *s1*
by (*rule Expr.hyps*)
ultimately show *?case*
by (*rule expr*)
next
case (*Lab s0 c s1 l L accC T A*)
from *Lab.prems*
have ⦇*prg* = *G*, *cls* = *accC*, *lcl* = *L*⦈⊢*c*::√
by (*elim wt-elim-cases*)
moreover
from *Lab.prems* **obtain** *C* **where**
⦇*prg*=*G*,*cls*=*accC*, *lcl*=*L*⦈⊢*dom* (*locals* (*store* ((*Norm s0*)::*state*)))»⟨*c*⟩$_s$»*C*
by (*elim da-elim-cases*) *simp*
moreover from *calculation*
have *P L accC* (*Norm s0*) ⟨*c*⟩$_s$ ♢ *s1*
by (*rule Lab.hyps*)
ultimately show *?case*
by (*rule lab*)
next
case (*Comp s0 c1 s1 c2 s2 L accC T A*)
note *eval-c1* = ‹*G*⊢*Norm s0* −*c1*→ *s1*›
note *eval-c2* = ‹*G*⊢*s1* −*c2*→ *s2*›
from *Comp.prems* **obtain**
wt-c1: ⦇*prg* = *G*, *cls* = *accC*, *lcl* = *L*⦈⊢*c1*::√ **and**
wt-c2: ⦇*prg* = *G*, *cls* = *accC*, *lcl* = *L*⦈⊢*c2*::√
by (*elim wt-elim-cases*)
from *Comp.prems*
obtain *C1 C2*
where *da-c1*: ⦇*prg*=*G*, *cls*=*accC*, *lcl*=*L*⦈⊢
dom (*locals* (*store* ((*Norm s0*)::*state*))) »⟨*c1*⟩$_s$» *C1* **and**
da-c2: ⦇*prg*=*G*, *cls*=*accC*, *lcl*=*L*⦈⊢ *nrm C1* »⟨*c2*⟩$_s$» *C2*
by (*elim da-elim-cases*) *simp*
from *wt-c1 da-c1*
have *P-c1*: *P L accC* (*Norm s0*) ⟨*c1*⟩$_s$ ♢ *s1*
by (*rule Comp.hyps*)
{
fix *Q*
assume *normal-s1*: *normal s1*
assume *elim*: ⋀ *C2'*.
⟦⦇*prg*=*G*,*cls*=*accC*,*lcl*=*L*⦈⊢*dom* (*locals* (*store s1*))»⟨*c2*⟩$_s$»*C2'*;
P L accC s1 ⟨*c2*⟩$_s$ ♢ *s2*⟧ ⟹ *Q*
have *Q*
proof −
obtain *C2'* **where**
da: ⦇*prg*=*G*, *cls*=*accC*, *lcl*=*L*⦈⊢ *dom* (*locals* (*store s1*)) »⟨*c2*⟩$_s$» *C2'*
proof −
from *eval-c1 wt-c1 da-c1 wf normal-s1*
have *nrm C1* ⊆ *dom* (*locals* (*store s1*))
by (*cases rule*: *da-good-approxE'*) *iprover*
with *da-c2* **show** *thesis*

by (*rule da-weakenE*) (*rule that*)
qed
with *wt-c2* **have** *P L accC s1* $\langle c2\rangle_s$ ♢ *s2*
by (*rule Comp.hyps*)
with *da* **show** *?thesis*
using *elim* **by** *iprover*
qed
}
with *eval-c1 eval-c2 wt-c1 wt-c2 da-c1 P-c1*
show *?case*
by (*rule comp*) *iprover+*
next
case (*If s0 e b s1 c1 c2 s2 L accC T A*)
note *eval-e* = ‹*G*⊢*Norm s0* −*e*−≻*b*→ *s1*›
note *eval-then-else* = ‹*G*⊢*s1* −(*if the-Bool b then c1 else c2*)→ *s2*›
from *If.prems*
obtain
wt-e: (|*prg*=*G*, *cls*=*accC*, *lcl*=*L*|)⊢*e*::−*PrimT Boolean* **and**
wt-then-else: (|*prg*=*G*, *cls*=*accC*, *lcl*=*L*|)⊢(*if the-Bool b then c1 else c2*)::√
by (*elim wt-elim-cases*) *auto*
from *If.prems* **obtain** *E C* **where**
da-e: (|*prg*=*G*,*cls*=*accC*,*lcl*=*L*|)⊢ *dom* (*locals* (*store* ((*Norm s0*)::*state*)))
»$\langle e\rangle_e$» *E* **and**
da-then-else:
(|*prg*=*G*,*cls*=*accC*,*lcl*=*L*|)⊢
(*dom* (*locals* (*store* ((*Norm s0*)::*state*))) ∪ *assigns-if* (*the-Bool b*) *e*)
»⟨*if the-Bool b then c1 else c2*⟩$_s$» *C*
by (*elim da-elim-cases*) (*cases the-Bool b*,*auto*)
from *wt-e da-e*
have *P-e*: *P L accC* (*Norm s0*) $\langle e\rangle_e$ $\lfloor b\rfloor_e$ *s1*
by (*rule If.hyps*)
{
fix *Q*
assume *normal-s1*: *normal s1*
assume *elim*: ⋀ *C*. ⟦(|*prg*=*G*,*cls*=*accC*,*lcl*=*L*|)⊢ (*dom* (*locals* (*store s1*)))
»⟨*if the-Bool b then c1 else c2*⟩$_s$» *C*;
P L accC s1 ⟨*if the-Bool b then c1 else c2*⟩$_s$ ♢ *s2*
⟧ ⟹ *Q*
have *Q*
proof −
obtain *C′* **where**
da: (|*prg*=*G*,*cls*=*accC*,*lcl*=*L*|)⊢
(*dom* (*locals* (*store s1*)))»⟨*if the-Bool b then c1 else c2*⟩$_s$ » *C′*
proof −
from *eval-e* **have**
dom (*locals* (*store* ((*Norm s0*)::*state*))) ⊆ *dom* (*locals* (*store s1*))
by (*rule dom-locals-eval-mono-elim*)
moreover
from *eval-e normal-s1 wt-e*
have *assigns-if* (*the-Bool b*) *e* ⊆ *dom* (*locals* (*store s1*))
by (*rule assigns-if-good-approx′*)
ultimately
have *dom* (*locals* (*store* ((*Norm s0*)::*state*)))
∪ *assigns-if* (*the-Bool b*) *e* ⊆ *dom* (*locals* (*store s1*))
by (*rule Un-least*)
with *da-then-else* **show** *thesis*
by (*rule da-weakenE*) (*rule that*)
qed
with *wt-then-else*

have *P L accC s1* $\langle$*if the-Bool b then c1 else c2*$\rangle_s$ $\diamondsuit$ *s2*
by (*rule If.hyps*)
with *da* **show** *?thesis* **using** *elim* **by** *iprover*
qed
}
with *eval-e eval-then-else wt-e wt-then-else da-e P-e*
show *?case*
by (*rule if*) *iprover+*
next
oops

end

Chapter 20

Evaln

1 Operational evaluation (big-step) semantics of Java expressions and statements

theory *Evaln* **imports** *TypeSafe* **begin**

Variant of *eval* relation with counter for bounded recursive depth. In principal *evaln* could replace *eval*.

Validity of the axiomatic semantics builds on *evaln*. For recursive method calls the axiomatic semantics rule assumes the method ok to derive a proof for the body. To prove the method rule sound we need to perform induction on the recursion depth. For the completeness proof of the axiomatic semantics the notion of the most general formula is used. The most general formula right now builds on the ordinary evaluation relation *eval*. So sometimes we have to switch between *evaln* and *eval* and vice versa. To make this switch easy *evaln* also does all the technical accessibility tests *check-field-access* and *check-method-access* like *eval*. If it would omit them *evaln* and *eval* would only be equivalent for welltyped, and definitely assigned terms.

inductive
  evaln :: [*prog*, *state*, *term*, *nat*, *vals*, *state*] ⇒ *bool*
    (-⊢- −-≻−-→ ′(-, -′) [*61*,*61*,*80*,*61*,*0*,*0*] *60*)
  and *evarn* :: [*prog*, *state*, *var*, *vvar*, *nat*, *state*] ⇒ *bool*
    (-⊢- −-=≻−-→ - [*61*,*61*,*90*,*61*,*61*,*61*] *60*)
  and *eval-n*:: [*prog*, *state*, *expr*, *val*, *nat*, *state*] ⇒ *bool*
    (-⊢- −-−≻−-→ - [*61*,*61*,*80*,*61*,*61*,*61*] *60*)
  and *evalsn* :: [*prog*, *state*, *expr list*, *val list*, *nat*, *state*] ⇒ *bool*
    (-⊢- −-≐≻−-→ - [*61*,*61*,*61*,*61*,*61*,*61*] *60*)
  and *execn* :: [*prog*, *state*, *stmt*, *nat*, *state*] ⇒ *bool*
    (-⊢- −-−-→ - [*61*,*61*,*65*, *61*,*61*] *60*)
  for *G* :: *prog*
where

  G⊢*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*: $[\![G\vdash Norm\ s0\ -Init\ statDeclC-n\rightarrow\ s1;\ G\vdash s1\ -e-\succ a-n\rightarrow\ s2;$
$(v,s2') = fvar\ statDeclC\ stat\ fn\ a\ s2;$
$s3 = check\text{-}field\text{-}access\ G\ accC\ statDeclC\ fn\ stat\ a\ s2']\!] \Longrightarrow$
$G\vdash Norm\ s0\ -\{accC,statDeclC,stat\}e..fn{=}\succ v-n\rightarrow\ s3$

| *AVar*: $[\![G\vdash\ Norm\ s0\ -e1-\succ a-n\rightarrow\ s1\ ;\ G\vdash s1\ -e2-\succ i-n\rightarrow\ s2;$
$(v,s2') = avar\ G\ i\ a\ s2]\!] \Longrightarrow$
$G\vdash Norm\ s0\ -e1.[e2]{=}\succ v-n\rightarrow\ s2'$

— evaluation of expressions

| *NewC*: $[\![G\vdash Norm\ s0\ -Init\ C-n\rightarrow\ s1;$
$G\vdash\quad s1\ -halloc\ (CInst\ C)\succ a\rightarrow\ s2]\!] \Longrightarrow$
$G\vdash Norm\ s0\ -NewC\ C-\succ Addr\ a-n\rightarrow\ s2$

| *NewA*: $[\![G\vdash Norm\ s0\ -init\text{-}comp\text{-}ty\ T-n\rightarrow\ s1;\ G\vdash s1\ -e-\succ i'-n\rightarrow\ s2;$
$G\vdash abupd\ (check\text{-}neg\ i')\ s2\ -halloc\ (Arr\ T\ (the\text{-}Intg\ i'))\succ a\rightarrow\ s3]\!] \Longrightarrow$
$G\vdash Norm\ s0\ -New\ T[e]-\succ Addr\ a-n\rightarrow\ s3$

| *Cast*: $[\![G\vdash Norm\ s0\ -e-\succ v-n\rightarrow\ s1;$
$s2 = abupd\ (raise\text{-}if\ (\neg G,snd\ s1\vdash v\ fits\ T)\ ClassCast)\ s1]\!] \Longrightarrow$
$G\vdash Norm\ s0\ -Cast\ T\ e-\succ v-n\rightarrow\ s2$

| *Inst*: $[\![G\vdash Norm\ s0\ -e-\succ v-n\rightarrow\ s1;$
$b = (v\neq Null\ \wedge\ G,store\ s1\vdash v\ fits\ RefT\ T)]\!] \Longrightarrow$
$G\vdash Norm\ s0\ -e\ InstOf\ T-\succ Bool\ b-n\rightarrow\ s1$

| *Lit*: $G\vdash Norm\ s\ -Lit\ v-\succ v-n\rightarrow\ Norm\ s$

| *UnOp*: $[\![G\vdash Norm\ s0\ -e-\succ v-n\rightarrow\ s1]\!]$
$\Longrightarrow G\vdash Norm\ s0\ -UnOp\ unop\ e-\succ(eval\text{-}unop\ unop\ v)-n\rightarrow\ s1$

| *BinOp*: $[\![G\vdash Norm\ s0\ -e1-\succ v1-n\rightarrow\ s1;$
$G\vdash s1\ -(if\ need\text{-}second\text{-}arg\ binop\ v1\ then\ (In1l\ e2)\ else\ (In1r\ Skip))$
$\succ-n\rightarrow\ (In1\ v2,s2)]\!]$
$\Longrightarrow G\vdash Norm\ s0\ -BinOp\ binop\ e1\ e2-\succ(eval\text{-}binop\ binop\ v1\ v2)-n\rightarrow\ s2$

| *Super*: $G\vdash Norm\ s\ -Super-\succ val\text{-}this\ s-n\rightarrow\ Norm\ s$

| *Acc*: $[\![G\vdash Norm\ s0\ -va{=}\succ(v,f)-n\rightarrow\ s1]\!] \Longrightarrow$
$G\vdash Norm\ s0\ -Acc\ va-\succ v-n\rightarrow\ s1$

| *Ass*: $[\![G\vdash Norm\ s0\ -va{=}\succ(w,f)-n\rightarrow\ s1;$
$G\vdash\quad s1\ -e-\succ v\quad -n\rightarrow\ s2]\!] \Longrightarrow$
$G\vdash Norm\ s0\ -va{:=}e-\succ v-n\rightarrow\ assign\ f\ v\ s2$

| *Cond*: $[\![G\vdash Norm\ s0\ -e0-\succ b-n\rightarrow\ s1;$
$G\vdash\quad s1\ -(if\ the\text{-}Bool\ b\ then\ e1\ else\ e2)-\succ v-n\rightarrow\ s2]\!] \Longrightarrow$
$G\vdash Norm\ s0\ -e0\ ?\ e1\ :\ e2-\succ v-n\rightarrow\ s2$

| *Call*:
$[\![G\vdash Norm\ s0\ -e-\succ a'-n\rightarrow\ s1;\ G\vdash s1\ -args\dot{=}\succ vs-n\rightarrow\ s2;$
$D = invocation\text{-}declclass\ G\ mode\ (store\ s2)\ a'\ statT\ (\!|name{=}mn,parTs{=}pTs|\!);$
$s3{=}init\text{-}lvars\ G\ D\ (\!|name{=}mn,parTs{=}pTs|\!)\ mode\ a'\ vs\ s2;$
$s3' = check\text{-}method\text{-}access\ G\ accC\ statT\ mode\ (\!|name{=}mn,parTs{=}pTs|\!)\ a'\ s3;$

$G \vdash s3' - Methd\ D\ (\!|name{=}mn, parTs{=}pTs|\!) - \succ v - n \rightarrow\ s4$
$]\!]$
$\Longrightarrow$
$G \vdash Norm\ s0\ -\{accC, statT, mode\} e \cdot mn(\{pTs\} args) - \succ v - n \rightarrow\ (restore\text{-}lvars\ s2\ s4)$

| *Methd*: $[\![G \vdash Norm\ s0\ - body\ G\ D\ sig - \succ v - n \rightarrow\ s1]\!] \Longrightarrow$
$G \vdash Norm\ s0\ - Methd\ D\ sig - \succ v - Suc\ n \rightarrow\ s1$

| *Body*: $[\![G \vdash Norm\ s0 - Init\ D - n \rightarrow\ s1;\ G \vdash s1\ - c - n \rightarrow\ s2;$
$s3 = (if\ (\exists\ l.\ abrupt\ s2 = Some\ (Jump\ (Break\ l)) \vee$
$abrupt\ s2 = Some\ (Jump\ (Cont\ l)))$
$then\ abupd\ (\lambda\ x.\ Some\ (Error\ CrossMethodJump))\ s2$
$else\ s2)]\!] \Longrightarrow$
$G \vdash Norm\ s0\ - Body\ D\ c$
$- \succ the\ (locals\ (store\ s2)\ Result) - n \rightarrow abupd\ (absorb\ Ret)\ s3$

— 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-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-class-obj G C s0*)
$-$(*if C* = *Object then Skip else Init* (*super c*))$-n\rightarrow$ *s1* $\wedge$
$G\vdash$*set-lvars Map.empty s1* $-$*init c*$-n\rightarrow$ *s2* $\wedge$
s3 = *restore-lvars s1 s2*)⟧
$\Longrightarrow$
$G\vdash$*Norm s0* $-$*Init C*$-n\rightarrow$ *s3*

monos
if-bool-eq-conj

declare *if-split* [*split del*] *if-split-asm* [*split del*]
option.split [*split del*] *option.split-asm* [*split del*]
not-None-eq [*simp del*]
split-paired-All [*simp del*] *split-paired-Ex* [*simp del*]
setup ‹*map-theory-simpset* (*fn ctxt* => *ctxt delloop split-all-tac*)›

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*• *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*• *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⊢Norm s −In1l ({accC,statT,mode}e·mn({pT}p)) ≻−n→ (v, s′)
G⊢Norm s −In1r (Init C)                       ≻−n→ (x, s′)
```

```
declare if-split     [split] if-split-asm     [split]
        option.split [split] option.split-asm [split]
        not-None-eq [simp]
        split-paired-All [simp] split-paired-Ex [simp]
declaration ‹K (Simplifier.map-ss (fn ss => ss addloop (split-all-tac, split-all-tac)))›
```

```
lemma evaln-Inj-elim: G⊢s −t≻−n→ (w,s′) ⟹ 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)
apply (erule evaln-cases , auto)
apply (induct-tac t)
apply  (rename-tac a, induct-tac a)
apply auto
done
```

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⊢s −In1l t≻−n→ (w, s′) = (∃ v. w=In1 v ∧ G⊢s −t−≻v −n→ s′)
  by (auto, frule evaln-Inj-elim, auto)
```

```
lemma evaln-var-eq: G⊢s −In2 t≻−n→ (w, s′) = (∃ vf. w=In2 vf ∧ G⊢s −t=≻vf−n→ s′)
  by (auto, frule evaln-Inj-elim, auto)
```

```
lemma evaln-exprs-eq: G⊢s −In3 t≻−n→ (w, s′) = (∃ vs. w=In3 vs ∧ G⊢s −t≐≻vs−n→ s′)
  by (auto, frule evaln-Inj-elim, auto)
```

```
lemma evaln-stmt-eq: G⊢s −In1r t≻−n→ (w, s′) = (w=◇ ∧ G⊢s −t −n→ s′)
  by (auto, frule evaln-Inj-elim, auto, frule evaln-Inj-elim, auto)
```

```
simproc-setup evaln-expr (G⊢s −In1l t≻−n→ (w, s′)) = ‹
  K (K (fn ct =>
    (case Thm.term-of ct of
      (- $ - $ - $ - $ - $ (Const - $ -) $ -) => NONE
    | - => SOME (mk-meta-eq @{thm evaln-expr-eq}))))›

simproc-setup evaln-var (G⊢s −In2 t≻−n→ (w, s′)) = ‹
  K (K (fn ct =>
    (case Thm.term-of ct of
      (- $ - $ - $ - $ - $ (Const - $ -) $ -) => NONE
    | - => SOME (mk-meta-eq @{thm evaln-var-eq}))))›

simproc-setup evaln-exprs (G⊢s −In3 t≻−n→ (w, s′)) = ‹
  K (K (fn ct =>
    (case Thm.term-of ct of
      (- $ - $ - $ - $ - $ (Const - $ -) $ -) => NONE
    | - => SOME (mk-meta-eq @{thm evaln-exprs-eq}))))›

simproc-setup evaln-stmt (G⊢s −In1r t≻−n→ (w, s′)) = ‹
  K (K (fn ct =>
    (case Thm.term-of ct of
```

(- $ - $ - $ - $ - $ (*Const* - $ -) $ -) => *NONE*
| - => *SOME* (*mk-meta-eq* @{*thm evaln-stmt-eq*}))))›

ML ‹*ML-Thms.bind-thms* (*evaln-AbruptIs*, *sum3-instantiate* ***context*** @{*thm evaln.Abrupt*})›
declare *evaln-AbruptIs* [*intro*!]

lemma *evaln-Callee*: *G*⊢*Norm s*−*In1l* (*Callee l e*)≻−*n*→ (*v*,*s*′) = *False*
proof −
{ **fix** *s t v s*′
assume *eval*: *G*⊢*s* −*t*≻−*n*→ (*v*,*s*′) **and**
normal: *normal s* **and**
callee: *t*=*In1l* (*Callee l e*)
then have *False* **by** *induct auto*
}
then show *?thesis*
by (*cases s*′) *fastforce*
qed

lemma *evaln-InsInitE*: *G*⊢*Norm s*−*In1l* (*InsInitE c e*)≻−*n*→ (*v*,*s*′) = *False*
proof −
{ **fix** *s t v s*′
assume *eval*: *G*⊢*s* −*t*≻−*n*→ (*v*,*s*′) **and**
normal: *normal s* **and**
callee: *t*=*In1l* (*InsInitE c e*)
then have *False* **by** *induct auto*
}
then show *?thesis*
by (*cases s*′) *fastforce*
qed

lemma *evaln-InsInitV*: *G*⊢*Norm s*−*In2* (*InsInitV c w*)≻−*n*→ (*v*,*s*′) = *False*
proof −
{ **fix** *s t v s*′
assume *eval*: *G*⊢*s* −*t*≻−*n*→ (*v*,*s*′) **and**
normal: *normal s* **and**
callee: *t*=*In2* (*InsInitV c w*)
then have *False* **by** *induct auto*
}
then show *?thesis*
by (*cases s*′) *fastforce*
qed

lemma *evaln-FinA*: *G*⊢*Norm s*−*In1r* (*FinA a c*)≻−*n*→ (*v*,*s*′) = *False*
proof −
{ **fix** *s t v s*′
assume *eval*: *G*⊢*s* −*t*≻−*n*→ (*v*,*s*′) **and**
normal: *normal s* **and**
callee: *t*=*In1r* (*FinA a c*)
then have *False* **by** *induct auto*
}
then show *?thesis*
by (*cases s*′) *fastforce*
qed

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*
apply (*erule evaln-cases* , *auto*)
done

lemma *evaln-abrupt*:
$\bigwedge s'.\ G\vdash(Some\ xc,s)\ -e\succ-n\rightarrow\ (w,s') = (s' = (Some\ xc,s) \wedge$
$w=undefined3\ e \wedge G\vdash(Some\ xc,s)\ -e\succ-n\rightarrow\ (undefined3\ e,(Some\ xc,s)))$
apply *auto*
apply (*frule evaln-abrupt-lemma*, *auto*)+
done

simproc-setup *evaln-abrupt* ($G\vdash(Some\ xc,s)\ -e\succ-n\rightarrow\ (w,s')$) = ‹
K (*K* (*fn ct* =>
(*case Thm.term-of ct of*
(- \$ - \$ - \$ - \$ - \$ - \$ (*Const* (***const-name*** ‹*Pair*›, -) \$ (*Const* (***const-name*** ‹*Some*›,-) \$ -)\$ -))
=> *NONE*
| - => *SOME* (*mk-meta-eq* @{*thm evaln-abrupt*}))))
›

lemma *evaln-LitI*: $G\vdash s\ -Lit\ v-\succ(if\ normal\ s\ then\ v\ else\ undefined)-n\rightarrow\ s$
apply (*case-tac s*, *case-tac a* = *None*)
by (*auto intro*!: *evaln.Lit*)

lemma *CondI*:
$\bigwedge s1.\ [\![G\vdash s\ -e-\succ b-n\rightarrow\ s1;\ G\vdash s1\ -(if\ the\text{-}Bool\ b\ then\ e1\ else\ e2)-\succ v-n\rightarrow\ s2]\!] \Longrightarrow$
$G\vdash s\ -e\ ?\ e1\ :\ e2-\succ(if\ normal\ s1\ then\ v\ else\ undefined)-n\rightarrow\ s2$
apply (*case-tac s*, *case-tac a* = *None*)
by (*auto intro*!: *evaln.Cond*)

lemma *evaln-SkipI* [*intro*!]: $G\vdash s\ -Skip-n\rightarrow\ s$
apply (*case-tac s*, *case-tac a* = *None*)
by (*auto intro*!: *evaln.Skip*)

lemma *evaln-ExprI*: $G\vdash s\ -e-\succ v-n\rightarrow\ s' \Longrightarrow G\vdash s\ -Expr\ e-n\rightarrow\ s'$
apply (*case-tac s*, *case-tac a* = *None*)
by (*auto intro*!: *evaln.Expr*)

lemma *evaln-CompI*: $[\![G\vdash s\ -c1-n\rightarrow\ s1;\ G\vdash s1\ -c2-n\rightarrow\ s2]\!] \Longrightarrow G\vdash s\ -c1;;\ c2-n\rightarrow\ s2$
apply (*case-tac s*, *case-tac a* = *None*)
by (*auto intro*!: *evaln.Comp*)

lemma *evaln-IfI*:
$[\![G\vdash s\ -e-\succ v-n\rightarrow\ s1;\ G\vdash s1\ -(if\ the\text{-}Bool\ v\ then\ c1\ else\ c2)-n\rightarrow\ s2]\!] \Longrightarrow$
$G\vdash s\ -If(e)\ c1\ Else\ c2-n\rightarrow\ s2$
apply (*case-tac s*, *case-tac a* = *None*)
by (*auto intro*!: *evaln.If*)

lemma *evaln-SkipD* [*dest*!]: $G\vdash s\ -Skip-n\rightarrow\ s' \Longrightarrow s' = s$
by (*erule evaln-cases*, *auto*)

```
lemma evaln-Skip-eq [simp]: G⊢s −Skip−n→ s′ = (s = s′)
apply auto
done
```

evaln implies eval

```
lemma evaln-eval:
  assumes evaln: G⊢s0 −t≻−n→ (v,s1)
  shows G⊢s0 −t≻→ (v,s1)
using evaln
proof (induct)
  case (Loop s0 e b n s1 c s2 l s3)
  note ‹G⊢Norm s0 −e−≻b→ s1›
  moreover
  have if the-Bool b
       then (G⊢s1 −c→ s2) ∧
            G⊢abupd (absorb (Cont l)) s2 −l• While(e) c→ s3
       else s3 = s1
    using Loop.hyps by simp
  ultimately show ?case by (rule eval.Loop)
next
  case (Try s0 c1 n s1 s2 C vn c2 s3)
  note ‹G⊢Norm s0 −c1→ s1›
  moreover
  note ‹G⊢s1 −sxalloc→ s2›
  moreover
  have if G,s2⊢catch C then G⊢new-xcpt-var vn s2 −c2→ s3 else s3 = s2
    using Try.hyps by simp
  ultimately show ?case by (rule eval.Try)
next
  case (Init C c s0 s3 n s1 s2)
  note ‹the (class G C) = c›
  moreover
  have if inited C (globs s0)
         then s3 = Norm s0
         else G⊢Norm ((init-class-obj G C) s0)
                −(if C = Object then Skip else Init (super c))→ s1 ∧
              G⊢(set-lvars Map.empty) s1 −init c→ s2 ∧
              s3 = (set-lvars (locals (store s1))) s2
    using Init.hyps by simp
  ultimately show ?case by (rule eval.Init)
qed (rule eval.intros,(assumption+ | assumption?))+
```

```
lemma Suc-le-D-lemma: ⟦Suc n <= m′; (⋀m. n <= m ⟹ P (Suc m)) ⟧ ⟹ P m′
apply (frule Suc-le-D)
apply fast
done
```

```
lemma evaln-nonstrict [rule-format (no-asm), elim]:
  G⊢s −t≻−n→ (w, s′) ⟹ ∀ m. n≤m ⟶ G⊢s −t≻−m→ (w, s′)
apply (erule evaln.induct)
apply (tactic ‹ALLGOALS (EVERY′ [strip-tac context,
  TRY o eresolve-tac context @{thms Suc-le-D-lemma},
  REPEAT o smp-tac context 1,
  resolve-tac context @{thms evaln.intros} THEN-ALL-NEW TRY o assume-tac context])›)
```

apply (*auto split del*: *if-split*)
done

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′)
by (*fast intro*: *max.cobounded1 max.cobounded2*)

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
by (*drule* (*1*) *evaln-max2*) *simp*

lemma *evaln-max3*:
⟦G⊢s1 −t1≻−n1→ (w1, s1′); G⊢s2 −t2≻−n2→ (w2, s2′); G⊢s3 −t3≻−n3→ (w3, s3′)⟧ ⟹
G⊢s1 −t1≻−max (max n1 n2) n3→ (w1, s1′) ∧
G⊢s2 −t2≻−max (max n1 n2) n3→ (w2, s2′) ∧
G⊢s3 −t3≻−max (max n1 n2) n3→ (w3, s3′)
apply (*drule* (*1*) *evaln-max2*, *erule thin-rl*)
apply (*fast intro*!: *max.cobounded1 max.cobounded2*)
done

corollary *evaln-max3E*:
⟦G⊢s1 −t1≻−n1→ (w1, s1′); G⊢s2 −t2≻−n2→ (w2, s2′); G⊢s3 −t3≻−n3→ (w3, s3′);
⟦G⊢s1 −t1≻−max (max n1 n2) n3→ (w1, s1′);
G⊢s2 −t2≻−max (max n1 n2) n3→ (w2, s2′);
G⊢s3 −t3≻−max (max n1 n2) n3→ (w3, s3′)
⟧ ⟹ P
⟧ ⟹ P
by (*drule* (*2*) *evaln-max3*) *simp*

lemma *le-max3I1*: (*n2*::*nat*) ≤ *max n1* (*max n2 n3*)
proof −
have *n2* ≤ *max n2 n3*
by (*rule max.cobounded1*)
also
have *max n2 n3* ≤ *max n1* (*max n2 n3*)
by (*rule max.cobounded2*)
finally
show *?thesis* **.**
qed

lemma *le-max3I2*: (*n3*::*nat*) ≤ *max n1* (*max n2 n3*)
proof −
have *n3* ≤ *max n2 n3*
by (*rule max.cobounded2*)
also
have *max n2 n3* ≤ *max n1* (*max n2 n3*)
by (*rule max.cobounded2*)
finally
show *?thesis* **.**
qed

declare [[*simproc del*: *wt-expr wt-var wt-exprs wt-stmt*]]

eval implies evaln

lemma *eval-evaln*:
  assumes *eval*: *G*⊢*s0* −*t*≻→ (*v*,*s1*)
  shows ∃ *n*. *G*⊢*s0* −*t*≻−*n*→ (*v*,*s1*)
using *eval*
proof (*induct*)
  case (*Abrupt xc s t*)
  obtain *n* **where**
    G⊢(*Some xc*, *s*) −*t*≻−*n*→ (*undefined3 t*, (*Some xc*, *s*))
    by (*iprover intro*: *evaln.Abrupt*)
  then show *?case* **..**
next
  case *Skip*
  show *?case* **by** (*blast intro*: *evaln.Skip*)
next
  case (*Expr s0 e v s1*)
  then obtain *n* **where**
    G⊢*Norm s0* −*e*−≻*v*−*n*→ *s1*
    by (*iprover*)
  then have *G*⊢*Norm s0* −*Expr e*−*n*→ *s1*
    by (*rule evaln.Expr*)
  then show *?case* **..**
next
  case (*Lab s0 c s1 l*)
  then obtain *n* **where**
    G⊢*Norm s0* −*c*−*n*→ *s1*
    by (*iprover*)
  then have *G*⊢*Norm s0* −*l*· *c*−*n*→ *abupd* (*absorb l*) *s1*
    by (*rule evaln.Lab*)
  then show *?case* **..**
next
  case (*Comp s0 c1 s1 c2 s2*)
  then obtain *n1 n2* **where**
    G⊢*Norm s0* −*c1*−*n1*→ *s1*
    G⊢*s1* −*c2*−*n2*→ *s2*
    by (*iprover*)
  then have *G*⊢*Norm s0* −*c1*;; *c2*−*max n1 n2*→ *s2*
    by (*blast intro*: *evaln.Comp dest*: *evaln-max2*)
  then show *?case* **..**
next
  case (*If s0 e b s1 c1 c2 s2*)
  then obtain *n1 n2* **where**
    G⊢*Norm s0* −*e*−≻*b*−*n1*→ *s1*
    G⊢*s1* −(*if the-Bool b then c1 else c2*)−*n2*→ *s2*
    by (*iprover*)
  then have *G*⊢*Norm s0* −*If*(*e*) *c1 Else c2*−*max n1 n2*→ *s2*
    by (*blast intro*: *evaln.If dest*: *evaln-max2*)
  then show *?case* **..**
next
  case (*Loop s0 e b s1 c s2 l s3*)
  from *Loop.hyps* **obtain** *n1* **where**
    G⊢*Norm s0* −*e*−≻*b*−*n1*→ *s1*
    by (*iprover*)
  moreover from *Loop.hyps* **obtain** *n2* **where**
    if the-Bool b

```
      then (G⊢s1 −c−n2→ s2 ∧
            G⊢(abupd (absorb (Cont l)) s2)−l· While(e) c−n2→ s3)
      else s3 = s1
    by simp (iprover intro: evaln-nonstrict max.cobounded1 max.cobounded2)
  ultimately
  have G⊢Norm s0 −l· While(e) c−max n1 n2→ s3
    apply −
    apply (rule evaln.Loop)
    apply  (iprover intro: evaln-nonstrict intro: max.cobounded1)
    apply  (auto intro: evaln-nonstrict intro: max.cobounded2)
    done
  then show ?case ..
next
  case (Jmp s j)
  fix n have G⊢Norm s −Jmp j−n→ (Some (Jump j), s)
    by (rule evaln.Jmp)
  then show ?case ..
next
  case (Throw s0 e a s1)
  then obtain n where
    G⊢Norm s0 −e−≻a−n→ s1
    by (iprover)
  then have G⊢Norm s0 −Throw e−n→ abupd (throw a) s1
    by (rule evaln.Throw)
  then show ?case ..
next
  case (Try s0 c1 s1 s2 catchC vn c2 s3)
  from Try.hyps obtain n1 where
    G⊢Norm s0 −c1−n1→ s1
    by (iprover)
  moreover
  note sxalloc = ‹G⊢s1 −sxalloc→ s2›
  moreover
  from Try.hyps obtain n2 where
    if G,s2⊢catch catchC then G⊢new-xcpt-var vn s2 −c2−n2→ s3 else s3 = s2
    by fastforce
  ultimately
  have G⊢Norm s0 −Try c1 Catch(catchC vn) c2−max n1 n2→ s3
    by (auto intro!: evaln.Try max.cobounded1 max.cobounded2)
  then show ?case ..
next
  case (Fin s0 c1 x1 s1 c2 s2 s3)
  from Fin obtain n1 n2 where
    G⊢Norm s0 −c1−n1→ (x1, s1)
    G⊢Norm s1 −c2−n2→ s2
    by iprover
  moreover
  note s3 = ‹s3 = (if ∃ err. x1 = Some (Error err)
                 then (x1, s1)
                 else abupd (abrupt-if (x1 ≠ None) x1) s2)›
  ultimately
  have
    G⊢Norm s0 −c1 Finally c2−max n1 n2→ s3
    by (blast intro: evaln.Fin dest: evaln-max2)
  then show ?case ..
next
  case (Init C c s0 s3 s1 s2)
  note cls = ‹the (class G C) = c›
  moreover from Init.hyps obtain n where
```

if inited C (globs s0) then s3 = Norm s0
else (G⊢Norm (init-class-obj G C s0)
−(if C = Object then Skip else Init (super c))−n→ s1 ∧
G⊢set-lvars Map.empty s1 −init c−n→ s2 ∧
s3 = restore-lvars s1 s2)
by (*auto intro*: *evaln-nonstrict max.cobounded1 max.cobounded2*)
ultimately have *G⊢Norm s0 −Init C−n→ s3*
by (*rule evaln.Init*)
then show *?case* **..**
next
case (*NewC s0 C s1 a s2*)
then obtain *n* **where**
G⊢Norm s0 −Init C−n→ s1
by (*iprover*)
with *NewC*
have *G⊢Norm s0 −NewC C−≻Addr a−n→ s2*
by (*iprover intro*: *evaln.NewC*)
then show *?case* **..**
next
case (*NewA s0 T s1 e i s2 a s3*)
then obtain *n1 n2* **where**
G⊢Norm s0 −init-comp-ty T−n1→ s1
G⊢s1 −e−≻i−n2→ s2
by (*iprover*)
moreover
note ‹*G⊢abupd (check-neg i) s2 −halloc Arr T (the-Intg i)≻a→ s3*›
ultimately
have *G⊢Norm s0 −New T[e]−≻Addr a−max n1 n2→ s3*
by (*blast intro*: *evaln.NewA dest*: *evaln-max2*)
then show *?case* **..**
next
case (*Cast s0 e v s1 s2 castT*)
then obtain *n* **where**
G⊢Norm s0 −e−≻v−n→ s1
by (*iprover*)
moreover
note ‹*s2 = abupd (raise-if (¬ G,snd s1⊢v fits castT) ClassCast) s1*›
ultimately
have *G⊢Norm s0 −Cast castT e−≻v−n→ s2*
by (*rule evaln.Cast*)
then show *?case* **..**
next
case (*Inst s0 e v s1 b T*)
then obtain *n* **where**
G⊢Norm s0 −e−≻v−n→ s1
by (*iprover*)
moreover
note ‹*b = (v ≠ Null ∧ G,snd s1⊢v fits RefT T)*›
ultimately
have *G⊢Norm s0 −e InstOf T−≻Bool b−n→ s1*
by (*rule evaln.Inst*)
then show *?case* **..**
next
case (*Lit s v*)
fix *n* **have** *G⊢Norm s −Lit v−≻v−n→ Norm s*
by (*rule evaln.Lit*)
then show *?case* **..**
next
case (*UnOp s0 e v s1 unop*)

```
  then obtain n where
    G⊢Norm s0 −e−≻v−n→ s1
    by (iprover)
  hence G⊢Norm s0 −UnOp unop e−≻eval-unop unop v−n→ s1
    by (rule evaln.UnOp)
  then show ?case ..
next
  case (BinOp s0 e1 v1 s1 binop e2 v2 s2)
  then obtain n1 n2 where
    G⊢Norm s0 −e1−≻v1−n1→ s1
    G⊢s1 −(if need-second-arg binop v1 then In1l e2
               else In1r Skip)≻−n2→ (In1 v2, s2)
    by (iprover)
  hence G⊢Norm s0 −BinOp binop e1 e2−≻(eval-binop binop v1 v2)−max n1 n2
          → s2
    by (blast intro!: evaln.BinOp dest: evaln-max2)
  then show ?case ..
next
  case (Super s )
  fix n have G⊢Norm s −Super−≻val-this s−n→ Norm s
    by (rule evaln.Super)
  then show ?case ..
next
  case (Acc s0 va v f s1)
  then obtain n where
    G⊢Norm s0 −va=≻(v, f)−n→ s1
    by (iprover)
  then
  have G⊢Norm s0 −Acc va−≻v−n→ s1
    by (rule evaln.Acc)
  then show ?case ..
next
  case (Ass s0 var w f s1 e v s2)
  then obtain n1 n2 where
    G⊢Norm s0 −var=≻(w, f)−n1→ s1
    G⊢s1 −e−≻v−n2→ s2
    by (iprover)
  then
  have G⊢Norm s0 −var:=e−≻v−max n1 n2→ assign f v s2
    by (blast intro: evaln.Ass dest: evaln-max2)
  then show ?case ..
next
  case (Cond s0 e0 b s1 e1 e2 v s2)
  then obtain n1 n2 where
    G⊢Norm s0 −e0−≻b−n1→ s1
    G⊢s1 −(if the-Bool b then e1 else e2)−≻v−n2→ s2
    by (iprover)
  then
  have G⊢Norm s0 −e0 ? e1 : e2−≻v−max n1 n2→ s2
    by (blast intro: evaln.Cond dest: evaln-max2)
  then show ?case ..
next
  case (Call s0 e a' s1 args vs s2 invDeclC mode statT mn pTs' s3 s3' accC' v s4)
  then obtain n1 n2 where
    G⊢Norm s0 −e−≻a'−n1→ s1
    G⊢s1 −args≐≻vs−n2→ s2
    by iprover
  moreover
  note ‹invDeclC = invocation-declclass G mode (store s2) a' statT
```

(|name=mn,parTs=pTs′|)›
moreover
note ‹s3 = init-lvars G invDeclC (|name=mn,parTs=pTs′|) mode a′ vs s2›
moreover
note ‹s3′=check-method-access G accC′ statT mode (|name=mn,parTs=pTs′|) a′ s3›
moreover
from *Call.hyps*
obtain *m* **where**
G⊢s3′ −Methd invDeclC (|name=mn, parTs=pTs′|)−≻v−m→ s4
by *iprover*
ultimately
have G⊢Norm s0 −{accC′,statT,mode}e·mn({pTs′}args)−≻v−max n1 (max n2 m)→
(set-lvars (locals (store s2))) s4
by (*auto intro*!: *evaln.Call max.cobounded1 le-max3I1 le-max3I2*)
thus *?case* **..**
next
case (*Methd s0 D sig v s1*)
then obtain *n* **where**
G⊢Norm s0 −body G D sig−≻v−n→ s1
by *iprover*
then have G⊢Norm s0 −Methd D sig−≻v−Suc n→ s1
by (*rule evaln.Methd*)
then show *?case* **..**
next
case (*Body s0 D s1 c s2 s3*)
from *Body.hyps* **obtain** *n1 n2* **where**
evaln-init: G⊢Norm s0 −Init D−n1→ s1 **and**
evaln-c: G⊢s1 −c−n2→ s2
by (*iprover*)
moreover
note ‹s3 = (if ∃ l. fst s2 = Some (Jump (Break l)) ∨
fst s2 = Some (Jump (Cont l))
then abupd (λx. Some (Error CrossMethodJump)) s2
else s2)›
ultimately
have
G⊢Norm s0 −Body D c−≻the (locals (store s2) Result)−max n1 n2
→ abupd (absorb Ret) s3
by (*iprover intro*: *evaln.Body dest*: *evaln-max2*)
then show *?case* **..**
next
case (*LVar s vn*)
obtain *n* **where**
G⊢Norm s −LVar vn=≻lvar vn s−n→ Norm s
by (*iprover intro*: *evaln.LVar*)
then show *?case* **..**
next
case (*FVar s0 statDeclC s1 e a s2 v s2′ stat fn s3 accC*)
then obtain *n1 n2* **where**
G⊢Norm s0 −Init statDeclC−n1→ s1
G⊢s1 −e−≻a−n2→ s2
by *iprover*
moreover
note ‹s3 = check-field-access G accC statDeclC fn stat a s2′›
and ‹(v, s2′) = fvar statDeclC stat fn a s2›
ultimately
have G⊢Norm s0 −{accC,statDeclC,stat}e..fn=≻v−max n1 n2→ s3
by (*iprover intro*: *evaln.FVar dest*: *evaln-max2*)
then show *?case* **..**

```
next
  case (AVar s0 e1 a s1 e2 i s2 v s2')
  then obtain n1 n2 where
    G⊢Norm s0 −e1−≻a−n1→ s1
    G⊢s1 −e2−≻i−n2→ s2
    by iprover
  moreover
  note ‹(v, s2') = avar G i a s2›
  ultimately
  have G⊢Norm s0 −e1.[e2]=≻v−max n1 n2→ s2'
    by (blast intro!: evaln.AVar dest: evaln-max2)
  then show ?case ..
next
  case (Nil s0)
  show ?case by (iprover intro: evaln.Nil)
next
  case (Cons s0 e v s1 es vs s2)
  then obtain n1 n2 where
    G⊢Norm s0 −e−≻v−n1→ s1
    G⊢s1 −es≐≻vs−n2→ s2
    by iprover
  then
  have G⊢Norm s0 −e # es≐≻v # vs−max n1 n2→ s2
    by (blast intro!: evaln.Cons dest: evaln-max2)
  then show ?case ..
qed

end
```

Chapter 21

Trans

theory *Trans* **imports** *Evaln* **begin**

definition
  groundVar :: *var* ⇒ *bool* **where**
  groundVar v ⟷ (*case v of*
      LVar ln ⇒ *True*
    | {*accC*,*statDeclC*,*stat*}*e*..*fn* ⇒ ∃ *a*. *e*=*Lit a*
    | *e1*.[*e2*] ⇒ ∃ *a i*. *e1*= *Lit a* ∧ *e2* = *Lit i*
    | *InsInitV c v* ⇒ *False*)

lemma *groundVar-cases*:
  assumes *ground*: *groundVar v*
  obtains (*LVar*) *ln* **where** *v*=*LVar ln*
    | (*FVar*) *accC statDeclC stat a fn* **where** *v*={*accC*,*statDeclC*,*stat*}(*Lit a*)..*fn*
    | (*AVar*) *a i* **where** *v*=(*Lit a*).[*Lit i*]
  using *ground LVar FVar AVar*
  by (*cases v*) (*auto simp add*: *groundVar-def*)

definition
  groundExprs :: *expr list* ⇒ *bool*
  where *groundExprs es* ⟷ (∀ *e* ∈ *set es*. ∃ *v*. *e* = *Lit v*)

primrec *the-val*:: *expr* ⇒ *val*
  where *the-val* (*Lit v*) = *v*

primrec *the-var*:: *prog* ⇒ *state* ⇒ *var* ⇒ (*vvar* × *state*) **where**
  the-var G s (*LVar ln*) = (*lvar ln* (*store s*),*s*)
| *the-var-FVar-def*: *the-var G s* ({*accC*,*statDeclC*,*stat*}*a*..*fn*) =*fvar statDeclC stat fn* (*the-val a*) *s*
| *the-var-AVar-def*: *the-var G s*(*a*.[*i*])   =*avar G* (*the-val i*) (*the-val a*) *s*

lemma *the-var-FVar-simp*[*simp*]:
the-var G s ({*accC*,*statDeclC*,*stat*}(*Lit a*)..*fn*) = *fvar statDeclC stat fn a s*
by (*simp*)
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*
by (*simp*)
declare *the-var-AVar-def* [*simp del*]

abbreviation
Ref :: *loc* ⇒ *expr*
where *Ref a* == *Lit* (*Addr a*)

abbreviation
SKIP :: *expr*
where *SKIP* == *Lit Unit*

inductive
step :: [*prog*,*term* × *state*,*term* × *state*] ⇒ *bool* (-⊢- ↦*1* -[*61*,*82*,*82*] *81*)
for *G* :: *prog*
where

Abrupt: ⟦∀ *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⊢(⟨*Cast T e*⟩,*None*,*s*) ↦*1* (⟨*Cast T e*′⟩,*s*′)
| *Cast*:
⟦*s*′ = *abupd* (*raise-if* (¬*G*,*s*⊢*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*: $G\vdash(\langle Lit\ b?\ e1{:}e2\rangle,Norm\ s) \mapsto 1\ (\langle if\ the\text{-}Bool\ b\ then\ e1\ else\ e2\rangle,Norm\ s)$

$| \ \mathit{CallTarget}$: $[\![G \vdash (\langle e \rangle, \mathit{Norm}\ s) \mapsto 1\ (\langle e' \rangle, s')]\!]$
$\Longrightarrow$
$G \vdash (\langle \{accC, statT, mode\} e \cdot mn(\{pTs\} args) \rangle, \mathit{Norm}\ s)$
$\mapsto 1\ (\langle \{accC, statT, mode\} e' \cdot mn(\{pTs\} args) \rangle, s')$

$| \ \mathit{CallArgs}$: $[\![G \vdash (\langle args \rangle, \mathit{Norm}\ s) \mapsto 1\ (\langle args' \rangle, s')]\!]$
$\Longrightarrow$
$G \vdash (\langle \{accC, statT, mode\} \mathit{Lit}\ a \cdot mn(\{pTs\} args) \rangle, \mathit{Norm}\ s)$
$\mapsto 1\ (\langle \{accC, statT, mode\} \mathit{Lit}\ a \cdot mn(\{pTs\} args') \rangle, s')$

$| \ \mathit{Call}$: $[\![\mathit{groundExprs}\ args;\ vs = \mathit{map}\ \mathit{the\text{-}val}\ args;$
$D = \mathit{invocation\text{-}declclass}\ G\ mode\ s\ a\ statT\ (\!|name{=}mn, parTs{=}pTs|\!);$
$s' {=} \mathit{init\text{-}lvars}\ G\ D\ (\!|name{=}mn, parTs{=}pTs|\!)\ mode\ a'\ vs\ (\mathit{Norm}\ s)]\!]$
$\Longrightarrow$
$G \vdash (\langle \{accC, statT, mode\} \mathit{Lit}\ a \cdot mn(\{pTs\} args) \rangle, \mathit{Norm}\ s)$
$\mapsto 1\ (\langle \mathit{Callee}\ (\mathit{locals}\ s)\ (\mathit{Methd}\ D\ (\!|name{=}mn, parTs{=}pTs|\!)) \rangle, s')$

$| \ \mathit{Callee}$: $[\![G \vdash (\langle e \rangle, \mathit{Norm}\ s) \mapsto 1\ (\langle e'{::}expr \rangle, s')]\!]$
$\Longrightarrow$
$G \vdash (\langle \mathit{Callee}\ \mathit{lcls\text{-}caller}\ e \rangle, \mathit{Norm}\ s) \mapsto 1\ (\langle e' \rangle, s')$

$| \ \mathit{CalleeRet}$: $G \vdash (\langle \mathit{Callee}\ \mathit{lcls\text{-}caller}\ (\mathit{Lit}\ v) \rangle, \mathit{Norm}\ s)$
$\mapsto 1\ (\langle \mathit{Lit}\ v \rangle, (\mathit{set\text{-}lvars}\ \mathit{lcls\text{-}caller}\ (\mathit{Norm}\ s)))$

$| \ \mathit{Methd}$: $G \vdash (\langle \mathit{Methd}\ D\ sig \rangle, \mathit{Norm}\ s) \mapsto 1\ (\langle \mathit{body}\ G\ D\ sig \rangle, \mathit{Norm}\ s)$

$| \ \mathit{Body}$: $G \vdash (\langle \mathit{Body}\ D\ c \rangle, \mathit{Norm}\ s) \mapsto 1\ (\langle \mathit{InsInitE}\ (\mathit{Init}\ D)\ (\mathit{Body}\ D\ c) \rangle, \mathit{Norm}\ s)$

$| \ \mathit{InsInitBody}$:
$[\![G \vdash (\langle c \rangle, \mathit{Norm}\ s) \mapsto 1\ (\langle c' \rangle, s')]\!]$
$\Longrightarrow$
$G \vdash (\langle \mathit{InsInitE}\ \mathit{Skip}\ (\mathit{Body}\ D\ c) \rangle, \mathit{Norm}\ s) \mapsto 1 (\langle \mathit{InsInitE}\ \mathit{Skip}\ (\mathit{Body}\ D\ c') \rangle, s')$

$| \ \mathit{InsInitBodyRet}$:
$G \vdash (\langle \mathit{InsInitE}\ \mathit{Skip}\ (\mathit{Body}\ D\ \mathit{Skip}) \rangle, \mathit{Norm}\ s)$
$\mapsto 1\ (\langle \mathit{Lit}\ (\mathit{the}\ ((\mathit{locals}\ s)\ \mathit{Result})) \rangle, \mathit{abupd}\ (\mathit{absorb}\ \mathit{Ret})\ (\mathit{Norm}\ s))$

$| \ \mathit{FVar}$: $[\![\neg\ \mathit{inited}\ \mathit{statDeclC}\ (\mathit{globs}\ s)]\!]$
$\Longrightarrow$
$G \vdash (\langle \{accC, statDeclC, stat\} e..fn \rangle, \mathit{Norm}\ s)$
$\mapsto 1\ (\langle \mathit{InsInitV}\ (\mathit{Init}\ \mathit{statDeclC})\ (\{accC, statDeclC, stat\} e..fn) \rangle, \mathit{Norm}\ s)$

$| \ \mathit{InsInitFVarE}$:
$[\![G \vdash (\langle e \rangle, \mathit{Norm}\ s) \mapsto 1\ (\langle e' \rangle, s')]\!]$
$\Longrightarrow$
$G \vdash (\langle \mathit{InsInitV}\ \mathit{Skip}\ (\{accC, statDeclC, stat\} e..fn) \rangle, \mathit{Norm}\ s)$
$\mapsto 1\ (\langle \mathit{InsInitV}\ \mathit{Skip}\ (\{accC, statDeclC, stat\} e'..fn) \rangle, s')$

$| \ \mathit{InsInitFVar}$:
$G \vdash (\langle \mathit{InsInitV}\ \mathit{Skip}\ (\{accC, statDeclC, stat\} \mathit{Lit}\ a..fn) \rangle, \mathit{Norm}\ s)$
$\mapsto 1\ (\langle \{accC, statDeclC, stat\} \mathit{Lit}\ a..fn \rangle, \mathit{Norm}\ s)$

— 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*.

$| \ \mathit{AVarE1}$: $[\![G \vdash (\langle e1 \rangle, \mathit{Norm}\ s) \mapsto 1\ (\langle e1' \rangle, s')]\!]$
$\Longrightarrow$
$G \vdash (\langle e1.[e2] \rangle, \mathit{Norm}\ s) \mapsto 1\ (\langle e1'.[e2] \rangle, s')$

| *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 $G\vdash p \mapsto n\ p' \equiv (p,p') \in \{(x,\ y).\ step\ G\ x\ y\}^{\frown\frown}n$

abbreviation
steptr:: [*prog*,*term* × *state*,*term* × *state*] ⇒ *bool* (-⊢- ↦∗ -[*61*,*82*,*82*] *81*)
where $G\vdash p \mapsto * \ p' \equiv (p,p') \in \{(x,\ y).\ step\ G\ x\ y\}^*$

end

Chapter 22

AxSem

1 Axiomatic semantics of Java expressions and statements (see also Eval.thy)

theory *AxSem* **imports** *Evaln TypeSafe* **begin**

design issues:

- a strong version of validity for triples with premises, namely one that takes the recursive depth needed to complete execution, enables correctness proof
- auxiliary variables are handled first-class (-> Thomas Kleymann)
- expressions not flattened to elementary assignments (as usual for axiomatic semantics) but treated first-class => explicit result value handling
- intermediate values not on triple, but on assertion level (with result entry)
- multiple results with semantical substitution mechnism not requiring a stack
- because of dynamic method binding, terms need to be dependent on state. this is also useful for conditional expressions and statements
- result values in triples exactly as in eval relation (also for xcpt states)
- validity: additional assumption of state conformance and well-typedness, which is required for soundness and thus rule hazard required of completeness

restrictions:

- all triples in a derivation are of the same type (due to weak polymorphism)

type-synonym *res* = *vals* — result entry

abbreviation (*input*)
  Val **where** *Val x* == *In1 x*

abbreviation (*input*)
  Var **where** *Var x* == *In2 x*

abbreviation (*input*)
  Vals **where** *Vals x* == *In3 x*

syntax
  -Val :: [*pttrn*] => *pttrn* (*Val*:- [*951*] *950*)
  -Var :: [*pttrn*] => *pttrn* (*Var*:- [*951*] *950*)

-Vals :: [*pttrn*] => *pttrn* (*Vals*:- [*951*] *950*)

translations
λ*Val*:*v* . *b* == ($\lambda v.$ *b*) $\circ$ *CONST the-In1*
λ*Var*:*v* . *b* == ($\lambda v.$ *b*) $\circ$ *CONST the-In2*
λ*Vals*:*v*. *b* == ($\lambda v.$ *b*) $\circ$ *CONST the-In3*

— relation on result values, state and auxiliary variables
type-synonym *'a assn* = *res* $\Rightarrow$ *state* $\Rightarrow$ *'a* $\Rightarrow$ *bool*
translations
(*type*) *'a assn* <= (*type*) *vals* $\Rightarrow$ *state* $\Rightarrow$ *'a* $\Rightarrow$ *bool*

definition
assn-imp :: *'a assn* $\Rightarrow$ *'a assn* $\Rightarrow$ *bool* (**infixr** $\Rrightarrow$ *25*)
where $(P \Rrightarrow Q) = (\forall Y\ s\ Z.\ P\ Y\ s\ Z \longrightarrow Q\ Y\ s\ Z)$

lemma *assn-imp-def2* [*iff*]: $(P \Rrightarrow Q) = (\forall Y\ s\ Z.\ P\ Y\ s\ Z \longrightarrow Q\ Y\ s\ Z)$
apply (*unfold assn-imp-def*)
apply (*rule HOL.refl*)
done

assertion transformers

2 peek-and

definition
peek-and :: *'a assn* $\Rightarrow$ (*state* $\Rightarrow$ *bool*) $\Rightarrow$ *'a assn* (**infixl** $\wedge.$ *13*)
where $(P \wedge.\ p) = (\lambda Y\ s\ Z.\ P\ Y\ s\ Z \wedge p\ s)$

lemma *peek-and-def2* [*simp*]: *peek-and P p Y s* = $(\lambda Z.\ (P\ Y\ s\ Z \wedge p\ s))$
apply (*unfold peek-and-def*)
apply (*simp* (*no-asm*))
done

lemma *peek-and-Not* [*simp*]: $(P \wedge.\ (\lambda s.\ \neg\ f\ s)) = (P \wedge.\ Not \circ f)$
apply (*rule ext*)
apply (*rule ext*)
apply (*simp* (*no-asm*))
done

lemma *peek-and-and* [*simp*]: *peek-and* (*peek-and P p*) *p* = *peek-and P p*
apply (*unfold peek-and-def*)
apply (*simp* (*no-asm*))
done

lemma *peek-and-commut*: $(P \wedge.\ p \wedge.\ q) = (P \wedge.\ q \wedge.\ p)$
apply (*rule ext*)
apply (*rule ext*)
apply (*rule ext*)
apply *auto*
done

abbreviation
Normal :: *'a assn* $\Rightarrow$ *'a assn*

where *Normal P* == *P* ∧. *normal*

lemma *peek-and-Normal* [*simp*]: *peek-and* (*Normal P*) *p* = *Normal* (*peek-and P p*)
apply (*rule ext*)
apply (*rule ext*)
apply (*rule ext*)
apply *auto*
done

3 assn-supd

definition
assn-supd :: *'a assn* ⇒ (*state* ⇒ *state*) ⇒ *'a assn* (**infixl** ;. *13*)
where (*P* ;. *f*) = (λ*Y s' Z*. ∃ *s*. *P Y s Z* ∧ *s'* = *f s*)

lemma *assn-supd-def2* [*simp*]: *assn-supd P f Y s' Z* = (∃ *s*. *P Y s Z* ∧ *s'* = *f s*)
apply (*unfold assn-supd-def*)
apply (*simp* (*no-asm*))
done

4 supd-assn

definition
supd-assn :: (*state* ⇒ *state*) ⇒ *'a assn* ⇒ *'a assn* (**infixr** .; *13*)
where (*f* .; *P*) = (λ*Y s*. *P Y* (*f s*))

lemma *supd-assn-def2* [*simp*]: (*f* .; *P*) *Y s* = *P Y* (*f s*)
apply (*unfold supd-assn-def*)
apply (*simp* (*no-asm*))
done

lemma *supd-assn-supdD* [*elim*]: ((*f* .; *Q*) ;. *f*) *Y s Z* ⟹ *Q Y s Z*
apply *auto*
done

lemma *supd-assn-supdI* [*elim*]: *Q Y s Z* ⟹ (*f* .; (*Q* ;. *f*)) *Y s Z*
apply (*auto simp del*: *split-paired-Ex*)
done

5 subst-res

definition
subst-res :: *'a assn* ⇒ *res* ⇒ *'a assn* (-←- [*60*,*61*] *60*)
where *P*←*w* = (λ*Y*. *P w*)

lemma *subst-res-def2* [*simp*]: (*P*←*w*) *Y* = *P w*
apply (*unfold subst-res-def*)
apply (*simp* (*no-asm*))
done

lemma *subst-subst-res* [*simp*]: *P*←*w*←*v* = *P*←*w*

apply (*rule ext*)
apply (*simp* (*no-asm*))
done

lemma *peek-and-subst-res* [*simp*]: (*P* ∧. *p*)←*w* = (*P*←*w* ∧. *p*)
apply (*rule ext*)
apply (*rule ext*)
apply (*simp* (*no-asm*))
done

6 subst-Bool

definition
subst-Bool :: ′*a assn* ⇒ *bool* ⇒ ′*a assn* (-←=- [*60*,*61*] *60*)
where *P*←=*b* = (λ*Y s Z*. ∃ *v*. *P* (*Val v*) *s Z* ∧ (*normal s* ⟶ *the-Bool v*=*b*))

lemma *subst-Bool-def2* [*simp*]:
(*P*←=*b*) *Y s Z* = (∃ *v*. *P* (*Val v*) *s Z* ∧ (*normal s* ⟶ *the-Bool v*=*b*))
apply (*unfold subst-Bool-def*)
apply (*simp* (*no-asm*))
done

lemma *subst-Bool-the-BoolI*: *P* (*Val b*) *s Z* ⟹ (*P*←=*the-Bool b*) *Y s Z*
apply *auto*
done

7 peek-res

definition
peek-res :: (*res* ⇒ ′*a assn*) ⇒ ′*a assn*
where *peek-res Pf* = (λ*Y*. *Pf Y Y*)

syntax
-peek-res :: *pttrn* ⇒ ′*a assn* ⇒ ′*a assn* (λ-:. - [*0*,*3*] *3*)
translations
λ*w*:. *P* == *CONST peek-res* (λ*w*. *P*)

lemma *peek-res-def2* [*simp*]: *peek-res P Y* = *P Y Y*
apply (*unfold peek-res-def*)
apply (*simp* (*no-asm*))
done

lemma *peek-res-subst-res* [*simp*]: *peek-res P*←*w* = *P w*←*w*
apply (*rule ext*)
apply (*simp* (*no-asm*))
done

lemma *peek-subst-res-allI*:
(⋀*a*. *T a* (*P* (*f a*)←*f a*)) ⟹ ∀ *a*. *T a* (*peek-res P*←*f a*)
apply (*rule allI*)
apply (*simp* (*no-asm*))
apply *fast*

done

8 ign-res

definition
ign-res :: *'a assn* ⇒ *'a assn* (-↓ [*1000*] *1000*)
where $P\downarrow = (\lambda Y\ s\ Z.\ \exists\ Y.\ P\ Y\ s\ Z)$

lemma *ign-res-def2* [*simp*]: $P\downarrow\ Y\ s\ Z = (\exists\ Y.\ P\ Y\ s\ Z)$
apply (*unfold ign-res-def*)
apply (*simp* (*no-asm*))
done

lemma *ign-ign-res* [*simp*]: $P\downarrow\downarrow = P\downarrow$
apply (*rule ext*)
apply (*rule ext*)
apply (*rule ext*)
apply (*simp* (*no-asm*))
done

lemma *ign-subst-res* [*simp*]: $P\downarrow\leftarrow w = P\downarrow$
apply (*rule ext*)
apply (*rule ext*)
apply (*rule ext*)
apply (*simp* (*no-asm*))
done

lemma *peek-and-ign-res* [*simp*]: $(P \wedge.\ p)\downarrow = (P\downarrow \wedge.\ p)$
apply (*rule ext*)
apply (*rule ext*)
apply (*rule ext*)
apply (*simp* (*no-asm*))
done

9 peek-st

definition
peek-st :: (*st* ⇒ *'a assn*) ⇒ *'a assn*
where *peek-st P* = (λ*Y s*. *P* (*store s*) *Y s*)

syntax
-peek-st :: *pttrn* ⇒ *'a assn* ⇒ *'a assn* (λ-.. - [*0*,*3*] *3*)
translations
λ*s*.. *P* == *CONST peek-st* (λ*s*. *P*)

lemma *peek-st-def2* [*simp*]: (λ*s*.. *Pf s*) *Y s* = *Pf* (*store s*) *Y s*
apply (*unfold peek-st-def*)
apply (*simp* (*no-asm*))
done

lemma *peek-st-triv* [*simp*]: (λ*s*.. *P*) = *P*
apply (*rule ext*)
apply (*rule ext*)

apply (*simp* (*no-asm*))
done

lemma *peek-st-st* [*simp*]: $(\lambda s.. \lambda s'.. P\ s\ s') = (\lambda s.. P\ s\ s)$
apply (*rule ext*)
apply (*rule ext*)
apply (*simp* (*no-asm*))
done

lemma *peek-st-split* [*simp*]: $(\lambda s.. \lambda Y\ s'. P\ s\ Y\ s') = (\lambda Y\ s.\ P\ (store\ s)\ Y\ s)$
apply (*rule ext*)
apply (*rule ext*)
apply (*simp* (*no-asm*))
done

lemma *peek-st-subst-res* [*simp*]: $(\lambda s.. P\ s)\leftarrow w = (\lambda s.. P\ s\leftarrow w)$
apply (*rule ext*)
apply (*simp* (*no-asm*))
done

lemma *peek-st-Normal* [*simp*]: $(\lambda s..(Normal\ (P\ s))) = Normal\ (\lambda s.. P\ s)$
apply (*rule ext*)
apply (*rule ext*)
apply (*simp* (*no-asm*))
done

10 ign-res-eq

definition
ign-res-eq :: $'a\ assn \Rightarrow res \Rightarrow 'a\ assn$ (-↓=- [*60*,*61*] *60*)
where $P{\downarrow}{=}w \equiv (\lambda Y{:}.\ P{\downarrow} \wedge.\ (\lambda s.\ Y{=}w))$

lemma *ign-res-eq-def2* [*simp*]: $(P{\downarrow}{=}w)\ Y\ s\ Z = ((\exists\ Y.\ P\ Y\ s\ Z) \wedge Y{=}w)$
apply (*unfold ign-res-eq-def*)
apply *auto*
done

lemma *ign-ign-res-eq* [*simp*]: $(P{\downarrow}{=}w){\downarrow} = P{\downarrow}$
apply (*rule ext*)
apply (*rule ext*)
apply (*rule ext*)
apply (*simp* (*no-asm*))
done

lemma *ign-res-eq-subst-res*: $P{\downarrow}{=}w{\leftarrow}w = P{\downarrow}$
apply (*rule ext*)
apply (*rule ext*)
apply (*rule ext*)
apply (*simp* (*no-asm*))
done

lemma *subst-Bool-ign-res-eq*: $((P\leftarrow=b)\downarrow=x)\ Y\ s\ Z = ((P\leftarrow=b)\ Y\ s\ Z\ \wedge\ Y=x)$
apply (*simp* (*no-asm*))
done

11 RefVar

definition
RefVar :: (*state* ⇒ *vvar* × *state*) ⇒ ′*a assn* ⇒ ′*a assn* (**infixr** ..; *13*)
where (*vf* ..; *P*) = (λ*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*))
apply (*unfold RefVar-def Let-def*)
apply (*simp* (*no-asm*) *add*: *split-beta*)
done

12 allocation

definition
Alloc :: *prog* ⇒ *obj-tag* ⇒ ′*a assn* ⇒ ′*a assn*
where *Alloc G otag P* = (λ*Y s Z*. ∀ *s*′ *a*. *G*⊢*s* −*halloc otag*≻*a*→ *s*′⟶ *P* (*Val* (*Addr a*)) *s*′ *Z*)

definition
SXAlloc :: *prog* ⇒ ′*a assn* ⇒ ′*a assn*
where *SXAlloc G P* = (λ*Y s Z*. ∀ *s*′. *G*⊢*s* −*sxalloc*→ *s*′ ⟶ *P Y s*′ *Z*)

lemma *Alloc-def2* [*simp*]: *Alloc G otag P Y s Z* =
(∀ *s*′ *a*. *G*⊢*s* −*halloc otag*≻*a*→ *s*′⟶ *P* (*Val* (*Addr a*)) *s*′ *Z*)
apply (*unfold Alloc-def*)
apply (*simp* (*no-asm*))
done

lemma *SXAlloc-def2* [*simp*]:
SXAlloc G P Y s Z = (∀ *s*′. *G*⊢*s* −*sxalloc*→ *s*′ ⟶ *P Y s*′ *Z*)
apply (*unfold SXAlloc-def*)
apply (*simp* (*no-asm*))
done

validity

definition
type-ok :: *prog* ⇒ *term* ⇒ *state* ⇒ *bool* **where**
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*)
type-synonym ′*a triples* = ′*a triple set*

abbreviation
var-triple :: [′*a assn*, *var* ,′*a assn*] ⇒ ′*a triple*

({(1-)}/ -=≻/ {(1-)} [3,80,3] 75)
where {P} e=≻ {Q} == {P} In2 e≻ {Q}

abbreviation
expr-triple :: ['a assn, expr ,'a assn] ⇒ 'a triple
({(1-)}/ -−≻/ {(1-)} [3,80,3] 75)
where {P} e−≻ {Q} == {P} In1l e≻ {Q}

abbreviation
exprs-triple :: ['a assn, expr list ,'a assn] ⇒ 'a triple
({(1-)}/ -≐≻/ {(1-)} [3,65,3] 75)
where {P} e≐≻ {Q} == {P} In3 e≻ {Q}

abbreviation
stmt-triple :: ['a assn, stmt, 'a assn] ⇒ 'a triple
({(1-)}/ .-./ {(1-)} [3,65,3] 75)
where {P} .c. {Q} == {P} In1r c≻ {Q}

notation (*ASCII*)
triple ({(1-)}/ ->/ {(1-)} [3,65,3]75) **and**
var-triple ({(1-)}/ -=>/ {(1-)} [3,80,3] 75) **and**
expr-triple ({(1-)}/ -->/ {(1-)} [3,80,3] 75) **and**
exprs-triple ({(1-)}/ -#>/ {(1-)} [3,65,3] 75)

lemma *inj-triple*: *inj* (λ(P,t,Q). {P} t≻ {Q})
apply (*rule inj-onI*)
apply *auto*
done

lemma *triple-inj-eq*: ({P} t≻ {Q} = {P'} t'≻ {Q'}) = (P=P' ∧ t=t' ∧ Q=Q')
apply *auto*
done

definition *mtriples* :: ('c ⇒ 'sig ⇒ 'a assn) ⇒ ('c ⇒ 'sig ⇒ expr) ⇒
('c ⇒ 'sig ⇒ 'a assn) ⇒ ('c × 'sig) set ⇒ 'a triples ({{(1-)}/ -−≻/ {(1-)} | -}[3,65,3,65]75)
where
{{P} tf−≻ {Q} | ms} = (λ(C,sig). {Normal(P C sig)} tf C sig−≻ {Q C sig})‘ms

definition
triple-valid :: prog ⇒ nat ⇒ 'a triple ⇒ bool (-⊨-:- [61,0, 58] 57)
where
G⊨n:t =
(case t of {P} t≻ {Q} ⇒
∀ Y s Z. P Y s Z ⟶ type-ok G t s ⟶
(∀ Y' s'. G⊢s −t≻−n→ (Y',s') ⟶ Q Y' s' Z))

abbreviation
triples-valid:: prog ⇒ nat ⇒ 'a triples ⇒ bool (-|⊨-:- [61,0, 58] 57)
where G|⊨n:ts == Ball ts (triple-valid G n)

notation (*ASCII*)
triples-valid (-||=-:- [61,0, 58] 57)

definition
ax-valids :: prog ⇒ 'b triples ⇒ 'a triples ⇒ bool (-,-|⊨- [61,58,58] 57)
where (G,A|⊨ts) = (∀ n. G|⊨n:A ⟶ G|⊨n:ts)

abbreviation
ax-valid :: prog ⇒ 'b triples ⇒ 'a triple ⇒ bool (-,-⊨- [61,58,58] 57)
where G,A ⊨t == G,A|⊨{t}

notation (ASCII)
ax-valid (-,-|=- [61,58,58] 57)

lemma triple-valid-def2: G⊨n:{P} t≻ {Q} =
(∀ Y s Z. P Y s Z
⟶ (∃ L. (normal s ⟶ (∃ C T A. ⦇prg=G,cls=C,lcl=L⦈⊢t::T ∧
⦇prg=G,cls=C,lcl=L⦈⊢dom (locals (store s))»t»A)) ∧
s::≼(G,L))
⟶ (∀ Y' s'. G⊢s −t≻−n→ (Y',s')⟶ Q Y' s' Z))
apply (unfold triple-valid-def type-ok-def)
apply (simp (no-asm))
done

declare split-paired-All [simp del] split-paired-Ex [simp del]
declare if-split [split del] if-split-asm [split del]
option.split [split del] option.split-asm [split del]
setup ‹map-theory-simpset (fn ctxt => ctxt delloop split-all-tac)›
setup ‹map-theory-claset (fn ctxt => ctxt delSWrapper split-all-tac)›

inductive
ax-derivs :: prog ⇒ 'a triples ⇒ 'a triples ⇒ bool (-,-|⊢- [61,58,58] 57)
and ax-deriv :: prog ⇒ 'a triples ⇒ 'a triple ⇒ bool (-,-⊢- [61,58,58] 57)
for G :: prog
where

G,A ⊢t ≡ G,A|⊢{t}

| empty: G,A|⊢{}
| insert:⟦G,A⊢t; G,A|⊢ts⟧ ⟹
G,A|⊢insert t ts

| asm: ts⊆A ⟹ G,A|⊢ts

| weaken:⟦G,A|⊢ts'; ts ⊆ ts'⟧ ⟹ G,A|⊢ts

| conseq:∀ Y s Z . P Y s Z ⟶ (∃ P' Q'. G,A⊢{P'} t≻ {Q'} ∧ (∀ Y' s'.
(∀ Y Z'. P' Y s Z' ⟶ Q' Y' s' Z') ⟶
Q Y' s' Z))
⟹ G,A⊢{P } t≻ {Q }

| hazard:G,A⊢{P ∧. Not ∘ type-ok G t} t≻ {Q}

| Abrupt: G,A⊢{P←(undefined3 t) ∧. Not ∘ normal} t≻ {P}

— variables
| LVar: G,A⊢{Normal (λs.. P←Var (lvar vn s))} LVar vn=≻ {P}

| FVar: ⟦G,A⊢{Normal P} .Init C. {Q};
G,A⊢{Q} e−≻ {λVal:a:. fvar C stat fn a ..; R}⟧ ⟹
G,A⊢{Normal P} {accC,C,stat}e..fn=≻ {R}

| *AVar*: ⟦G,A⊢{Normal P} e1−≻ {Q};
∀ a. G,A⊢{Q← Val a} e2−≻ {λVal:i:. avar G i a ..; R}⟧ ⟹
G,A⊢{Normal P} e1.[e2]=≻ {R}
— expressions

| *NewC*: ⟦G,A⊢{Normal P} .Init C. {Alloc G (CInst C) Q}⟧ ⟹
G,A⊢{Normal P} NewC C−≻ {Q}

| *NewA*: ⟦G,A⊢{Normal P} .init-comp-ty T. {Q}; G,A⊢{Q} e−≻
{λVal:i:. abupd (check-neg i) .; Alloc G (Arr T (the-Intg i)) R}⟧ ⟹
G,A⊢{Normal P} New T[e]−≻ {R}

| *Cast*: ⟦G,A⊢{Normal P} e−≻ {λVal:v:. λs..
abupd (raise-if (¬G,s⊢v fits T) ClassCast) .; Q← Val v}⟧ ⟹
G,A⊢{Normal P} Cast T e−≻ {Q}

| *Inst*: ⟦G,A⊢{Normal P} e−≻ {λVal:v:. λs..
Q← Val (Bool (v≠Null ∧ G,s⊢v fits RefT T))}⟧ ⟹
G,A⊢{Normal P} e InstOf T−≻ {Q}

| *Lit*: G,A⊢{Normal (P← Val v)} Lit v−≻ {P}

| *UnOp*: ⟦G,A⊢{Normal P} e−≻ {λVal:v:. Q← Val (eval-unop unop v)}⟧
⟹
G,A⊢{Normal P} UnOp unop e−≻ {Q}

| *BinOp*:
⟦G,A⊢{Normal P} e1−≻ {Q};
∀ v1. G,A⊢{Q← Val v1}
(if need-second-arg binop v1 then (In1l e2) else (In1r Skip))≻
{λVal:v2:. R← Val (eval-binop binop v1 v2)}⟧
⟹
G,A⊢{Normal P} BinOp binop e1 e2−≻ {R}

| *Super*: G,A⊢{Normal (λs.. P← Val (val-this s))} Super−≻ {P}

| *Acc*: ⟦G,A⊢{Normal P} va=≻ {λVar:(v,f):. Q← Val v}⟧ ⟹
G,A⊢{Normal P} Acc va−≻ {Q}

| *Ass*: ⟦G,A⊢{Normal P} va=≻ {Q};
∀ vf. G,A⊢{Q← Var vf} e−≻ {λVal:v:. assign (snd vf) v .; R}⟧ ⟹
G,A⊢{Normal P} va:=e−≻ {R}

| *Cond*: ⟦G,A ⊢{Normal P} e0−≻ {P′};
∀ b. G,A⊢{P′←=b} (if b then e1 else e2)−≻ {Q}⟧ ⟹
G,A⊢{Normal P} e0 ? e1 : e2−≻ {Q}

| *Call*:
⟦G,A⊢{Normal P} e−≻ {Q}; ∀ a. G,A⊢{Q← Val a} args≐≻ {R a};
∀ a vs invC declC l. G,A⊢{(R a← Vals vs ∧.
(λs. declC=invocation-declclass G mode (store s) a statT ⦇name=mn,parTs=pTs⦈ ∧
invC = invocation-class mode (store s) a statT ∧
l = locals (store s)) ;.
init-lvars G declC ⦇name=mn,parTs=pTs⦈ mode a vs) ∧.
(λs. normal s ⟶ G⊢mode→invC≼statT)}
Methd declC ⦇name=mn,parTs=pTs⦈−≻ {set-lvars l .; S}⟧ ⟹
G,A⊢{Normal P} {accC,statT,mode}e·mn({pTs}args)−≻ {S}

| *Methd*: $[\![G,A\cup \{\{P\}\ Methd{-}\!\succ\ \{Q\}\ |\ ms\}\ |\!\vdash\ \{\{P\}\ body\ G{-}\!\succ\ \{Q\}\ |\ ms\}]\!] \Longrightarrow$
$G,A|\!\vdash\{\{P\}\ Methd{-}\!\succ\ \{Q\}\ |\ ms\}$

| *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*: $G,A\vdash\{Normal\ (P\leftarrow Vals\ [])\}\ []\dot{=}\!\succ\ \{P\}$

| *Cons*: $[\![G,A\vdash\{Normal\ P\}\ e{-}\!\succ\ \{Q\};$
$\forall\, v.\ G,A\vdash\{Q\leftarrow Val\ v\}\ es\dot{=}\!\succ\ \{\lambda Vals{:}vs{:}.\ R\leftarrow Vals\ (v\#vs)\}]\!] \Longrightarrow$
$G,A\vdash\{Normal\ P\}\ e\#es\dot{=}\!\succ\ \{R\}$

— statements

| *Skip*: $G,A\vdash\{Normal\ (P\leftarrow\Diamond)\}\ .Skip.\ \{P\}$

| *Expr*: $[\![G,A\vdash\{Normal\ P\}\ e{-}\!\succ\ \{Q\leftarrow\Diamond\}]\!] \Longrightarrow$
$G,A\vdash\{Normal\ P\}\ .Expr\ e.\ \{Q\}$

| *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*: $[\![G,A\vdash\{P\}\ e{-}\!\succ\ \{P'\};$
$G,A\vdash\{Normal\ (P'\leftarrow=True)\}\ .c.\ \{abupd\ (absorb\ (Cont\ l))\ .;\ P\}]\!] \Longrightarrow$
$G,A\vdash\{P\}\ .l\bullet\ While(e)\ c.\ \{(P'\leftarrow=False)\downarrow=\Diamond\}$

| *Jmp*: $G,A\vdash\{Normal\ (abupd\ (\lambda a.\ (Some\ (Jump\ j)))\ .;\ P\leftarrow\Diamond)\}\ .Jmp\ j.\ \{P\}$

| *Throw*: $[\![G,A\vdash\{Normal\ P\}\ e{-}\!\succ\ \{\lambda Val{:}a{:}.\ abupd\ (throw\ a)\ .;\ Q\leftarrow\Diamond\}]\!] \Longrightarrow$
$G,A\vdash\{Normal\ P\}\ .Throw\ e.\ \{Q\}$

| *Try*: $[\![G,A\vdash\{Normal\ P\}\ .c1.\ \{SXAlloc\ G\ Q\};$
$G,A\vdash\{Q\ \wedge.\ (\lambda s.\ \ G,s\vdash catch\ C)\ ;.\ new\text{-}xcpt\text{-}var\ vn\}\ .c2.\ \{R\};$
$(Q\ \wedge.\ (\lambda s.\ \neg G,s\vdash catch\ C))\Rightarrow R]\!] \Longrightarrow$
$G,A\vdash\{Normal\ P\}\ .Try\ c1\ Catch(C\ vn)\ c2.\ \{R\}$

| *Fin*: $[\![G,A\vdash\{Normal\ P\}\ .c1.\ \{Q\};$
$\forall\, x.\ G,A\vdash\{Q\ \wedge.\ (\lambda s.\ x = fst\ s)\ ;.\ abupd\ (\lambda x.\ None)\}$
$.c2.\ \{abupd\ (abrupt\text{-}if\ (x\neq None)\ x)\ .;\ R\}]\!] \Longrightarrow$
$G,A\vdash\{Normal\ P\}\ .c1\ Finally\ c2.\ \{R\}$

| *Done*: $G,A\vdash\{Normal\ (P\leftarrow\Diamond\ \wedge.\ initd\ C)\}\ .Init\ C.\ \{P\}$

| *Init*: $[\![the\ (class\ G\ C) = c;$
$G,A\vdash\{Normal\ ((P\ \wedge.\ Not\circ initd\ C)\ ;.\ supd\ (init\text{-}class\text{-}obj\ G\ C))\}$
$.(if\ C = Object\ then\ Skip\ else\ Init\ (super\ c)).\ \{Q\};$
$\forall\, l.\ G,A\vdash\{Q\ \wedge.\ (\lambda s.\ l = locals\ (store\ s))\ ;.\ set\text{-}lvars\ Map.empty\}$

.init c. {*set-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*}

definition
adapt-pre :: *'a assn* ⇒ *'a assn* ⇒ *'a assn* ⇒ *'a assn*
where *adapt-pre P Q Q'* = (*λY 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*
apply (*unfold ax-valids-def*)
apply *fast*
done

lemma *ax-thin* [*rule-format* (*no-asm*)]:
G,(*A'*::*'a triple set*)|⊢(*ts*::*'a triple set*) ⟹ ∀ *A*. *A'* ⊆ *A* ⟶ *G*,*A*|⊢*ts*
apply (*erule ax-derivs.induct*)
apply (*tactic ALLGOALS* (*EVERY'*[*clarify-tac* ***context***, *REPEAT o smp-tac* ***context*** *1*]))
apply (*rule ax-derivs.empty*)
apply (*erule* (*1*) *ax-derivs.insert*)
apply (*fast intro*: *ax-derivs.asm*)

apply (*fast intro*: *ax-derivs.weaken*)
apply (*rule ax-derivs.conseq*, *intro strip*, *tactic smp-tac* ***context*** *3 1*,*clarify*,
tactic smp-tac ***context*** *1 1*,*rule exI*, *rule exI*, *erule* (*1*) *conjI*)

prefer *18*
apply (*rule ax-derivs.Methd*, *drule spec*, *erule mp*, *fast*)
apply (*tactic* ‹*TRYALL* (*resolve-tac* ***context*** ((*funpow 5 tl*) @{*thms ax-derivs.intros*}))›)
apply *auto*
done

lemma *ax-thin-insert*: *G*,(*A*::*'a triple set*)⊢(*t*::*'a triple*) ⟹ *G*,*insert x A*⊢*t*
apply (*erule ax-thin*)
apply *fast*
done

lemma *subset-mtriples-iff*:
ts ⊆ {{*P*} *mb*−≻ {*Q*} | *ms*} = (∃ *ms'*. *ms'*⊆*ms* ∧ *ts* = {{*P*} *mb*−≻ {*Q*} | *ms'*})
apply (*unfold mtriples-def*)
apply (*rule subset-image-iff*)
done

lemma *weaken*:
G,(*A*::*'a triple set*)|⊢(*ts'*::*'a triple set*) ⟹ ∀ *ts*. *ts* ⊆ *ts'* ⟶ *G*,*A*|⊢*ts*
apply (*erule ax-derivs.induct*)

apply (*tactic ALLGOALS* (*strip-tac* ***context***))
apply (*tactic* ‹*ALLGOALS*(*REPEAT o* (*EVERY'*[*dresolve-tac* ***context*** @{*thms subset-singletonD*},
eresolve-tac ***context*** [*disjE*],
fast-tac (***context*** *addSIs* @{*thms ax-derivs.empty*})]))›)
apply (*tactic TRYALL* (*hyp-subst-tac* ***context***))
apply (*simp*, *rule ax-derivs.empty*)
apply (*drule subset-insertD*)
apply (*blast intro*: *ax-derivs.insert*)
apply (*fast intro*: *ax-derivs.asm*)

apply (*fast intro*: *ax-derivs.weaken*)
apply (*rule ax-derivs.conseq*, *clarify*, *tactic smp-tac* ***context*** *3 1*, *blast*)

apply (*tactic* ‹*TRYALL* (*resolve-tac* ***context*** ((*funpow 5 tl*) @{*thms ax-derivs.intros*})
THEN-ALL-NEW fast-tac ***context***)›)

apply (*clarsimp simp add*: *subset-mtriples-iff*)
apply (*rule ax-derivs.Methd*)
apply (*drule spec*)
apply (*erule impE*)
apply (*rule exI*)
apply (*erule conjI*)
apply (*rule HOL.refl*)
oops

rules derived from conseq

In the following rules we often have to give some type annotations like: $G,A \vdash \{P\}\ t \succ\ \{Q\}$. Given only the term above without annotations, Isabelle would infer a more general type were we could have different types of auxiliary variables in the assumption set (A) and in the triple itself (P and Q). But *ax-derivs.Methd* enforces the same type in the inductive definition of the derivation. So we have to restrict the types to be able to apply the rules.

lemma *conseq12*: $[\![G,(A::'a\ triple\ set) \vdash \{P'::'a\ assn\}\ t \succ\ \{Q'\};$
$\forall Y\ s\ Z.\ P\ Y\ s\ Z \longrightarrow (\forall Y'\ s'.\ (\forall Y\ Z'.\ P'\ Y\ s\ Z' \longrightarrow Q'\ Y'\ s'\ Z') \longrightarrow$
$Q\ Y'\ s'\ Z)]\!]$
$\Longrightarrow\ G,A \vdash \{P\ ::'a\ assn\}\ t \succ\ \{Q\ \}$
apply (*rule ax-derivs.conseq*)
apply *clarsimp*
apply *blast*
done

— Nice variant, since it is so symmetric we might be able to memorise it.

lemma *conseq12'*: $[\![G,(A::'a\ triple\ set) \vdash \{P'::'a\ assn\}\ t \succ\ \{Q'\};\ \forall s\ Y'\ s'.$
$(\forall Y\ Z.\ P'\ Y\ s\ Z \longrightarrow Q'\ Y'\ s'\ Z) \longrightarrow$
$(\forall Y\ Z.\ P\ \ Y\ s\ Z \longrightarrow Q\ \ Y'\ s'\ Z)]\!]$
$\Longrightarrow\ G,A \vdash \{P::'a\ assn\ \}\ t \succ\ \{Q\ \}$
apply (*erule conseq12*)
apply *fast*
done

lemma *conseq12-from-conseq12'*: $[\![G,(A::'a\ triple\ set) \vdash \{P'::'a\ assn\}\ t \succ\ \{Q'\};$
$\forall Y\ s\ Z.\ P\ Y\ s\ Z \longrightarrow (\forall Y'\ s'.\ (\forall Y\ Z'.\ P'\ Y\ s\ Z' \longrightarrow Q'\ Y'\ s'\ Z') \longrightarrow$
$Q\ Y'\ s'\ Z)]\!]$
$\Longrightarrow\ G,A \vdash \{P::'a\ assn\}\ t \succ\ \{Q\ \}$
apply (*erule conseq12'*)

apply *blast*
done

lemma *conseq1*: $[\![G,(A::'a\ triple\ set)\vdash\{P'::'a\ assn\}\ t\succ\ \{Q\};\ P \Rightarrow P']\!]$
$\Longrightarrow G,A\vdash\{P::'a\ assn\}\ t\succ\ \{Q\}$
apply (*erule conseq12*)
apply *blast*
done

lemma *conseq2*: $[\![G,(A::'a\ triple\ set)\vdash\{P::'a\ assn\}\ t\succ\ \{Q'\};\ Q' \Rightarrow Q]\!]$
$\Longrightarrow G,A\vdash\{P::'a\ assn\}\ t\succ\ \{Q\}$
apply (*erule conseq12*)
apply *blast*
done

lemma *ax-escape*:
$[\![\forall\ Y\ s\ Z.\ P\ Y\ s\ Z$
$\longrightarrow G,(A::'a\ triple\ set)\vdash\{\lambda Y'\ s'\ (Z'::'a).\ (Y',s') = (Y,s)\}$
$t\succ$
$\{\lambda Y\ s\ Z'.\ Q\ Y\ s\ Z\}$
$]\!] \Longrightarrow\ G,A\vdash\{P::'a\ assn\}\ t\succ\ \{Q::'a\ assn\}$
apply (*rule ax-derivs.conseq*)
apply *force*
done

lemma *ax-constant*: $[\![\ C \Longrightarrow G,(A::'a\ triple\ set)\vdash\{P::'a\ assn\}\ t\succ\ \{Q\}]\!]$
$\Longrightarrow G,A\vdash\{\lambda Y\ s\ Z.\ C \wedge P\ Y\ s\ Z\}\ t\succ\ \{Q\}$
apply (*rule ax-escape*)
apply *clarify*
apply (*rule conseq12*)
apply *fast*
apply *auto*
done

lemma *ax-impossible* [*intro*]:
$G,(A::'a\ triple\ set)\vdash\{\lambda Y\ s\ Z.\ False\}\ t\succ\ \{Q::'a\ assn\}$
apply (*rule ax-escape*)
apply *clarify*
done

lemma *ax-nochange-lemma*: $[\![P\ Y\ s;\ All\ ((=)\ w)]\!] \Longrightarrow P\ w\ s$
apply *auto*
done

lemma *ax-nochange*:
$G,(A::(res \times state)\ triple\ set)\vdash\{\lambda Y\ s\ Z.\ (Y,s)=Z\}\ t\succ\ \{\lambda Y\ s\ Z.\ (Y,s)=Z\}$
$\Longrightarrow G,A\vdash\{P::(res \times state)\ assn\}\ t\succ\ \{P\}$
apply (*erule conseq12*)

apply *auto*
apply (*erule* (*1*) *ax-nochange-lemma*)
done

lemma *ax-trivial*: $G,(A::'a\ triple\ set)\vdash\{P::'a\ assn\}\ \ t\succ\ \{\lambda Y\ s\ Z.\ True\}$
apply (*rule ax-derivs.conseq*)
apply *auto*
done

lemma *ax-disj*:
$[\![G,(A::'a\ triple\ set)\vdash\{P1::'a\ assn\}\ t\succ\ \{Q1\};\ G,A\vdash\{P2::'a\ assn\}\ t\succ\ \{Q2\}]\!]$
$\implies\ G,A\vdash\{\lambda Y\ s\ Z.\ P1\ Y\ s\ Z\ \vee\ P2\ Y\ s\ Z\}\ t\succ\ \{\lambda Y\ s\ Z.\ Q1\ Y\ s\ Z\ \vee\ Q2\ Y\ s\ Z\}$
apply (*rule ax-escape*)
apply *safe*
apply (*erule conseq12*, *fast*)+
done

lemma *ax-supd-shuffle*:
$(\exists\, Q.\ G,(A::'a\ triple\ set)\vdash\{P::'a\ assn\}\ .c1.\ \{Q\}\ \wedge\ G,A\vdash\{Q\ ;.\ f\}\ .c2.\ \{R\}) =$
$(\exists\, Q'.\ G,A\vdash\{P\}\ .c1.\ \{f\ .;\ Q'\}\ \wedge\ G,A\vdash\{Q'\}\ .c2.\ \{R\})$
apply (*best elim*!: *conseq1 conseq2*)
done

lemma *ax-cases*:
$[\![G,(A::'a\ triple\ set)\vdash\{P\ \wedge.\qquad C\}\ t\succ\ \{Q::'a\ assn\};$
$G,A\vdash\{P\ \wedge.\ Not\ \circ\ C\}\ t\succ\ \{Q\}]\!]\implies G,A\vdash\{P\}\ t\succ\ \{Q\}$
apply (*unfold peek-and-def*)
apply (*rule ax-escape*)
apply *clarify*
apply (*case-tac C s*)
apply (*erule conseq12*, *force*)+
done

lemma *ax-adapt*: $G,(A::'a\ triple\ set)\vdash\{P::'a\ assn\}\ t\succ\ \{Q\}$
$\implies G,A\vdash\{adapt\text{-}pre\ P\ Q\ Q'\}\ t\succ\ \{Q'\}$
apply (*unfold adapt-pre-def*)
apply (*erule conseq12*)
apply *fast*
done

lemma *adapt-pre-adapts*: $G,(A::'a\ triple\ set)\models\{P::'a\ assn\}\ t\succ\ \{Q\}$
$\longrightarrow G,A\models\{adapt\text{-}pre\ P\ Q\ Q'\}\ t\succ\ \{Q'\}$
apply (*unfold adapt-pre-def*)
apply (*simp add*: *ax-valids-def triple-valid-def2*)
apply *fast*
done

lemma *adapt-pre-weakest*:
$\forall G\ (A::'a\ triple\ set)\ t.\ G,A \models \{P\}\ t\succ\ \{Q\} \longrightarrow G,A \models \{P'\}\ t\succ\ \{Q'\} \Longrightarrow$
$P' \Rightarrow adapt\text{-}pre\ P\ Q\ (Q'::'a\ assn)$
apply (*unfold adapt-pre-def*)
apply (*drule spec*)
apply (*drule-tac x = {} **in** spec*)
apply (*drule-tac x = In1r Skip **in** spec*)
apply (*simp add: ax-valids-def triple-valid-def2*)
oops

lemma *peek-and-forget1-Normal*:
$G,(A::'a\ triple\ set)\vdash\{Normal\ P\}\ t\succ\ \{Q::'a\ assn\}$
$\Longrightarrow G,A\vdash\{Normal\ (P \wedge.\ p)\}\ t\succ\ \{Q\}$
apply (*erule conseq1*)
apply (*simp* (*no-asm*))
done

lemma *peek-and-forget1*:
$G,(A::'a\ triple\ set)\vdash\{P::'a\ assn\}\ t\succ\ \{Q\}$
$\Longrightarrow G,A\vdash\{P \wedge.\ p\}\ t\succ\ \{Q\}$
apply (*erule conseq1*)
apply (*simp* (*no-asm*))
done

lemmas *ax-NormalD* = *peek-and-forget1* [*of* - - - - - *normal*]

lemma *peek-and-forget2*:
$G,(A::'a\ triple\ set)\vdash\{P::'a\ assn\}\ t\succ\ \{Q \wedge.\ p\}$
$\Longrightarrow G,A\vdash\{P\}\ t\succ\ \{Q\}$
apply (*erule conseq2*)
apply (*simp* (*no-asm*))
done

lemma *ax-subst-Val-allI*:
$\forall v.\ G,(A::'a\ triple\ set)\vdash\{(P' \qquad v\ )\leftarrow Val\ v\}\ t\succ\ \{(Q\ v)::'a\ assn\}$
$\Longrightarrow\ \forall v.\ G,A\vdash\{(\lambda w{:}.\ P'\ (the\text{-}In1\ w))\leftarrow Val\ v\}\ t\succ\ \{Q\ v\}$
apply (*force elim*!: *conseq1*)
done

lemma *ax-subst-Var-allI*:
$\forall v.\ G,(A::'a\ triple\ set)\vdash\{(P' \qquad v\ )\leftarrow Var\ v\}\ t\succ\ \{(Q\ v)::'a\ assn\}$
$\Longrightarrow\ \forall v.\ G,A\vdash\{(\lambda w{:}.\ P'\ (the\text{-}In2\ w))\leftarrow Var\ v\}\ t\succ\ \{Q\ v\}$
apply (*force elim*!: *conseq1*)
done

lemma *ax-subst-Vals-allI*:
$(\forall v.\ G,(A::'a\ triple\ set)\vdash\{(\quad P' \qquad v\ )\leftarrow Vals\ v\}\ t\succ\ \{(Q\ v)::'a\ assn\})$
$\Longrightarrow\ \forall v.\ G,A\vdash\{(\lambda w{:}.\ P'\ (the\text{-}In3\ w))\leftarrow Vals\ v\}\ t\succ\ \{Q\ v\}$
apply (*force elim*!: *conseq1*)
done

alternative axioms

lemma *ax-Lit2*:
  G,(*A*::′*a triple set*)⊢{*Normal P*::′*a assn*} *Lit v*−≻ {*Normal* (*P*↓=*Val v*)}
apply (*rule ax-derivs.Lit* [*THEN conseq1*])
apply *force*
done

lemma *ax-Lit2-test-complete*:
  G,(*A*::′*a triple set*)⊢{*Normal* (*P*←*Val v*)::′*a assn*} *Lit v*−≻ {*P*}
apply (*rule ax-Lit2* [*THEN conseq2*])
apply *force*
done

lemma *ax-LVar2*: *G*,(*A*::′*a triple set*)⊢{*Normal P*::′*a assn*} *LVar vn*=≻ {*Normal* (λ*s*.. *P*↓=*Var* (*lvar vn s*))}
apply (*rule ax-derivs.LVar* [*THEN conseq1*])
apply *force*
done

lemma *ax-Super2*: *G*,(*A*::′*a triple set*)⊢
  {*Normal P*::′*a assn*} *Super*−≻ {*Normal* (λ*s*.. *P*↓=*Val* (*val-this s*))}
apply (*rule ax-derivs.Super* [*THEN conseq1*])
apply *force*
done

lemma *ax-Nil2*:
  G,(*A*::′*a triple set*)⊢{*Normal P*::′*a assn*} []≐≻ {*Normal* (*P*↓=*Vals* [])}
apply (*rule ax-derivs.Nil* [*THEN conseq1*])
apply *force*
done

misc derived structural rules

lemma *ax-finite-mtriples-lemma*: ⟦*F* ⊆ *ms*; *finite ms*; ∀ (*C*,*sig*)∈*ms*.
    G,(*A*::′*a triple set*)⊢{*Normal* (*P C sig*)::′*a assn*} *mb C sig*−≻ {*Q C sig*}⟧ ⟹
      G,*A*|⊢{{*P*} *mb*−≻ {*Q*} | *F*}
apply (*frule* (*1*) *finite-subset*)
apply (*erule rev-mp*)
apply (*erule thin-rl*)
apply (*erule finite-induct*)
apply (*unfold mtriples-def*)
apply (*clarsimp intro*!: *ax-derivs.empty ax-derivs.insert*)+
apply *force*
done
lemmas *ax-finite-mtriples* = *ax-finite-mtriples-lemma* [*OF subset-refl*]

lemma *ax-derivs-insertD*:
 G,(*A*::′*a triple set*)|⊢*insert* (*t*::′*a triple*) *ts* ⟹ *G*,*A*⊢*t* ∧ *G*,*A*|⊢*ts*
apply (*fast intro*: *ax-derivs.weaken*)
done

lemma *ax-methods-spec*:
⟦*G*,(*A*::′*a triple set*)|⊢*case-prod f* ‘ *ms*; (*C*,*sig*) ∈ *ms*⟧⟹ *G*,*A*⊢((*f C sig*)::′*a triple*)
apply (*erule ax-derivs.weaken*)

apply (*force del*: *image-eqI intro*: *rev-image-eqI*)
done

lemma *ax-finite-pointwise-lemma* [*rule-format*]: ⟦*F* ⊆ *ms*; *finite ms*⟧ ⟹
((∀ (*C*,*sig*)∈*F*. *G*,(*A*::′*a triple set*)⊢(*f C sig*::′*a triple*)) ⟶ (∀ (*C*,*sig*)∈*ms*. *G*,*A*⊢(*g C sig*::′*a triple*))) ⟶
G,*A*|⊢*case-prod f* ' *F* ⟶ *G*,*A*|⊢*case-prod g* ' *F*
apply (*frule* (*1*) *finite-subset*)
apply (*erule rev-mp*)
apply (*erule thin-rl*)
apply (*erule finite-induct*)
apply *clarsimp*+
apply (*drule ax-derivs-insertD*)
apply (*rule ax-derivs.insert*)
apply (*simp* (*no-asm-simp*) *only*: *split-tupled-all*)
apply (*auto elim*: *ax-methods-spec*)
done
lemmas *ax-finite-pointwise* = *ax-finite-pointwise-lemma* [*OF subset-refl*]

lemma *ax-no-hazard*:
G,(*A*::′*a triple set*)⊢{*P* ∧. *type-ok G t*} *t*≻ {*Q*::′*a assn*} ⟹ *G*,*A*⊢{*P*} *t*≻ {*Q*}
apply (*erule ax-cases*)
apply (*rule ax-derivs.hazard* [*THEN conseq1*])
apply *force*
done

lemma *ax-free-wt*:
(∃ *T L C*. (|*prg*=*G*,*cls*=*C*,*lcl*=*L*|)⊢*t*::*T*)
⟶ *G*,(*A*::′*a triple set*)⊢{*Normal P*} *t*≻ {*Q*::′*a assn*} ⟹
G,*A*⊢{*Normal P*} *t*≻ {*Q*}
apply (*rule ax-no-hazard*)
apply (*rule ax-escape*)
apply *clarify*
apply (*erule mp* [*THEN conseq12*])
apply (*auto simp add*: *type-ok-def*)
done

ML ‹*ML-Thms.bind-thms* (*ax-Abrupts*, *sum3-instantiate* ***context*** @{*thm ax-derivs.Abrupt*})›
declare *ax-Abrupts* [*intro*!]

lemmas *ax-Normal-cases* = *ax-cases* [*of - - - normal*]

lemma *ax-Skip* [*intro*!]: *G*,(*A*::′*a triple set*)⊢{*P*←◇} .*Skip*. {*P*::′*a assn*}
apply (*rule ax-Normal-cases*)
apply (*rule ax-derivs.Skip*)
apply *fast*
done
lemmas *ax-SkipI* = *ax-Skip* [*THEN conseq1*]

derived rules for methd call

lemma *ax-Call-known-DynT*:
⟦*G*⊢*IntVir*→*C*≼*statT*;
∀ *a vs l*. *G*,*A*⊢{(*R a*←*Vals vs* ∧. (λ*s*. *l* = *locals* (*store s*)) ;.
init-lvars G C (|*name*=*mn*,*parTs*=*pTs*|) *IntVir a vs*)}

Methd C (|name=mn,parTs=pTs|)−≻ {set-lvars l .; S};
∀ *a. G,A⊢{Q← Val a} args*≐≻
{R a ∧. (λ*s. C = obj-class (the (heap (store s) (the-Addr a)))* ∧
C = invocation-declclass
G IntVir (store s) a statT (|name=mn,parTs=pTs|))};
G,(A::'a triple set)⊢{Normal P} e−≻ {Q::'a assn}⟧
⟹ *G,A⊢{Normal P} {accC,statT,IntVir}e·mn({pTs}args)−≻ {S}*
apply (*erule ax-derivs.Call*)
apply *safe*
apply (*erule spec*)
apply (*rule ax-escape, clarsimp*)
apply (*drule spec, drule spec, drule spec,erule conseq12*)
apply *force*
done

lemma *ax-Call-Static*:
⟦∀ *a vs l. G,A⊢{R a← Vals vs* ∧. (λ*s. l = locals (store s)) ;.*
init-lvars G C (|name=mn,parTs=pTs|) Static any-Addr vs}
Methd C (|name=mn,parTs=pTs|)−≻ {set-lvars l .; S};
G,A⊢{Normal P} e−≻ {Q};
∀ *a. G,(A::'a triple set)⊢{Q← Val a} args*≐≻ *{(R::val ⇒ 'a assn) a*
∧. (λ *s. C=invocation-declclass*
G Static (store s) a statT (|name=mn,parTs=pTs|))}
⟧ ⟹ *G,A⊢{Normal P} {accC,statT,Static}e·mn({pTs}args)−≻ {S}*
apply (*erule ax-derivs.Call*)
apply *safe*
apply (*erule spec*)
apply (*rule ax-escape, clarsimp*)
apply (*erule-tac V = P* ⟶ *Q* **for** *P Q* **in** *thin-rl*)
apply (*drule spec,drule spec,drule spec, erule conseq12*)
apply (*force simp add: init-lvars-def Let-def*)
done

lemma *ax-Methd1*:
⟦*G,A*∪*{{P} Methd−≻ {Q} | ms}*|⊢ *{{P} body G−≻ {Q} | ms}*; *(C,sig)*∈ *ms*⟧ ⟹
G,A⊢{Normal (P C sig)} Methd C sig−≻ {Q C sig}
apply (*drule ax-derivs.Methd*)
apply (*unfold mtriples-def*)
apply (*erule (1) ax-methods-spec*)
done

lemma *ax-MethdN*:
G,insert({Normal P} Methd C sig−≻ {Q}) A⊢
{Normal P} body G C sig−≻ {Q} ⟹
G,A⊢{Normal P} Methd C sig−≻ {Q}
apply (*rule ax-Methd1*)
apply (*rule-tac [2] singletonI*)
apply (*unfold mtriples-def*)
apply *clarsimp*
done

lemma *ax-StatRef*:
G,(A::'a triple set)⊢{Normal (P← Val Null)} StatRef rt−≻ {P::'a assn}
apply (*rule ax-derivs.Cast*)

apply (*rule ax-Lit2* [*THEN conseq2*])
apply *clarsimp*
done

rules derived from Init and Done

lemma *ax-InitS*: ⟦*the* (*class G C*) = *c*; *C* ≠ *Object*;
∀ *l*. *G*,*A*⊢{*Q* ∧. (λ*s*. *l* = *locals* (*store s*)) ;. *set-lvars Map.empty*}
.*init c*. {*set-lvars l* .; *R*};
G,*A*⊢{*Normal* ((*P* ∧. *Not* ∘ *initd C*) ;. *supd* (*init-class-obj G C*))}
.*Init* (*super c*). {*Q*}⟧ ⟹
G,(*A*::′*a triple set*)⊢{*Normal* (*P* ∧. *Not* ∘ *initd C*)} .*Init C*. {*R*::′*a assn*}
apply (*erule ax-derivs.Init*)
apply (*simp* (*no-asm-simp*))
apply *assumption*
done

lemma *ax-Init-Skip-lemma*:
∀ *l*. *G*,(*A*::′*a triple set*)⊢{*P*←♢ ∧. (λ*s*. *l* = *locals* (*store s*)) ;. *set-lvars l*′}
.*Skip*. {(*set-lvars l* .; *P*)::′*a assn*}
apply (*rule allI*)
apply (*rule ax-SkipI*)
apply *clarsimp*
done

lemma *ax-triv-InitS*: ⟦*the* (*class G C*) = *c*;*init c* = *Skip*; *C* ≠ *Object*;
P←♢ ⇒ (*supd* (*init-class-obj G C*) .; *P*);
G,*A*⊢{*Normal* (*P* ∧. *initd C*)} .*Init* (*super c*). {(*P* ∧. *initd C*)←♢}⟧ ⟹
G,(*A*::′*a triple set*)⊢{*Normal P*←♢} .*Init C*. {(*P* ∧. *initd C*)::′*a assn*}
apply (*rule-tac C* = *initd C* **in** *ax-cases*)
apply (*rule conseq1*, *rule ax-derivs.Done*, *clarsimp*)
apply (*simp* (*no-asm*))
apply (*erule* (*1*) *ax-InitS*)
apply *simp*
apply (*rule ax-Init-Skip-lemma*)
apply (*erule conseq1*)
apply *force*
done

lemma *ax-Init-Object*: *wf-prog G* ⟹ *G*,(*A*::′*a triple set*)⊢
{*Normal* ((*supd* (*init-class-obj G Object*) .; *P*←♢) ∧. *Not* ∘ *initd Object*)}
.*Init Object*. {(*P* ∧. *initd Object*)::′*a assn*}
apply (*rule ax-derivs.Init*)
apply (*drule class-Object*, *force*)
apply (*simp-all* (*no-asm*))
apply (*rule-tac* [*2*] *ax-Init-Skip-lemma*)
apply (*rule ax-SkipI*, *force*)
done

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*}
apply (*rule-tac C* = *initd Object* **in** *ax-cases*)
apply (*rule conseq1*, *rule ax-derivs.Done*, *clarsimp*)

apply (*erule ax-Init-Object* [*THEN conseq1*])
apply *force*
done

introduction rules for Alloc and SXAlloc

lemma *ax-SXAlloc-Normal*:
$G,(A::'a\ triple\ set)\vdash\{P::'a\ assn\}\ .c.\ \{Normal\ Q\}$
$\Longrightarrow G,A\vdash\{P\}\ .c.\ \{SXAlloc\ G\ Q\}$
apply (*erule conseq2*)
apply (*clarsimp elim*!: *sxalloc-elim-cases simp add*: *split-tupled-all*)
done

lemma *ax-Alloc*:
$G,(A::'a\ triple\ set)\vdash\{P::'a\ assn\}\ t\succ$
$\{Normal\ (\lambda Y\ (x,s)\ Z.\ (\forall a.\ new\text{-}Addr\ (heap\ s) = Some\ a \longrightarrow$
$Q\ (Val\ (Addr\ a))\ (Norm(init\text{-}obj\ G\ (CInst\ C)\ (Heap\ a)\ s))\ Z)) \wedge.$
$heap\text{-}free\ (Suc\ (Suc\ 0))\}$
$\Longrightarrow G,A\vdash\{P\}\ t\succ\ \{Alloc\ G\ (CInst\ C)\ Q\}$
apply (*erule conseq2*)
apply (*auto elim*!: *halloc-elim-cases*)
done

lemma *ax-Alloc-Arr*:
$G,(A::'a\ triple\ set)\vdash\{P::'a\ assn\}\ t\succ$
$\{\lambda Val{:}i{:}.\ Normal\ (\lambda Y\ (x,s)\ Z.\ \neg the\text{-}Intg\ i<0\ \wedge$
$(\forall a.\ new\text{-}Addr\ (heap\ s) = Some\ a \longrightarrow$
$Q\ (Val\ (Addr\ a))\ (Norm\ (init\text{-}obj\ G\ (Arr\ T\ (the\text{-}Intg\ i))\ (Heap\ a)\ s))\ Z)) \wedge.$
$heap\text{-}free\ (Suc\ (Suc\ 0))\}$
$\Longrightarrow$
$G,A\vdash\{P\}\ t\succ\ \{\lambda Val{:}i{:}.\ abupd\ (check\text{-}neg\ i)\ .;\ Alloc\ G\ (Arr\ T(the\text{-}Intg\ i))\ Q\}$
apply (*erule conseq2*)
apply (*auto elim*!: *halloc-elim-cases*)
done

lemma *ax-SXAlloc-catch-SXcpt*:
$[\![G,(A::'a\ triple\ set)\vdash\{P::'a\ assn\}\ t\succ$
$\{(\lambda Y\ (x,s)\ Z.\ x=Some\ (Xcpt\ (Std\ xn))\ \wedge$
$(\forall a.\ new\text{-}Addr\ (heap\ s) = Some\ a \longrightarrow$
$Q\ Y\ (Some\ (Xcpt\ (Loc\ a)),init\text{-}obj\ G\ (CInst\ (SXcpt\ xn))\ (Heap\ a)\ s)\ Z))$
$\wedge.\ heap\text{-}free\ (Suc\ (Suc\ 0))\}]\!]$
$\Longrightarrow$
$G,A\vdash\{P\}\ t\succ\ \{SXAlloc\ G\ (\lambda Y\ s\ Z.\ Q\ Y\ s\ Z \wedge G,s\vdash catch\ SXcpt\ xn)\}$
apply (*erule conseq2*)
apply (*auto elim*!: *sxalloc-elim-cases halloc-elim-cases*)
done

end

Chapter 23

AxSound

1 Soundness proof for Axiomatic semantics of Java expressions and statements

theory *AxSound* **imports** *AxSem* **begin**

validity

definition
triple-valid2 :: *prog* ⇒ *nat* ⇒ ′*a triple* ⇒ *bool* (-⊨-::-[*61*,*0*, *58*] *57*)
where
G⊨n::t =
(*case t of* {P} t≻ {Q} ⇒
∀ Y s Z. P Y s Z ⟶ (∀ L. s::≼(G,L)
⟶ (∀ T C A. (*normal s* ⟶ ((|*prg*=G,*cls*=C,*lcl*=L|)⊢t::T ∧
(|*prg*=G,*cls*=C,*lcl*=L|)⊢*dom* (*locals* (*store s*))»t»A)) ⟶
(∀ Y' s'. G⊢s −t≻−n→ (Y',s') ⟶ Q Y' s' Z ∧ s'::≼(G,L)))))

This definition differs from the ordinary *triple-valid-def* manly in the conclusion: We also ensures conformance of the result state. So we don't have to apply the type soundness lemma all the time during induction. This definition is only introduced for the soundness proof of the axiomatic semantics, in the end we will conclude to the ordinary definition.

definition
ax-valids2 :: *prog* ⇒ ′*a triples* ⇒ ′*a triples* ⇒ *bool* (-,-|⊨::- [*61*,*58*,*58*] *57*)
where G,A|⊨::ts = (∀ n. (∀ t∈A. G⊨n::t) ⟶ (∀ t∈ts. G⊨n::t))

lemma *triple-valid2-def2*: G⊨n::{P} t≻ {Q} =
(∀ Y s Z. P Y s Z ⟶ (∀ Y' s'. G⊢s −t≻−n→ (Y',s')⟶
(∀ L. s::≼(G,L) ⟶ (∀ T C A. (*normal s* ⟶ ((|*prg*=G,*cls*=C,*lcl*=L|)⊢t::T ∧
(|*prg*=G,*cls*=C,*lcl*=L|)⊢*dom* (*locals* (*store s*))»t»A)) ⟶
Q Y' s' Z ∧ s'::≼(G,L)))))
apply (*unfold triple-valid2-def*)
apply (*simp* (*no-asm*) *add*: *split-paired-All*)
apply *blast*
done

lemma *triple-valid2-eq* [*rule-format* (*no-asm*)]:
wf-prog G ==> *triple-valid2 G* = *triple-valid G*
apply (*rule ext*)
apply (*rule ext*)
apply (*rule triple.induct*)
apply (*simp* (*no-asm*) *add*: *triple-valid-def2 triple-valid2-def2*)
apply (*rule iffI*)
apply *fast*

apply *clarify*
apply (*tactic smp-tac* ***context*** *3 1*)
apply (*case-tac normal s*)
apply *clarsimp*
apply (*elim conjE impE*)
apply *blast*

apply (*tactic smp-tac* ***context*** *2 1*)
apply (*drule evaln-eval*)
apply (*drule* (*1*) *eval-type-sound* [*THEN conjunct1*],*simp*, *assumption*+)
apply *simp*

apply *clarsimp*
done

lemma *ax-valids2-eq*: *wf-prog G* $\Longrightarrow$ *G*,*A*|$\models$::*ts* = *G*,*A*|$\models$*ts*
apply (*unfold ax-valids-def ax-valids2-def*)
apply (*force simp add*: *triple-valid2-eq*)
done

lemma *triple-valid2-Suc* [*rule-format* (*no-asm*)]: *G*$\models$*Suc n*::*t* $\longrightarrow$ *G*$\models$*n*::*t*
apply (*induct-tac t*)
apply (*subst triple-valid2-def2*)
apply (*subst triple-valid2-def2*)
apply (*fast intro*: *evaln-nonstrict-Suc*)
done

lemma *Methd-triple-valid2-0*: *G*$\models$*0*::{*Normal P*} *Methd C sig*$-\succ$ {*Q*}
by (*auto elim*!: *evaln-elim-cases simp add*: *triple-valid2-def2*)

lemma *Methd-triple-valid2-SucI*:
$[\![$*G*$\models$*n*::{*Normal P*} *body G C sig*$-\succ${*Q*}$]\!]$
$\Longrightarrow$ *G*$\models$*Suc n*::{*Normal P*} *Methd C sig*$-\succ$ {*Q*}
apply (*simp* (*no-asm-use*) *add*: *triple-valid2-def2*)
apply (*intro strip*, *tactic smp-tac* ***context*** *3 1*, *clarify*)
apply (*erule wt-elim-cases*, *erule da-elim-cases*, *erule evaln-elim-cases*)
apply (*unfold body-def Let-def*)
apply (*clarsimp simp add*: *inj-term-simps*)
apply *blast*
done

lemma *triples-valid2-Suc*:
Ball ts (*triple-valid2 G* (*Suc n*)) $\Longrightarrow$ *Ball ts* (*triple-valid2 G n*)
apply (*fast intro*: *triple-valid2-Suc*)
done

lemma *G*|$\models$*n*:*insert t A* = (*G*$\models$*n*:*t* $\wedge$ *G*|$\models$*n*:*A*)
oops

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 −
  {
    fix n
    assume recursive: ⋀ n. ∀ t∈(A ∪ {{P} Methd−≻ {Q} | ms}). G⊨n::t
                          ⟹ ∀ t∈{{P} body G−≻ {Q} | ms}. G⊨n::t
    have ∀ t∈A. G⊨n::t ⟹ ∀ t∈{{P} Methd−≻ {Q} | ms}. G⊨n::t
    proof (induct n)
      case 0
      show ∀ t∈{{P} Methd−≻ {Q} | ms}. G⊨0::t
      proof −
        {
          fix C sig
          assume (C,sig) ∈ ms
          have G⊨0::{Normal (P C sig)} Methd C sig−≻ {Q C sig}
            by (rule Methd-triple-valid2-0)
        }
        thus ?thesis
          by (simp add: mtriples-def split-def)
      qed
    next
      case (Suc m)
      note hyp = ‹∀ t∈A. G⊨m::t ⟹ ∀ t∈{{P} Methd−≻ {Q} | ms}. G⊨m::t›
      note prem = ‹∀ t∈A. G⊨Suc m::t›
      show ∀ t∈{{P} Methd−≻ {Q} | ms}. G⊨Suc m::t
      proof −
        {
          fix C sig
          assume m: (C,sig) ∈ ms
          have G⊨Suc m::{Normal (P C sig)} Methd C sig−≻ {Q C sig}
          proof −
            from prem have prem-m: ∀ t∈A. G⊨m::t
              by (rule triples-valid2-Suc)
            hence ∀ t∈{{P} Methd−≻ {Q} | ms}. G⊨m::t
              by (rule hyp)
            with prem-m
            have ∀ t∈(A ∪ {{P} Methd−≻ {Q} | ms}). G⊨m::t
              by (simp add: ball-Un)
            hence ∀ t∈{{P} body G−≻ {Q} | ms}. G⊨m::t
              by (rule recursive)
            with m have G⊨m::{Normal (P C sig)} body G C sig−≻ {Q C sig}
              by (auto simp add: mtriples-def split-def)
            thus ?thesis
              by (rule Methd-triple-valid2-SucI)
          qed
        }
        thus ?thesis
          by (simp add: mtriples-def split-def)
      qed
    qed
  }
  with recursive show ?thesis
    by (unfold ax-valids2-def) blast
qed
```

```
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}}
apply (simp (no-asm) add: ax-valids2-def triple-valid2-def2)
apply clarsimp
done


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 −
  from evaln have G⊢s0 −t≻→ (v, s1)
    by (rule evaln-eval)
  from this wt da wf elim show P
    by (rule da-good-approxE') iprover+
qed


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} }
apply (simp add: ax-valids2-def triple-valid2-def2)
apply (intro allI impI)
apply (case-tac normal s)
apply   clarsimp
apply   (rule I,(assumption|simp)+)

apply   (rule I,auto)
done


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} }
apply (simp add: ax-valids2-def triple-valid2-def2)
```

apply (*intro allI impI*)
apply (*case-tac normal s*)
apply *clarsimp*
apply (*rule I*,(*assumption*|*simp*)+)

apply (*rule I*,*auto*)
done

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*} }
apply (*simp add*: *ax-valids2-def triple-valid2-def2*)
apply (*intro allI impI*)
apply (*elim exE conjE*)
apply (*rule I*)
by *auto*

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*} }
apply (*simp add*: *ax-valids2-def triple-valid2-def2*)
apply (*intro allI impI*)
apply (*elim exE conjE*)
apply *simp*
apply (*rule I*)
by *auto*

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*} }
apply (*simp add*: *ax-valids2-def triple-valid2-def2*)
apply (*intro allI impI*)
apply (*elim exE conjE*)
apply *simp*
apply (*rule I*)
by *auto*

lemma *valid-expr-list-NormalI*:
assumes *I*: ⋀ *n s0 L accC T C vs s1 Y Z*.
⟦∀ *t*∈*A*. *G*⊨*n*::*t*; *s0*::⪯(*G*,*L*); *normal s0*;
(|*prg*=*G*,*cls*=*accC*,*lcl*=*L*|)⊢*t*::≐*T*;
(|*prg*=*G*,*cls*=*accC*,*lcl*=*L*|)⊢*dom* (*locals* (*store s0*))»⟨*t*⟩$_l$»*C*;

$G\vdash s0\ -t\dot{=}\succ vs-n\rightarrow\ s1;\ (Normal\ P)\ Y\ s0\ Z]\!]$
$\Longrightarrow Q\ (In3\ vs)\ s1\ Z \wedge s1::\preceq(G,L)$
shows $G,A|\models::\{\ \{Normal\ P\}\ \langle t\rangle_l\succ\ \{Q\}\ \}$
apply (*simp add*: *ax-valids2-def triple-valid2-def2*)
apply (*intro allI impI*)
apply (*elim exE conjE*)
apply *simp*
apply (*rule I*)
by *auto*

lemma *validE* [*consumes 5*]:
assumes *valid*: $G,A|\models::\{\ \{P\}\ t\succ\ \{Q\}\ \}$
and *P*: $P\ Y\ s0\ Z$
and *valid-A*: $\forall t\in A.\ G\models n::t$
and *conf*: $s0::\preceq(G,L)$
and *eval*: $G\vdash s0\ -t\succ-n\rightarrow\ (v,s1)$
and *wt*: $normal\ s0 \Longrightarrow (|prg=G,cls=accC,lcl=L|)\vdash t::T$
and *da*: $normal\ s0 \Longrightarrow (|prg=G,cls=accC,lcl=L|)\vdash dom\ (locals\ (store\ s0))\gg t\gg C$
and *elim*: $[\![Q\ v\ s1\ Z;\ s1::\preceq(G,L)]\!] \Longrightarrow concl$
shows *concl*
using *assms*
by (*simp add*: *ax-valids2-def triple-valid2-def2*) *fast*

lemma *all-empty*: $(\forall x.\ P) = P$
by *simp*

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 −
from *evaln* **have** $G\vdash s0\ -t\succ\rightarrow\ (v,s1)$
by (*rule evaln-eval*)
from *this wt da wf conf-s0* **show** *?thesis*
by (*rule eval-type-sound*)
qed

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 −
from *evaln* **have** $G\vdash\ s0\ -t\succ\rightarrow\ (v,s1)$ **by** (*rule evaln-eval*)
from *this hyps* **show** *?thesis*
by (*rule dom-locals-eval-mono-elim*) *iprover+*
qed

```
lemma evaln-no-abrupt:
  ⋀s s'. ⟦G⊢s −t≻−n→ (w,s'); normal s'⟧ ⟹ normal s
by (erule evaln-cases,auto)

declare inj-term-simps [simp]

lemma ax-sound2:
  assumes   wf: wf-prog G
    and   deriv: G,A|⊢ts
  shows G,A|⊨::ts
using deriv
proof (induct)
  case (empty A)
  show ?case
    by (simp add: ax-valids2-def triple-valid2-def2)
next
  case (insert A t ts)
  note valid-t = ‹G,A|⊨::{t}›
  moreover
  note valid-ts = ‹G,A|⊨::ts›
  {
    fix n assume valid-A: ∀ t∈A. G⊨n::t
    have G⊨n::t and ∀ t∈ts. G⊨n::t
    proof −
      from valid-A valid-t show G⊨n::t
        by (simp add: ax-valids2-def)
    next
      from valid-A valid-ts show ∀ t∈ts. G⊨n::t
        by (unfold ax-valids2-def) blast
    qed
    hence ∀ t'∈insert t ts. G⊨n::t'
      by simp
  }
  thus ?case
    by (unfold ax-valids2-def) blast
next
  case (asm ts A)
  from ‹ts ⊆ A›
  show G,A|⊨::ts
    by (auto simp add: ax-valids2-def triple-valid2-def)
next
  case (weaken A ts' ts)
  note ‹G,A|⊨::ts'›
  moreover note ‹ts ⊆ ts'›
  ultimately show G,A|⊨::ts
    by (unfold ax-valids2-def triple-valid2-def) blast
next
  case (conseq P A t Q)
  note con = ‹∀ Y s Z. P Y s Z ⟶
           (∃ P' Q'.
               (G,A⊢{P'} t≻ {Q'} ∧ G,A|⊨::{ {P'} t≻ {Q'} }) ∧
               (∀ Y' s'. (∀ Y Z'. P' Y s Z' ⟶ Q' Y' s' Z') ⟶ Q Y' s' Z))›
  show G,A|⊨::{ {P} t≻ {Q} }
  proof (rule validI)
    fix n s0 L accC T C v s1 Y Z
    assume valid-A: ∀ t∈A. G⊨n::t
    assume conf: s0::≼(G,L)
```

```
    assume wt: normal s0 ⟹ (|prg=G,cls=accC,lcl=L|)⊢t::T
    assume da: normal s0
               ⟹ (|prg=G,cls=accC,lcl=L|)⊢dom (locals (store s0)) »t» C
    assume eval: G⊢s0 −t≻−n→ (v, s1)
    assume P: P Y s0 Z
    show Q v s1 Z ∧ s1::≼(G, L)
    proof −
      from valid-A conf wt da eval P con
      have Q v s1 Z
        apply (simp add: ax-valids2-def triple-valid2-def2)
        apply (tactic smp-tac context 3 1)
        apply clarify
        apply (tactic smp-tac context 1 1)
        apply (erule allE,erule allE, erule mp)
        apply (intro strip)
        apply (tactic smp-tac context 3 1)
        apply (tactic smp-tac context 2 1)
        apply (tactic smp-tac context 1 1)
        by blast
      moreover have s1::≼(G, L)
      proof (cases normal s0)
        case True
        from eval wt [OF True] da [OF True] conf wf
        show ?thesis
          by (rule evaln-type-sound [elim-format]) simp
      next
        case False
        with eval have s1=s0
          by auto
        with conf show ?thesis by simp
      qed
      ultimately show ?thesis ..
    qed
  qed
next
  case (hazard A P t Q)
  show G,A|⊨::{ {P ∧. Not ∘ type-ok G t} t≻ {Q} }
    by (simp add: ax-valids2-def triple-valid2-def2 type-ok-def) fast
next
  case (Abrupt A P t)
  show G,A|⊨::{ {P←undefined3 t ∧. Not ∘ normal} t≻ {P} }
  proof (rule validI)
    fix n s0 L accC T C v s1 Y Z
    assume conf-s0: s0::≼(G, L)
    assume eval: G⊢s0 −t≻−n→ (v, s1)
    assume (P←undefined3 t ∧. Not ∘ normal) Y s0 Z
    then obtain P: P (undefined3 t) s0 Z and abrupt-s0: ¬ normal s0
      by simp
    from eval abrupt-s0 obtain s1=s0 and v=undefined3 t
      by auto
    with P conf-s0
    show P v s1 Z ∧ s1::≼(G, L)
      by simp
  qed
next
  case (LVar A P vn)
  show G,A|⊨::{ {Normal (λs.. P←In2 (lvar vn s))} LVar vn=≻ {P} }
  proof (rule valid-var-NormalI)
    fix n s0 L accC T C vf s1 Y Z
```

assume *conf-s0*: *s0*::≼(*G*, *L*)
assume *normal-s0*: *normal s0*
assume *wt*: (|*prg* = *G*, *cls* = *accC*, *lcl* = *L*|)⊢*LVar vn*::=*T*
assume *da*: (|*prg*=*G*,*cls*=*accC*,*lcl*=*L*|)⊢ *dom* (*locals* (*store s0*)) »⟨*LVar vn*⟩$_v$» *C*
assume *eval*: *G*⊢*s0* −*LVar vn*=≻*vf*−*n*→ *s1*
assume *P*: (*Normal* (λ*s*.. *P*←*In2* (*lvar vn s*))) *Y s0 Z*
show *P* (*In2 vf*) *s1 Z* ∧ *s1*::≼(*G*, *L*)
proof
from *eval normal-s0* **obtain** *s1*=*s0 vf*=*lvar vn* (*store s0*)
by (*fastforce elim*: *evaln-elim-cases*)
with *P* **show** *P* (*In2 vf*) *s1 Z*
by *simp*
next
from *eval wt da conf-s0 wf*
show *s1*::≼(*G*, *L*)
by (*rule evaln-type-sound* [*elim-format*]) *simp*
qed
qed
next
case (*FVar A P statDeclC Q e stat fn R accC*)
note *valid-init* = ‹*G*,*A*|⊨::{ {*Normal P*} .*Init statDeclC*. {*Q*} }›
note *valid-e* = ‹*G*,*A*|⊨::{ {*Q*} *e*−≻ {λ*Val*:*a*:. *fvar statDeclC stat fn a* ..; *R*} }›
show *G*,*A*|⊨::{ {*Normal P*} {*accC*,*statDeclC*,*stat*}*e*..*fn*=≻ {*R*} }
proof (*rule valid-var-NormalI*)
fix *n s0 L accC′ T V vf s3 Y Z*
assume *valid-A*: ∀ *t*∈*A*. *G*⊨*n*::*t*
assume *conf-s0*: *s0*::≼(*G*,*L*)
assume *normal-s0*: *normal s0*
assume *wt*: (|*prg*=*G*,*cls*=*accC′*,*lcl*=*L*|)⊢{*accC*,*statDeclC*,*stat*}*e*..*fn*::=*T*
assume *da*: (|*prg*=*G*,*cls*=*accC′*,*lcl*=*L*|)
⊢ *dom* (*locals* (*store s0*)) »⟨{*accC*,*statDeclC*,*stat*}*e*..*fn*⟩$_v$» *V*
assume *eval*: *G*⊢*s0* −{*accC*,*statDeclC*,*stat*}*e*..*fn*=≻*vf*−*n*→ *s3*
assume *P*: (*Normal P*) *Y s0 Z*
show *R* ⌊*vf*⌋$_v$ *s3 Z* ∧ *s3*::≼(*G*, *L*)
proof −
from *wt* **obtain** *statC f* **where**
wt-e: (|*prg*=*G*, *cls*=*accC*, *lcl*=*L*|)⊢*e*::−*Class statC* **and**
accfield: *accfield G accC statC fn* = *Some* (*statDeclC*,*f*) **and**
eq-accC: *accC*=*accC′* **and**
stat: *stat*=*is-static f* **and**
T: *T*=(*type f*)
by (*cases*) (*auto simp add*: *member-is-static-simp*)
from *da eq-accC*
have *da-e*: (|*prg*=*G*, *cls*=*accC*, *lcl*=*L*|)⊢*dom* (*locals* (*store s0*))»⟨*e*⟩$_e$» *V*
by *cases simp*
from *eval* **obtain** *a s1 s2 s2′* **where**
eval-init: *G*⊢*s0* −*Init statDeclC*−*n*→ *s1* **and**
eval-e: *G*⊢*s1* −*e*−≻*a*−*n*→ *s2* **and**
fvar: (*vf*,*s2′*)=*fvar statDeclC stat fn a s2* **and**
s3: *s3* = *check-field-access G accC statDeclC fn stat a s2′*
using *normal-s0* **by** (*fastforce elim*: *evaln-elim-cases*)
have *wt-init*: (|*prg*=*G*, *cls*=*accC*, *lcl*=*L*|)⊢(*Init statDeclC*)::√
proof −
from *wf wt-e*
have *iscls-statC*: *is-class G statC*
by (*auto dest*: *ty-expr-is-type type-is-class*)
with *wf accfield*
have *iscls-statDeclC*: *is-class G statDeclC*
by (*auto dest*!: *accfield-fields dest*: *fields-declC*)

```
      thus ?thesis by simp
    qed
    obtain I where
      da-init: (|prg=G,cls=accC,lcl=L|)
                ⊢ dom (locals (store s0)) »⟨Init statDeclC⟩ₛ» I
      by (auto intro: da-Init [simplified] assigned.select-convs)
    from valid-init P valid-A conf-s0 eval-init wt-init da-init
    obtain Q: Q ◇ s1 Z and conf-s1: s1::≼(G, L)
      by (rule validE)
    obtain
      R: R ⌊vf⌋ᵥ s2' Z and
      conf-s2: s2::≼(G, L) and
      conf-a: normal s2 ⟶ G,store s2⊢a::≼Class statC
    proof (cases normal s1)
      case True
      obtain V' where
        da-e':
        (|prg=G,cls=accC,lcl=L|) ⊢dom (locals (store s1))»⟨e⟩ₑ» V'
      proof −
        from eval-init
        have (dom (locals (store s0))) ⊆ (dom (locals (store s1)))
          by (rule dom-locals-evaln-mono-elim)
        with da-e show thesis
          by (rule da-weakenE) (rule that)
      qed
      with valid-e Q valid-A conf-s1 eval-e wt-e
      obtain R ⌊vf⌋ᵥ s2' Z and s2::≼(G, L)
        by (rule validE) (simp add: fvar [symmetric])
      moreover
      from eval-e wt-e da-e' conf-s1 wf
      have normal s2 ⟶ G,store s2⊢a::≼Class statC
        by (rule evaln-type-sound [elim-format]) simp
      ultimately show ?thesis ..
    next
      case False
      with valid-e Q valid-A conf-s1 eval-e
      obtain  R ⌊vf⌋ᵥ s2' Z and s2::≼(G, L)
        by (cases rule: validE) (simp add: fvar [symmetric])+
      moreover from False eval-e have ¬ normal s2
        by auto
      hence normal s2 ⟶ G,store s2⊢a::≼Class statC
        by auto
      ultimately show ?thesis ..
    qed
    from accfield wt-e eval-init eval-e conf-s2 conf-a fvar stat s3 wf
    have eq-s3-s2': s3=s2'
      using normal-s0 by (auto dest!: error-free-field-access evaln-eval)
    moreover
    from eval wt da conf-s0 wf
    have s3::≼(G, L)
      by (rule evaln-type-sound [elim-format]) simp
    ultimately show ?thesis using Q R by simp
  qed
 qed
next
 case (AVar A P e1 Q e2 R)
 note valid-e1 = ‹G,A|⊨::{ {Normal P} e1−≻ {Q} }›
 have valid-e2: ⋀ a. G,A|⊨::{ {Q←In1 a} e2−≻ {λVal:i:. avar G i a ..; R} }
   using AVar.hyps by simp
```

```
show G,A|⊨::{ {Normal P} e1.[e2]=≻ {R} }
proof (rule valid-var-NormalI)
  fix n s0 L accC T V vf s2' Y Z
  assume valid-A: ∀ t∈A. G⊨n::t
  assume conf-s0: s0::≼(G,L)
  assume normal-s0: normal s0
  assume wt: (|prg=G,cls=accC,lcl=L|)⊢e1.[e2]::=T
  assume da: (|prg=G,cls=accC,lcl=L|)
             ⊢ dom (locals (store s0)) »⟨e1.[e2]⟩v» V
  assume eval: G⊢s0 −e1.[e2]=≻vf−n→ s2'
  assume P: (Normal P) Y s0 Z
  show R ⌊vf⌋v s2' Z ∧ s2'::≼(G, L)
  proof −
    from wt obtain
      wt-e1: (|prg=G,cls=accC,lcl=L|)⊢e1::−T.[] and
      wt-e2: (|prg=G,cls=accC,lcl=L|)⊢e2::−PrimT Integer
      by (rule wt-elim-cases) simp
    from da obtain E1 where
      da-e1: (|prg=G,cls=accC,lcl=L|) ⊢dom (locals (store s0))»⟨e1⟩e» E1 and
      da-e2: (|prg=G,cls=accC,lcl=L|)⊢ nrm E1 »⟨e2⟩e» V
      by (rule da-elim-cases) simp
    from eval obtain s1 a i s2 where
      eval-e1: G⊢s0 −e1−≻a−n→ s1 and
      eval-e2: G⊢s1 −e2−≻i−n→ s2 and
      avar: avar G i a s2 =(vf, s2')
      using normal-s0 by (fastforce elim: evaln-elim-cases)
    from valid-e1 P valid-A conf-s0 eval-e1 wt-e1 da-e1
    obtain Q: Q ⌊a⌋e s1 Z and conf-s1: s1::≼(G, L)
      by (rule validE)
    from Q have Q': ⋀ v. (Q←In1 a) v s1 Z
      by simp
    have R ⌊vf⌋v s2' Z
    proof (cases normal s1)
      case True
      obtain V' where
        (|prg=G,cls=accC,lcl=L|) ⊢dom (locals (store s1))»⟨e2⟩e» V'
      proof −
        from eval-e1 wt-e1 da-e1 wf True
        have nrm E1 ⊆ dom (locals (store s1))
          by (cases rule: da-good-approx-evalnE) iprover
        with da-e2 show thesis
          by (rule da-weakenE) (rule that)
      qed
      with valid-e2 Q' valid-A conf-s1 eval-e2 wt-e2
      show ?thesis
        by (rule validE) (simp add: avar)
    next
      case False
      with valid-e2 Q' valid-A conf-s1 eval-e2
      show ?thesis
        by (cases rule: validE) (simp add: avar)+
    qed
    moreover
    from eval wt da conf-s0 wf
    have s2'::≼(G, L)
      by (rule evaln-type-sound [elim-format]) simp
    ultimately show ?thesis ..
  qed
qed
```

```
next
  case (NewC A P C Q)
  note valid-init = ‹G,A|⊨::{ {Normal P} .Init C. {Alloc G (CInst C) Q} }›
  show G,A|⊨::{ {Normal P} NewC C−≻ {Q} }
  proof (rule valid-expr-NormalI)
    fix n s0 L accC T E v s2 Y Z
    assume valid-A: ∀ t∈A. G⊨n::t
    assume conf-s0: s0::⪯(G,L)
    assume normal-s0: normal s0
    assume wt: (|prg=G,cls=accC,lcl=L|)⊢NewC C::−T
    assume da: (|prg=G,cls=accC,lcl=L|)
                 ⊢ dom (locals (store s0)) »⟨NewC C⟩e» E
    assume eval: G⊢s0 −NewC C−≻v−n→ s2
    assume P: (Normal P) Y s0 Z
    show Q ⌊v⌋e s2 Z ∧ s2::⪯(G, L)
    proof −
      from wt obtain is-cls-C: is-class G C
        by (rule wt-elim-cases) (auto dest: is-acc-classD)
      hence wt-init: (|prg=G, cls=accC, lcl=L|)⊢Init C::√
        by auto
      obtain I where
        da-init: (|prg=G,cls=accC,lcl=L|)⊢ dom (locals (store s0)) »⟨Init C⟩s» I
        by (auto intro: da-Init [simplified] assigned.select-convs)
      from eval obtain s1 a where
        eval-init: G⊢s0 −Init C−n→ s1 and
        alloc: G⊢s1 −halloc CInst C≻a→ s2 and
        v: v=Addr a
        using normal-s0 by (fastforce elim: evaln-elim-cases)
      from valid-init P valid-A conf-s0 eval-init wt-init da-init
      obtain (Alloc G (CInst C) Q) ♢ s1 Z
        by (rule validE)
      with alloc v have Q ⌊v⌋e s2 Z
        by simp
      moreover
      from eval wt da conf-s0 wf
      have s2::⪯(G, L)
        by (rule evaln-type-sound [elim-format]) simp
      ultimately show ?thesis ..
    qed
  qed
next
  case (NewA A P T Q e R)
  note valid-init = ‹G,A|⊨::{ {Normal P} .init-comp-ty T. {Q} }›
  note valid-e = ‹G,A|⊨::{ {Q} e−≻ {λVal:i:. abupd (check-neg i) .;
                                      Alloc G (Arr T (the-Intg i)) R}}›
  show G,A|⊨::{ {Normal P} New T[e]−≻ {R} }
  proof (rule valid-expr-NormalI)
    fix n s0 L accC arrT E v s3 Y Z
    assume valid-A: ∀ t∈A. G⊨n::t
    assume conf-s0: s0::⪯(G,L)
    assume normal-s0: normal s0
    assume wt: (|prg=G,cls=accC,lcl=L|)⊢New T[e]::−arrT
    assume da: (|prg=G,cls=accC,lcl=L|)⊢dom (locals (store s0)) »⟨New T[e]⟩e» E
    assume eval: G⊢s0 −New T[e]−≻v−n→ s3
    assume P: (Normal P) Y s0 Z
    show R ⌊v⌋e s3 Z ∧ s3::⪯(G, L)
    proof −
      from wt obtain
        wt-init: (|prg=G,cls=accC,lcl=L|)⊢init-comp-ty T::√ and
```

```
      wt-e: (|prg=G,cls=accC,lcl=L|)⊢e::−PrimT Integer
      by (rule wt-elim-cases) (auto intro: wt-init-comp-ty )
    from da obtain
      da-e:(|prg=G,cls=accC,lcl=L|)⊢ dom (locals (store s0)) »⟨e⟩e» E
      by cases simp
    from eval obtain s1 i s2 a where
      eval-init: G⊢s0 −init-comp-ty T−n→ s1 and
      eval-e: G⊢s1 −e−≻i−n→ s2 and
      alloc: G⊢abupd (check-neg i) s2 −halloc Arr T (the-Intg i)≻a→ s3 and
      v: v=Addr a
      using normal-s0 by (fastforce elim: evaln-elim-cases)
    obtain I where
      da-init:
      (|prg=G,cls=accC,lcl=L|)⊢dom (locals (store s0)) »⟨init-comp-ty T⟩s» I
    proof (cases ∃ C. T = Class C)
      case True
      thus ?thesis
        by − (rule that, (auto intro: da-Init [simplified]
                                 assigned.select-convs
                        simp add: init-comp-ty-def))

    next
      case False
      thus ?thesis
        by − (rule that, (auto intro: da-Skip [simplified]
                                assigned.select-convs
                      simp add: init-comp-ty-def))

    qed
    with valid-init P valid-A conf-s0 eval-init wt-init
    obtain Q: Q ◇ s1 Z and conf-s1: s1::≼(G, L)
      by (rule validE)
    obtain E' where
     (|prg=G,cls=accC,lcl=L|)⊢ dom (locals (store s1)) »⟨e⟩e» E'
    proof −
      from eval-init
      have dom (locals (store s0)) ⊆ dom (locals (store s1))
        by (rule dom-locals-evaln-mono-elim)
      with da-e show thesis
        by (rule da-weakenE) (rule that)
    qed
    with valid-e Q valid-A conf-s1 eval-e wt-e
    have (λVal:i:. abupd (check-neg i) .;
                Alloc G (Arr T (the-Intg i)) R) ⌊i⌋e s2 Z
      by (rule validE)
    with alloc v have R ⌊v⌋e s3 Z
      by simp
    moreover
    from eval wt da conf-s0 wf
    have s3::≼(G, L)
      by (rule evaln-type-sound [elim-format]) simp
    ultimately show ?thesis ..
  qed
 qed
next
 case (Cast A P e T Q)
 note valid-e = ‹G,A|⊨::{ {Normal P} e−≻
              {λVal:v:. λs.. abupd (raise-if (¬ G,s⊢v fits T) ClassCast) .;
               Q←In1 v} }›
```

show *G*,*A*|⊨::{ {*Normal P*} *Cast T e*−≻ {*Q*} }
proof (*rule valid-expr-NormalI*)
fix *n s0 L accC castT E v s2 Y Z*
assume *valid-A*: ∀ *t*∈*A*. *G*⊨*n*::*t*
assume *conf-s0*: *s0*::⪯(*G*,*L*)
assume *normal-s0*: *normal s0*
assume *wt*: (|*prg*=*G*,*cls*=*accC*,*lcl*=*L*|)⊢*Cast T e*::−*castT*
assume *da*: (|*prg*=*G*,*cls*=*accC*,*lcl*=*L*|)⊢*dom* (*locals* (*store s0*)) »⟨*Cast T e*⟩$_e$» *E*
assume *eval*: *G*⊢*s0* −*Cast T e*−≻*v*−*n*→ *s2*
assume *P*: (*Normal P*) *Y s0 Z*
show *Q* ⌊*v*⌋$_e$ *s2 Z* ∧ *s2*::⪯(*G*, *L*)
proof −
from *wt* **obtain** *eT* **where**
wt-e: (|*prg* = *G*, *cls* = *accC*, *lcl* = *L*|)⊢*e*::−*eT*
by *cases simp*
from *da* **obtain**
da-e: (|*prg*=*G*,*cls*=*accC*,*lcl*=*L*|)⊢ *dom* (*locals* (*store s0*)) »⟨*e*⟩$_e$» *E*
by *cases simp*
from *eval* **obtain** *s1* **where**
eval-e: *G*⊢*s0* −*e*−≻*v*−*n*→ *s1* **and**
s2: *s2* = *abupd* (*raise-if* (¬ *G*,*snd s1*⊢*v fits T*) *ClassCast*) *s1*
using *normal-s0* **by** (*fastforce elim*: *evaln-elim-cases*)
from *valid-e P valid-A conf-s0 eval-e wt-e da-e*
have (λ*Val*:*v*:. λ*s*.. *abupd* (*raise-if* (¬ *G*,*s*⊢*v fits T*) *ClassCast*) .;
Q←*In1 v*) ⌊*v*⌋$_e$ *s1 Z*
by (*rule validE*)
with *s2* **have** *Q* ⌊*v*⌋$_e$ *s2 Z*
by *simp*
moreover
from *eval wt da conf-s0 wf*
have *s2*::⪯(*G*, *L*)
by (*rule evaln-type-sound* [*elim-format*]) *simp*
ultimately show *?thesis* **..**
qed
qed
next
case (*Inst A P e Q T*)
assume *valid-e*: *G*,*A*|⊨::{ {*Normal P*} *e*−≻
{λ*Val*:*v*:. λ*s*.. *Q*←*In1* (*Bool* (*v* ≠ *Null* ∧ *G*,*s*⊢*v fits RefT T*))} }
show *G*,*A*|⊨::{ {*Normal P*} *e InstOf T*−≻ {*Q*} }
proof (*rule valid-expr-NormalI*)
fix *n s0 L accC instT E v s1 Y Z*
assume *valid-A*: ∀ *t*∈*A*. *G*⊨*n*::*t*
assume *conf-s0*: *s0*::⪯(*G*,*L*)
assume *normal-s0*: *normal s0*
assume *wt*: (|*prg*=*G*,*cls*=*accC*,*lcl*=*L*|)⊢*e InstOf T*::−*instT*
assume *da*: (|*prg*=*G*,*cls*=*accC*,*lcl*=*L*|)⊢*dom* (*locals* (*store s0*))»⟨*e InstOf T*⟩$_e$» *E*
assume *eval*: *G*⊢*s0* −*e InstOf T*−≻*v*−*n*→ *s1*
assume *P*: (*Normal P*) *Y s0 Z*
show *Q* ⌊*v*⌋$_e$ *s1 Z* ∧ *s1*::⪯(*G*, *L*)
proof −
from *wt* **obtain** *eT* **where**
wt-e: (|*prg* = *G*, *cls* = *accC*, *lcl* = *L*|)⊢*e*::−*eT*
by *cases simp*
from *da* **obtain**
da-e: (|*prg*=*G*,*cls*=*accC*,*lcl*=*L*|)⊢ *dom* (*locals* (*store s0*)) »⟨*e*⟩$_e$» *E*
by *cases simp*
from *eval* **obtain** *a* **where**
eval-e: *G*⊢*s0* −*e*−≻*a*−*n*→ *s1* **and**

```
      v: v = Bool (a ≠ Null ∧ G,store s1⊢a fits RefT T)
      using normal-s0 by (fastforce elim: evaln-elim-cases)
    from valid-e P valid-A conf-s0 eval-e wt-e da-e
    have (λVal:v:. λs.. Q←In1 (Bool (v ≠ Null ∧ G,s⊢v fits RefT T)))
           ⌊a⌋e s1 Z
      by (rule validE)
    with v have Q ⌊v⌋e s1 Z
      by simp
    moreover
    from eval wt da conf-s0 wf
    have s1::≼(G, L)
      by (rule evaln-type-sound [elim-format]) simp
    ultimately show ?thesis ..
  qed
 qed
next
 case (Lit A P v)
 show G,A|⊨::{ {Normal (P←In1 v)} Lit v−≻ {P} }
 proof (rule valid-expr-NormalI)
  fix n L s0 s1 v'  Y Z
  assume conf-s0: s0::≼(G, L)
  assume normal-s0: normal s0
  assume eval: G⊢s0 −Lit v−≻v'−n→ s1
  assume P: (Normal (P←In1 v)) Y s0 Z
  show P ⌊v'⌋e s1 Z ∧ s1::≼(G, L)
  proof −
    from eval have s1=s0 and v'=v
      using normal-s0 by (auto elim: evaln-elim-cases)
    with P conf-s0 show ?thesis by simp
  qed
 qed
next
 case (UnOp A P e Q unop)
 assume valid-e: G,A|⊨::{ {Normal P}e−≻{λVal:v:. Q←In1 (eval-unop unop v)} }
 show G,A|⊨::{ {Normal P} UnOp unop e−≻ {Q} }
 proof (rule valid-expr-NormalI)
  fix n s0 L accC T E v s1 Y Z
  assume valid-A: ∀t∈A. G⊨n::t
  assume conf-s0: s0::≼(G,L)
  assume normal-s0: normal s0
  assume wt: (|prg=G,cls=accC,lcl=L|)⊢UnOp unop e::−T
  assume da: (|prg=G,cls=accC,lcl=L|)⊢dom (locals (store s0))»⟨UnOp unop e⟩e»E
  assume eval: G⊢s0 −UnOp unop e−≻v−n→ s1
  assume P: (Normal P) Y s0 Z
  show Q ⌊v⌋e s1 Z ∧ s1::≼(G, L)
  proof −
    from wt obtain eT where
      wt-e: (|prg = G, cls = accC, lcl = L|)⊢e::−eT
      by cases simp
    from da obtain
      da-e: (|prg=G,cls=accC,lcl=L|)⊢ dom (locals (store s0)) »⟨e⟩e» E
      by cases simp
    from eval obtain ve where
      eval-e: G⊢s0 −e−≻ve−n→ s1 and
      v: v = eval-unop unop ve
      using normal-s0 by (fastforce elim: evaln-elim-cases)
    from valid-e P valid-A conf-s0 eval-e wt-e da-e
    have (λVal:v:. Q←In1 (eval-unop unop v)) ⌊ve⌋e s1 Z
      by (rule validE)
```

```
      with v have Q ⌊v⌋e s1 Z
        by simp
      moreover
      from eval wt da conf-s0 wf
      have s1::≼(G, L)
        by (rule evaln-type-sound [elim-format]) simp
      ultimately show ?thesis ..
    qed
  qed
next
  case (BinOp A P e1 Q binop e2 R)
  assume valid-e1: G,A|⊨::{ {Normal P} e1−≻ {Q} }
  have valid-e2: ⋀ v1.  G,A|⊨::{ {Q←In1 v1}
              (if need-second-arg binop v1 then In1l e2 else In1r Skip)≻
              {λVal:v2:. R←In1 (eval-binop binop v1 v2)} }
    using BinOp.hyps by simp
  show G,A|⊨::{ {Normal P} BinOp binop e1 e2−≻ {R} }
  proof (rule valid-expr-NormalI)
    fix n s0 L accC T E v s2 Y Z
    assume valid-A: ∀ t∈A. G⊨n::t
    assume conf-s0: s0::≼(G,L)
    assume normal-s0: normal s0
    assume wt: (|prg=G,cls=accC,lcl=L|)⊢BinOp binop e1 e2::−T
    assume da: (|prg=G,cls=accC,lcl=L|)
                 ⊢dom (locals (store s0)) »⟨BinOp binop e1 e2⟩e» E
    assume eval: G⊢s0 −BinOp binop e1 e2−≻v−n→ s2
    assume P: (Normal P) Y s0 Z
    show R ⌊v⌋e s2 Z ∧ s2::≼(G, L)
    proof −
      from wt obtain e1T e2T where
        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
        by cases simp
      have wt-Skip: (|prg = G, cls = accC, lcl = L|)⊢Skip::√
        by simp

      from da obtain E1 where
        da-e1: (|prg=G,cls=accC,lcl=L|) ⊢ dom (locals (store s0)) »⟨e1⟩e» E1
        by cases simp+
      from eval obtain v1 s1 v2 where
        eval-e1: G⊢s0 −e1−≻v1−n→ s1 and
        eval-e2: G⊢s1 −(if need-second-arg binop v1 then ⟨e2⟩e else ⟨Skip⟩s)
                    ≻−n→ (⌊v2⌋e, s2) and
        v: v=eval-binop binop v1 v2
        using normal-s0 by (fastforce elim: evaln-elim-cases)
      from valid-e1 P valid-A conf-s0 eval-e1 wt-e1 da-e1
      obtain Q: Q ⌊v1⌋e s1 Z and conf-s1: s1::≼(G,L)
        by (rule validE)
      from Q have Q': ⋀ v. (Q←In1 v1) v s1 Z
        by simp
      have (λVal:v2:. R←In1 (eval-binop binop v1 v2)) ⌊v2⌋e s2 Z
      proof (cases normal s1)
        case True
        from eval-e1 wt-e1 da-e1 conf-s0 wf
        have conf-v1: G,store s1⊢v1::≼e1T
          by (rule evaln-type-sound [elim-format]) (insert True,simp)
        from eval-e1
        have G⊢s0 −e1−≻v1→ s1
```

```
        by (rule evaln-eval)
      from da wt-e1 wt-e2 wt-binop conf-s0 True this conf-v1 wf
      obtain E2 where
        da-e2: (|prg=G,cls=accC,lcl=L|)⊢ dom (locals (store s1))
               »(if need-second-arg binop v1 then ⟨e2⟩e else ⟨Skip⟩s)» E2
        by (rule da-e2-BinOp [elim-format]) iprover
      from wt-e2 wt-Skip obtain T2
        where (|prg=G,cls=accC,lcl=L|)
              ⊢(if need-second-arg binop v1 then ⟨e2⟩e else ⟨Skip⟩s)::T2
        by (cases need-second-arg binop v1) auto
      note ve=validE [OF valid-e2,OF  Q' valid-A conf-s1 eval-e2 this da-e2]

      thus ?thesis
        by (rule ve)
    next
      case False
      note ve=validE [OF valid-e2,OF Q' valid-A conf-s1 eval-e2]
      with False show ?thesis
        by iprover
    qed
    with v have R ⌊v⌋e s2 Z
      by simp
    moreover
    from eval wt da conf-s0 wf
    have s2::≼(G, L)
      by (rule evaln-type-sound [elim-format]) simp
    ultimately show ?thesis ..
  qed
 qed
next
 case (Super A P)
 show G,A|⊨::{ {Normal (λs.. P←In1 (val-this s))} Super−≻ {P} }
 proof (rule valid-expr-NormalI)
  fix n L s0 s1 v  Y Z
  assume conf-s0: s0::≼(G, L)
  assume normal-s0:  normal s0
  assume eval: G⊢s0 −Super−≻v−n→ s1
  assume P: (Normal (λs.. P←In1 (val-this s))) Y s0 Z
  show P ⌊v⌋e s1 Z ∧ s1::≼(G, L)
  proof −
    from eval have s1=s0 and  v=val-this (store s0)
      using normal-s0 by (auto elim: evaln-elim-cases)
    with P conf-s0 show ?thesis by simp
  qed
 qed
next
 case (Acc A P var Q)
 note valid-var = ‹G,A|⊨::{ {Normal P} var=≻ {λVar:(v, f):. Q←In1 v} }›
 show G,A|⊨::{ {Normal P} Acc var−≻ {Q} }
 proof (rule valid-expr-NormalI)
  fix n s0 L accC T E v s1 Y Z
  assume valid-A: ∀ t∈A. G⊨n::t
  assume conf-s0: s0::≼(G,L)
  assume normal-s0: normal s0
  assume wt: (|prg=G,cls=accC,lcl=L|)⊢Acc var::−T
  assume da: (|prg=G,cls=accC,lcl=L|)⊢dom (locals (store s0))»⟨Acc var⟩e»E
  assume eval: G⊢s0 −Acc var−≻v−n→ s1
  assume P: (Normal P) Y s0 Z
  show Q ⌊v⌋e s1 Z ∧ s1::≼(G, L)
```

```
    proof -
      from wt obtain
        wt-var: ⦇prg=G,cls=accC,lcl=L⦈⊢var::=T
        by cases simp
      from da obtain V where
        da-var: ⦇prg=G,cls=accC,lcl=L⦈ ⊢ dom (locals (store s0)) »⟨var⟩ᵥ» V
        by (cases ∃ n. var=LVar n) (insert da.LVar,auto elim!: da-elim-cases)
      from eval obtain upd where
        eval-var: G⊢s0 −var=≻(v, upd)−n→ s1
        using normal-s0 by (fastforce elim: evaln-elim-cases)
      from valid-var P valid-A conf-s0 eval-var wt-var da-var
      have (λVar:(v, f):. Q←In1 v) ⌊(v, upd)⌋ᵥ s1 Z
        by (rule validE)
      then have Q ⌊v⌋ₑ s1 Z
        by simp
      moreover
      from eval wt da conf-s0 wf
      have s1::≼(G, L)
        by (rule evaln-type-sound [elim-format]) simp
      ultimately show ?thesis ..
    qed
  qed
next
  case (Ass A P var Q e R)
  note valid-var = ‹G,A|⊨::{ {Normal P} var=≻ {Q} }›
  have valid-e: ⋀ vf.
                  G,A|⊨::{ {Q←In2 vf} e−≻ {λVal:v:. assign (snd vf) v .; R} }
    using Ass.hyps by simp
  show G,A|⊨::{ {Normal P} var:=e−≻ {R} }
  proof (rule valid-expr-NormalI)
    fix n s0 L accC T E v s3 Y Z
    assume valid-A: ∀ t∈A. G⊨n::t
    assume conf-s0: s0::≼(G,L)
    assume normal-s0: normal s0
    assume wt: ⦇prg=G,cls=accC,lcl=L⦈⊢var:=e::−T
    assume da: ⦇prg=G,cls=accC,lcl=L⦈⊢dom (locals (store s0))»⟨var:=e⟩ₑ»E
    assume eval: G⊢s0 −var:=e−≻v−n→ s3
    assume P: (Normal P) Y s0 Z
    show R ⌊v⌋ₑ s3 Z ∧ s3::≼(G, L)
    proof -
      from wt obtain varT where
        wt-var: ⦇prg=G,cls=accC,lcl=L⦈⊢var::=varT and
        wt-e: ⦇prg=G,cls=accC,lcl=L⦈⊢e::−T
        by cases simp
      from eval obtain w upd s1 s2 where
        eval-var: G⊢s0 −var=≻(w, upd)−n→ s1 and
        eval-e: G⊢s1 −e−≻v−n→ s2 and
        s3: s3=assign upd v s2
        using normal-s0 by (auto elim: evaln-elim-cases)
      have R ⌊v⌋ₑ s3 Z
      proof (cases ∃ vn. var = LVar vn)
        case False
        with da obtain V where
          da-var: ⦇prg=G,cls=accC,lcl=L⦈
                    ⊢ dom (locals (store s0)) »⟨var⟩ᵥ» V and
          da-e:   ⦇prg=G,cls=accC,lcl=L⦈ ⊢ nrm V »⟨e⟩ₑ» E
          by cases simp+
        from valid-var P valid-A conf-s0 eval-var wt-var da-var
        obtain Q: Q ⌊(w,upd)⌋ᵥ s1 Z and conf-s1: s1::≼(G,L)
```

by (*rule validE*)
hence Q': $\bigwedge$ *v*. (Q←*In2* (*w*,*upd*)) *v s1 Z*
by *simp*
have (λ*Val*:*v*:. *assign* (*snd* (*w*,*upd*)) *v* .; *R*) $\lfloor v \rfloor_e$ *s2 Z*
proof (*cases normal s1*)
case *True*
obtain E' **where**
da-e': (|*prg*=*G*,*cls*=*accC*,*lcl*=*L*|)⊢ *dom* (*locals* (*store s1*)) »$\langle e \rangle_e$» E'
proof −
from *eval-var wt-var da-var wf True*
have *nrm V* ⊆ *dom* (*locals* (*store s1*))
by (*cases rule*: *da-good-approx-evalnE*) *iprover*
with *da-e* **show** *thesis*
by (*rule da-weakenE*) (*rule that*)
qed
note *ve*=*validE* [*OF valid-e*,*OF* Q' *valid-A conf-s1 eval-e wt-e da-e'*]
show *?thesis*
by (*rule ve*)
next
case *False*
note *ve*=*validE* [*OF valid-e*,*OF* Q' *valid-A conf-s1 eval-e*]
with *False* **show** *?thesis*
by *iprover*
qed
with *s3* **show** *R* $\lfloor v \rfloor_e$ *s3 Z*
by *simp*
next
case *True*
then obtain *vn* **where**
vn: *var* = *LVar vn*
by *auto*
with *da* **obtain** *E* **where**
da-e: (|*prg*=*G*,*cls*=*accC*,*lcl*=*L*|) ⊢ *dom* (*locals* (*store s0*)) »$\langle e \rangle_e$» *E*
by *cases simp+*
from *da.LVar vn* **obtain** *V* **where**
da-var: (|*prg*=*G*,*cls*=*accC*,*lcl*=*L*|)
⊢ *dom* (*locals* (*store s0*)) »$\langle var \rangle_v$» *V*
by *auto*
from *valid-var P valid-A conf-s0 eval-var wt-var da-var*
obtain *Q*: *Q* $\lfloor (w,upd) \rfloor_v$ *s1 Z* **and** *conf-s1*: *s1*::≼(*G*,*L*)
by (*rule validE*)
hence Q': $\bigwedge$ *v*. (Q←*In2* (*w*,*upd*)) *v s1 Z*
by *simp*
have (λ*Val*:*v*:. *assign* (*snd* (*w*,*upd*)) *v* .; *R*) $\lfloor v \rfloor_e$ *s2 Z*
proof (*cases normal s1*)
case *True*
obtain E' **where**
da-e': (|*prg*=*G*,*cls*=*accC*,*lcl*=*L*|)
⊢ *dom* (*locals* (*store s1*)) »$\langle e \rangle_e$» E'
proof −
from *eval-var*
have *dom* (*locals* (*store s0*)) ⊆ *dom* (*locals* (*store* (*s1*)))
by (*rule dom-locals-evaln-mono-elim*)
with *da-e* **show** *thesis*
by (*rule da-weakenE*) (*rule that*)
qed
note *ve*=*validE* [*OF valid-e*,*OF* Q' *valid-A conf-s1 eval-e wt-e da-e'*]
show *?thesis*
by (*rule ve*)

```
        next
          case False
          note ve=validE [OF valid-e,OF Q' valid-A conf-s1 eval-e]
          with False show ?thesis
            by iprover
        qed
        with s3 show R ⌊v⌋e s3 Z
          by simp
      qed
      moreover
      from eval wt da conf-s0 wf
      have s3::≼(G, L)
        by (rule evaln-type-sound [elim-format]) simp
      ultimately show ?thesis ..
    qed
  qed
next
  case (Cond A P e0 P' e1 e2 Q)
  note valid-e0 = ‹G,A|⊨::{ {Normal P} e0−≻ {P'} }›
  have valid-then-else:⋀ b.  G,A|⊨::{ {P'←=b} (if b then e1 else e2)−≻ {Q} }
    using Cond.hyps by simp
  show G,A|⊨::{ {Normal P} e0 ? e1 : e2−≻ {Q} }
  proof (rule valid-expr-NormalI)
    fix n s0 L accC T E v s2 Y Z
    assume valid-A: ∀ t∈A. G⊨n::t
    assume conf-s0: s0::≼(G,L)
    assume normal-s0: normal s0
    assume wt: (|prg=G,cls=accC,lcl=L|)⊢e0 ? e1 : e2::−T
    assume da: (|prg=G,cls=accC,lcl=L|)⊢dom (locals (store s0))»⟨e0 ? e1:e2⟩e»E
    assume eval: G⊢s0 −e0 ? e1 : e2−≻v−n→ s2
    assume P: (Normal P) Y s0 Z
    show Q ⌊v⌋e s2 Z ∧ s2::≼(G, L)
    proof −
      from wt obtain T1 T2 where
        wt-e0: (|prg=G,cls=accC,lcl=L|)⊢e0::−PrimT Boolean and
        wt-e1: (|prg=G,cls=accC,lcl=L|)⊢e1::−T1 and
        wt-e2: (|prg=G,cls=accC,lcl=L|)⊢e2::−T2
        by cases simp
      from da obtain E0 E1 E2 where
        da-e0: (|prg=G,cls=accC,lcl=L|)⊢ dom (locals (store s0)) »⟨e0⟩e» E0 and
        da-e1: (|prg=G,cls=accC,lcl=L|)
                 ⊢(dom (locals (store s0)) ∪ assigns-if True e0)»⟨e1⟩e» E1 and
        da-e2: (|prg=G,cls=accC,lcl=L|)
                 ⊢(dom (locals (store s0)) ∪ assigns-if False e0)»⟨e2⟩e» E2
        by cases simp+
      from eval obtain b s1 where
        eval-e0: G⊢s0 −e0−≻b−n→ s1 and
        eval-then-else: G⊢s1 −(if the-Bool b then e1 else e2)−≻v−n→ s2
        using normal-s0 by (fastforce elim: evaln-elim-cases)
      from valid-e0 P valid-A conf-s0 eval-e0 wt-e0 da-e0
      obtain P' ⌊b⌋e s1 Z and conf-s1: s1::≼(G,L)
        by (rule validE)
      hence P': ⋀ v. (P'←=(the-Bool b)) v s1 Z
        by (cases normal s1) auto
      have Q ⌊v⌋e s2 Z
      proof (cases normal s1)
        case True
        note normal-s1=this
        from wt-e1 wt-e2 obtain T' where
```

```
          wt-then-else:
          (|prg=G,cls=accC,lcl=L|)⊢(if the-Bool b then e1 else e2)::−T'
          by (cases the-Bool b) simp+
        have s0-s1: dom (locals (store s0))
                    ∪ assigns-if (the-Bool b) e0 ⊆ dom (locals (store s1))
        proof −
          from eval-e0
          have eval-e0': G⊢s0 −e0−≻b→ s1
            by (rule evaln-eval)
          hence
            dom (locals (store s0)) ⊆ dom (locals (store s1))
            by (rule dom-locals-eval-mono-elim)
          moreover
          from eval-e0' True wt-e0
          have assigns-if (the-Bool b) e0 ⊆ dom (locals (store s1))
            by (rule assigns-if-good-approx')
          ultimately show ?thesis by (rule Un-least)
        qed
        obtain E' where
          da-then-else:
          (|prg=G,cls=accC,lcl=L|)
             ⊢dom (locals (store s1))»⟨if the-Bool b then e1 else e2⟩e» E'
        proof (cases the-Bool b)
          case True
          with that da-e1 s0-s1 show ?thesis
            by simp (erule da-weakenE,auto)
        next
          case False
          with that da-e2 s0-s1 show ?thesis
            by simp (erule da-weakenE,auto)
        qed
        with valid-then-else P' valid-A conf-s1 eval-then-else wt-then-else
        show ?thesis
          by (rule validE)
      next
        case False
        with valid-then-else P' valid-A conf-s1 eval-then-else
        show ?thesis
          by (cases rule: validE) iprover+
      qed
      moreover
      from eval wt da conf-s0 wf
      have s2::≼(G, L)
        by (rule evaln-type-sound [elim-format]) simp
      ultimately show ?thesis ..
    qed
  qed
next
  case (Call A P e Q args R mode statT mn pTs' S accC')
  note valid-e = ‹G,A|⊨::{ {Normal P} e−≻ {Q} }›
  have valid-args: ⋀ a. G,A|⊨::{ {Q←In1 a} args≐≻ {R a} }
    using Call.hyps by simp
  have valid-methd: ⋀ a vs invC declC l.
        G,A|⊨::{ {R a←In3 vs ∧.
                (λs. declC =
                    invocation-declclass G mode (store s) a statT
                     (|name = mn, parTs = pTs'|) ∧
                    invC = invocation-class mode (store s) a statT ∧
                    l = locals (store s)) ;.
```

```
            init-lvars G declC (|name = mn, parTs = pTs'|) mode a vs ∧.
            (λs. normal s ⟶ G⊢mode→invC≼statT)}
         Methd declC (|name=mn,parTs=pTs'|)−≻ {set-lvars l .; S} }
  using Call.hyps by simp
show G,A|⊨::{ {Normal P} {accC',statT,mode}e·mn( {pTs'}args)−≻ {S} }
proof (rule valid-expr-NormalI)
  fix n s0 L accC T E v s5 Y Z
  assume valid-A: ∀ t∈A. G⊨n::t
  assume conf-s0: s0::≼(G,L)
  assume normal-s0: normal s0
  assume wt: (|prg=G,cls=accC,lcl=L|)⊢{accC',statT,mode}e·mn( {pTs'}args)::−T
  assume da: (|prg=G,cls=accC,lcl=L|)⊢dom (locals (store s0))
              »⟨{accC',statT,mode}e·mn( {pTs'}args)⟩e» E
  assume eval: G⊢s0 −{accC',statT,mode}e·mn( {pTs'}args)−≻v−n→ s5
  assume P: (Normal P) Y s0 Z
  show S ⌊v⌋e s5 Z ∧ s5::≼(G, L)
  proof −
    from wt obtain pTs statDeclT statM where
             wt-e: (|prg=G,cls=accC,lcl=L|)⊢e::−RefT statT and
          wt-args: (|prg=G,cls=accC,lcl=L|)⊢args::≐pTs and
            statM: max-spec G accC statT (|name=mn,parTs=pTs|)
                   = {((statDeclT,statM),pTs')} and
             mode: mode = invmode statM e and
                T: T =(resTy statM) and
      eq-accC-accC': accC=accC'
      by cases fastforce+
    from da obtain C where
      da-e: (|prg=G,cls=accC,lcl=L|)⊢ (dom (locals (store s0)))»⟨e⟩e» C and
      da-args: (|prg=G,cls=accC,lcl=L|)⊢ nrm C »⟨args⟩l» E
      by cases simp
    from eval eq-accC-accC' obtain a s1 vs s2 s3 s3' s4 invDeclC where
      evaln-e: G⊢s0 −e−≻a−n→ s1 and
      evaln-args: G⊢s1 −args≐≻vs−n→ s2 and
      invDeclC: invDeclC = invocation-declclass
             G mode (store s2) a statT (|name=mn,parTs=pTs'|) and
      s3: s3 = init-lvars G invDeclC (|name=mn,parTs=pTs'|) mode a vs s2 and
      check: s3' = check-method-access G
                       accC' statT mode (|name = mn, parTs = pTs'|) a s3 and
      evaln-methd:
         G⊢s3' −Methd invDeclC (|name=mn,parTs=pTs'|)−≻v−n→ s4 and
      s5: s5=(set-lvars (locals (store s2))) s4
      using normal-s0 by (auto elim: evaln-elim-cases)

    from evaln-e
    have eval-e: G⊢s0 −e−≻a→ s1
      by (rule evaln-eval)

    from eval-e - wt-e wf
    have s1-no-return: abrupt s1 ≠ Some (Jump Ret)
      by (rule eval-expression-no-jump
              [where ?Env=(|prg=G,cls=accC,lcl=L|),simplified])
         (insert normal-s0,auto)

    from valid-e P valid-A conf-s0 evaln-e wt-e da-e
    obtain Q ⌊a⌋e s1 Z and conf-s1: s1::≼(G,L)
      by (rule validE)
    hence Q: ⋀ v. (Q←In1 a) v s1 Z
      by simp
    obtain
```

```
    R: (R a) ⌊vs⌋l s2 Z and
    conf-s2: s2::≼(G,L) and
    s2-no-return: abrupt s2 ≠ Some (Jump Ret)
  proof (cases normal s1)
    case True
    obtain E' where
      da-args':
      (|prg=G,cls=accC,lcl=L|)⊢ dom (locals (store s1)) »⟨args⟩l» E'
    proof −
      from evaln-e wt-e da-e wf True
      have nrm C ⊆  dom (locals (store s1))
        by (cases rule: da-good-approx-evalnE) iprover
      with da-args show thesis
        by (rule da-weakenE) (rule that)
    qed
    with valid-args Q valid-A conf-s1 evaln-args wt-args
    obtain (R a) ⌊vs⌋l s2 Z s2::≼(G,L)
      by (rule validE)
    moreover
    from evaln-args
    have e: G⊢s1 −args≐≻vs→ s2
      by (rule evaln-eval)
    from this s1-no-return wt-args wf
    have abrupt s2 ≠ Some (Jump Ret)
      by (rule eval-expression-list-no-jump
             [where ?Env=(|prg=G,cls=accC,lcl=L|),simplified])
    ultimately show ?thesis ..
  next
    case False
    with valid-args Q valid-A conf-s1 evaln-args
    obtain (R a) ⌊vs⌋l s2 Z s2::≼(G,L)
      by (cases rule: validE) iprover+
    moreover
    from False evaln-args have s2=s1
      by auto
    with s1-no-return have abrupt s2 ≠ Some (Jump Ret)
      by simp
    ultimately show ?thesis ..
  qed

  obtain invC where
    invC: invC = invocation-class mode (store s2) a statT
    by simp
  with s3
  have invC': invC = (invocation-class mode (store s3) a statT)
    by (cases s2,cases mode) (auto simp add: init-lvars-def2 )
  obtain l where
    l: l = locals (store s2)
    by simp

  from eval wt da conf-s0 wf
  have conf-s5: s5::≼(G, L)
    by (rule evaln-type-sound [elim-format]) simp
  let PROP ?R = ⋀ v.
       (R a←In3 vs ∧.
           (λs. invDeclC = invocation-declclass G mode (store s) a statT
                           (|name = mn, parTs = pTs'|) ∧
                 invC = invocation-class mode (store s) a statT ∧
                    l = locals (store s)) ;.
```

```
          init-lvars G invDeclC (|name = mn, parTs = pTs'|) mode a vs ∧.
          (λs. normal s ⟶ G⊢mode→invC≼statT)
        ) v s3' Z
{
  assume abrupt-s3: ¬ normal s3
  have S ⌊v⌋e s5 Z
  proof −
    from abrupt-s3 check have eq-s3'-s3: s3'=s3
      by (auto simp add: check-method-access-def Let-def)
    with R s3 invDeclC invC l abrupt-s3
    have R': PROP ?R
      by auto
    have conf-s3': s3'::≼(G, Map.empty)

    proof −
      from s2-no-return s3
      have abrupt s3 ≠ Some (Jump Ret)
        by (cases s2) (auto simp add: init-lvars-def2 split: if-split-asm)
      moreover
      obtain abr2 str2 where s2: s2=(abr2,str2)
        by (cases s2)
      from s3 s2 conf-s2 have (abrupt s3,str2)::≼(G, L)
        by (auto simp add: init-lvars-def2 split: if-split-asm)
      ultimately show ?thesis
        using s3 s2 eq-s3'-s3
        apply (simp add: init-lvars-def2)
        apply (rule conforms-set-locals [OF - wlconf-empty])
        by auto
    qed
    from valid-methd R' valid-A conf-s3' evaln-methd abrupt-s3 eq-s3'-s3
    have (set-lvars l .; S) ⌊v⌋e s4 Z
      by (cases rule: validE) simp+
    with s5 l show ?thesis
      by simp
  qed
} note abrupt-s3-lemma = this

have S ⌊v⌋e s5 Z
proof (cases normal s2)
  case False
  with s3 have abrupt-s3: ¬ normal s3
    by (cases s2) (simp add: init-lvars-def2)
  thus ?thesis
    by (rule abrupt-s3-lemma)
next
  case True
  note normal-s2 = this
  with evaln-args
  have normal-s1: normal s1
    by (rule evaln-no-abrupt)
  obtain E' where
    da-args':
    (|prg=G,cls=accC,lcl=L|)⊢ dom (locals (store s1)) »⟨args⟩l» E'
  proof −
    from evaln-e wt-e da-e wf normal-s1
    have nrm C ⊆ dom (locals (store s1))
      by (cases rule: da-good-approx-evalnE) iprover
    with da-args show thesis
      by (rule da-weakenE) (rule that)
```

qed
from *evaln-args*
have *eval-args*: $G \vdash s1 \ -args \dot{=}\succ vs \rightarrow \ s2$
  by (*rule evaln-eval*)
from *evaln-e wt-e da-e conf-s0 wf*
have *conf-a*: G, *store* $s1 \vdash a{::}\preceq RefT\ statT$
  by (*rule evaln-type-sound* [*elim-format*]) (*insert normal-s1*,*simp*)
with *normal-s1 normal-s2 eval-args*
have *conf-a-s2*: G, *store* $s2 \vdash a{::}\preceq RefT\ statT$
  by (*auto dest*: *eval-gext*)
from *evaln-args wt-args da-args′ conf-s1 wf*
have *conf-args*: *list-all2* (*conf G* (*store s2*)) *vs pTs*
  by (*rule evaln-type-sound* [*elim-format*]) (*insert normal-s2*,*simp*)
from *statM*
obtain
  statM′: (*statDeclT*,*statM*)∈*mheads G accC statT* (|*name*=*mn*,*parTs*=*pTs′*|)
  and
  pTs-widen: $G \vdash pTs[\preceq]pTs'$
  by (*blast dest*: *max-spec2mheads*)
show *?thesis*
proof (*cases normal s3*)
  case *False*
  thus *?thesis*
    by (*rule abrupt-s3-lemma*)
next
  case *True*
  note *normal-s3* = *this*
  with *s3* **have** *notNull*: *mode* = *IntVir* ⟶ *a* ≠ *Null*
    by (*cases s2*) (*auto simp add*: *init-lvars-def2*)
  from *conf-s2 conf-a-s2 wf notNull invC*
  have *dynT-prop*: $G \vdash mode \rightarrow invC \preceq statT$
    by (*cases s2*) (*auto intro*: *DynT-propI*)

  with *wt-e statM′ invC mode wf*
  obtain *dynM* **where**
    dynM: *dynlookup G statT invC* (|*name*=*mn*,*parTs*=*pTs′*|) = *Some dynM* **and**
    acc-dynM: $G \vdash Methd$ (|*name*=*mn*,*parTs*=*pTs′*|) *dynM*
                    in invC dyn-accessible-from accC
    by (*force dest*!: *call-access-ok*)
  with *invC′ check eq-accC-accC′*
  have *eq-s3′-s3*: *s3′*=*s3*
    by (*auto simp add*: *check-method-access-def Let-def*)

  with *dynT-prop R s3 invDeclC invC l*
  have *R′*: *PROP ?R*
    by *auto*

  from *dynT-prop wf wt-e statM′ mode invC invDeclC dynM*
  obtain
    dynM: *dynlookup G statT invC* (|*name*=*mn*,*parTs*=*pTs′*|) = *Some dynM* **and**
    wf-dynM: *wf-mdecl G invDeclC* ((|*name*=*mn*,*parTs*=*pTs′*|),*mthd dynM*) **and**
      dynM′: *methd G invDeclC* (|*name*=*mn*,*parTs*=*pTs′*|) = *Some dynM* **and**
    iscls-invDeclC: *is-class G invDeclC* **and**
         invDeclC′: *invDeclC* = *declclass dynM* **and**
      invC-widen: $G \vdash invC \preceq_C invDeclC$ **and**
     resTy-widen: $G \vdash resTy\ dynM \preceq resTy\ statM$ **and**
    is-static-eq: *is-static dynM* = *is-static statM* **and**
    involved-classes-prop:
     (*if invmode statM e* = *IntVir*

```
    then ∀ statC. statT = ClassT statC ⟶ G⊢invC≼C statC
    else ((∃ statC. statT = ClassT statC ∧ G⊢statC≼C invDeclC) ∨
          (∀ statC. statT ≠ ClassT statC ∧ invDeclC = Object)) ∧
           statDeclT = ClassT invDeclC)
  by (cases rule: DynT-mheadsE) simp
obtain L' where
  L':L'=(λ k.
        (case k of
           EName e
           ⇒ (case e of
                 VNam v
                 ⇒((table-of (lcls (mbody (mthd dynM))))
                     (pars (mthd dynM)[↦]pTs')) v
               | Res ⇒ Some (resTy dynM))
         | This ⇒ if is-static statM
                    then None else Some (Class invDeclC)))
  by simp
from wf-dynM [THEN wf-mdeclD1, THEN conjunct1] normal-s2 conf-s2 wt-e
  wf eval-args conf-a mode notNull wf-dynM involved-classes-prop
have conf-s3: s3::≼(G,L')
  apply −

  apply (drule conforms-init-lvars [of G invDeclC
          (|name=mn,parTs=pTs'|) dynM store s2 vs pTs abrupt s2
          L statT invC a (statDeclT,statM) e])
  apply (rule wf)
  apply (rule conf-args)
  apply (simp add: pTs-widen)
  apply (cases s2,simp)
  apply (rule dynM')
  apply (force dest: ty-expr-is-type)
  apply (rule invC-widen)
  apply (force dest: eval-gext)
  apply simp
  apply simp
  apply (simp add: invC)
  apply (simp add: invDeclC)
  apply (simp add: normal-s2)
  apply (cases s2, simp add: L' init-lvars-def2 s3
                cong add: lname.case-cong ename.case-cong)
  done
with eq-s3'-s3 have conf-s3': s3'::≼(G,L') by simp
from is-static-eq wf-dynM L'
obtain mthdT where
  (|prg=G,cls=invDeclC,lcl=L'|)
    ⊢Body invDeclC (stmt (mbody (mthd dynM)))::−mthdT and
  mthdT-widen: G⊢mthdT≼resTy dynM
  by − (drule wf-mdecl-bodyD,
        auto simp add: callee-lcl-def
             cong add: lname.case-cong ename.case-cong)
with dynM' iscls-invDeclC invDeclC'
have
  wt-methd:
  (|prg=G,cls=invDeclC,lcl=L'|)
    ⊢(Methd invDeclC (|name = mn, parTs = pTs'|))::−mthdT
  by (auto intro: wt.Methd)
obtain M where
  da-methd:
  (|prg=G,cls=invDeclC,lcl=L'|)
```

```
   ⊢ dom (locals (store s3'))
        »⟨Methd invDeclC (|name=mn,parTs=pTs'|)⟩ₑ» M
proof -
  from wf-dynM
  obtain M' where
    da-body:
    (|prg=G, cls=invDeclC
     ,lcl=callee-lcl invDeclC (|name = mn, parTs = pTs'|) (mthd dynM)
     |) ⊢ parameters (mthd dynM) »⟨stmt (mbody (mthd dynM))⟩» M' and
    res: Result ∈ nrm M'
    by (rule wf-mdeclE) iprover
  from da-body is-static-eq L' have
    (|prg=G, cls=invDeclC,lcl=L'|)
       ⊢ parameters (mthd dynM) »⟨stmt (mbody (mthd dynM))⟩» M'
    by (simp add: callee-lcl-def
        cong add: lname.case-cong ename.case-cong)
  moreover have parameters (mthd dynM) ⊆ dom (locals (store s3'))
  proof -
    from is-static-eq
    have (invmode (mthd dynM) e) = (invmode statM e)
      by (simp add: invmode-def)
    moreover
    have length (pars (mthd dynM)) = length vs
    proof -
      from normal-s2 conf-args
      have length vs = length pTs
        by (simp add: list-all2-iff)
      also from pTs-widen
      have ... = length pTs'
        by (simp add: widens-def list-all2-iff)
      also from wf-dynM
      have ... = length (pars (mthd dynM))
        by (simp add: wf-mdecl-def wf-mhead-def)
      finally show ?thesis ..
    qed
    moreover note s3 dynM' is-static-eq normal-s2 mode
    ultimately
    have parameters (mthd dynM) = dom (locals (store s3))
      using dom-locals-init-lvars
        [of mthd dynM G invDeclC (|name=mn,parTs=pTs'|) vs e a s2]
      by simp
    thus ?thesis using eq-s3'-s3 by simp
  qed
  ultimately obtain M2 where
    da:
    (|prg=G, cls=invDeclC,lcl=L'|)
      ⊢ dom (locals (store s3')) »⟨stmt (mbody (mthd dynM))⟩» M2 and
    M2: nrm M' ⊆ nrm M2
    by (rule da-weakenE)
  from res M2 have Result ∈ nrm M2
    by blast
  moreover from wf-dynM
  have jumpNestingOkS {Ret} (stmt (mbody (mthd dynM)))
    by (rule wf-mdeclE)
  ultimately
  obtain M3 where
    (|prg=G, cls=invDeclC,lcl=L'|) ⊢ dom (locals (store s3'))
          »⟨Body (declclass dynM) (stmt (mbody (mthd dynM)))⟩» M3
    using da
```

```
          by (iprover intro: da.Body assigned.select-convs)
        from - this [simplified]
        show thesis
          by (rule da.Methd [simplified,elim-format])
             (auto intro: dynM′ that)
      qed
      from valid-methd R′ valid-A conf-s3′ evaln-methd wt-methd da-methd
      have (set-lvars l .; S) ⌊v⌋e s4 Z
        by (cases rule: validE) iprover+
      with s5 l show ?thesis
        by simp
    qed
  qed
  with conf-s5 show ?thesis by iprover
 qed
qed
next
 case (Methd A P Q ms)
 note valid-body = ‹G,A ∪ {{P} Methd−≻ {Q} | ms}|⊨::{{P} body G−≻ {Q} | ms}›
 show G,A|⊨::{{P} Methd−≻ {Q} | ms}
   by (rule Methd-sound) (rule Methd.hyps)
next
 case (Body A P D Q c R)
 note valid-init = ‹G,A|⊨::{ {Normal P} .Init D. {Q} }›
 note valid-c = ‹G,A|⊨::{ {Q} .c.
              {λs.. abupd (absorb Ret) .; R←In1 (the (locals s Result))} }›
 show G,A|⊨::{ {Normal P} Body D c−≻ {R} }
 proof (rule valid-expr-NormalI)
   fix n s0 L accC T E v s4 Y Z
   assume valid-A: ∀ t∈A. G⊨n::t
   assume conf-s0: s0::≼(G,L)
   assume normal-s0: normal s0
   assume wt: ⦇prg=G,cls=accC,lcl=L⦈⊢Body D c::−T
   assume da: ⦇prg=G,cls=accC,lcl=L⦈⊢dom (locals (store s0))»⟨Body D c⟩e»E
   assume eval: G⊢s0 −Body D c−≻v−n→ s4
   assume P: (Normal P) Y s0 Z
   show R ⌊v⌋e s4 Z ∧ s4::≼(G, L)
   proof −
     from wt obtain
       iscls-D: is-class G D and
       wt-init: ⦇prg=G,cls=accC,lcl=L⦈⊢Init D::√ and
       wt-c: ⦇prg=G,cls=accC,lcl=L⦈⊢c::√
       by cases auto
     obtain I where
       da-init:⦇prg=G,cls=accC,lcl=L⦈ ⊢ dom (locals (store s0)) »⟨Init D⟩s» I
       by (auto intro: da-Init [simplified] assigned.select-convs)
     from da obtain C where
       da-c: ⦇prg=G,cls=accC,lcl=L⦈⊢ (dom (locals (store s0)))»⟨c⟩s» C and
       jmpOk: jumpNestingOkS {Ret} c
       by cases simp
     from eval obtain s1 s2 s3 where
       eval-init: G⊢s0 −Init D−n→ s1 and
       eval-c: G⊢s1 −c−n→ s2 and
       v: v = the (locals (store s2) Result) and
       s3: s3 =(if ∃ l. abrupt s2 = Some (Jump (Break l)) ∨
                        abrupt s2 = Some (Jump (Cont l))
                then abupd (λx. Some (Error CrossMethodJump)) s2 else s2)and
       s4: s4 = abupd (absorb Ret) s3
       using normal-s0 by (fastforce elim: evaln-elim-cases)
```

obtain C' **where**
da-c': (|*prg*=*G*,*cls*=*accC*,*lcl*=*L*|)⊢ (*dom* (*locals* (*store s1*)))»⟨*c*⟩$_s$» C'
proof −
from *eval-init*
have (*dom* (*locals* (*store s0*))) ⊆ (*dom* (*locals* (*store s1*)))
by (*rule dom-locals-evaln-mono-elim*)
with *da-c* **show** *thesis* **by** (*rule da-weakenE*) (*rule that*)
qed
from *valid-init P valid-A conf-s0 eval-init wt-init da-init*
obtain *Q*: *Q* ♢ *s1 Z* **and** *conf-s1*: *s1*::≼(*G*,*L*)
by (*rule validE*)
from *valid-c Q valid-A conf-s1 eval-c wt-c da-c'*
have *R*: (λ*s*.. *abupd* (*absorb Ret*) .; *R*←*In1* (*the* (*locals s Result*)))
♢ *s2 Z*
by (*rule validE*)
have *s3*=*s2*
proof −
from *eval-init* [*THEN evaln-eval*] *wf*
have *s1-no-jmp*: ⋀ *j*. *abrupt s1* ≠ *Some* (*Jump j*)
by − (*rule eval-statement-no-jump* [*OF* - - - *wt-init*],
insert normal-s0,*auto*)
from *eval-c* [*THEN evaln-eval*] - *wt-c wf*
have ⋀ *j*. *abrupt s2* = *Some* (*Jump j*) ⟹ *j*=*Ret*
by (*rule jumpNestingOk-evalE*) (*auto intro*: *jmpOk simp add*: *s1-no-jmp*)
moreover note *s3*
ultimately show *?thesis*
by (*force split*: *if-split*)
qed
with *R v s4*
have *R* ⌊*v*⌋$_e$ *s4 Z*
by *simp*
moreover
from *eval wt da conf-s0 wf*
have *s4*::≼(*G*, *L*)
by (*rule evaln-type-sound* [*elim-format*]) *simp*
ultimately show *?thesis* **..**
qed
qed
next
case (*Nil A P*)
show *G*,*A*|⊨::{ {*Normal* (*P*←⌊[]⌋$_l$)} []≐≻ {*P*} }
proof (*rule valid-expr-list-NormalI*)
fix *s0 s1 vs n L Y Z*
assume *conf-s0*: *s0*::≼(*G*,*L*)
assume *normal-s0*: *normal s0*
assume *eval*: *G*⊢*s0* −[]≐≻*vs*−*n*→ *s1*
assume *P*: (*Normal* (*P*←⌊[]⌋$_l$)) *Y s0 Z*
show *P* ⌊*vs*⌋$_l$ *s1 Z* ∧ *s1*::≼(*G*, *L*)
proof −
from *eval* **obtain** *vs*=[] *s1*=*s0*
using *normal-s0* **by** (*auto elim*: *evaln-elim-cases*)
with *P conf-s0* **show** *?thesis*
by *simp*
qed
qed
next
case (*Cons A P e Q es R*)
note *valid-e* = ‹*G*,*A*|⊨::{ {*Normal P*} *e*−≻ {*Q*} }›
have *valid-es*: ⋀ *v*. *G*,*A*|⊨::{ {*Q*←⌊*v*⌋$_e$} *es*≐≻ {λ*Vals*:*vs*:. *R*←⌊(*v* # *vs*)⌋$_l$} }

```
    using Cons.hyps by simp
  show G,A|⊨::{ {Normal P} e # es≐≻ {R} }
  proof (rule valid-expr-list-NormalI)
    fix n s0 L accC T E v s2 Y Z
    assume valid-A: ∀ t∈A. G⊨n::t
    assume conf-s0: s0::≼(G,L)
    assume normal-s0: normal s0
    assume wt: (|prg=G,cls=accC,lcl=L|)⊢e # es::≐T
    assume da: (|prg=G,cls=accC,lcl=L|)⊢dom (locals (store s0)) »⟨e # es⟩ₗ» E
    assume eval: G⊢s0 −e # es≐≻v−n→ s2
    assume P: (Normal P) Y s0 Z
    show R ⌊v⌋ₗ s2 Z ∧ s2::≼(G, L)
    proof −
      from wt obtain eT esT where
        wt-e: (|prg=G,cls=accC,lcl=L|)⊢e::−eT and
        wt-es: (|prg=G,cls=accC,lcl=L|)⊢es::≐esT
        by cases simp
      from da obtain E1 where
        da-e: (|prg=G,cls=accC,lcl=L|)⊢ (dom (locals (store s0)))»⟨e⟩ₑ» E1 and
        da-es: (|prg=G,cls=accC,lcl=L|)⊢ nrm E1 »⟨es⟩ₗ» E
        by cases simp
      from eval obtain s1 ve vs where
        eval-e: G⊢s0 −e−≻ve−n→ s1 and
        eval-es: G⊢s1 −es≐≻vs−n→ s2 and
        v: v=ve#vs
        using normal-s0 by (fastforce elim: evaln-elim-cases)
      from valid-e P valid-A conf-s0 eval-e wt-e da-e
      obtain Q: Q ⌊ve⌋ₑ s1 Z and conf-s1: s1::≼(G,L)
        by (rule validE)
      from Q have Q′: ⋀ v. (Q←⌊ve⌋ₑ) v s1 Z
        by simp
      have (λVals:vs:. R←⌊(ve # vs)⌋ₗ) ⌊vs⌋ₗ s2 Z
      proof (cases normal s1)
        case True
        obtain E′ where
          da-es′: (|prg=G,cls=accC,lcl=L|)⊢ dom (locals (store s1)) »⟨es⟩ₗ» E′
        proof −
          from eval-e wt-e da-e wf True
          have nrm E1 ⊆ dom (locals (store s1))
            by (cases rule: da-good-approx-evalnE) iprover
          with da-es show thesis
            by (rule da-weakenE) (rule that)
        qed
        from valid-es Q′ valid-A conf-s1 eval-es wt-es da-es′
        show ?thesis
          by (rule validE)
      next
        case False
        with valid-es Q′ valid-A conf-s1 eval-es
        show ?thesis
          by (cases rule: validE) iprover+
      qed
      with v have R ⌊v⌋ₗ s2 Z
        by simp
      moreover
      from eval wt da conf-s0 wf
      have s2::≼(G, L)
        by (rule evaln-type-sound [elim-format]) simp
      ultimately show ?thesis ..
```

```
    qed
  qed
next
  case (Skip A P)
  show G,A|⊨::{ {Normal (P←◇)} .Skip. {P} }
  proof (rule valid-stmt-NormalI)
    fix s0 s1 n L Y Z
    assume conf-s0: s0::≼(G,L)
    assume normal-s0: normal s0
    assume eval: G⊢s0 −Skip−n→ s1
    assume P: (Normal (P←◇)) Y s0 Z
    show P ◇ s1 Z ∧ s1::≼(G, L)
    proof −
      from eval obtain s1=s0
        using normal-s0 by (fastforce elim: evaln-elim-cases)
      with P conf-s0 show ?thesis
        by simp
    qed
  qed
next
  case (Expr A P e Q)
  note valid-e = ‹G,A|⊨::{ {Normal P} e−≻ {Q←◇} }›
  show G,A|⊨::{ {Normal P} .Expr e. {Q} }
  proof (rule valid-stmt-NormalI)
    fix n s0 L accC C s1 Y Z
    assume valid-A: ∀ t∈A. G⊨n::t
    assume conf-s0: s0::≼(G,L)
    assume normal-s0: normal s0
    assume wt: ⦇prg=G,cls=accC,lcl=L⦈⊢Expr e::√
    assume da: ⦇prg=G,cls=accC,lcl=L⦈⊢dom (locals (store s0)) »⟨Expr e⟩ₛ» C
    assume eval: G⊢s0 −Expr e−n→ s1
    assume P: (Normal P) Y s0 Z
    show Q ◇ s1 Z ∧ s1::≼(G, L)
    proof −
      from wt obtain eT where
        wt-e: ⦇prg = G, cls = accC, lcl = L⦈⊢e::−eT
        by cases simp
      from da obtain E where
        da-e: ⦇prg=G,cls=accC, lcl=L⦈⊢dom (locals (store s0))»⟨e⟩ₑ»E
        by cases simp
      from eval obtain v where
        eval-e: G⊢s0 −e−≻v−n→ s1
        using normal-s0 by (fastforce elim: evaln-elim-cases)
      from valid-e P valid-A conf-s0 eval-e wt-e da-e
      obtain Q: (Q←◇) ⌊v⌋ₑ s1 Z and s1::≼(G,L)
        by (rule validE)
      thus ?thesis by simp
    qed
  qed
next
  case (Lab A P c l Q)
  note valid-c = ‹G,A|⊨::{ {Normal P} .c. {abupd (absorb l) .; Q} }›
  show G,A|⊨::{ {Normal P} .l• c. {Q} }
  proof (rule valid-stmt-NormalI)
    fix n s0 L accC C s2 Y Z
    assume valid-A: ∀ t∈A. G⊨n::t
    assume conf-s0: s0::≼(G,L)
    assume normal-s0: normal s0
    assume wt: ⦇prg=G,cls=accC,lcl=L⦈⊢l• c::√
```

```
    assume da: (|prg=G,cls=accC,lcl=L|)⊢dom (locals (store s0)) »⟨l• c⟩s» C
    assume eval: G⊢s0 −l• c−n→ s2
    assume P: (Normal P) Y s0 Z
    show Q ◇ s2 Z ∧ s2::≼(G, L)
    proof −
      from wt obtain
        wt-c: (|prg = G, cls = accC, lcl = L|)⊢c::√
        by cases simp
      from da obtain E where
        da-c: (|prg=G,cls=accC, lcl=L|)⊢dom (locals (store s0))»⟨c⟩s»E
        by cases simp
      from eval obtain s1 where
        eval-c: G⊢s0 −c−n→ s1 and
        s2: s2 = abupd (absorb l) s1
        using normal-s0 by (fastforce elim: evaln-elim-cases)
      from valid-c P valid-A conf-s0 eval-c wt-c da-c
      obtain Q: (abupd (absorb l) .; Q) ◇ s1 Z
        by (rule validE)
      with s2 have Q ◇ s2 Z
        by simp
      moreover
      from eval wt da conf-s0 wf
      have s2::≼(G, L)
        by (rule evaln-type-sound [elim-format]) simp
      ultimately show ?thesis ..
    qed
  qed
next
  case (Comp A P c1 Q c2 R)
  note valid-c1 = ‹G,A|⊨::{ {Normal P} .c1. {Q} }›
  note valid-c2 = ‹G,A|⊨::{ {Q} .c2. {R} }›
  show G,A|⊨::{ {Normal P} .c1;; c2. {R} }
  proof (rule valid-stmt-NormalI)
    fix n s0 L accC C s2 Y Z
    assume valid-A: ∀ t∈A. G|⊨n::t
    assume conf-s0:  s0::≼(G,L)
    assume normal-s0: normal s0
    assume wt: (|prg=G,cls=accC,lcl=L|)⊢(c1;; c2)::√
    assume da: (|prg=G,cls=accC,lcl=L|)⊢dom (locals (store s0))»⟨c1;;c2⟩s»C
    assume eval: G⊢s0 −c1;; c2−n→ s2
    assume P: (Normal P) Y s0 Z
    show R ◇ s2 Z ∧ s2::≼(G,L)
    proof −
      from eval  obtain s1 where
        eval-c1: G⊢s0 −c1 −n→ s1 and
        eval-c2: G⊢s1 −c2 −n→ s2
        using normal-s0 by (fastforce elim: evaln-elim-cases)
      from wt obtain
        wt-c1: (|prg = G, cls = accC, lcl = L|)⊢c1::√ and
        wt-c2: (|prg = G, cls = accC, lcl = L|)⊢c2::√
        by cases simp
      from da obtain C1 C2 where
        da-c1: (|prg=G,cls=accC,lcl=L|)⊢ dom (locals (store s0)) »⟨c1⟩s» C1 and
        da-c2: (|prg=G,cls=accC,lcl=L|)⊢nrm C1 »⟨c2⟩s» C2
        by cases simp
      from valid-c1 P valid-A conf-s0 eval-c1 wt-c1 da-c1
      obtain Q: Q ◇ s1 Z and conf-s1: s1::≼(G,L)
        by (rule validE)
      have R ◇ s2 Z
```

```
    proof (cases normal s1)
      case True
      obtain C2' where
        ⦇prg=G,cls=accC,lcl=L⦈⊢ dom (locals (store s1)) »⟨c2⟩ₛ» C2'
      proof −
        from eval-c1 wt-c1 da-c1 wf True
        have nrm C1 ⊆ dom (locals (store s1))
          by (cases rule: da-good-approx-evalnE) iprover
        with da-c2 show thesis
          by (rule da-weakenE) (rule that)
      qed
      with valid-c2 Q valid-A conf-s1 eval-c2 wt-c2
      show ?thesis
        by (rule validE)
    next
      case False
      from valid-c2 Q valid-A conf-s1 eval-c2 False
      show ?thesis
        by (cases rule: validE) iprover+
    qed
    moreover
    from eval wt da conf-s0 wf
    have s2::≼(G, L)
      by (rule evaln-type-sound [elim-format]) simp
    ultimately show ?thesis ..
  qed
 qed
next
 case (If A P e P' c1 c2 Q)
 note valid-e = ‹G,A|⊨::{ {Normal P} e−≻ {P'} }›
 have valid-then-else: ⋀ b. G,A|⊨::{ {P'←=b} .(if b then c1 else c2). {Q} }
   using If.hyps by simp
 show G,A|⊨::{ {Normal P} .If(e) c1 Else c2. {Q} }
 proof (rule valid-stmt-NormalI)
   fix n s0 L accC C s2 Y Z
   assume valid-A: ∀ t∈A. G⊨n::t
   assume conf-s0:  s0::≼(G,L)
   assume normal-s0: normal s0
   assume wt: ⦇prg=G,cls=accC,lcl=L⦈⊢If(e) c1 Else c2::√
   assume da: ⦇prg=G,cls=accC,lcl=L⦈
              ⊢dom (locals (store s0))»⟨If(e) c1 Else c2⟩ₛ»C
   assume eval: G⊢s0 −If(e) c1 Else c2−n→ s2
   assume P: (Normal P) Y s0 Z
   show Q ◇ s2 Z ∧ s2::≼(G,L)
   proof −
     from eval obtain b s1 where
       eval-e: G⊢s0 −e−≻b−n→ s1 and
       eval-then-else: G⊢s1 −(if the-Bool b then c1 else c2)−n→ s2
       using normal-s0 by (auto elim: evaln-elim-cases)
     from wt obtain
       wt-e: ⦇prg=G, cls=accC, lcl=L⦈⊢e::−PrimT Boolean and
       wt-then-else: ⦇prg=G,cls=accC,lcl=L⦈⊢(if the-Bool b then c1 else c2)::√
       by cases (simp split: if-split)
     from da obtain E S where
       da-e: ⦇prg=G,cls=accC,lcl=L⦈⊢ dom (locals (store s0)) »⟨e⟩ₑ» E and
       da-then-else:
       ⦇prg=G,cls=accC,lcl=L⦈⊢
            (dom (locals (store s0)) ∪ assigns-if (the-Bool b) e)
              »⟨if the-Bool b then c1 else c2⟩ₛ» S
```

```
        by cases (cases the-Bool b,auto)
      from valid-e P valid-A conf-s0 eval-e wt-e da-e
      obtain P' ⌊b⌋e s1 Z and conf-s1: s1::≼(G,L)
        by (rule validE)
      hence P': ⋀v. (P'←=the-Bool b) v s1 Z
        by (cases normal s1) auto
      have Q ◇ s2 Z
      proof (cases normal s1)
        case True
        have s0-s1: dom (locals (store s0))
                    ∪ assigns-if (the-Bool b) e ⊆ dom (locals (store s1))
        proof −
          from eval-e
          have eval-e': G⊢s0 −e−≻b→ s1
            by (rule evaln-eval)
          hence
            dom (locals (store s0)) ⊆ dom (locals (store s1))
            by (rule dom-locals-eval-mono-elim)
          moreover
          from eval-e' True wt-e
          have assigns-if (the-Bool b) e ⊆ dom (locals (store s1))
            by (rule assigns-if-good-approx')
          ultimately show ?thesis by (rule Un-least)
        qed
        with da-then-else
        obtain S' where
          (|prg=G,cls=accC,lcl=L|)
             ⊢dom (locals (store s1))»⟨if the-Bool b then c1 else c2⟩s» S'
          by (rule da-weakenE)
        with valid-then-else P' valid-A conf-s1 eval-then-else wt-then-else
        show ?thesis
          by (rule validE)
      next
        case False
        with valid-then-else P' valid-A conf-s1 eval-then-else
        show ?thesis
          by (cases rule: validE) iprover+
      qed
      moreover
      from eval wt da conf-s0 wf
      have s2::≼(G, L)
        by (rule evaln-type-sound [elim-format]) simp
      ultimately show ?thesis ..
    qed
  qed
next
  case (Loop A P e P' c l)
  note valid-e = ‹G,A|⊨::{ {P} e−≻ {P'} }›
  note valid-c = ‹G,A|⊨::{ {Normal (P'←=True)}
                         .c.
                         {abupd (absorb (Cont l)) .; P} }›
  show G,A|⊨::{ {P} .l· While(e) c. {P'←=False↓=◇} }
  proof (rule valid-stmtI)
    fix n s0 L accC C s3 Y Z
    assume valid-A: ∀ t∈A. G⊨n::t
    assume conf-s0:  s0::≼(G,L)
    assume wt: normal s0 ⟹ (|prg=G,cls=accC,lcl=L|)⊢l· While(e) c::√
    assume da: normal s0 ⟹ (|prg=G,cls=accC,lcl=L|)
                  ⊢ dom (locals (store s0)) »⟨l· While(e) c⟩s» C
```

```
    assume eval: G⊢s0 −l· While(e) c−n→ s3
    assume P: P Y s0 Z
    show (P′←=False↓=♢) ♢ s3 Z ∧ s3::≼(G,L)
    proof −
```

— From the given hypothesises *valid-e* and *valid-c* we can only reach the state after unfolding the loop once, i.e. *P* ♢ *s2 Z*, where *s2* is the state after executing *c*. To gain validity of the further execution of while, to finally get (*P′*←=*False*↓=♢) ♢ *s3 Z* we have to get a hypothesis about the subsequent unfoldings (the whole loop again), too. We can achieve this, by performing induction on the evaluation relation, with all the necessary preconditions to apply *valid-e* and *valid-c* in the goal.

```
      {
        fix t s s′ v
        assume G⊢s −t≻−n→ (v, s′)
        hence ⋀ Y′ T E.
              ⟦t = ⟨l· While(e) c⟩s; ∀ t∈A. G⊨n::t; P Y′ s Z; s::≼(G, L);
               normal s ⟹ (|prg=G,cls=accC,lcl=L|)⊢t::T;
               normal s ⟹ (|prg=G,cls=accC,lcl=L|)⊢dom (locals (store s))»t»E
              ⟧⟹ (P′←=False↓=♢) v s′ Z
          (is PROP ?Hyp n t s v s′)
        proof (induct)
          case (Loop s0′ e′ b n′ s1′ c′ s2′ l′ s3′ Y′ T E)
          note while = ‹(⟨l′· While(e′) c′⟩s::term) = ⟨l· While(e) c⟩s›
          hence eqs: l′=l e′=e c′=c by simp-all
          note valid-A = ‹∀ t∈A. G⊨n′::t›
          note P = ‹P Y′ (Norm s0′) Z›
          note conf-s0′ = ‹Norm s0′::≼(G, L)›
          have wt: (|prg=G,cls=accC,lcl=L|)⊢⟨l· While(e) c⟩s::T
            using Loop.prems eqs by simp
          have da: (|prg=G,cls=accC,lcl=L|)⊢
                  dom (locals (store ((Norm s0′)::state)))»⟨l· While(e) c⟩s»E
            using Loop.prems eqs by simp
          have evaln-e: G⊢Norm s0′ −e−≻b−n′→ s1′
            using Loop.hyps eqs by simp
          show (P′←=False↓=♢) ♢ s3′ Z
          proof −
            from wt obtain
              wt-e: (|prg=G,cls=accC,lcl=L|)⊢e::−PrimT Boolean and
              wt-c: (|prg=G,cls=accC,lcl=L|)⊢c::√
              by cases (simp add: eqs)
            from da obtain E S where
              da-e: (|prg=G,cls=accC,lcl=L|)
                    ⊢ dom (locals (store ((Norm s0′)::state))) »⟨e⟩e» E and
              da-c: (|prg=G,cls=accC,lcl=L|)
                    ⊢ (dom (locals (store ((Norm s0′)::state)))
                          ∪ assigns-if True e) »⟨c⟩s» S
              by cases (simp add: eqs)
            from evaln-e
            have eval-e: G⊢Norm s0′ −e−≻b→ s1′
              by (rule evaln-eval)
            from valid-e P valid-A conf-s0′ evaln-e wt-e da-e
            obtain P′: P′ ⌊b⌋e s1′ Z and conf-s1′: s1′::≼(G,L)
              by (rule validE)
            show (P′←=False↓=♢) ♢ s3′ Z
            proof (cases normal s1′)
              case True
              note normal-s1′=this
              show ?thesis
              proof (cases the-Bool b)
                case True
                with P′ normal-s1′ have P″: (Normal (P′←=True)) ⌊b⌋e s1′ Z
```

```
    by auto
  from True Loop.hyps obtain
    eval-c: G⊢s1′ −c−n′→ s2′ and
    eval-while:
       G⊢abupd (absorb (Cont l)) s2′ −l· While(e) c−n′→ s3′
    by (simp add: eqs)
  from True Loop.hyps have
    hyp: PROP ?Hyp n′ ⟨l· While(e) c⟩ₛ
           (abupd (absorb (Cont l′)) s2′) ♢ s3′
    apply (simp only: True if-True eqs)
    apply (elim conjE)
    apply (tactic smp-tac context 3 1)
    apply fast
    done
  from eval-e
  have s0′-s1′: dom (locals (store ((Norm s0′)::state)))
                 ⊆ dom (locals (store s1′))
    by (rule dom-locals-eval-mono-elim)
  obtain S′ where
    da-c′:
     (|prg=G,cls=accC,lcl=L|)⊢(dom (locals (store s1′)))»⟨c⟩ₛ» S′
  proof −
    note s0′-s1′
    moreover
    from eval-e normal-s1′ wt-e
    have assigns-if True e ⊆ dom (locals (store s1′))
      by (rule assigns-if-good-approx′ [elim-format])
         (simp add: True)
    ultimately
    have dom (locals (store ((Norm s0′)::state)))
            ∪ assigns-if True e ⊆ dom (locals (store s1′))
      by (rule Un-least)
    with da-c show thesis
      by (rule da-weakenE) (rule that)
  qed
  with valid-c P′′ valid-A conf-s1′ eval-c wt-c
  obtain (abupd (absorb (Cont l)) .; P) ♢ s2′ Z and
    conf-s2′: s2′::≼(G,L)
    by (rule validE)
  hence P-s2′: P ♢ (abupd (absorb (Cont l)) s2′) Z
    by simp
  from conf-s2′
  have conf-absorb: abupd (absorb (Cont l)) s2′ ::≼(G, L)
    by (cases s2′) (auto intro: conforms-absorb)
  moreover
  obtain E′ where
    da-while′:
     (|prg=G,cls=accC,lcl=L|)⊢
       dom (locals(store (abupd (absorb (Cont l)) s2′)))
        »⟨l· While(e) c⟩ₛ» E′
  proof −
    note s0′-s1′
    also
    from eval-c
    have G⊢s1′ −c→ s2′
      by (rule evaln-eval)
    hence dom (locals (store s1′)) ⊆ dom (locals (store s2′))
      by (rule dom-locals-eval-mono-elim)
    also
```

```
          have ...⊆dom (locals (store (abupd (absorb (Cont l)) s2')))
            by simp
          finally
          have dom (locals (store ((Norm s0')::state))) ⊆ ... .
          with da show thesis
            by (rule da-weakenE) (rule that)
        qed
        from valid-A P-s2' conf-absorb wt da-while'
        show (P'←=False↓=◇) ◇ s3' Z
          using hyp by (simp add: eqs)
      next
        case False
        with Loop.hyps obtain s3'=s1'
          by simp
        with P' False show ?thesis
          by auto
      qed
    next
      case False
      note abnormal-s1'=this
      have s3'=s1'
      proof -
        from False obtain abr where abr: abrupt s1' = Some abr
          by (cases s1') auto
        from eval-e - wt-e wf
        have no-jmp: ⋀ j. abrupt s1' ≠ Some (Jump j)
          by (rule eval-expression-no-jump
                [where ?Env=(|prg=G,cls=accC,lcl=L|),simplified])
             simp
        show ?thesis
        proof (cases the-Bool b)
          case True
          with Loop.hyps obtain
            eval-c: G⊢s1' −c−n'→ s2' and
            eval-while:
             G⊢abupd (absorb (Cont l)) s2' −l• While(e) c−n'→ s3'
            by (simp add: eqs)
          from eval-c abr have s2'=s1' by auto
          moreover from calculation no-jmp
          have abupd (absorb (Cont l)) s2'=s2'
            by (cases s1') (simp add: absorb-def)
          ultimately show ?thesis
            using eval-while abr
            by auto
        next
          case False
          with Loop.hyps show ?thesis by simp
        qed
      qed
      with P' False show ?thesis
        by auto
    qed
  qed
next
  case (Abrupt abr s t' n' Y' T E)
  note t' = ‹t' = ⟨l• While(e) c⟩s›
  note conf = ‹(Some abr, s)::≼(G, L)›
  note P = ‹P Y' (Some abr, s) Z›
  note valid-A = ‹∀ t∈A. G⊨n'::t›
```

```
        show (P′←=False↓=◇) (undefined3 t′) (Some abr, s) Z
        proof −
          have eval-e:
            G⊢(Some abr,s) −⟨e⟩e≻−n′→ (undefined3 ⟨e⟩e,(Some abr,s))
            by auto
          from valid-e P valid-A conf eval-e
          have P′ (undefined3 ⟨e⟩e) (Some abr,s) Z
            by (cases rule: validE [where ?P=P]) simp+
          with t′ show ?thesis
            by auto
        qed
      qed simp-all
    } note generalized=this
    from eval - valid-A P conf-s0 wt da
    have (P′←=False↓=◇) ◇ s3 Z
      by (rule generalized) simp-all
    moreover
    have s3::≼(G, L)
    proof (cases normal s0)
      case True
      from eval wt [OF True] da [OF True] conf-s0 wf
      show ?thesis
        by (rule evaln-type-sound [elim-format]) simp
    next
      case False
      with eval have s3=s0
        by auto
      with conf-s0 show ?thesis
        by simp
    qed
    ultimately show ?thesis ..
  qed
 qed
next
 case (Jmp A j P)
 show G,A|⊨::{ {Normal (abupd (λa. Some (Jump j)) .; P←◇)} .Jmp j. {P} }
 proof (rule valid-stmt-NormalI)
  fix n s0 L accC C s1 Y Z
  assume valid-A: ∀ t∈A. G⊨n::t
  assume conf-s0: s0::≼(G,L)
  assume normal-s0: normal s0
  assume wt: ⦇prg=G,cls=accC,lcl=L⦈⊢Jmp j::√
  assume da: ⦇prg=G,cls=accC,lcl=L⦈
              ⊢dom (locals (store s0))»⟨Jmp j⟩s»C
  assume eval: G⊢s0 −Jmp j−n→ s1
  assume P: (Normal (abupd (λa. Some (Jump j)) .; P←◇)) Y s0 Z
  show P ◇ s1 Z ∧ s1::≼(G,L)
  proof −
    from eval obtain s where
      s: s0=Norm s s1=(Some (Jump j), s)
      using normal-s0 by (auto elim: evaln-elim-cases)
    with P have P ◇ s1 Z
      by simp
    moreover
    from eval wt da conf-s0 wf
    have s1::≼(G,L)
      by (rule evaln-type-sound [elim-format]) simp
    ultimately show ?thesis ..
  qed
```

```
  qed
next
  case (Throw A P e Q)
  note valid-e = ‹G,A|⊨::{ {Normal P} e−≻ {λVal:a:. abupd (throw a) .; Q←◇} }›
  show G,A|⊨::{ {Normal P} .Throw e. {Q} }
  proof (rule valid-stmt-NormalI)
    fix n s0 L accC C s2 Y Z
    assume valid-A: ∀t∈A. G⊨n::t
    assume conf-s0:  s0::≼(G,L)
    assume normal-s0: normal s0
    assume wt: (|prg=G,cls=accC,lcl=L|)⊢Throw e::√
    assume da: (|prg=G,cls=accC,lcl=L|)
                ⊢dom (locals (store s0))»⟨Throw e⟩s»C
    assume eval: G⊢s0 −Throw e−n→ s2
    assume P: (Normal P) Y s0 Z
    show Q ◇ s2 Z ∧ s2::≼(G,L)
    proof −
      from eval obtain s1 a where
        eval-e: G⊢s0 −e−≻a−n→ s1 and
        s2: s2 = abupd (throw a) s1
        using normal-s0 by (auto elim: evaln-elim-cases)
      from wt obtain T where
        wt-e: (|prg=G,cls=accC,lcl=L|)⊢e::−T
        by cases simp
      from da obtain E where
        da-e: (|prg=G,cls=accC,lcl=L|)⊢ dom (locals (store s0)) »⟨e⟩e» E
        by cases simp
      from valid-e P valid-A conf-s0 eval-e wt-e da-e
      obtain (λVal:a:. abupd (throw a) .; Q←◇) ⌊a⌋e s1 Z
        by (rule validE)
      with s2 have Q ◇ s2 Z
        by simp
      moreover
      from eval wt da conf-s0 wf
      have s2::≼(G,L)
        by (rule evaln-type-sound [elim-format]) simp
      ultimately show ?thesis ..
    qed
  qed
next
  case (Try A P c1 Q C vn c2 R)
  note valid-c1 = ‹G,A|⊨::{ {Normal P} .c1. {SXAlloc G Q} }›
  note valid-c2 = ‹G,A|⊨::{ {Q ∧. (λs. G,s⊢catch C) ;. new-xcpt-var vn}
                          .c2.
                          {R} }›
  note Q-R = ‹(Q ∧. (λs. ¬ G,s⊢catch C)) ⇒ R›
  show G,A|⊨::{ {Normal P} .Try c1 Catch(C vn) c2. {R} }
  proof (rule valid-stmt-NormalI)
    fix n s0 L accC E s3 Y Z
    assume valid-A: ∀t∈A. G⊨n::t
    assume conf-s0:  s0::≼(G,L)
    assume normal-s0: normal s0
    assume wt: (|prg=G,cls=accC,lcl=L|)⊢Try c1 Catch(C vn) c2::√
    assume da: (|prg=G,cls=accC,lcl=L|)
                ⊢dom (locals (store s0)) »⟨Try c1 Catch(C vn) c2⟩s» E
    assume eval: G⊢s0 −Try c1 Catch(C vn) c2−n→ s3
    assume P: (Normal P) Y s0 Z
    show R ◇ s3 Z ∧ s3::≼(G,L)
    proof −
```

```
from eval obtain s1 s2 where
  eval-c1: G⊢s0 −c1−n→ s1 and
  sxalloc: G⊢s1 −sxalloc→ s2 and
  s3: if G,s2⊢catch C
        then G⊢new-xcpt-var vn s2 −c2−n→ s3
        else s3 = s2
  using normal-s0 by (fastforce elim: evaln-elim-cases)
from wt obtain
  wt-c1: (|prg=G,cls=accC,lcl=L|)⊢c1::√ and
  wt-c2: (|prg=G,cls=accC,lcl=L(VName vn↦Class C)|)⊢c2::√
  by cases simp
from da obtain C1 C2 where
  da-c1: (|prg=G,cls=accC,lcl=L|)⊢ dom (locals (store s0)) »⟨c1⟩s» C1 and
  da-c2: (|prg=G,cls=accC,lcl=L(VName vn↦Class C)|)
         ⊢ (dom (locals (store s0)) ∪ {VName vn}) »⟨c2⟩s» C2
  by cases simp
from valid-c1 P valid-A conf-s0 eval-c1 wt-c1 da-c1
obtain sxQ: (SXAlloc G Q) ♢ s1 Z and conf-s1: s1::≼(G,L)
  by (rule validE)
from sxalloc sxQ
have Q: Q ♢ s2 Z
  by auto
have R ♢ s3 Z
proof (cases ∃ x. abrupt s1 = Some (Xcpt x))
  case False
  from sxalloc wf
  have s2=s1
    by (rule sxalloc-type-sound [elim-format])
      (insert False, auto split: option.splits abrupt.splits )
  with False
  have no-catch: ¬  G,s2⊢catch C
    by (simp add: catch-def)
  moreover
  from no-catch s3
  have s3=s2
    by simp
  ultimately show ?thesis
    using Q Q-R by simp
next
  case True
  note exception-s1 = this
  show ?thesis
  proof (cases G,s2⊢catch C)
    case False
    with s3
    have s3=s2
      by simp
    with False Q Q-R show ?thesis
      by simp
  next
    case True
    with s3 have eval-c2: G⊢new-xcpt-var vn s2 −c2−n→ s3
      by simp
    from conf-s1 sxalloc wf
    have conf-s2: s2::≼(G, L)
      by (auto dest: sxalloc-type-sound
               split: option.splits abrupt.splits)
    from exception-s1 sxalloc wf
    obtain a
```

```
          where xcpt-s2: abrupt s2 = Some (Xcpt (Loc a))
          by (auto dest!: sxalloc-type-sound
                     split: option.splits abrupt.splits)
        with True
        have G⊢obj-ty (the (globs (store s2) (Heap a)))≼Class C
          by (cases s2) simp
        with xcpt-s2 conf-s2 wf
        have conf-new-xcpt: new-xcpt-var vn s2 ::≼(G, L(VName vn↦Class C))
          by (auto dest: Try-lemma)
        obtain C2' where
          da-c2':
          (|prg=G,cls=accC,lcl=L(VName vn↦Class C)|)
            ⊢ (dom (locals (store (new-xcpt-var vn s2)))) »⟨c2⟩s» C2'
        proof -
          have (dom (locals (store s0)) ∪ {VName vn})
                  ⊆ dom (locals (store (new-xcpt-var vn s2)))
          proof -
            from eval-c1
            have dom (locals (store s0))
                    ⊆ dom (locals (store s1))
              by (rule dom-locals-evaln-mono-elim)
            also
            from sxalloc
            have ... ⊆ dom (locals (store s2))
              by (rule dom-locals-sxalloc-mono)
            also
            have ... ⊆ dom (locals (store (new-xcpt-var vn s2)))
              by (cases s2) (simp add: new-xcpt-var-def, blast)
            also
            have {VName vn} ⊆ ...
              by (cases s2) simp
            ultimately show ?thesis
              by (rule Un-least)
          qed
          with da-c2 show thesis
            by (rule da-weakenE) (rule that)
        qed
        from Q eval-c2 True
        have (Q ∧. (λs. G,s⊢catch C) ;. new-xcpt-var vn)
                ♢ (new-xcpt-var vn s2) Z
          by auto
        from valid-c2 this valid-A conf-new-xcpt eval-c2 wt-c2 da-c2'
        show R ♢ s3 Z
          by (rule validE)
      qed
    qed
    moreover
    from eval wt da conf-s0 wf
    have s3::≼(G,L)
      by (rule evaln-type-sound [elim-format]) simp
    ultimately show ?thesis ..
  qed
qed
next
  case (Fin A P c1 Q c2 R)
  note valid-c1 = ‹G,A|⊨::{ {Normal P} .c1. {Q} }›
  have valid-c2: ⋀ abr. G,A|⊨::{ {Q ∧. (λs. abr = fst s) ;. abupd (λx. None)}
                              .c2.
                              {abupd (abrupt-if (abr ≠ None) abr) .; R} }
```

using *Fin.hyps* **by** *simp*
show $G,A|\models::\{ \{Normal\ P\}\ .c1\ Finally\ c2.\ \{R\}\ \}$
proof (*rule valid-stmt-NormalI*)
fix *n s0 L accC E s3 Y Z*
assume *valid-A*: $\forall t \in A.\ G \models n::t$
assume *conf-s0*: $s0::\preceq(G,L)$
assume *normal-s0*: *normal s0*
assume *wt*: $(\|prg=G,cls=accC,lcl=L\|)\vdash c1\ Finally\ c2::\surd$
assume *da*: $(\|prg=G,cls=accC,lcl=L\|)$
$\vdash dom\ (locals\ (store\ s0))\ »\langle c1\ Finally\ c2\rangle_s»\ E$
assume *eval*: $G\vdash s0\ -c1\ Finally\ c2-n\rightarrow\ s3$
assume *P*: $(Normal\ P)\ Y\ s0\ Z$
show $R\ \diamondsuit\ s3\ Z \wedge s3::\preceq(G,L)$
proof −
from *eval* **obtain** *s1 abr1 s2* **where**
eval-c1: $G\vdash s0\ -c1-n\rightarrow\ (abr1,\ s1)$ **and**
eval-c2: $G\vdash Norm\ s1\ -c2-n\rightarrow\ s2$ **and**
s3: $s3 = (if\ \exists\, err.\ abr1 = Some\ (Error\ err)$
$then\ (abr1,\ s1)$
$else\ abupd\ (abrupt\text{-}if\ (abr1 \neq None)\ abr1)\ s2)$
using *normal-s0* **by** (*fastforce elim*: *evaln-elim-cases*)
from *wt* **obtain**
wt-c1: $(\|prg=G,cls=accC,lcl=L\|)\vdash c1::\surd$ **and**
wt-c2: $(\|prg=G,cls=accC,lcl=L\|)\vdash c2::\surd$
by *cases simp*
from *da* **obtain** *C1 C2* **where**
da-c1: $(\|prg=G,cls=accC,lcl=L\|) \vdash dom\ (locals\ (store\ s0))\ »\langle c1\rangle_s»\ C1$ **and**
da-c2: $(\|prg=G,cls=accC,lcl=L\|) \vdash dom\ (locals\ (store\ s0))\ »\langle c2\rangle_s»\ C2$
by *cases simp*
from *valid-c1 P valid-A conf-s0 eval-c1 wt-c1 da-c1*
obtain *Q*: $Q\ \diamondsuit\ (abr1,s1)\ Z$ **and** *conf-s1*: $(abr1,s1)::\preceq(G,L)$
by (*rule validE*)
from *Q*
have *Q'*: $(Q \wedge.\ (\lambda s.\ abr1 = fst\ s)\ ;.\ abupd\ (\lambda x.\ None))\ \diamondsuit\ (Norm\ s1)\ Z$
by *auto*
from *eval-c1 wt-c1 da-c1 conf-s0 wf*
have *error-free* $(abr1,s1)$
by (*rule evaln-type-sound* [*elim-format*]) (*insert normal-s0*,*simp*)
with *s3* **have** *s3'*: $s3 = abupd\ (abrupt\text{-}if\ (abr1 \neq None)\ abr1)\ s2$
by (*simp add*: *error-free-def*)
from *conf-s1*
have *conf-Norm-s1*: $Norm\ s1::\preceq(G,L)$
by (*rule conforms-NormI*)
obtain *C2'* **where**
da-c2': $(\|prg=G,cls=accC,lcl=L\|)$
$\vdash dom\ (locals\ (store\ ((Norm\ s1)::state)))\ »\langle c2\rangle_s»\ C2'$
proof −
from *eval-c1*
have $dom\ (locals\ (store\ s0)) \subseteq dom\ (locals\ (store\ (abr1,s1)))$
by (*rule dom-locals-evaln-mono-elim*)
hence $dom\ (locals\ (store\ s0))$
$\subseteq dom\ (locals\ (store\ ((Norm\ s1)::state)))$
by *simp*
with *da-c2* **show** *thesis*
by (*rule da-weakenE*) (*rule that*)
qed
from *valid-c2 Q' valid-A conf-Norm-s1 eval-c2 wt-c2 da-c2'*
have $(abupd\ (abrupt\text{-}if\ (abr1 \neq None)\ abr1)\ .;\ R)\ \diamondsuit\ s2\ Z$
by (*rule validE*)

```
      with s3' have R ◇ s3 Z
        by simp
      moreover
      from eval wt da conf-s0 wf
      have s3::≼(G,L)
        by (rule evaln-type-sound [elim-format]) simp
      ultimately show ?thesis ..
    qed
  qed
next
  case (Done A P C)
  show G,A|⊨::{ {Normal (P←◇ ∧. initd C)} .Init C. {P} }
  proof (rule valid-stmt-NormalI)
    fix n s0 L accC E s3 Y Z
    assume valid-A: ∀ t∈A. G|⊨n::t
    assume conf-s0:  s0::≼(G,L)
    assume normal-s0: normal s0
    assume wt: (|prg=G,cls=accC,lcl=L|)⊢Init C::√
    assume da: (|prg=G,cls=accC,lcl=L|)
                 ⊢dom (locals (store s0)) »⟨Init C⟩ₛ» E
    assume eval: G⊢s0 −Init C−n→ s3
    assume P: (Normal (P←◇ ∧. initd C)) Y s0 Z
    show P ◇ s3 Z ∧ s3::≼(G,L)
    proof −
      from P have inited: inited C (globs (store s0))
        by simp
      with eval have s3=s0
        using normal-s0 by (auto elim: evaln-elim-cases)
      with P conf-s0 show ?thesis
        by simp
    qed
  qed
next
  case (Init C c A P Q R)
  note c = ‹the (class G C) = c›
  note valid-super =
        ‹G,A|⊨::{ {Normal (P ∧. Not ∘ initd C ;. supd (init-class-obj G C))}
                 .(if C = Object then Skip else Init (super c)).
                 {Q} }›
  have valid-init:
        ⋀ l.  G,A|⊨::{ {Q ∧. (λs. l = locals (snd s)) ;. set-lvars Map.empty}
                        .init c.
                        {set-lvars l .; R} }
    using Init.hyps by simp
  show G,A|⊨::{ {Normal (P ∧. Not ∘ initd C)} .Init C. {R} }
  proof (rule valid-stmt-NormalI)
    fix n s0 L accC E s3 Y Z
    assume valid-A: ∀ t∈A. G|⊨n::t
    assume conf-s0:  s0::≼(G,L)
    assume normal-s0: normal s0
    assume wt: (|prg=G,cls=accC,lcl=L|)⊢Init C::√
    assume da: (|prg=G,cls=accC,lcl=L|)
                 ⊢dom (locals (store s0)) »⟨Init C⟩ₛ» E
    assume eval: G⊢s0 −Init C−n→ s3
    assume P: (Normal (P ∧. Not ∘ initd C)) Y s0 Z
    show R ◇ s3 Z ∧ s3::≼(G,L)
    proof −
      from P have not-inited: ¬ inited C (globs (store s0)) by simp
      with eval c obtain s1 s2 where
```

```
  eval-super:
  G⊢Norm ((init-class-obj G C) (store s0))
     −(if C = Object then Skip else Init (super c))−n→ s1 and
  eval-init: G⊢(set-lvars Map.empty) s1 −init c−n→ s2 and
  s3: s3 = (set-lvars (locals (store s1))) s2
  using normal-s0 by (auto elim!: evaln-elim-cases)
from wt c have
  cls-C: class G C = Some c
  by cases auto
from wf cls-C have
  wt-super: (|prg=G,cls=accC,lcl=L|)
                 ⊢(if C = Object then Skip else Init (super c))::√
  by (cases C=Object)
     (auto dest: wf-prog-cdecl wf-cdecl-supD is-acc-classD)
obtain S where
  da-super:
  (|prg=G,cls=accC,lcl=L|)
    ⊢ dom (locals (store ((Norm
                   ((init-class-obj G C) (store s0)))::state)))
         »⟨if C = Object then Skip else Init (super c)⟩s» S
proof (cases C=Object)
  case True
  with da-Skip show ?thesis
    using that by (auto intro: assigned.select-convs)
next
  case False
  with da-Init show ?thesis
    by − (rule that, auto intro: assigned.select-convs)
qed
from normal-s0 conf-s0 wf cls-C not-inited
have conf-init-cls: (Norm ((init-class-obj G C) (store s0)))::≼(G, L)
  by (auto intro: conforms-init-class-obj)
from P
have P′: (Normal (P ∧. Not ∘ initd C ;. supd (init-class-obj G C)))
           Y (Norm ((init-class-obj G C) (store s0))) Z
  by auto

from valid-super P′ valid-A conf-init-cls eval-super wt-super da-super
obtain Q: Q ◇ s1 Z and conf-s1: s1::≼(G,L)
  by (rule validE)

from cls-C wf have wt-init: (|prg=G, cls=C,lcl=Map.empty|)⊢(init c)::√
  by (rule wf-prog-cdecl [THEN wf-cdecl-wt-init])
from cls-C wf obtain I where
  (|prg=G,cls=C,lcl=Map.empty|)⊢ {} »⟨init c⟩s» I
  by (rule wf-prog-cdecl [THEN wf-cdeclE,simplified]) blast

then obtain I′ where
  da-init:
  (|prg=G,cls=C,lcl=Map.empty|)⊢dom (locals (store ((set-lvars Map.empty) s1)))
      »⟨init c⟩s» I′
  by (rule da-weakenE) simp
have conf-s1-empty: (set-lvars Map.empty) s1::≼(G, Map.empty)
proof −
  from eval-super have
    G⊢Norm ((init-class-obj G C) (store s0))
       −(if C = Object then Skip else Init (super c))→ s1
    by (rule evaln-eval)
  from this wt-super wf
```

```
        have s1-no-ret: ⋀ j. abrupt s1 ≠ Some (Jump j)
          by − (rule eval-statement-no-jump
                [where ?Env=(|prg=G,cls=accC,lcl=L|)], auto split: if-split)
        with conf-s1
        show ?thesis
          by (cases s1) (auto intro: conforms-set-locals)
      qed

      obtain l where l: l = locals (store s1)
        by simp
      with Q
      have Q': (Q ∧. (λs. l = locals (snd s)) ;. set-lvars Map.empty)
                ♢ ((set-lvars Map.empty) s1) Z
        by auto
      from valid-init Q' valid-A conf-s1-empty eval-init wt-init da-init
      have (set-lvars l .; R) ♢ s2 Z
        by (rule validE)
      with s3 l have R ♢ s3 Z
        by simp
      moreover
      from eval wt da conf-s0 wf
      have s3::≼(G,L)
        by (rule evaln-type-sound [elim-format]) simp
      ultimately show ?thesis ..
    qed
  qed
next
  case (InsInitV A P c v Q)
  show G,A|⊨::{ {Normal P} InsInitV c v=≻ {Q} }
  proof (rule valid-var-NormalI)
    fix s0 vf n s1 L Z
    assume normal s0
    moreover
    assume G⊢s0 −InsInitV c v=≻vf−n→ s1
    ultimately have False
      by (cases s0) (simp add: evaln-InsInitV)
    thus Q ⌊vf⌋_v s1 Z ∧ s1::≼(G, L)..
  qed
next
  case (InsInitE A P c e Q)
  show G,A|⊨::{ {Normal P} InsInitE c e−≻ {Q} }
  proof (rule valid-expr-NormalI)
    fix s0 v n s1 L Z
    assume normal s0
    moreover
    assume G⊢s0 −InsInitE c e−≻v−n→ s1
    ultimately have False
      by (cases s0) (simp add: evaln-InsInitE)
    thus Q ⌊v⌋_e s1 Z ∧ s1::≼(G, L)..
  qed
next
  case (Callee A P l e Q)
  show G,A|⊨::{ {Normal P} Callee l e−≻ {Q} }
  proof (rule valid-expr-NormalI)
    fix s0 v n s1 L Z
    assume normal s0
    moreover
    assume G⊢s0 −Callee l e−≻v−n→ s1
    ultimately have False
```

    by (*cases s0*) (*simp add*: *evaln-Callee*)
   thus $Q\ \lfloor v \rfloor_e\ s1\ Z \wedge s1{::}\preceq(G,\ L)$..
  qed
next
  case (*FinA A P a c Q*)
  show $G,A|\models{::}\{\ \{Normal\ P\}\ .FinA\ a\ c.\ \{Q\}\ \}$
  proof (*rule valid-stmt-NormalI*)
   fix *s0 v n s1 L Z*
   assume *normal s0*
   moreover
   assume $G\vdash s0\ -FinA\ a\ c-n\rightarrow\ s1$
   ultimately have *False*
    by (*cases s0*) (*simp add*: *evaln-FinA*)
   thus $Q\ \diamondsuit\ s1\ Z \wedge s1{::}\preceq(G,\ L)$..
  qed
qed
declare *inj-term-simps* [*simp del*]

theorem *ax-sound*:
 $wf\text{-}prog\ G \Longrightarrow G,(A{::}'a\ triple\ set)|\vdash(ts{::}'a\ triple\ set) \Longrightarrow G,A|\models ts$
apply (*subst ax-valids2-eq* [*symmetric*])
apply *assumption*
apply (*erule* (*1*) *ax-sound2*)
done

lemma *sound-valid2-lemma*:
$[\![\forall v\ n.\ Ball\ A\ (triple\text{-}valid2\ G\ n) \longrightarrow P\ v\ n;\ Ball\ A\ (triple\text{-}valid2\ G\ n)]\!]$
 $\Longrightarrow P\ v\ n$
by *blast*

end

Chapter 24

AxCompl

1 Completeness proof for Axiomatic semantics of Java expressions and statements

theory *AxCompl* **imports** *AxSem* **begin**

design issues:

- proof structured by Most General Formulas (-> Thomas Kleymann)

set of not yet initialzed classes

definition
  nyinitcls :: *prog* ⇒ *state* ⇒ *qtname set*
  where *nyinitcls G s* = {*C*. *is-class G C* ∧ ¬ *initd C s*}

lemma *nyinitcls-subset-class*: *nyinitcls G s* ⊆ {*C*. *is-class G C*}
apply (*unfold nyinitcls-def*)
apply *fast*
done

lemmas *finite-nyinitcls* [*simp*] =
  finite-is-class [*THEN nyinitcls-subset-class* [*THEN finite-subset*]]

lemma *card-nyinitcls-bound*: *card* (*nyinitcls G s*) ≤ *card* {*C*. *is-class G C*}
apply (*rule nyinitcls-subset-class* [*THEN finite-is-class* [*THEN card-mono*]])
done

lemma *nyinitcls-set-locals-cong* [*simp*]:
  nyinitcls G (*x*,*set-locals l s*) = *nyinitcls G* (*x*,*s*)
  by (*simp add*: *nyinitcls-def*)

lemma *nyinitcls-abrupt-cong* [*simp*]: *nyinitcls G* (*f x*, *y*) = *nyinitcls G* (*x*, *y*)
  by (*simp add*: *nyinitcls-def*)

lemma *nyinitcls-abupd-cong* [*simp*]: *nyinitcls G* (*abupd f s*) = *nyinitcls G s*
  by (*simp add*: *nyinitcls-def*)

lemma *card-nyinitcls-abrupt-congE* [*elim*!]:
card (*nyinitcls G* (*x*, *s*)) ≤ *n* ⟹ *card* (*nyinitcls G* (*y*, *s*)) ≤ *n*
unfolding *nyinitcls-def* **by** *auto*

lemma *nyinitcls-new-xcpt-var* [*simp*]:
nyinitcls G (*new-xcpt-var vn s*) = *nyinitcls G s*
by (*induct s*) (*simp-all add*: *nyinitcls-def*)

lemma *nyinitcls-init-lvars* [*simp*]:
nyinitcls G ((*init-lvars G C sig mode a′ pvs*) *s*) = *nyinitcls G s*
by (*induct s*) (*simp add*: *init-lvars-def2 split*: *if-split*)

lemma *nyinitcls-emptyD*: ⟦*nyinitcls G s* = {}; *is-class G C*⟧ ⟹ *initd C s*
unfolding *nyinitcls-def* **by** *fast*

lemma *card-Suc-lemma*:
⟦*card* (*insert a A*) ≤ *Suc n*; *a*∉*A*; *finite A*⟧ ⟹ *card A* ≤ *n*
by *auto*

lemma *nyinitcls-le-SucD*:
⟦*card* (*nyinitcls G* (*x*,*s*)) ≤ *Suc n*; ¬*inited C* (*globs s*); *class G C*=*Some y*⟧ ⟹
card (*nyinitcls G* (*x*,*init-class-obj G C s*)) ≤ *n*
apply (*subgoal-tac*
nyinitcls G (*x*,*s*) = *insert C* (*nyinitcls G* (*x*,*init-class-obj G C s*)))
apply *clarsimp*
apply (*erule-tac V*=*nyinitcls G* (*x*, *s*) = *rhs* **for** *rhs* **in** *thin-rl*)
apply (*rule card-Suc-lemma* [*OF - - finite-nyinitcls*])
apply (*auto dest*!: *not-initedD elim*!:
simp add: *nyinitcls-def inited-def split*: *if-split-asm*)
done

lemma *inited-gext′*: ⟦*s*≤|*s′*;*inited C* (*globs s*)⟧ ⟹ *inited C* (*globs s′*)
by (*rule inited-gext*)

lemma *nyinitcls-gext*: *snd s*≤|*snd s′* ⟹ *nyinitcls G s′* ⊆ *nyinitcls G s*
unfolding *nyinitcls-def* **by** (*force dest*!: *inited-gext′*)

lemma *card-nyinitcls-gext*:
⟦*snd s*≤|*snd s′*; *card* (*nyinitcls G s*) ≤ *n*⟧⟹ *card* (*nyinitcls G s′*) ≤ *n*
apply (*rule le-trans*)
apply (*rule card-mono*)
apply (*rule finite-nyinitcls*)
apply (*erule nyinitcls-gext*)
apply *assumption*
done

init-le

definition
init-le :: *prog* ⇒ *nat* ⇒ *state* ⇒ *bool* (-⊢*init*≤- [*51*,*51*] *50*)
where *G*⊢*init*≤*n* = (λ*s*. *card* (*nyinitcls G s*) ≤ *n*)

lemma *init-le-def2* [*simp*]: $(G\vdash init\le n)\ s = (card\ (nyinitcls\ G\ s)\le n)$
apply (*unfold init-le-def*)
apply *auto*
done

lemma *All-init-leD*:
$\forall n{::}nat.\ G,(A{::}'a\ triple\ set)\vdash\{P \wedge.\ G\vdash init\le n\}\ t\succ\ \{Q{::}'a\ assn\}$
$\Longrightarrow G,A\vdash\{P\}\ t\succ\ \{Q\}$
apply (*drule spec*)
apply (*erule conseq1*)
apply *clarsimp*
apply (*rule card-nyinitcls-bound*)
done

Most General Triples and Formulas

definition
remember-init-state :: *state assn* ($\doteq$)
where $\doteq\ \equiv \lambda Y\ s\ Z.\ s = Z$

lemma *remember-init-state-def2* [*simp*]: $\doteq\ Y = (=)$
apply (*unfold remember-init-state-def*)
apply (*simp* (*no-asm*))
done

definition
MGF ::[*state assn*, *term*, *prog*] $\Rightarrow$ *state triple* ({-} -≻ {-→}[*3*,*65*,*3*]*62*)
where $\{P\}\ t\succ\ \{G\rightarrow\} = \{P\}\ t\succ\ \{\lambda Y\ s'\ s.\ G\vdash s\ -t\succ\rightarrow\ (Y,s')\}$

definition
MGFn :: [*nat*, *term*, *prog*] $\Rightarrow$ *state triple* ({=:-} -≻ {-→}[*3*,*65*,*3*]*62*)
where $\{=:n\}\ t\succ\ \{G\rightarrow\} = \{\doteq \wedge.\ G\vdash init\le n\}\ t\succ\ \{G\rightarrow\}$

lemma *MGF-valid*: *wf-prog G* $\Longrightarrow G,\{\}\models\{\doteq\}\ t\succ\ \{G\rightarrow\}$
apply (*unfold MGF-def*)
apply (*simp add*: *ax-valids-def triple-valid-def2*)
apply (*auto elim*: *evaln-eval*)
done

lemma *MGF-res-eq-lemma* [*simp*]:
$(\forall Y'\ Y\ s.\ Y = Y' \wedge P\ s \longrightarrow Q\ s) = (\forall s.\ P\ s \longrightarrow Q\ s)$
by *auto*

lemma *MGFn-def2*:
$G,A\vdash\{=:n\}\ t\succ\ \{G\rightarrow\} = G,A\vdash\{\doteq \wedge.\ G\vdash init\le n\}$
$t\succ\ \{\lambda Y\ s'\ s.\ G\vdash s\ -t\succ\rightarrow\ (Y,s')\}$
unfolding *MGFn-def MGF-def* **by** *fast*

lemma *MGF-MGFn-iff*:

$G,(A::state\ triple\ set)\vdash\{\doteq\}\ t\succ\ \{G\rightarrow\} = (\forall\, n.\ G,A\vdash\{=:n\}\ t\succ\ \{G\rightarrow\})$
apply (*simp add*: *MGFn-def2 MGF-def*)
apply *safe*
apply (*erule-tac* [*2*] *All-init-leD*)
apply (*erule conseq1*)
apply *clarsimp*
done

lemma *MGFnD*:
$G,(A::state\ triple\ set)\vdash\{=:n\}\ t\succ\ \{G\rightarrow\} \Longrightarrow$
$G,A\vdash\{(\lambda Y'\ s'\ s.\ s' = s \qquad \wedge\ P\ s)\ \wedge.\ G\vdash init\le n\}$
$t\succ\ \{(\lambda Y'\ s'\ s.\ G\vdash s{-}t{\succ}{\rightarrow}(Y',s')\ \wedge\ P\ s)\ \wedge.\ G\vdash init\le n\}$
apply (*unfold init-le-def*)
apply (*simp* (*no-asm-use*) *add*: *MGFn-def2*)
apply (*erule conseq12*)
apply *clarsimp*
apply (*erule* (*1*) *eval-gext* [*THEN card-nyinitcls-gext*])
done
lemmas *MGFnD′* = *MGFnD* [*of - - - - λx. 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*: $G,A\vdash\{Normal\ \doteq\}\ t\succ\ \{G\rightarrow\} \Longrightarrow$
$G,(A::state\ triple\ set)\vdash\{\doteq::state\ assn\}\ t\succ\ \{G\rightarrow\}$
apply (*unfold MGF-def*)
apply (*rule ax-Normal-cases*)
apply (*erule conseq1*)
apply *clarsimp*
apply (*rule ax-derivs.Abrupt* [*THEN conseq1*])
apply (*clarsimp simp add*: *Let-def*)
done

lemma *MGFNormalD*:
$G,(A::state\ triple\ set)\vdash\{\doteq\}\ t\succ\ \{G\rightarrow\} \Longrightarrow G,A\vdash\{Normal\ \doteq\}\ t\succ\ \{G\rightarrow\}$
apply (*unfold MGF-def*)
apply (*erule conseq1*)
apply *clarsimp*
done

Additionally to *MGFNormalI*, we also expand the definition of the most general formula here

lemma *MGFn-NormalI*:
$G,(A::state\ triple\ set)\vdash\{Normal((\lambda Y'\ s'\ s.\ s'{=}s\ \wedge\ normal\ s)\ \wedge.\ G\vdash init\le n)\}t\succ$
$\{\lambda Y\ s'\ s.\ G\vdash s\ {-}t{\succ}{\rightarrow}\ (Y,s')\} \Longrightarrow G,A\vdash\{=:n\}t\succ\{G\rightarrow\}$
apply (*simp* (*no-asm-use*) *add*: *MGFn-def2*)
apply (*rule ax-Normal-cases*)
apply (*erule conseq1*)
apply *clarsimp*
apply (*rule ax-derivs.Abrupt* [*THEN conseq1*])
apply (*clarsimp simp add*: *Let-def*)
done

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*:
$(\exists\ T\ L\ C.\ (\!|prg{=}G,cls{=}C,lcl{=}L|\!)\vdash t::T)$
$\longrightarrow G,(A::state\ triple\ set)\vdash\{=:n\}\ t\succ\ \{G\rightarrow\}$

$\Longrightarrow$ $G,A\vdash\{=:n\}\ t\succ\ \{G\rightarrow\}$
apply (*rule MGFn-NormalI*)
apply (*rule ax-free-wt*)
apply (*auto elim*: *conseq12 simp add*: *MGFn-def MGF-def*)
done

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*:
$(\forall\ T\ L\ C\ .\ (\!|prg{=}G,cls{=}C,lcl{=}L|\!)\vdash t{::}T$
$\longrightarrow G,(A{::}state\ triple\ set)$
$\vdash\{Normal((\lambda Y'\ s'\ s.\ s'{=}s \wedge normal\ s) \wedge.\ G\vdash init{\leq}n) \wedge.\ (\lambda\ s.\ s{::}{\preceq}(G,\ L))\}$
$t\succ$
$\{\lambda Y\ s'\ s.\ G\vdash s\ {-}t{\succ}{\rightarrow}\ (Y,s')\})$
$\Longrightarrow G,A\vdash\{=:n\}t\succ\{G\rightarrow\}$
apply (*rule MGFn-NormalI*)
apply (*rule ax-no-hazard*)
apply (*rule ax-escape*)
apply (*intro strip*)
apply (*simp only*: *type-ok-def peek-and-def*)
apply (*erule conjE*)+
apply (*erule exE*,*erule exE*, *erule exE*, *erule exE*,*erule conjE*,*drule* (*1*) *mp*,
erule conjE)
apply (*drule spec*,*drule spec*, *drule spec*, *drule* (*1*) *mp*)
apply (*erule conseq12*)
apply *blast*
done

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*:
$(\forall\ T\ L\ C\ B.\ (\!|prg{=}G,cls{=}C,lcl{=}L|\!)\vdash t{::}T$
$\longrightarrow G,(A{::}state\ triple\ set)$
$\vdash\{Normal((\lambda Y'\ s'\ s.\ s'{=}s \wedge normal\ s) \wedge.\ G\vdash init{\leq}n) \wedge.\ (\lambda\ s.\ s{::}{\preceq}(G,\ L))$
$\wedge.\ (\lambda\ s.\ (\!|prg{=}G,cls{=}C,lcl{=}L|\!)\vdash dom\ (locals\ (store\ s))\gg t\gg B)\}$
$t\succ$
$\{\lambda Y\ s'\ s.\ G\vdash s\ {-}t{\succ}{\rightarrow}\ (Y,s')\})$
$\Longrightarrow G,A\vdash\{=:n\}t\succ\{G\rightarrow\}$
apply (*rule MGFn-NormalI*)
apply (*rule ax-no-hazard*)
apply (*rule ax-escape*)
apply (*intro strip*)
apply (*simp only*: *type-ok-def peek-and-def*)
apply (*erule conjE*)+
apply (*erule exE*,*erule exE*, *erule exE*, *erule exE*,*erule conjE*,*drule* (*1*) *mp*,
erule conjE)
apply (*drule spec*,*drule spec*, *drule spec*,*drule spec*, *drule* (*1*) *mp*)
apply (*erule conseq12*)
apply *blast*
done

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 (*rule MGFn-free-wt* [*rule-format*],*elim exE*,*rule MGFn-NormalI*)
  fix *T L accC*
  assume ⦇*prg*=*G*, *cls*=*accC*, *lcl*= *L*⦈⊢⟨*Init C*⟩$_s$::*T*
  hence *is-cls*: *is-class G C*
    by *cases simp*
  show *G*,*A*⊢{*Normal* ((λ*Y′ s′ s*. *s′* = *s* ∧ *normal s*) ∧. *G*⊢*init*≤*n*)}
          .*Init C*.
          {λ*Y s′ s*. *G*⊢*s* −⟨*Init C*⟩$_s$≻→ (*Y*, *s′*)}
      (**is** *G*,*A*⊢{*Normal ?P*} .*Init C*. {*?R*})
  proof (*rule ax-cases* [**where** *?C*=*initd C*])
    show *G*,*A*⊢{*Normal ?P* ∧. *initd C*} .*Init C*. {*?R*}
      by (*rule ax-derivs.Done* [*THEN conseq1*]) (*fastforce intro*: *init-done*)
  next
    have *G*,*A*⊢{*Normal* (*?P* ∧. *Not* ∘ *initd C*)} .*Init C*. {*?R*}
    proof (*cases n*)
      case *0*
      with *is-cls*
      show *?thesis*
        by − (*rule ax-impossible* [*THEN conseq1*],*fastforce dest*: *nyinitcls-emptyD*)
    next
      case (*Suc m*)
      with *mgf-hyp* **have** *mgf-hyp′*: ⋀ *t*. *G*,*A*⊢{=:*m*} *t*≻ {*G*→}
        by *simp*
      from *is-cls* **obtain** *c* **where** *c*: *the* (*class G C*) = *c*
        by *auto*
      let *?Q*= (λ*Y s′* (*x*,*s*) .
          G⊢ (*x*,*init-class-obj G C s*)
            − (*if C*=*Object then Skip else Init* (*super* (*the* (*class G C*))))→ *s′*
          ∧ *x*=*None* ∧ ¬*inited C* (*globs s*)) ∧. *G*⊢*init*≤*m*
      from *c*
      show *?thesis*
      proof (*rule ax-derivs.Init* [**where** *?Q*=*?Q*])
        let *?P′* = *Normal* ((λ*Y s′ s*. *s′* = *supd* (*init-class-obj G C*) *s*
                          ∧ *normal s* ∧ ¬ *initd C s*) ∧. *G*⊢*init*≤*m*)
        show *G*,*A*⊢{*Normal* (*?P* ∧. *Not* ∘ *initd C* ;. *supd* (*init-class-obj G C*))}
                .(*if C* = *Object then Skip else Init* (*super c*)).
                {*?Q*}
        proof (*rule conseq1* [**where** *?P′*=*?P′*])
          show *G*,*A*⊢{*?P′*} .(*if C* = *Object then Skip else Init* (*super c*)). {*?Q*}
          proof (*cases C*=*Object*)
            case *True*
            have *G*,*A*⊢{*?P′*} .*Skip*. {*?Q*}
              by (*rule ax-derivs.Skip* [*THEN conseq1*])
                (*auto simp add*: *True intro*: *eval.Skip*)
            with *True* **show** *?thesis*
              by *simp*
          next
            case *False*
            from *mgf-hyp′*
            have *G*,*A*⊢{*?P′*} .*Init* (*super c*). {*?Q*}
              by (*rule MGFnD′* [*THEN conseq12*]) (*fastforce simp add*: *False c*)
            with *False* **show** *?thesis*
              by *simp*
          qed
        next
          from *Suc is-cls*
          show *Normal* (*?P* ∧. *Not* ∘ *initd C* ;. *supd* (*init-class-obj G C*))
              ⇒ *?P′*
            by (*fastforce elim*: *nyinitcls-le-SucD*)

```
        qed
      next
        from mgf-hyp′
        show ∀ l. G,A⊢{?Q ∧. (λs. l = locals (snd s)) ;. set-lvars Map.empty}
                      .init c.
                      {set-lvars l .; ?R}
          apply (rule MGFnD′ [THEN conseq12, THEN allI])
          apply (clarsimp simp add: split-paired-all)
          apply (rule eval.Init [OF c])
          apply (insert c)
          apply auto
          done
      qed
    qed
    thus G,A⊢{Normal ?P ∧. Not ∘ initd C} .Init C. {?R}
      by clarsimp
  qed
qed
lemmas MGFn-InitD = MGFn-Init [THEN MGFnD, THEN ax-NormalD]


lemma MGFn-Call:
  assumes mgf-methds:
          ∀ C sig. G,(A::state triple set)⊢{=:n} ⟨(Methd C sig)⟩e≻ {G→}
  and mgf-e: G,A⊢{=:n} ⟨e⟩e≻ {G→}
  and mgf-ps: G,A⊢{=:n} ⟨ps⟩l≻ {G→}
  and wf: wf-prog G
  shows G,A⊢{=:n} ⟨{accC,statT,mode}e·mn({pTs′}ps)⟩e≻ {G→}
proof (rule MGFn-free-wt-da-NormalConformI [rule-format],clarsimp)
  note inj-term-simps [simp]
  fix T L accC′ E
  assume wt: (|prg=G,cls=accC′,lcl = L|)⊢⟨({accC,statT,mode}e·mn( {pTs′}ps))⟩e::T
  then obtain pTs statDeclT statM where
               wt-e: (|prg=G,cls=accC,lcl=L|)⊢e::−RefT statT and
            wt-args: (|prg=G,cls=accC,lcl=L|)⊢ps::≐pTs and
              statM: max-spec G accC statT (|name=mn,parTs=pTs|)
                     = {((statDeclT,statM),pTs′)} and
               mode: mode = invmode statM e and
                  T: T =Inl (resTy statM) and
        eq-accC-accC′: accC=accC′
        by cases fastforce+
  let ?Q=(λY s1 (x,s) . x = None ∧
             (∃ a. G⊢Norm s −e−≻a→ s1 ∧
                  (normal s1 ⟶ G, store s1⊢a::≼RefT statT)
                  ∧ Y = In1 a) ∧
             (∃ P. normal s1
                 ⟶ (|prg=G,cls=accC′,lcl=L|)⊢dom (locals (store s1))»⟨ps⟩l»P))
         ∧. G⊢init≤n ∧. (λ s. s::≼(G, L))::state assn
  let ?R=λa. ((λY (x2,s2) (x,s) . x = None ∧
              (∃ s1 pvs. G⊢Norm s −e−≻a→ s1 ∧
                        (normal s1 ⟶ G, store s1⊢a::≼RefT statT)∧
                        Y = ⌊pvs⌋l ∧ G⊢s1 −ps≐≻pvs→ (x2,s2)))
            ∧. G⊢init≤n ∧. (λ s. s::≼(G, L)))::state assn

  show G,A⊢{Normal ((λY′ s′ s. s′ = s ∧ abrupt s = None) ∧. G⊢init≤n ∧.
                     (λs. s::≼(G, L)) ∧.
                     (λs. (|prg=G, cls=accC′,lcl=L|) ⊢ dom (locals (store s))
                          » ⟨{accC,statT,mode}e·mn( {pTs′}ps)⟩e» E))}
            {accC,statT,mode}e·mn( {pTs′}ps)−≻
```

```
        {λY s' s. ∃ v. Y = ⌊v⌋e ∧
                    G⊢s −{accC,statT,mode}e·mn( {pTs'}ps)−≻v→ s'}
  (is G,A⊢{Normal ?P} {accC,statT,mode}e·mn( {pTs'}ps)−≻ {?S})
proof (rule ax-derivs.Call [where ?Q=?Q and ?R=?R])
  from mgf-e
  show G,A⊢{Normal ?P} e−≻ {?Q}
  proof (rule MGFnD' [THEN conseq12],clarsimp)
    fix s0 s1 a
    assume conf-s0: Norm s0::≼(G, L)
    assume da: (|prg=G,cls=accC',lcl=L|)⊢
                  dom (locals s0) »⟨{accC,statT,mode}e·mn( {pTs'}ps)⟩e» E
    assume eval-e: G⊢Norm s0 −e−≻a→ s1
    show (abrupt s1 = None ⟶ G,store s1⊢a::≼RefT statT) ∧
         (abrupt s1 = None ⟶
           (∃ P. (|prg=G,cls=accC',lcl=L|)⊢ dom (locals (store s1)) »⟨ps⟩l» P))
         ∧ s1::≼(G, L)
    proof −
      from da obtain C where
        da-e: (|prg=G,cls=accC,lcl=L|)⊢
                dom (locals (store ((Norm s0)::state)))»⟨e⟩e» C and
        da-ps: (|prg=G,cls=accC,lcl=L|)⊢ nrm C »⟨ps⟩l» E
        by cases (simp add: eq-accC-accC')
      from eval-e conf-s0 wt-e da-e wf
      obtain (abrupt s1 = None ⟶ G,store s1⊢a::≼RefT statT)
        and  s1::≼(G, L)
        by (rule eval-type-soundE) simp
      moreover
      {
        assume normal-s1: normal s1
        have ∃ P. (|prg=G,cls=accC,lcl=L|)⊢ dom (locals (store s1)) »⟨ps⟩l» P
        proof −
          from eval-e wt-e da-e wf normal-s1
          have nrm C ⊆  dom (locals (store s1))
            by (cases rule: da-good-approxE') iprover
          with da-ps show ?thesis
            by (rule da-weakenE) iprover
        qed
      }
      ultimately show ?thesis
        using eq-accC-accC' by simp
    qed
  qed
next
  show ∀ a. G,A⊢{?Q←In1 a} ps≐≻ {?R a} (is ∀  a. ?PS a)
  proof
    fix a
    show ?PS a
    proof (rule MGFnD' [OF mgf-ps, THEN conseq12],
           clarsimp simp add: eq-accC-accC' [symmetric])
      fix s0 s1 s2 vs
      assume conf-s1: s1::≼(G, L)
      assume eval-e: G⊢Norm s0 −e−≻a→ s1
      assume conf-a: abrupt s1 = None ⟶ G,store s1⊢a::≼RefT statT
      assume eval-ps: G⊢s1 −ps≐≻vs→ s2
      assume da-ps: abrupt s1 = None ⟶
                      (∃ P. (|prg=G,cls=accC,lcl=L|)⊢
                              dom (locals (store s1)) »⟨ps⟩l» P)
      show (∃ s1. G⊢Norm s0 −e−≻a→ s1 ∧
              (abrupt s1 = None ⟶ G,store s1⊢a::≼RefT statT) ∧
```

```
            G⊢s1 −ps≐≻vs→ s2) ∧
          s2::≼(G, L)
      proof (cases normal s1)
        case True
        with da-ps obtain P where
         (|prg=G,cls=accC,lcl=L|)⊢ dom (locals (store s1)) »⟨ps⟩l» P
          by auto
        from eval-ps conf-s1 wt-args this wf
        have s2::≼(G, L)
          by (rule eval-type-soundE)
        with eval-e conf-a eval-ps
        show ?thesis
          by auto
      next
        case False
        with eval-ps have s2=s1 by auto
        with eval-e conf-a eval-ps conf-s1
        show ?thesis
          by auto
      qed
    qed
  qed
next
  show ∀ a vs invC declC l.
    G,A⊢{?R a←⌊vs⌋l ∧.
          (λs. declC =
               invocation-declclass G mode (store s) a statT
                  (|name=mn, parTs=pTs'|) ∧
               invC = invocation-class mode (store s) a statT ∧
               l = locals (store s)) ;.
          init-lvars G declC (|name=mn, parTs=pTs'|) mode a vs ∧.
          (λs. normal s ⟶ G⊢mode→invC≼statT)}
       Methd declC (|name=mn,parTs=pTs'|)−≻
       {set-lvars l .; ?S}
    (is ∀ a vs invC declC l. ?METHD a vs invC declC l)
  proof (intro allI)
    fix a vs invC declC l
    from mgf-methds [rule-format]
    show ?METHD a vs invC declC l
    proof (rule MGFnD' [THEN conseq12],clarsimp)
      fix s4 s2 s1::state
      fix s0 v
      let ?D= invocation-declclass G mode (store s2) a statT
                (|name=mn,parTs=pTs'|)
      let ?s3= init-lvars G ?D (|name=mn, parTs=pTs'|) mode a vs s2
      assume inv-prop: abrupt ?s3=None
          ⟶ G⊢mode→invocation-class mode (store s2) a statT≼statT
      assume conf-s2: s2::≼(G, L)
      assume conf-a: abrupt s1 = None ⟶ G,store s1⊢a::≼RefT statT
      assume eval-e: G⊢Norm s0 −e−≻a→ s1
      assume eval-ps: G⊢s1 −ps≐≻vs→ s2
      assume eval-mthd: G⊢?s3 −Methd ?D (|name=mn,parTs=pTs'|)−≻v→ s4
      show G⊢Norm s0 −{accC,statT,mode}e·mn( {pTs'}ps)−≻v
                → (set-lvars (locals (store s2))) s4
      proof −
        obtain D where D: D=?D by simp
        obtain s3 where s3: s3=?s3 by simp
        obtain s3' where
          s3': s3' = check-method-access G accC statT mode
```

```
                    (|name=mn,parTs=pTs'|) a s3
          by simp
        have eq-s3'-s3: s3'=s3
        proof -
          from inv-prop s3 mode
          have normal s3 ⟹
           G⊢invmode statM e→invocation-class mode (store s2) a statT≼statT
            by auto
          with eval-ps wt-e statM conf-s2 conf-a [rule-format]
          have check-method-access G accC statT (invmode statM e)
                 (|name=mn,parTs=pTs'|) a s3 = s3
            by (rule error-free-call-access) (auto simp add: s3 mode wf)
          thus ?thesis
            by (simp add: s3' mode)
        qed
        with eval-mthd D s3
        have G⊢s3' −Methd D (|name=mn,parTs=pTs'|)−≻v→ s4
          by simp
        with eval-e eval-ps D - s3'
        show ?thesis
          by (rule eval-Call) (auto simp add: s3 mode D)
      qed
    qed
  qed
 qed
qed


lemma eval-expression-no-jump':
 assumes eval: G⊢s0 −e−≻v→ s1
 and    no-jmp: abrupt s0 ≠ Some (Jump j)
 and       wt: (|prg=G, cls=C,lcl=L|)⊢e::−T
 and       wf: wf-prog G
shows abrupt s1 ≠ Some (Jump j)
using eval no-jmp wt wf
by − (rule eval-expression-no-jump
          [where ?Env=(|prg=G, cls=C,lcl=L|),simplified],auto)
```

To derive the most general formula for the loop statement, we need to come up with a proper loop invariant, which intuitively states that we are currently inside the evaluation of the loop. To define such an invariant, we unroll the loop in iterated evaluations of the expression and evaluations of the loop body.

```
definition
 unroll :: prog ⇒ label ⇒ expr ⇒ stmt ⇒ (state × state) set where
 unroll G l e c = {(s,t). ∃ 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'
using unroll
```

```
proof (induct rule: converse-rtrancl-induct)
  show G⊢t −l• While(e) c→ s′
  proof (cases normal t)
    case False
    with eval-e have s′=t by auto
    with False show ?thesis by auto
  next
    case True
    note normal-t = this
    show ?thesis
    proof (cases normal s′)
      case True
      with normal-t eval-e normal-termination
      show ?thesis
        by (auto intro: eval.Loop)
    next
      case False
      note abrupt-s′ = this
      from eval-e - wt wf
      have no-cont: abrupt s′ ≠ Some (Jump (Cont l))
        by (rule eval-expression-no-jump′) (insert normal-t,simp)
      have
        if the-Bool v
             then (G⊢s′ −c→ s′ ∧
                    G⊢(abupd (absorb (Cont l)) s′) −l• While(e) c→ s′)
             else s′ = s′
      proof (cases the-Bool v)
        case False thus ?thesis by simp
      next
        case True
        with abrupt-s′ have G⊢s′ −c→ s′ by auto
        moreover from abrupt-s′ no-cont
        have no-absorb: (abupd (absorb (Cont l)) s′)=s′
          by (cases s′) (simp add: absorb-def split: if-split)
        moreover
        from no-absorb abrupt-s′
        have G⊢(abupd (absorb (Cont l)) s′) −l• While(e) c→ s′
          by auto
        ultimately show ?thesis
          using True by simp
      qed
      with eval-e
      show ?thesis
        using normal-t by (auto intro: eval.Loop)
    qed
  qed
next
  fix s s3
  assume unroll: (s,s3) ∈ unroll G l e c
  assume while: G⊢s3 −l• While(e) c→ s′
  show G⊢s −l• While(e) c→ s′
  proof −
    from unroll obtain v s1 s2 where
      normal-s1: normal s1 and
      eval-e: G⊢s −e−≻v→ s1 and
      continue: the-Bool v and
      eval-c: G⊢s1 −c→ s2 and
      s3: s3=(abupd (absorb (Cont l)) s2)
      by  (unfold unroll-def) fast
```

```
    from eval-e normal-s1 have
      normal s
      by (rule eval-no-abrupt-lemma [rule-format])
    with while eval-e continue eval-c s3 show ?thesis
      by (auto intro!: eval.Loop)
  qed
qed


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 (rule MGFn-free-wt [rule-format],elim exE)
  fix T L C
  assume wt: (|prg = G, cls = C, lcl = L|)⊢⟨l· While(e) c⟩ₛ::T
  then obtain eT where
    wt-e: (|prg = G, cls = C, lcl = L|)⊢e::−eT
    by cases simp
  show ?thesis
  proof (rule MGFn-NormalI)
    show G,A⊢{Normal ((λY' s' s. s' = s ∧ normal s) ∧. G⊢init≤n)}
           .l· While(e) c.
           {λY s' s. G⊢s −In1r (l· While(e) c)≻→ (Y, s')}
    proof (rule conseq12
         [where ?P'=(λ Y s' s. (s,s') ∈ (unroll G l e c)* ) ∧. G⊢init≤n
          and  ?Q'=((λ Y s' s. (∃ t b. (s,t) ∈ (unroll G l e c)* ∧
                     Y=⌊b⌋ₑ ∧ G⊢t −e−≻b→ s'))
                   ∧. G⊢init≤n)←=False↓=◇])
      show  G,A⊢{(λY s' s. (s, s') ∈ (unroll G l e c)*) ∧. G⊢init≤n}
              .l· While(e) c.
             {((λY s' s. (∃ t b. (s, t) ∈ (unroll G l e c)* ∧
                           Y = In1 b ∧ G⊢t −e−≻b→ s'))
                        ∧. G⊢init≤n)←=False↓=◇}
      proof (rule ax-derivs.Loop)
        from mfg-e
        show G,A⊢{(λY s' s. (s, s') ∈ (unroll G l e c)*) ∧. G⊢init≤n}
                e−≻
               {(λY s' s. (∃ t b. (s, t) ∈ (unroll G l e c)* ∧
                            Y = In1 b ∧ G⊢t −e−≻b→ s'))
                ∧. G⊢init≤n}
        proof (rule MGFnD' [THEN conseq12],clarsimp)
          fix s Z s' v
          assume (Z, s) ∈ (unroll G l e c)*
          moreover
          assume G⊢s −e−≻v→ s'
          ultimately
          show ∃ t. (Z, t) ∈ (unroll G l e c)* ∧ G⊢t −e−≻v→ s'
            by blast
        qed
      next
        from mfg-c
        show G,A⊢{Normal (((λY s' s. ∃ t b. (s, t) ∈ (unroll G l e c)* ∧
                                Y = ⌊b⌋ₑ ∧ G⊢t −e−≻b→ s')
                    ∧. G⊢init≤n)←=True)}
               .c.
               {abupd (absorb (Cont l)) .;
                ((λY s' s. (s, s') ∈ (unroll G l e c)*) ∧. G⊢init≤n)}
```

```
        proof (rule MGFnD' [THEN conseq12],clarsimp)
          fix Z s' s v t
          assume unroll: (Z, t) ∈ (unroll G l e c)*
          assume eval-e: G⊢t −e−≻v→ Norm s
          assume true: the-Bool v
          assume eval-c: G⊢Norm s −c→ s'
          show (Z, abupd (absorb (Cont l)) s') ∈ (unroll G l e c)*
          proof −
            note unroll
            also
            from eval-e true eval-c
            have (t,abupd (absorb (Cont l)) s') ∈ unroll G l e c
              by (unfold unroll-def) force
            ultimately show ?thesis ..
          qed
        qed
      qed
    next
      show
       ∀ Y s Z.
        (Normal ((λY' s' s. s' = s ∧ normal s) ∧. G⊢init≤n)) Y s Z
        ⟶ (∀ Y' s'.
             (∀ Y Z'.
               ((λY s' s. (s, s') ∈ (unroll G l e c)*) ∧. G⊢init≤n) Y s Z'
               ⟶ (((λY s' s. ∃ t b. (s,t) ∈ (unroll G l e c)*
                                    ∧ Y=⌊b⌋_e ∧ G⊢t −e−≻b→ s')
                  ∧. G⊢init≤n)←=False↓=◇) Y' s' Z')
             ⟶ G⊢Z −⟨l• While(e) c⟩_s≻→ (Y', s'))
      proof (clarsimp)
        fix Y' s' s
        assume asm:
          ∀ Z'. (Z', Norm s) ∈ (unroll G l e c)*
               ⟶ card (nyinitcls G s') ≤ n ∧
                  (∃ v. (∃ t. (Z', t) ∈ (unroll G l e c)* ∧ G⊢t −e−≻v→ s') ∧
                  (fst s' = None ⟶ ¬ the-Bool v)) ∧ Y' = ◇
        show Y' = ◇ ∧ G⊢Norm s −l• While(e) c→ s'
        proof −
          from asm obtain v t where
            — Z' gets instantiated with Norm s
            unroll: (Norm s, t) ∈ (unroll G l e c)* and
            eval-e: G⊢t −e−≻v→ s' and
            normal-termination: normal s' ⟶ ¬ the-Bool v and
             Y': Y' = ◇
            by auto
          from unroll eval-e normal-termination wt-e wf
          have G⊢Norm s −l• While(e) c→ s'
            by (rule unroll-while)
          with Y'
          show ?thesis
            by simp
        qed
      qed
    qed
  qed
qed


lemma MGFn-FVar:
  fixes A :: state triple set
```

assumes *mgf-init*: *G*,*A*⊢{=:*n*} ⟨*Init statDeclC*$\rangle_s$≻ {*G*→}
and *mgf-e*: *G*,*A*⊢{=:*n*} ⟨*e*$\rangle_e$≻ {*G*→}
and *wf*: *wf-prog G*
shows *G*,*A*⊢{=:*n*} ⟨{*accC*,*statDeclC*,*stat*}*e*..*fn*$\rangle_v$≻ {*G*→}
proof (*rule MGFn-free-wt-da-NormalConformI* [*rule-format*],*clarsimp*)
note *inj-term-simps* [*simp*]
fix *T L accC′ V*
assume *wt*: (|*prg* = *G*, *cls* = *accC′*, *lcl* = *L*|)⊢⟨{*accC*,*statDeclC*,*stat*}*e*..*fn*$\rangle_v$::*T*
then obtain *statC f* **where**
wt-e: (|*prg*=*G*, *cls*=*accC′*, *lcl*=*L*|)⊢*e*::−*Class statC* **and**
accfield: *accfield G accC′ statC fn* = *Some* (*statDeclC*,*f*) **and**
eq-accC: *accC*=*accC′* **and**
stat: *stat*=*is-static f*
by (*cases*) (*auto simp add*: *member-is-static-simp*)
let *?Q*=(λ*Y s1* (*x*,*s*) . *x* = *None* ∧
(*G*⊢*Norm s* −*Init statDeclC*→ *s1*) ∧
(∃ *E*. (|*prg*=*G*,*cls*=*accC′*,*lcl*=*L*|)⊢*dom* (*locals* (*store s1*)) »⟨*e*$\rangle_e$» *E*))
∧. *G*⊢*init*≤*n* ∧. (λ *s*. *s*::≼(*G*, *L*))
show *G*,*A*⊢{*Normal*
((λ*Y′ s′ s*. *s′* = *s* ∧ *abrupt s* = *None*) ∧. *G*⊢*init*≤*n* ∧.
(λ*s*. *s*::≼(*G*, *L*)) ∧.
(λ*s*. (|*prg*=*G*,*cls*=*accC′*,*lcl*=*L*|)
⊢ *dom* (*locals* (*store s*)) » ⟨{*accC*,*statDeclC*,*stat*}*e*..*fn*$\rangle_v$» *V*))
} {*accC*,*statDeclC*,*stat*}*e*..*fn*=≻
{λ*Y s′ s*. ∃ *vf*. *Y* = ⌊*vf*$\rfloor_v$ ∧
G⊢*s* −{*accC*,*statDeclC*,*stat*}*e*..*fn*=≻*vf*→ *s′*}
(**is** *G*,*A*⊢{*Normal ?P*} {*accC*,*statDeclC*,*stat*}*e*..*fn*=≻ {*?R*})
proof (*rule ax-derivs.FVar* [**where** *?Q*=*?Q*])
from *mgf-init*
show *G*,*A*⊢{*Normal ?P*} .*Init statDeclC*. {*?Q*}
proof (*rule MGFnD′* [*THEN conseq12*],*clarsimp*)
fix *s s′*
assume *conf-s*: *Norm s*::≼(*G*, *L*)
assume *da*: (|*prg*=*G*,*cls*=*accC′*,*lcl*=*L*|)
⊢ *dom* (*locals s*) »⟨{*accC*,*statDeclC*,*stat*}*e*..*fn*$\rangle_v$» *V*
assume *eval-init*: *G*⊢*Norm s* −*Init statDeclC*→ *s′*
show (∃ *E*. (|*prg*=*G*, *cls*=*accC′*, *lcl*=*L*|)⊢ *dom* (*locals* (*store s′*)) »⟨*e*$\rangle_e$» *E*) ∧
s′::≼(*G*, *L*)
proof −
from *da*
obtain *E* **where**
(|*prg*=*G*, *cls*=*accC′*, *lcl*=*L*|)⊢ *dom* (*locals s*) »⟨*e*$\rangle_e$» *E*
by *cases simp*
moreover
from *eval-init*
have *dom* (*locals s*) ⊆ *dom* (*locals* (*store s′*))
by (*rule dom-locals-eval-mono* [*elim-format*]) *simp*
ultimately obtain *E′* **where**
(|*prg*=*G*, *cls*=*accC′*, *lcl*=*L*|)⊢ *dom* (*locals* (*store s′*)) »⟨*e*$\rangle_e$» *E′*
by (*rule da-weakenE*)
moreover
have *s′*::≼(*G*, *L*)
proof −
have *wt-init*: (|*prg*=*G*, *cls*=*accC*, *lcl*=*L*|)⊢(*Init statDeclC*)::√
proof −
from *wf wt-e*
have *iscls-statC*: *is-class G statC*
by (*auto dest*: *ty-expr-is-type type-is-class*)
with *wf accfield*

```
          have iscls-statDeclC: is-class G statDeclC
            by (auto dest!: accfield-fields dest: fields-declC)
          thus ?thesis by simp
        qed
        obtain I where
          da-init: (|prg=G,cls=accC,lcl=L|)
             ⊢ dom (locals (store ((Norm s)::state))) »⟨Init statDeclC⟩s» I
          by (auto intro: da-Init [simplified] assigned.select-convs)
        from eval-init conf-s wt-init da-init  wf
        show ?thesis
          by (rule eval-type-soundE)
      qed
      ultimately show ?thesis by iprover
    qed
  qed
next
  from mgf-e
  show G,A⊢{?Q} e−≻ {λVal:a:. fvar statDeclC stat fn a ..; ?R}
  proof (rule MGFnD' [THEN conseq12],clarsimp)
    fix s0 s1 s2 E a
    let ?fvar = fvar statDeclC stat fn a s2
    assume eval-init: G⊢Norm s0 −Init statDeclC→ s1
    assume eval-e: G⊢s1 −e−≻a→ s2
    assume conf-s1: s1::≼(G, L)
    assume da-e: (|prg=G,cls=accC',lcl=L|)⊢ dom (locals (store s1)) »⟨e⟩e» E
    show G⊢Norm s0 −{accC,statDeclC,stat}e..fn=≻fst ?fvar→ snd ?fvar
    proof −
      obtain v s2' where
        v: v=fst ?fvar and s2': s2'=snd ?fvar
        by simp
      obtain s3 where
        s3: s3= check-field-access G accC' statDeclC fn stat a s2'
        by simp
      have eq-s3-s2': s3=s2'
      proof −
        from eval-e conf-s1 wt-e da-e wf obtain
          conf-s2: s2::≼(G, L)  and
          conf-a: normal s2 ⟹ G,store s2⊢a::≼Class statC
          by (rule eval-type-soundE) simp
        from accfield wt-e eval-init eval-e conf-s2 conf-a - wf
        show ?thesis
          by (rule  error-free-field-access
                  [where ?v=v and ?s2'=s2',elim-format])
            (simp add: s3 v s2' stat)+
      qed
      from eval-init eval-e
      show ?thesis
        apply (rule eval.FVar [where ?s2'=s2'])
        apply  (simp add: s2')
        apply  (simp add: s3 [symmetric]   eq-s3-s2' eq-accC s2' [symmetric])
        done
    qed
  qed
qed
qed
```

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 (rule MGFn-free-wt-da-NormalConformI [rule-format],clarsimp)
  fix T L accC C
  assume wt: ⦇prg=G,cls=accC,lcl=L⦈⊢In1r (c1 Finally c2)::T
  then obtain
    wt-c1: ⦇prg=G,cls=accC,lcl=L⦈⊢c1::√ and
    wt-c2: ⦇prg=G,cls=accC,lcl=L⦈⊢c2::√
    by cases simp
  let ?Q = (λY' s' s. normal s ∧ G⊢s −c1→ s' ∧
              (∃ C1. ⦇prg=G,cls=accC,lcl=L⦈⊢dom (locals (store s)) »⟨c1⟩ₛ» C1)
              ∧ s::≼(G, L))
            ∧. G⊢init≤n
  show G,A⊢{Normal
             ((λY' s' s. s' = s ∧ abrupt s = None) ∧. G⊢init≤n ∧.
             (λs. s::≼(G, L)) ∧.
             (λs. ⦇prg=G,cls=accC,lcl =L⦈
                   ⊢dom (locals (store s)) »⟨c1 Finally c2⟩ₛ» C))}
            .c1 Finally c2.
            {λY s' s. Y = ◇ ∧ G⊢s −c1 Finally c2→ s'}
    (is G,A⊢{Normal ?P} .c1 Finally c2. {?R})
  proof (rule ax-derivs.Fin [where ?Q=?Q])
    from mgf-c1
    show G,A⊢{Normal ?P} .c1. {?Q}
    proof (rule MGFnD' [THEN conseq12],clarsimp)
      fix s0
      assume ⦇prg=G,cls=accC,lcl=L⦈⊢ dom (locals s0) »⟨c1 Finally c2⟩ₛ» C
      thus ∃ C1. ⦇prg=G,cls=accC,lcl=L⦈⊢ dom (locals s0) »⟨c1⟩ₛ» C1
        by cases (auto simp add: inj-term-simps)
    qed
  next
    from mgf-c2
    show ∀ abr. G,A⊢{?Q ∧. (λs. abr = abrupt s) ;. abupd (λabr. None)} .c2.
          {abupd (abrupt-if (abr ≠ None) abr) .; ?R}
    proof (rule MGFnD' [THEN conseq12, THEN allI],clarsimp)
      fix s0 s1 s2 C1
      assume da-c1:⦇prg=G,cls=accC,lcl=L⦈⊢ dom (locals s0) »⟨c1⟩ₛ» C1
      assume conf-s0: Norm s0::≼(G, L)
      assume eval-c1: G⊢Norm s0 −c1→ s1
      assume eval-c2: G⊢abupd (λabr. None) s1 −c2→ s2
      show G⊢Norm s0 −c1 Finally c2
              → abupd (abrupt-if (∃ y. abrupt s1 = Some y) (abrupt s1)) s2
      proof −
        obtain abr1 str1 where s1: s1=(abr1,str1)
          by (cases s1)
        with eval-c1 eval-c2 obtain
          eval-c1': G⊢Norm s0 −c1→ (abr1,str1) and
          eval-c2': G⊢Norm str1 −c2→ s2
          by simp
        obtain s3 where
          s3: s3 = (if ∃ err. abr1 = Some (Error err)
                      then (abr1, str1)
                      else abupd (abrupt-if (abr1 ≠ None) abr1) s2)
          by simp
        from eval-c1' conf-s0 wt-c1 - wf
        have error-free (abr1,str1)
          by (rule eval-type-soundE) (insert da-c1,auto)
```

```
        with s3 have eq-s3: s3=abupd (abrupt-if (abr1 ≠ None) abr1) s2
          by (simp add: error-free-def)
        from eval-c1' eval-c2' s3
        show ?thesis
          by (rule eval.Fin [elim-format]) (simp add: s1 eq-s3)
      qed
    qed
  qed
qed


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
  fix l show abrupt s2 ≠ Some (Jump (Break l)) ∧
             abrupt s2 ≠ Some (Jump (Cont l))
  proof −
    fix accC from clsD have wt-init: (|prg=G, cls=accC, lcl=L|)⊢(Init D)::√
      by auto
    from eval-init wf
    have s1-no-jmp: ⋀ j. abrupt s1 ≠ Some (Jump j)
      by − (rule eval-statement-no-jump [OF - - - wt-init],auto)
    from eval-c - wt-c wf
    show ?thesis
      apply (rule jumpNestingOk-eval [THEN conjE, elim-format])
      using jmpOk s1-no-jmp
      apply auto
      done
  qed
qed


lemma MGFn-Body:
  assumes wf: wf-prog G
  and      mgf-init: G,A⊢{=:n} ⟨Init D⟩s≻ {G→}
  and      mgf-c: G,A⊢{=:n} ⟨c⟩s≻ {G→}
  shows  G,(A::state triple set)⊢{=:n} ⟨Body D c⟩e≻ {G→}
proof (rule MGFn-free-wt-da-NormalConformI [rule-format],clarsimp)
  fix T L accC E
  assume wt: (|prg=G, cls=accC,lcl=L|)⊢⟨Body D c⟩e::T
  let ?Q=(λY' s' s. normal s ∧ G⊢s −Init D→ s' ∧ jumpNestingOkS {Ret} c)
        ∧. G⊢init≤n
  show G,A⊢{Normal
             ((λY' s' s. s' = s ∧ fst s = None) ∧. G⊢init≤n ∧.
              (λs. s::≼(G, L)) ∧.
              (λs. (|prg=G,cls=accC,lcl=L|)
                     ⊢ dom (locals (store s)) »⟨Body D c⟩e» E))}
           Body D c−≻
           {λY s' s. ∃ v. Y = In1 v ∧ G⊢s −Body D c−≻v→ s'}
    (is G,A⊢{Normal ?P} Body D c−≻ {?R})
  proof (rule ax-derivs.Body [where ?Q=?Q])
    from mgf-init
```

```
    show G,A⊢{Normal ?P} .Init D. {?Q}
    proof (rule MGFnD' [THEN conseq12],clarsimp)
      fix s0
      assume da: (|prg=G,cls=accC,lcl=L|)⊢ dom (locals s0) »⟨Body D c⟩e» E
      thus jumpNestingOkS {Ret} c
        by cases simp
    qed
  next
    from mgf-c
    show G,A⊢{?Q}.c.{λs.. abupd (absorb Ret) .; ?R←⌊the (locals s Result)⌋e}
    proof (rule MGFnD' [THEN conseq12],clarsimp)
      fix s0 s1 s2
      assume eval-init: G⊢Norm s0 −Init D→ s1
      assume eval-c: G⊢s1 −c→ s2
      assume nestingOk: jumpNestingOkS {Ret} c
      show G⊢Norm s0 −Body D c−≻the (locals (store s2) Result)
              → abupd (absorb Ret) s2
      proof −
        from wt obtain d where
          d: class G D=Some d and
          wt-c: (|prg = G, cls = accC, lcl = L|)⊢c::√
          by cases auto
        obtain s3 where
          s3: s3= (if ∃ l. fst s2 = Some (Jump (Break l)) ∨
                           fst s2 = Some (Jump (Cont l))
                      then abupd (λx. Some (Error CrossMethodJump)) s2
                      else s2)
          by simp
        from eval-init eval-c nestingOk wt-c d wf
        have eq-s3-s2: s3=s2
          by (rule Body-no-break [elim-format]) (simp add: s3)
        from eval-init eval-c s3
        show ?thesis
          by (rule eval.Body [elim-format]) (simp add: eq-s3-s2)
      qed
    qed
  qed
qed


lemma MGFn-lemma:
  assumes mgf-methds:
           ⋀ n. ∀ C sig. G,(A::state triple set)⊢{=:n} ⟨Methd C sig⟩e≻ {G→}
  and wf: wf-prog G
  shows ⋀ t. G,A⊢{=:n} t≻ {G→}
proof (induct rule: full-nat-induct)
  fix n t
  assume hyp: ∀ m. Suc m ≤ n ⟶ (∀ t. G,A⊢{=:m} t≻ {G→})
  show G,A⊢{=:n} t≻ {G→}
  proof −
  {
    fix v e c es
    have G,A⊢{=:n} ⟨v⟩v≻ {G→} and
      G,A⊢{=:n} ⟨e⟩e≻ {G→} and
      G,A⊢{=:n} ⟨c⟩s≻ {G→} and
      G,A⊢{=:n} ⟨es⟩l≻ {G→}
    proof (induct rule: compat-var.induct compat-expr.induct compat-stmt.induct compat-expr-list.induct)
      case (LVar v)
      show G,A⊢{=:n} ⟨LVar v⟩v≻ {G→}
```

```
      apply (rule MGFn-NormalI)
      apply (rule ax-derivs.LVar [THEN conseq1])
      apply (clarsimp)
      apply (rule eval.LVar)
      done
  next
    case (FVar accC statDeclC stat e fn)
    from MGFn-Init [OF hyp] and ‹G,A⊢{=:n} ⟨e⟩ₑ≻ {G→}› and wf
    show ?case
      by (rule MGFn-FVar)
  next
    case (AVar e1 e2)
    note mgf-e1 = ‹G,A⊢{=:n} ⟨e1⟩ₑ≻ {G→}›
    note mgf-e2 = ‹G,A⊢{=:n} ⟨e2⟩ₑ≻ {G→}›
    show G,A⊢{=:n} ⟨e1.[e2]⟩ᵥ≻ {G→}
      apply (rule MGFn-NormalI)
      apply (rule ax-derivs.AVar)
      apply (rule MGFnD [OF mgf-e1, THEN ax-NormalD])
      apply (rule allI)
      apply (rule MGFnD' [OF mgf-e2, THEN conseq12])
      apply (fastforce intro: eval.AVar)
      done
  next
    case (InsInitV c v)
    show ?case
      by (rule MGFn-NormalI) (rule ax-derivs.InsInitV)
  next
    case (NewC C)
    show ?case
      apply (rule MGFn-NormalI)
      apply (rule ax-derivs.NewC)
      apply (rule MGFn-InitD [OF hyp, THEN conseq2])
      apply (fastforce intro: eval.NewC)
      done
  next
    case (NewA T e)
    thus ?case
      apply −
      apply (rule MGFn-NormalI)
      apply (rule ax-derivs.NewA
             [where ?Q = (λY' s' s. normal s ∧ G⊢s −In1r (init-comp-ty T)
                          ≻→ (Y',s')) ∧. G⊢init≤n])
      apply (simp add: init-comp-ty-def split: if-split)
      apply (rule conjI, clarsimp)
      apply  (rule MGFn-InitD [OF hyp, THEN conseq2])
      apply  (clarsimp intro: eval.Init)
      apply clarsimp
      apply (rule ax-derivs.Skip [THEN conseq1])
      apply (clarsimp intro: eval.Skip)
      apply (erule MGFnD' [THEN conseq12])
      apply (fastforce intro: eval.NewA)
      done
  next
    case (Cast C e)
    thus ?case
      apply −
      apply (rule MGFn-NormalI)
      apply (erule MGFnD'[THEN conseq12,THEN ax-derivs.Cast])
      apply (fastforce intro: eval.Cast)
```

```
    done
next
  case (Inst e C)
  thus ?case
    apply -
    apply (rule MGFn-NormalI)
    apply (erule MGFnD'[THEN conseq12,THEN ax-derivs.Inst])
    apply (fastforce intro: eval.Inst)
    done
next
  case (Lit v)
  show ?case
    apply -
    apply (rule MGFn-NormalI)
    apply (rule ax-derivs.Lit [THEN conseq1])
    apply (fastforce intro: eval.Lit)
    done
next
  case (UnOp unop e)
  thus ?case
    apply -
    apply (rule MGFn-NormalI)
    apply (rule ax-derivs.UnOp)
    apply (erule MGFnD' [THEN conseq12])
    apply (fastforce intro: eval.UnOp)
    done
next
  case (BinOp binop e1 e2)
  thus ?case
    apply -
    apply (rule MGFn-NormalI)
    apply (rule ax-derivs.BinOp)
    apply  (erule MGFnD [THEN ax-NormalD])
    apply (rule allI)
    apply (case-tac need-second-arg binop v1)
    apply  simp
    apply  (erule MGFnD' [THEN conseq12])
    apply  (fastforce intro: eval.BinOp)
    apply simp
    apply (rule ax-Normal-cases)
    apply  (rule ax-derivs.Skip [THEN conseq1])
    apply  clarsimp
    apply  (rule eval-BinOp-arg2-indepI)
    apply   simp
    apply  simp
    apply (rule ax-derivs.Abrupt [THEN conseq1], clarsimp simp add: Let-def)
    apply (fastforce intro: eval.BinOp)
    done
next
  case Super
  show ?case
    apply -
    apply (rule MGFn-NormalI)
    apply (rule ax-derivs.Super [THEN conseq1])
    apply (fastforce intro: eval.Super)
    done
next
  case (Acc v)
  thus ?case
```

```
    apply –
    apply (rule MGFn-NormalI)
    apply (erule MGFnD'[THEN conseq12,THEN ax-derivs.Acc])
    apply (fastforce intro: eval.Acc simp add: split-paired-all)
    done
next
  case (Ass v e)
  thus G,A⊢{=:n} ⟨v:=e⟩ₑ≻ {G→}
    apply –
    apply (rule MGFn-NormalI)
    apply (rule ax-derivs.Ass)
    apply  (erule MGFnD [THEN ax-NormalD])
    apply (rule allI)
    apply (erule MGFnD'[THEN conseq12])
    apply (fastforce intro: eval.Ass simp add: split-paired-all)
    done
next
  case (Cond e1 e2 e3)
  thus G,A⊢{=:n} ⟨e1 ? e2 : e3⟩ₑ≻ {G→}
    apply –
    apply (rule MGFn-NormalI)
    apply (rule ax-derivs.Cond)
    apply  (erule MGFnD [THEN ax-NormalD])
    apply (rule allI)
    apply (rule ax-Normal-cases)
    prefer 2
    apply  (rule ax-derivs.Abrupt [THEN conseq1],clarsimp simp add: Let-def)
    apply  (fastforce intro: eval.Cond)
    apply (case-tac b)
    apply  simp
    apply  (erule MGFnD'[THEN conseq12])
    apply  (fastforce intro: eval.Cond)
    apply simp
    apply (erule MGFnD'[THEN conseq12])
    apply (fastforce intro: eval.Cond)
    done
next
  case (Call accC statT mode e mn pTs' ps)
  note mgf-e = ‹G,A⊢{=:n} ⟨e⟩ₑ≻ {G→}›
  note mgf-ps = ‹G,A⊢{=:n} ⟨ps⟩ₗ≻ {G→}›
  from mgf-methds mgf-e mgf-ps wf
  show G,A⊢{=:n} ⟨{accC,statT,mode}e·mn({pTs'}ps)⟩ₑ≻ {G→}
    by (rule MGFn-Call)
next
  case (Methd D mn)
  from mgf-methds
  show G,A⊢{=:n} ⟨Methd D mn⟩ₑ≻ {G→}
    by simp
next
  case (Body D c)
  note mgf-c = ‹G,A⊢{=:n} ⟨c⟩ₛ≻ {G→}›
  from wf MGFn-Init [OF hyp] mgf-c
  show G,A⊢{=:n} ⟨Body D c⟩ₑ≻ {G→}
    by (rule MGFn-Body)
next
  case (InsInitE c e)
  show ?case
    by (rule MGFn-NormalI) (rule ax-derivs.InsInitE)
next
```

```
    case (Callee l e)
    show ?case
      by (rule MGFn-NormalI) (rule ax-derivs.Callee)
  next
    case Skip
    show ?case
      apply −
      apply (rule MGFn-NormalI)
      apply (rule ax-derivs.Skip [THEN conseq1])
      apply (fastforce intro: eval.Skip)
      done
  next
    case (Expr e)
    thus ?case
      apply −
      apply (rule MGFn-NormalI)
      apply (erule MGFnD'[THEN conseq12,THEN ax-derivs.Expr])
      apply (fastforce intro: eval.Expr)
      done
  next
    case (Lab l c)
    thus G,A⊢{=:n} ⟨l· c⟩ₛ≻ {G→}
      apply −
      apply (rule MGFn-NormalI)
      apply (erule MGFnD' [THEN conseq12, THEN ax-derivs.Lab])
      apply (fastforce intro: eval.Lab)
      done
  next
    case (Comp c1 c2)
    thus G,A⊢{=:n} ⟨c1;; c2⟩ₛ≻ {G→}
      apply −
      apply (rule MGFn-NormalI)
      apply (rule ax-derivs.Comp)
      apply  (erule MGFnD [THEN ax-NormalD])
      apply (erule MGFnD' [THEN conseq12])
      apply (fastforce intro: eval.Comp)
      done
  next
    case (If' e c1 c2)
    thus G,A⊢{=:n} ⟨If(e) c1 Else c2⟩ₛ≻ {G→}
      apply −
      apply (rule MGFn-NormalI)
      apply (rule ax-derivs.If)
      apply  (erule MGFnD [THEN ax-NormalD])
      apply (rule allI)
      apply (rule ax-Normal-cases)
      prefer 2
      apply  (rule ax-derivs.Abrupt [THEN conseq1],clarsimp simp add: Let-def)
      apply  (fastforce intro: eval.If)
      apply (case-tac b)
      apply  simp
      apply  (erule MGFnD' [THEN conseq12])
      apply  (fastforce intro: eval.If)
      apply simp
      apply (erule MGFnD' [THEN conseq12])
      apply (fastforce intro: eval.If)
      done
  next
    case (Loop l e c)
```

```
    note mgf-e = ‹G,A⊢{=:n} ⟨e⟩e≻ {G→}›
    note mgf-c = ‹G,A⊢{=:n} ⟨c⟩s≻ {G→}›
    from mgf-e mgf-c wf
    show G,A⊢{=:n} ⟨l• While(e) c⟩s≻ {G→}
      by (rule MGFn-Loop)
  next
    case (Jmp j)
    thus ?case
      apply -
      apply (rule MGFn-NormalI)
      apply (rule ax-derivs.Jmp [THEN conseq1])
      apply (auto intro: eval.Jmp)
      done
  next
    case (Throw e)
    thus ?case
      apply -
      apply (rule MGFn-NormalI)
      apply (erule MGFnD' [THEN conseq12, THEN ax-derivs.Throw])
      apply (fastforce intro: eval.Throw)
      done
  next
    case (TryC c1 C vn c2)
    thus G,A⊢{=:n} ⟨Try c1 Catch(C vn) c2⟩s≻ {G→}
      apply -
      apply (rule MGFn-NormalI)
      apply (rule ax-derivs.Try [where
        ?Q = (λY' s' s. normal s ∧ (∃s''. G⊢s −⟨c1⟩s≻→ (Y',s'') ∧
                        G⊢s'' −sxalloc→ s')) ∧. G⊢init≤n])
      apply  (erule MGFnD [THEN ax-NormalD, THEN conseq2])
      apply  (fastforce elim: sxalloc-gext [THEN card-nyinitcls-gext])
      apply (erule MGFnD'[THEN conseq12])
      apply (fastforce intro: eval.Try)
      apply (fastforce intro: eval.Try)
      done
  next
    case (Fin c1 c2)
    note mgf-c1 = ‹G,A⊢{=:n} ⟨c1⟩s≻ {G→}›
    note mgf-c2 = ‹G,A⊢{=:n} ⟨c2⟩s≻ {G→}›
    from wf mgf-c1 mgf-c2
    show G,A⊢{=:n} ⟨c1 Finally c2⟩s≻ {G→}
      by (rule MGFn-Fin)
  next
    case (FinA abr c)
    show ?case
      by (rule MGFn-NormalI) (rule ax-derivs.FinA)
  next
    case (Init C)
    from hyp
    show G,A⊢{=:n} ⟨Init C⟩s≻ {G→}
      by (rule MGFn-Init)
  next
    case Nil-expr
    show G,A⊢{=:n} ⟨[]⟩l≻ {G→}
      apply -
      apply (rule MGFn-NormalI)
      apply (rule ax-derivs.Nil [THEN conseq1])
      apply (fastforce intro: eval.Nil)
      done
```

next
case (*Cons-expr e es*)
thus $G,A\vdash\{=:n\}\ \langle e\#\ es\rangle_l\succ\ \{G\rightarrow\}$
apply −
apply (*rule MGFn-NormalI*)
apply (*rule ax-derivs.Cons*)
apply (*erule MGFnD* [*THEN ax-NormalD*])
apply (*rule allI*)
apply (*erule MGFnD'*[*THEN conseq12*])
apply (*fastforce intro*: *eval.Cons*)
done
qed
}
thus *?thesis*
by (*cases rule*: *term-cases*) *auto*
qed
qed

lemma *MGF-asm*:
$[\![\forall\ C\ sig.\ is\text{-}methd\ G\ C\ sig \longrightarrow G,A\vdash\{\doteq\}\ In1l\ (Methd\ C\ sig)\succ\ \{G\rightarrow\};\ wf\text{-}prog\ G]\!]$
$\Longrightarrow G,(A{::}state\ triple\ set)\vdash\{\doteq\}\ t\succ\ \{G\rightarrow\}$
apply (*simp* (*no-asm-use*) *add*: *MGF-MGFn-iff*)
apply (*rule allI*)
apply (*rule MGFn-lemma*)
apply (*intro strip*)
apply (*rule MGFn-free-wt*)
apply (*force dest*: *wt-Methd-is-methd*)
apply *assumption*
done

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-call'U*
shows $\forall A.\ A \subseteq uA \longrightarrow n \leq card\ uA \longrightarrow card\ A = card\ uA - n$
$\longrightarrow (\forall t.\ P\ A\ \{mgf\ t\})$
using *finU uA*
apply −
apply (*induct-tac n*)
apply (*tactic ALLGOALS* (*clarsimp-tac* ***context***))
apply (*tactic* ‹*dresolve-tac* ***context*** [*Thm.permute-prems 0 1* @{*thm card-seteq*}] *1*›)
apply *simp*
apply (*erule finite-imageI*)
apply (*simp add*: *MGF-asm ax-derivs-asm*)
apply (*rule MGF-asm*)
apply (*rule ballI*)
apply (*case-tac mgf-call pn* ∈ *A*)
apply (*fast intro*: *ax-derivs-asm*)
apply (*rule MGF-nested-Methd*)
apply (*rule ballI*)
apply (*drule spec*, *erule impE*, *erule-tac* [*2*] *impE*, *erule-tac* [*3*] *spec*)
apply *hypsubst-thin*
apply *fast*

apply (*drule finite-subset*)
apply (*erule finite-imageI*)
apply *auto*
done

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 $P\ \{\}\ \{mgf\ t\}$
using *ax-derivs-asm MGF-nested-Methd MGF-asm finU*
by (*rule nesting-lemma'*) (*auto intro*!: *le-refl*)

lemma *MGF-nested-Methd*: $[\![$
$G, insert\ (\{Normal\ \doteq\}\ \langle Methd\ \ C\ sig\rangle_e \succ\{G\rightarrow\})\ A$
$\vdash\{Normal\ \doteq\}\ \langle body\ G\ C\ sig\rangle_e \succ\{G\rightarrow\}$
$]\!] \Longrightarrow\ \ G,A\vdash\{Normal\ \doteq\}\ \ \langle Methd\ \ C\ sig\rangle_e \succ\{G\rightarrow\}$
apply (*unfold MGF-def*)
apply (*rule ax-MethdN*)
apply (*erule conseq2*)
apply *clarsimp*
apply (*erule MethdI*)
done

lemma *MGF-deriv*: $wf\text{-}prog\ G \Longrightarrow G,(\{\}{::}state\ triple\ set)\vdash\{\doteq\}\ t\succ\ \{G\rightarrow\}$
apply (*rule MGFNormalI*)
apply (*rule-tac* $mgf = \lambda t.\ \{Normal\ \doteq\}\ t\succ\ \{G\rightarrow\}$ **and**
$bdy = \lambda\ (C,sig)\ .\{\langle body\ G\ C\ sig\rangle_e\ \}$ **and**
$f = \lambda\ (C,sig)\ .\ \langle Methd\ C\ sig\rangle_e$ **in** *nesting-lemma*)
apply (*erule ax-derivs.asm*)
apply (*clarsimp simp add*: *split-tupled-all*)
apply (*erule MGF-nested-Methd*)
apply (*erule-tac* [*2*] *finite-is-methd* [*OF wf-ws-prog*])
apply (*rule MGF-asm* [*THEN MGFNormalD*])
apply (*auto intro*: *MGFNormalI*)
done

simultaneous version

lemma *MGF-simult-Methd-lemma*: *finite ms* $\Longrightarrow$
$G,A \cup (\lambda(C,sig).\ \{Normal\ \doteq\}\ \langle Methd\ \ C\ sig\rangle_e\succ\ \{G\rightarrow\})\ \text{`}\ ms$
$|\vdash(\lambda(C,sig).\ \{Normal\ \doteq\}\ \langle body\ G\ C\ sig\rangle_e\succ\ \{G\rightarrow\})\ \text{`}\ ms \Longrightarrow$
$G,A|\vdash(\lambda(C,sig).\ \{Normal\ \doteq\}\ \langle Methd\ \ C\ sig\rangle_e\succ\ \{G\rightarrow\})\ \text{`}\ ms$
apply (*unfold MGF-def*)
apply (*rule ax-derivs.Methd* [*unfolded mtriples-def*])
apply (*erule ax-finite-pointwise*)
prefer *2*
apply (*rule ax-derivs.asm*)
apply *fast*
apply *clarsimp*
apply (*rule conseq2*)
apply (*erule* (*1*) *ax-methods-spec*)

apply *clarsimp*
apply (*erule eval-Methd*)
done

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 (*case-prod* (*is-methd G*))
apply (*frule finite-is-methd* [*OF wf-ws-prog*])
apply (*rule MGF-simult-Methd-lemma*)
apply *assumption*
apply (*erule ax-finite-pointwise*)
prefer *2*
apply (*rule ax-derivs.asm*)
apply *blast*
apply *clarsimp*
apply (*rule MGF-asm* [*THEN MGFNormalD*])
apply (*auto intro*: *MGFNormalI*)
done

corollaries

lemma *eval-to-evaln*: $[\![G\vdash s\ -t\succ\rightarrow\ (Y',\ s');type\text{-}ok\ G\ t\ s;\ wf\text{-}prog\ G]\!]$
$\Longrightarrow\ \ \exists\,n.\ G\vdash s\ -t\succ-n\rightarrow\ (Y',\ s')$
apply (*cases normal s*)
apply (*force simp add*: *type-ok-def intro*: *eval-evaln*)
apply (*force intro*: *evaln.Abrupt*)
done

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 (*rule ax-no-hazard*)
from *mgf*
have $G,(\{\}::state\ triple\ set)\vdash\{\doteq\}\ t\succ\ \{\lambda Y\ s'\ s.\ G\vdash s\ -t\succ\rightarrow\ (Y,\ s')\}$
by (*unfold MGF-def*)
thus $G,(\{\}::state\ triple\ set)\vdash\{P\ \wedge.\ type\text{-}ok\ G\ t\}\ t\succ\ \{Q\}$
proof (*rule conseq12*,*clarsimp*)
fix $Y\ s\ Z\ Y'\ s'$
assume *P*: $P\ Y\ s\ Z$
assume *type-ok*: *type-ok G t s*
assume *eval-t*: $G\vdash s\ -t\succ\rightarrow\ (Y',\ s')$
show $Q\ Y'\ s'\ Z$
proof −
from *eval-t type-ok wf*
obtain *n* **where** *evaln*: $G\vdash s\ -t\succ-n\rightarrow\ (Y',\ s')$
by (*rule eval-to-evaln* [*elim-format*]) *iprover*
from *valid* **have**
valid-expanded:
$\forall\,n\ Y\ s\ Z.\ P\ Y\ s\ Z\ \longrightarrow\ type\text{-}ok\ G\ t\ s$
$\longrightarrow\ (\forall\,Y'\ s'.\ G\vdash s\ -t\succ-n\rightarrow\ (Y',\ s')\ \longrightarrow\ Q\ Y'\ s'\ Z)$
by (*simp add*: *ax-valids-def triple-valid-def*)
from *P type-ok evaln*
show $Q\ Y'\ s'\ Z$
by (*rule valid-expanded* [*rule-format*])
qed

qed
qed

theorem *ax-complete*:
assumes *wf*: *wf-prog G*
and *valid*: G,{}⊨{P::*state assn*} $t\succ$ {Q}
shows G,({}::*state triple set*)⊢{P} $t\succ$ {Q}
proof −
from *wf* **have** G,({}::*state triple set*)⊢{$\doteq$} $t\succ$ {$G\rightarrow$}
by (*rule MGF-deriv*)
from *valid this wf*
show *?thesis*
by (*rule MGF-complete*)
qed

end

Chapter 25

AxExample

1 Example of a proof based on the Bali axiomatic semantics

theory *AxExample*
imports *AxSem Example*
begin

definition
arr-inv :: *st* ⇒ *bool* **where**
arr-inv = (λ*s*. ∃ *obj a T el*. *globs s* (*Stat Base*) = *Some obj* ∧
values obj (*Inl* (*arr*, *Base*)) = *Some* (*Addr a*) ∧
heap s a = *Some* (|*tag*=*Arr T 2*,*values*=*el*|))

lemma *arr-inv-new-obj*:
⋀*a*. ⟦*arr-inv s*; *new-Addr* (*heap s*)=*Some a*⟧ ⟹ *arr-inv* (*gupd*(*Inl a*↦*x*) *s*)
apply (*unfold arr-inv-def*)
apply (*force dest*!: *new-AddrD2*)
done

lemma *arr-inv-set-locals* [*simp*]: *arr-inv* (*set-locals l s*) = *arr-inv s*
apply (*unfold arr-inv-def*)
apply (*simp* (*no-asm*))
done

lemma *arr-inv-gupd-Stat* [*simp*]:
Base ≠ *C* ⟹ *arr-inv* (*gupd*(*Stat C*↦*obj*) *s*) = *arr-inv s*
apply (*unfold arr-inv-def*)
apply (*simp* (*no-asm-simp*))
done

lemma *ax-inv-lupd* [*simp*]: *arr-inv* (*lupd*(*x*↦*y*) *s*) = *arr-inv s*
apply (*unfold arr-inv-def*)
apply (*simp* (*no-asm*))
done

declare *if-split-asm* [*split del*]
declare *lvar-def* [*simp*]

ML ‹
fun inst1-tac ctxt s t xs st =

```
  (case AList.lookup (op =) (rev (Term.add-var-names (Thm.prop-of st) [])) s of
    SOME i => PRIMITIVE (Rule-Insts.read-instantiate ctxt [(((s, i), Position.none), t)] xs) st
  | NONE => Seq.empty);

fun ax-tac ctxt =
  REPEAT o resolve-tac ctxt [allI] THEN'
  resolve-tac ctxt
    @{thms ax-Skip ax-StatRef ax-MethdN ax-Alloc ax-Alloc-Arr ax-SXAlloc-Normal ax-derivs.intros(8−)};
›


theorem ax-test: tprg,({}::'a triple set)⊢
  {Normal (λY s Z::'a. heap-free four s ∧ ¬initd Base s ∧ ¬ initd Ext s)}
  .test [Class Base].
  {λY s Z. abrupt s = Some (Xcpt (Std IndOutBound))}
apply (unfold test-def arr-viewed-from-def)
apply (tactic ax-tac context 1 )
defer
apply  (tactic ax-tac context 1 )
defer
apply   (tactic ‹inst1-tac context Q
           λY s Z. arr-inv (snd s) ∧ tprg,s⊢catch SXcpt NullPointer []›)
prefer 2
apply   simp
apply  (rule-tac P' = Normal (λY s Z. arr-inv (snd s)) in conseq1)
prefer 2
apply   clarsimp
apply  (rule-tac Q' = (λY s Z. Q Y s Z)←=False↓=◇ and Q = Q for Q in conseq2)
prefer 2
apply   simp
apply  (tactic ax-tac context 1 )
prefer 2
apply   (rule ax-impossible [THEN conseq1], clarsimp)
apply   (rule-tac P' = Normal P and P = P for P in conseq1)
prefer 2
apply   clarsimp
apply  (tactic ax-tac context 1)
apply  (tactic ax-tac context 1 )
prefer 2
apply   (rule ax-subst-Val-allI)
apply   (tactic ‹inst1-tac context P' λa. Normal (PP a←x) [PP, x]›)
apply   (simp del: avar-def2 peek-and-def2)
apply   (tactic ax-tac context 1)
apply  (tactic ax-tac context 1)

apply   (rule-tac Q' = Normal (λVar:(v, f) u ua. fst (snd (avar tprg (Intg 2) v u)) = Some (Xcpt (Std
IndOutBound))) in conseq2)
prefer 2
apply   (clarsimp simp add: split-beta)
apply  (tactic ax-tac context 1 )
apply   (tactic ax-tac context 2 )
apply  (rule ax-derivs.Done [THEN conseq1])
apply  (clarsimp simp add: arr-inv-def inited-def in-bounds-def)
defer
apply (rule ax-SXAlloc-catch-SXcpt)
apply (rule-tac Q' = (λY (x, s) Z. x = Some (Xcpt (Std NullPointer)) ∧ arr-inv s) ∧. heap-free two in
conseq2)
prefer 2
apply  (simp add: arr-inv-new-obj)
```

apply (*tactic ax-tac* ***context*** *1*)
apply (*rule-tac C* = *Ext* **in** *ax-Call-known-DynT*)
apply (*unfold DynT-prop-def*)
apply (*simp* (*no-asm*))
apply (*intro strip*)
apply (*rule-tac P′* = *Normal P* **and** *P* = *P* **for** *P* **in** *conseq1*)
apply (*tactic ax-tac* ***context*** *1*)
apply (*rule ax-thin* [*OF* - *empty-subsetI*])
apply (*simp* (*no-asm*) *add*: *body-def2*)
apply (*tactic ax-tac* ***context*** *1*)

defer
apply (*simp* (*no-asm*))
apply (*tactic ax-tac* ***context*** *1*)

apply (*rule-tac* [*2*] *ax-derivs.Abrupt*)

apply (*rule ax-derivs.Expr*)
apply (*tactic ax-tac* ***context*** *1*)
prefer *2*
apply (*rule ax-subst-Var-allI*)
apply (*tactic* ‹*inst1-tac* ***context*** *P′* λ*a vs l vf*. *PP a vs l vf*←*x* ∧. *p* [*PP*, *x*, *p*]›)
apply (*rule allI*)
apply (*tactic* ‹*simp-tac* (***context*** *delloop split-all-tac delsimps* [@{*thm peek-and-def2*}, @{*thm heap-def2*}, @{*thm subst-res-def2*}, @{*thm normal-def2*}]) *1*›)
apply (*rule ax-derivs.Abrupt*)
apply (*simp* (*no-asm*))
apply (*tactic ax-tac* ***context*** *1*)
apply (*tactic ax-tac* ***context*** *2*, *tactic ax-tac* ***context*** *2*, *tactic ax-tac* ***context*** *2*)
apply (*tactic ax-tac* ***context*** *1*)
apply (*tactic* ‹*inst1-tac* ***context*** *R* λ*a′*. *Normal* ((λ*Vals*:*vs* (*x*, *s*) *Z*. *arr-inv s* ∧ *inited Ext* (*globs s*) ∧ *a′* ≠ *Null* ∧ *vs* = [*Null*]) ∧. *heap-free two*) []›)
apply *fastforce*
prefer *4*
apply (*rule ax-derivs.Done* [*THEN conseq1*],*force*)
apply (*rule ax-subst-Val-allI*)
apply (*tactic* ‹*inst1-tac* ***context*** *P′* λ*a*. *Normal* (*PP a*←*x*) [*PP*, *x*]›)
apply (*simp* (*no-asm*) *del*: *peek-and-def2 heap-free-def2 normal-def2 o-apply*)
apply (*tactic ax-tac* ***context*** *1*)
prefer *2*
apply (*rule ax-subst-Val-allI*)
apply (*tactic* ‹*inst1-tac* ***context*** *P′* λ*aa v*. *Normal* (*QQ aa v*←*y*) [*QQ*, *y*]›)
apply (*simp del*: *peek-and-def2 heap-free-def2 normal-def2*)
apply (*tactic ax-tac* ***context*** *1*)
apply (*tactic ax-tac* ***context*** *1*)
apply (*tactic ax-tac* ***context*** *1*)
apply (*tactic ax-tac* ***context*** *1*)

apply (*simp* (*no-asm*))

apply (*rule-tac Q′* = *Normal* ((λ*Y* (*x*, *s*) *Z*. *arr-inv s* ∧ (∃ *a*. *the* (*locals s* (*VName e*)) = *Addr a* ∧ *obj-class* (*the* (*globs s* (*Inl a*))) = *Ext* ∧
invocation-declclass tprg IntVir s (*the* (*locals s* (*VName e*))) (*ClassT Base*)
(|*name* = *foo*, *parTs* = [*Class Base*]|) = *Ext*)) ∧. *initd Ext* ∧. *heap-free two*)
in *conseq2*)
prefer *2*
apply *clarsimp*
apply (*tactic ax-tac* ***context*** *1*)
apply (*tactic ax-tac* ***context*** *1*)

defer
apply (*rule ax-subst-Var-allI*)
apply (*tactic ‹inst1-tac* ***context*** *P′ λvf. Normal* (*PP vf* ∧. *p*) [*PP*, *p*]›)
apply (*simp* (*no-asm*) *del*: *split-paired-All peek-and-def2 initd-def2 heap-free-def2 normal-def2*)
apply (*tactic ax-tac* ***context*** *1*)
apply (*tactic ax-tac* ***context*** *1*)

apply (*rule-tac Q′* = *Normal* ((*λY s Z. arr-inv* (*store s*) ∧ *vf*=*lvar* (*VName e*) (*store s*)) ∧. *heap-free three* ∧. *initd Ext*) **in** *conseq2*)
prefer *2*
apply (*simp add*: *invocation-declclass-def dynmethd-def*)
apply (*unfold dynlookup-def*)
apply (*simp add*: *dynmethd-Ext-foo*)
apply (*force elim*!: *arr-inv-new-obj atleast-free-SucD atleast-free-weaken*)

apply (*rule ax-InitS*)
apply *force*
apply (*simp* (*no-asm*))
apply (*tactic ‹simp-tac* (***context*** *delloop split-all-tac*) *1*›)
apply (*rule ax-Init-Skip-lemma*)
apply (*tactic ‹simp-tac* (***context*** *delloop split-all-tac*) *1*›)
apply (*rule ax-InitS* [*THEN conseq1*])
apply *force*
apply (*simp* (*no-asm*))
apply (*unfold arr-viewed-from-def*)
apply (*rule allI*)
apply (*rule-tac P′* = *Normal P* **and** *P* = *P* **for** *P* **in** *conseq1*)
apply (*tactic ‹simp-tac* (***context*** *delloop split-all-tac*) *1*›)
apply (*tactic ax-tac* ***context*** *1*)
apply (*tactic ax-tac* ***context*** *1*)
apply (*rule-tac* [*2*] *ax-subst-Var-allI*)
apply (*tactic ‹inst1-tac* ***context*** *P′ λvf l vfa. Normal* (*P vf l vfa*) [*P*]›)
apply (*tactic ‹simp-tac* (***context*** *delloop split-all-tac delsimps* [@{*thm split-paired-All*}, @{*thm peek-and-def2*}, @{*thm heap-free-def2*}, @{*thm initd-def2*}, @{*thm normal-def2*}, @{*thm supd-lupd*}]) *2*›)
apply (*tactic ax-tac* ***context*** *2*)
apply (*tactic ax-tac* ***context*** *3*)
apply (*tactic ax-tac* ***context*** *3*)
apply (*tactic ‹inst1-tac* ***context*** *P λvf l vfa. Normal* (*P vf l vfa*←◇) [*P*]›)
apply (*tactic ‹simp-tac* (***context*** *delloop split-all-tac*) *2*›)
apply (*tactic ax-tac* ***context*** *2*)
apply (*tactic ax-tac* ***context*** *1*)
apply (*tactic ax-tac* ***context*** *2*)
apply (*rule ax-derivs.Done* [*THEN conseq1*])
apply (*tactic ‹inst1-tac* ***context*** *Q λvf. Normal* ((*λY s Z. vf*=*lvar* (*VName e*) (*snd s*)) ∧. *heap-free four* ∧. *initd Base* ∧. *initd Ext*) []›)
apply (*clarsimp split del*: *if-split*)
apply (*frule atleast-free-weaken* [*THEN atleast-free-weaken*])
apply (*drule initedD*)
apply (*clarsimp elim*!: *atleast-free-SucD simp add*: *arr-inv-def*)
apply *force*
apply (*tactic ‹simp-tac* (***context*** *delloop split-all-tac*) *1*›)
apply (*rule ax-triv-Init-Object* [*THEN peek-and-forget2*, *THEN conseq1*])
apply (*rule wf-tprg*)
apply *clarsimp*
apply (*tactic ‹inst1-tac* ***context*** *P λvf. Normal* ((*λY s Z. vf* = *lvar* (*VName e*) (*snd s*)) ∧. *heap-free four* ∧. *initd Ext*) []›)
apply *clarsimp*
apply (*tactic ‹inst1-tac* ***context*** *PP λvf. Normal* ((*λY s Z. vf* = *lvar* (*VName e*) (*snd s*)) ∧. *heap-free four* ∧. *Not* ∘ *initd Base*) []›)

apply *clarsimp*

apply (*rule conseq1*)
apply (*tactic ax-tac* ***context*** *1*)
apply *clarsimp*
done

lemma *Loop-Xcpt-benchmark*:
$Q = (\lambda Y\ (x,s)\ Z.\ x \neq None \longrightarrow the\text{-}Bool\ (the\ (locals\ s\ i))) \Longrightarrow$
$G,(\{\}::'a\ triple\ set)\vdash\{Normal\ (\lambda Y\ s\ Z::'a.\ True)\}$
$.lab1\cdot\ While(Lit\ (Bool\ True))\ (If(Acc\ (LVar\ i))\ (Throw\ (Acc\ (LVar\ xcpt)))\ Else$
$(Expr\ (Ass\ (LVar\ i)\ (Acc\ (LVar\ j))))).\ \{Q\}$
apply (*rule-tac* $P' = Q$ **and** $Q' = Q\leftarrow=False\downarrow=\Diamond$ **in** *conseq12*)
apply *safe*
apply (*tactic ax-tac* ***context*** *1*)
apply (*rule ax-Normal-cases*)
prefer *2*
apply (*rule ax-derivs.Abrupt* [*THEN conseq1*], *clarsimp simp add*: *Let-def*)
apply (*rule conseq1*)
apply (*tactic ax-tac* ***context*** *1*)
apply *clarsimp*
prefer *2*
apply *clarsimp*
apply (*tactic ax-tac* ***context*** *1*)
apply (*tactic*
‹*inst1-tac* ***context*** P' *Normal* ($\lambda s..\ (\lambda Y\ s\ Z.\ True)\downarrow=Val\ (the\ (locals\ s\ i))$) []›)
apply (*tactic ax-tac* ***context*** *1*)
apply (*rule conseq1*)
apply (*tactic ax-tac* ***context*** *1*)
apply *clarsimp*
apply (*rule allI*)
apply (*rule ax-escape*)
apply *auto*
apply (*rule conseq1*)
apply (*tactic ax-tac* ***context*** *1*)
apply (*tactic ax-tac* ***context*** *1*)
apply (*tactic ax-tac* ***context*** *1*)
apply *clarsimp*
apply (*rule-tac* $Q' = Normal\ (\lambda Y\ s\ Z.\ True)$ **in** *conseq2*)
prefer *2*
apply *clarsimp*
apply (*rule conseq1*)
apply (*tactic ax-tac* ***context*** *1*)
apply (*tactic ax-tac* ***context*** *1*)
prefer *2*
apply (*rule ax-subst-Var-allI*)
apply (*tactic* ‹*inst1-tac* ***context*** P' $\lambda b\ Y\ ba\ Z\ vf.\ \lambda Y\ (x,s)\ Z.\ x{=}None \wedge snd\ vf = snd\ (lvar\ i\ s)$ []›)
apply (*rule allI*)
apply (*rule-tac* $P' = Normal\ P$ **and** $P = P$ **for** P **in** *conseq1*)
prefer *2*
apply *clarsimp*
apply (*tactic ax-tac* ***context*** *1*)
apply (*rule conseq1*)
apply (*tactic ax-tac* ***context*** *1*)
apply *clarsimp*
apply (*tactic ax-tac* ***context*** *1*)
apply *clarsimp*

done

end