

Java Source and Bytecode Formalizations in Isabelle: Bali

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

January 18, 2026

Contents

1	Overview	5
2	Basis	9
1	Definitions extending HOL as logical basis of Bali	9
3	Table	15
1	Abstract tables and their implementation as lists	15
4	Name	23
1	Java names	23
5	Value	25
1	Java values	25
6	Type	27
1	Java types	27
7	Term	29
1	Java expressions and statements	29
8	Decl	39
1	Field, method, interface, and class declarations, whole Java programs	39
2	Modifier	39
3	Declaration (base "class" for member,interface and class declarations	41
4	Member (field or method)	41
5	Field	41
6	Method	41
7	Interface	43
8	Class	44
9	TypeRel	53
1	The relations between Java types	53
10	DeclConcepts	65
1	Advanced concepts on Java declarations like overriding, inheritance, dynamic method lookup	65
2	accessibility of types (cf. 6.6.1)	65
3	accessibility of members	66
4	imethds	89
5	accimethd	90
6	methd	90
7	accmethd	93
8	dynmethd	93

9	dynlookup	99
10	fields	99
11	accfield	101
12	is methd	101
11	WellType	105
1	Well-typedness of Java programs	105
12	DefiniteAssignment	117
1	Definite Assignment	117
2	Very restricted calculation fallback calculation	119
3	Analysis of constant expressions	120
4	Main analysis for boolean expressions	121
5	Lifting set operations to range of tables (map to a set)	124
13	WellForm	147
1	Well-formedness of Java programs	147
2	accessibility concerns	185
14	State	195
1	State for evaluation of Java expressions and statements	195
2	access	199
3	memory allocation	199
4	initialization	199
5	update	200
6	update	205
15	Eval	209
1	Operational evaluation (big-step) semantics of Java expressions and statements	209
16	Example	229
1	Example Bali program	229
17	Conform	253
1	Conformance notions for the type soundness proof for Java	253
18	DefiniteAssignmentCorrect	263
1	Correctness of Definite Assignment	263
19	TypeSafe	337
1	The type soundness proof for Java	337
2	accessibility	352
3	Ideas for the future	400
20	Evaln	405
1	Operational evaluation (big-step) semantics of Java expressions and statements	405
21	Trans	421
22	AxSem	427
1	Axiomatic semantics of Java expressions and statements (see also Eval.thy) .	427
2	peek-and	428
3	assn-supd	429
4	supd-assn	429

5	subst-res	429
6	subst-Bool	429
7	peek-res	430
8	ign-res	430
9	peek-st	431
10	ign-res-eq	432
11	RefVar	432
12	allocation	432

23 AxSound 447

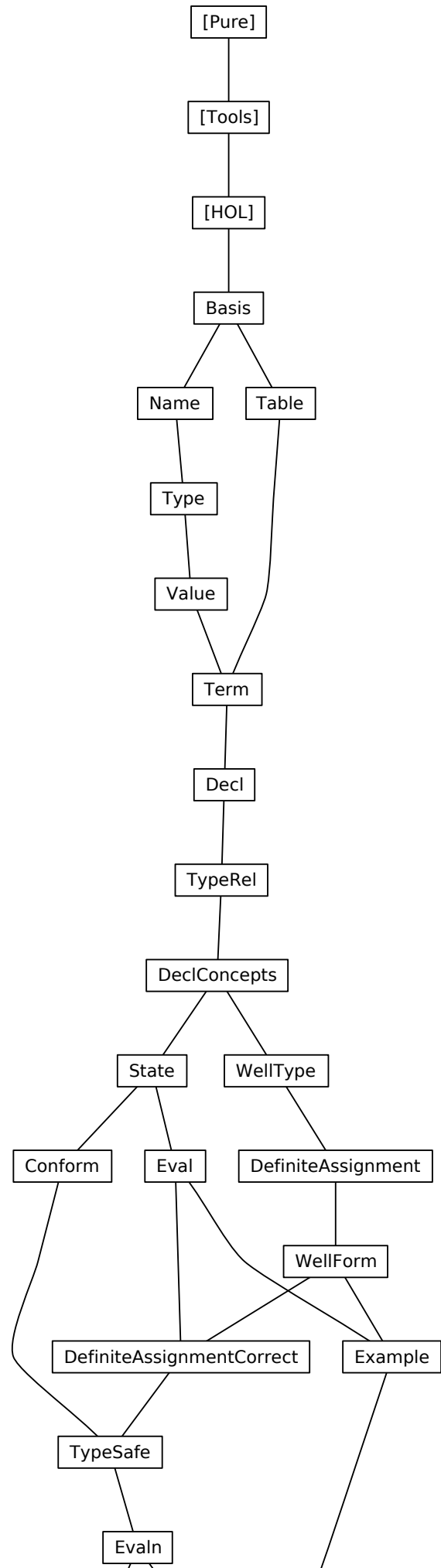
1	Soundness proof for Axiomatic semantics of Java expressions and statements	447
---	--	-----

24 AxCompl 493

1	Completeness proof for Axiomatic semantics of Java expressions and statements	493
---	---	-----

25 AxExample 519

1	Example of a proof based on the Bali axiomatic semantics	519
---	--	-----



Chapter 1

Overview

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

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

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

The following features are missing in Bali wrt. JavaCard:

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

- Inner Classes

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

Overview of the theories:

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

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

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

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

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

Term JavaCard terms. Variables, expressions and statements.

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

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

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

WellType Typesystem on the JavaCard term level.

DefiniteAssignment The definite assignment analysis on the JavaCard term level.

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

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

Eval Operational (big step) semantics for JavaCard.

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

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

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

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

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

Trans A smallstep operational semantics for JavaCard.

AxSem An axiomatic semantics (Hoare logic) for JavaCard.

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

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

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

Chapter 2

Basis

1 Definitions extending HOL as logical basis of Bali

```
theory Basis
imports Main
begin

misc

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 |> Simplifier.add-loop (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-prod f p)} = {(a,b). P (f a b)}
  by auto

lemma subset-insertD: A ⊆ insert x B ⟹ A ⊆ B ∧ x ∉ A ∨ (∃ B'. A = insert x B' ∧ B' ⊆ B)
  apply (case-tac x ∈ A)
  apply (rule disjI2)
  apply (rule-tac x = A - {x} in exI)
  apply fast+
done

abbreviation nat3 :: nat (⟨3⟩) where 3 ≡ Suc 2
abbreviation nat4 :: nat (⟨4⟩) where 4 ≡ Suc 3

lemma irrefl-tranclI': r-1 ∩ r+ = {} ⟹ ∀ x. (x, x) ∉ r+
  by (blast elim: tranclE dest: trancl-into-rtrancl)

lemma trancl-rtrancl-trancl: [(x, y) ∈ r+; (y, z) ∈ r*] ⟹ (x, z) ∈ r+
  by (auto dest: tranclD rtrancl-trans rtrancl-into-trancl2)

lemma rtrancl-into-trancl3: [(a, b) ∈ r*; a ≠ b] ⟹ (a, b) ∈ r+
  apply (drule rtranclD)
  apply auto
done

lemma rtrancl-into-rtrancl2: [(a, b) ∈ r; (b, c) ∈ r*] ⟹ (a, c) ∈ r*
  by (auto intro: rtrancl-trans)
```

lemma *triangle-lemma*:

assumes *unique*: $\bigwedge a\ b\ c. \llbracket (a,b) \in r; (a,c) \in r \rrbracket \implies b = c$
and *ax*: $(a,x) \in r^*$ **and** *ay*: $(a,y) \in r^*$
shows $(x,y) \in r^* \vee (y,x) \in r^*$
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*: $(v, y) \in r^* \implies (x, y) \in r^* \vee (y, x) \in r^*$
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*: $\llbracket \text{Ball } s\ P; \bigwedge x. P\ x \longrightarrow Q\ x \rrbracket \implies \text{Ball } s\ Q$
by *auto*

lemma *finite-SetCompr2*:

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

proof –

have $\{f\ y\ x \mid x\ y. P\ y\} = (\bigcup y \in \text{Collect } P. \text{range } (f\ y))$
by *auto*
with *that* **show** *?thesis* **by** *simp*

qed

lemma *list-all2-trans*: $\forall a\ b\ c. P1\ a\ b \longrightarrow P2\ b\ c \longrightarrow P3\ a\ c \implies$

$\forall xs2\ xs3. \text{list-all2 } P1\ xs1\ xs2 \longrightarrow \text{list-all2 } P2\ xs2\ xs3 \longrightarrow \text{list-all2 } P3\ xs1\ xs3$
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*: $(x, y) \in set\ l \implies x \in fst\ `set\ l$
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** $\langle '(+)\rangle\ 80$)

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

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

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

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

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

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

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

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

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

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

ML \langle

```
fun sum3-instantiate ctxt thm =
  map (fn s =>
    simplify (ctxt |> Simplifier.del-simps @{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

$-Oall :: [pttrn, 'a\ option, bool] \Rightarrow bool \quad (\langle (3! \text{ :-:} / -) \rangle [0,0,10] 10)$
 $-Oex :: [pttrn, 'a\ option, bool] \Rightarrow bool \quad (\langle (3? \text{ :-:} / -) \rangle [0,0,10] 10)$

syntax (symbols)

$-Oall :: [pttrn, 'a\ option, bool] \Rightarrow bool \quad (\langle (3\forall \text{ -}\in\text{:-} / -) \rangle [0,0,10] 10)$
 $-Oex :: [pttrn, 'a\ option, bool] \Rightarrow bool \quad (\langle (3\exists \text{ -}\in\text{:-} / -) \rangle [0,0,10] 10)$

syntax-consts

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

translations

$\forall x \in A: P \Leftarrow \forall x \in CONST\ set-option\ A. P$
 $\exists x \in A: P \Leftarrow \exists x \in CONST\ set-option\ A. P$

Special map update

Deemed too special for theory Map.

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

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

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

lemma $chg-map-other\ [simp]: a \neq b \Longrightarrow chg-map\ f\ a\ m\ b = m\ b$
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 ($\forall y. (x,y) \notin \text{set } l$))
by (*auto simp: unique-def dest: fst-in-set-lemma*)

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

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

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

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

list patterns

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

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

syntax

-lpttrn :: [*pttrn, pttrn*] \Rightarrow *pttrn* ($\langle \text{-\#/-} \rangle$ [901,900] 900)

syntax-consts

-lpttrn \rightleftharpoons *lsplit*

translations

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

$\lambda x \# xs. b \rightleftharpoons \text{CONST } \text{lsplit } (\lambda x\ xs. b)$

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* \rightarrow *'b*

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

map of / table of

abbreviation

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

translations

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

lemma *map-add-find-left[simp]*: $n\ k = \text{None} \Rightarrow (m\ ++\ n)\ k = m\ k$
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 =
  (λk.
    (case new k of
      None      ⇒ old k
    | Some new-val ⇒ (case old k of
        None      ⇒ Some new-val
      | Some old-val ⇒ (if cond new-val old-val
          then Some new-val
        else Some old-val))))

```

lemma *cond-override-empty1*[simp]: *cond-override c Map.empty t = t*
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*:
 $\llbracket \text{old } k = \text{Some } ov; \text{new } k = \text{Some } nv; C \text{ } nv \text{ } ov \rrbracket$
 $\implies (\text{cond-override } C \text{ old new}) k = \text{Some } nv$
by (*auto simp add: cond-override-def*)

lemma *cond-override-noOverride*:
 $\llbracket \text{old } k = \text{Some } ov; \text{new } k = \text{Some } nv; \neg (C \text{ } nv \text{ } ov) \rrbracket$
 $\implies (\text{cond-override } C \text{ old new}) k = \text{Some } ov$
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*:
 $\llbracket \text{finite } (\text{dom } s); \text{finite } (\text{dom } t) \rrbracket \implies \text{finite } (\text{dom } (\text{cond-override } C \text{ } s \text{ } t))$
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 = (λk. (case t k of
 None ⇒ None
 | Some x ⇒ if c k x then Some x else None))

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

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

lemma *filter-tab-False*[simp]: *filter-tab (λx 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) ⊆ ran t*
by (*auto simp add: filter-tab-def ran-def*)

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

lemma *finite-range-filter-tab*:
 $\text{finite } (\text{range } t) \implies \text{finite } (\text{range } (\text{filter-tab } c \ t))$
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!]*:
 $\text{filter-tab } c \ t \ k = \text{Some } x \implies (t \ k = \text{Some } x) \wedge c \ k \ x$
by (*auto simp add: filter-tab-def*)

lemma *filter-tab-SomeI*: $\llbracket t \ k = \text{Some } x; C \ k \ x \rrbracket \implies \text{filter-tab } C \ t \ k = \text{Some } x$
by (*simp add: filter-tab-def*)

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

lemma *filter-tab-all-True-Some*:
 $\llbracket \forall k \ y. t \ k = \text{Some } y \longrightarrow p \ k \ y; t \ k = \text{Some } v \rrbracket \implies \text{filter-tab } p \ t \ k = \text{Some } v$
by (*auto simp add: filter-tab-def fun-eq-iff*)

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

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

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

lemma *filter-tab-eq*: $\llbracket a=b \rrbracket \implies \text{filter-tab } C \ a = \text{filter-tab } C \ b$
by (*auto simp add: fun-eq-iff filter-tab-def*)

lemma *finite-dom-filter-tab*:
 $\text{finite } (\text{dom } t) \implies \text{finite } (\text{dom } (\text{filter-tab } C \ t))$
apply (*rule-tac B=dom t in finite-subset*)
by (*rule filter-tab-dom-subset*)

lemma *filter-tab-weaken*:
 $\llbracket \forall a \in t \ k. \exists b \in s \ k. P \ a \ b; \bigwedge k \ x \ y. \llbracket t \ k = \text{Some } x; s \ k = \text{Some } y \rrbracket \implies \text{cond } k \ x \longrightarrow \text{cond } k \ y \rrbracket \implies \forall a \in \text{filter-tab } \text{cond } t \ k. \exists b \in \text{filter-tab } \text{cond } s \ k. P \ a \ b$
by (*force simp add: filter-tab-def*)

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

$\text{cond-override overC (filter-tab filterC t) (filter-tab filterC s)}$
 $= \text{filter-tab filterC (cond-override overC t s)}$
by (auto simp add: fun-eq-iff cond-override-def filter-tab-def)

Misc

lemma *Ball-set-table*: $(\forall (x,y) \in \text{set } l. P x y) \implies \forall x. \forall y \in \text{map-of } l x. P x y$
apply (erule rev-mp)
apply (induct l)
apply simp
apply (simp (no-asm))
apply auto
done

lemma *Ball-set-tableD*:
 $\llbracket (\forall (x,y) \in \text{set } l. P x y); x \in \text{set-option (table-of } l \text{ xa)} \rrbracket \implies P xa x$
apply (frule *Ball-set-table*)
by auto

declare *map-of-SomeD* [elim]

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

lemma *set-get-eq*:
 $\text{unique } l \implies (k, \text{the (table-of } l k)) \in \text{set } l = (\text{table-of } l k \neq \text{None})$
by (auto dest!: weak-map-of-SomeI)

lemma *inj-Pair-const2*: $\text{inj } (\lambda k. (k, C))$
apply (rule inj-onI)
apply auto
done

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

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

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

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

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

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

lemma *table-of-mapf-SomeD* [dest!]:

table-of (map ($\lambda(k,x). (k, f x)$) *t*) *k* = *Some z* $\implies (\exists y \in \text{table-of } t \ k: z = f y)$
by (induct *t*) auto

lemma *table-of-mapf-NoneD* [dest!]:

table-of (map ($\lambda(k,x). (k, f x)$) *t*) *k* = *None* $\implies (\text{table-of } t \ k = \text{None})$
by (induct *t*) auto

lemma *table-of-mapkey-SomeD* [dest!]:

table-of (map ($\lambda(k,x). ((k,C),x)$) *t*) (*k,D*) = *Some x* $\implies C = D \wedge \text{table-of } t \ k = \text{Some } x$
by (induct *t*) auto

lemma *table-of-mapkey-SomeD2* [dest!]:

table-of (map ($\lambda(k,x). ((k,C),x)$) *t*) *ek* = *Some x* \implies
 $C = \text{snd } ek \wedge \text{table-of } t \ (\text{fst } ek) = \text{Some } x$
by (induct *t*) auto

lemma *table-append-Some-iff*: *table-of* (*xs@ys*) *k* = *Some z* =

(*table-of xs k* = *Some z* \vee (*table-of xs k* = *None* \wedge *table-of ys k* = *Some z*))

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* $\implies \text{unique } xs \longrightarrow \text{table-of } xs \ k = \text{Some } z$
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* = ($\lambda k. \bigcup t \in ts. t \ k$)

definition *overrides-t* :: ('a, 'b) tables \Rightarrow ('a, 'b) tables \Rightarrow ('a, 'b) tables

(infixl $\langle \oplus \oplus \rangle$ 100)

where *s* $\oplus \oplus$ *t* = ($\lambda k. \text{if } t \ k = \{\} \text{ then } s \ k \text{ else } t \ k$)

definition

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

($\langle \text{- hidings - entails -} \rangle$ 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

($\langle \text{- hiding - entails -} \rangle$ 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

($\langle \text{- hiding - under - entails -} \rangle$ 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]: $t \in ts \implies x \in t \ k \implies x \in \text{Un-tables } ts \ k$

by (auto simp add: Un-tables-def)

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

lemma *Un-tables-empty* [*simp*]: $\text{Un-tables } \{\} = (\lambda k. \{\})$
by (*simp add: Un-tables-def*)

overrides

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

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

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

hiding entails

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

lemma *empty-hiding-entails* [*simp*]: $\text{Map.empty hiding } s \text{ entails } R$
by (*simp add: hiding-entails-def*)

lemma *hiding-empty-entails* [*simp*]: $t \text{ hiding Map.empty entails } R$
by (*simp add: hiding-entails-def*)

cond hiding entails

lemma *cond-hiding-entailsD*:
 $\llbracket t \text{ hiding } s \text{ under } C \text{ entails } R; t \ k = \text{Some } x; s \ k = \text{Some } y; C \ x \ y \rrbracket \implies R \ x \ y$
by (*simp add: cond-hiding-entails-def*)

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

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

lemma *hidings-entailsD*: $\llbracket t \text{ hidings } s \text{ entails } R; x \in t \ k; y \in s \ k \rrbracket \implies R \ x \ y$
by (*simp add: hidings-entails-def*)

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

lemma *empty-hidings-entails* [*intro!*]:

$(\lambda k. \{\})$ *hidings* s entails **Rapply** (*unfold hidings-entails-def*)
by (*simp (no-asm)*)

primrec *atleast-free* :: $('a \rightarrow 'b) \Rightarrow \text{nat} \Rightarrow \text{bool}$
where
 atleast-free $m\ 0 = \text{True}$
 | *atleast-free-Suc*: *atleast-free* $m\ (\text{Suc } n) = (\exists a. m\ a = \text{None} \wedge (\forall b. \text{atleast-free } (m(a \mapsto b))\ n))$

lemma *atleast-free-weaken* [*rule-format (no-asm)*]:
 $\forall m. \text{atleast-free } m\ (\text{Suc } n) \longrightarrow \text{atleast-free } m\ n$
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*:
 $[| h\ a = \text{None}; \forall \text{obj}. \text{atleast-free } (h(a|-\>\text{obj}))\ n |] \Longrightarrow \text{atleast-free } h\ (\text{Suc } n)$
by *force*

declare *fun-upd-apply* [*simp del*]
lemma *atleast-free-SucD-lemma* [*rule-format (no-asm)*]:
 $\forall m\ a. m\ a = \text{None} \longrightarrow (\forall c. \text{atleast-free } (m(a \mapsto c))\ n) \longrightarrow$
 $(\forall b\ d. a \neq b \longrightarrow \text{atleast-free } (m(b \mapsto d))\ n)$
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 simp-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* $h\ (\text{Suc } n) \Longrightarrow \text{atleast-free } (h(a|-\>b))\ n$
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**

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

typeddecl *pname* — package name

typeddecl *mname* — method name

typeddecl *vname* — variable or field name

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

datatype *ename* — expression name

 = *VName vname*

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

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

 = *ENAME ename*

 | *This*

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

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

abbreviation *Result* :: *lname*

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

datatype *xname* — names of standard exceptions

 = *Throwable*

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

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

lemma *xn-cases*:

xn = *Throwable* \vee *xn* = *NullPointerException* \vee

xn = *OutOfMemory* \vee *xn* = *ClassCast* \vee

xn = *NegativeArraySize* \vee *xn* = *IndexOutOfBoundsException* \vee *xn* = *ArrayStore*

apply (*induct xn*)

apply *auto*

done

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

 = *Object'*

 | *SXcpt'* *xname*

 | *TName* *tname*

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

pid :: *pname*
tid :: *tname*

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

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

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

instance ..

end

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

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

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

instance ..

end

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

translations
 (*type*) *qname* <= (*type*) (\llbracket *pid*::*pname*,*tid*::*tname* \rrbracket)
 (*type*) 'a *qname-scheme* <= (*type*) (\llbracket *pid*::*pname*,*tid*::*tname*,...::'a \rrbracket)

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

definition
 Object :: *qname*
 where *Object* = (\llbracket *pid* = *java-lang*, *tid* = *Object* \rrbracket)

definition *SXcpt* :: *xname* \Rightarrow *qname*
 where *SXcpt* = ($\lambda x.$ (\llbracket *pid* = *java-lang*, *tid* = *SXcpt* ' *x* \rrbracket)

lemma *Object-neq-SXcpt* [*simp*]: *Object* \neq *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**

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

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

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

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

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

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

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

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

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

end

Chapter 6

Type

1 Java types

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

simplifications:

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

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

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

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

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

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

lemma *the-Class-eq* [simp]: *the-Class (Class C)* = *C*
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 effort

simplifications:

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

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

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

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

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

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

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

type-synonym
abopt = *abrupt option*

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

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

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

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

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

— function codes for unary operations

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

— function codes for binary operations

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

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

datatype *var*
 = *LVar lname* — local variable (incl. parameters)
 | *FVar qname qname bool expr vname* ($\langle \{-, -, -\} \dots \rangle [10, 10, 10, 85, 99] 90$)
 — 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* ($\langle \cdot, [-] \rangle [90, 10] 90$)
 — array component
 — *e1*.*[e2]*: *e1* array reference; *e2* index
 | *InsInitV stmt var*
 — insertion of initialization before evaluation
 — of var (technical term for smallstep semantics.)

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

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

	<i>Ass var expr</i>	($\leftarrow := \rightarrow$) [90,85]85)	— variable assign
	<i>Cond expr expr expr</i>	($\leftarrow ? - : \rightarrow$) [85,85,80]80)	— conditional
	<i>Call qname ref-ty inv-mode expr mname (ty list) (expr list)</i>	($\leftarrow \{-, -, \} \dots \leftarrow \{ \} \rightarrow$) [10,10,10,85,99,10,10]85)	— method call
			— $\{accC, statT, mode\}e.mn(\{pTs\}args)$ "
			— <i>accC</i> : accessing class (static class were
			— the call code is declared. Annotation only needed for
			— evaluation to check accessibility)
			— <i>statT</i> : static declaration class/interface of
			— method
			— <i>mode</i> : invocation mode
			— <i>e</i> : reference to object
			— <i>mn</i> : field name
			— <i>pTs</i> : types of parameters
			— <i>args</i> : the actual parameters/arguments
	<i>Methd qname sig</i>		— (folded) method (see below)
	<i>Body qname stmt</i>		— (unfolded) method body
	<i>InsInitE stmt expr</i>		— insertion of initialization before
			— evaluation of <i>expr</i> (technical term for smallstep sem.)
	<i>Callee locals expr</i>		— save callers locals in callee-Frame
			— (technical term for smallstep semantics)
and <i>stmt</i>			
=	<i>Skip</i>		— empty statement
	<i>Expr expr</i>		— expression statement
	<i>Lab jump stmt</i>	($\leftarrow \cdot \rightarrow$) [99,66]66)	— labeled statement; handles break
	<i>Comp stmt stmt</i>	($\leftarrow ; \rightarrow$) [66,65]65)	
	<i>If' expr stmt stmt</i>	($\leftarrow If'(-) - Else \rightarrow$) [80,79,79]70)	
	<i>Loop label expr stmt</i>	($\leftarrow \cdot While'(-) \rightarrow$) [99,80,79]70)	
	<i>Jmp jump</i>		— break, continue, return
	<i>Throw expr</i>		
	<i>TryC stmt qname vname stmt</i>	($\leftarrow Try - Catch'(-) \rightarrow$) [79,99,80,79]70)	
			— <i>Try c1 Catch(C vn) c2</i>
			— <i>c1</i> : block were exception may be thrown
			— <i>C</i> : execption class to catch
			— <i>vn</i> : local name for exception used in <i>c2</i>
			— <i>c2</i> : block to execute when exception is cateched
	<i>Fin stmt stmt</i>	($\leftarrow Finally \rightarrow$) [79,79]70)	
	<i>FinA abopt stmt</i>		— Save abruption of first statement
			— technical term for smallstep sem.)
	<i>Init qname</i>		— class initialization

```
datatype-compat var expr stmt
```

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

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

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

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

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

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

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

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

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

ML ⟨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 (⟨⟨-⟩_e⟩ 1000)
 where ⟨e⟩_e == In1l e

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

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

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

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

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

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

instantiation stmt :: inj-term

begin

definition

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

instance ..

end

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

by (*simp add: stmt-inj-term-def*)

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

by (*simp add: stmt-inj-term-simp*)

instantiation *expr :: inj-term*

begin

definition

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

instance ..

end

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

by (*simp add: expr-inj-term-def*)

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

by (*simp add: expr-inj-term-simp*)

instantiation *var :: inj-term*

begin

definition

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

instance ..

end

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

by (*simp add: var-inj-term-def*)

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

by (*simp add: var-inj-term-simp*)

class *expr-of* =

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

instantiation *expr :: expr-of*

begin

definition

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

instance ..

end

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

definition

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

instance ..

end

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

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

by (*simp add: inj-term-list-def expr-of-expr-def*)

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

by (*simp add: expr-list-inj-term-def*)

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

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

by (*simp add: inj-term-simps*)

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

by (*simp add: inj-term-simps*)

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

by (*simp add: inj-term-simps*)

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

by (*simp add: inj-term-simps*)

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

by (*simp add: inj-term-simps*)

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

by (*simp add: inj-term-simps*)

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

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*:

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

$\implies P\ t$

apply (*cases t*)

apply (*rename-tac a, case-tac a*)

apply *auto*

done

Evaluation of unary operations

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

where

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

Evaluation of binary operations

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

where

| *eval-binop* *Mul* *v1 v2* = *Intg* ((*the-Intg* *v1*) * (*the-Intg* *v2*))
 | *eval-binop* *Div* *v1 v2* = *Intg* ((*the-Intg* *v1*) div (*the-Intg* *v2*))
 | *eval-binop* *Mod* *v1 v2* = *Intg* ((*the-Intg* *v1*) mod (*the-Intg* *v2*))
 | *eval-binop* *Plus* *v1 v2* = *Intg* ((*the-Intg* *v1*) + (*the-Intg* *v2*))
 | *eval-binop* *Minus* *v1 v2* = *Intg* ((*the-Intg* *v1*) - (*the-Intg* *v2*))

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

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

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

definition

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

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

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

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

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

lemma *need-second-arg-strict*[*simp*]:
 $\llbracket \text{binop} \neq \text{CondAnd}; \text{binop} \neq \text{CondOr} \rrbracket \Longrightarrow \text{need-second-arg } \text{binop } b$
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* $\Rightarrow (b = \text{Package} \vee b = \text{Protected} \vee b = \text{Public})$
 | *Package* $\Rightarrow (b = \text{Protected} \vee b = \text{Public})$
 | *Protected* $\Rightarrow (b = \text{Public})$
 | *Public* $\Rightarrow \text{False}$)

definition

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

instance**proof**

fix $x\ y\ z :: \text{acc-modi}$
show $x < y \longleftrightarrow x \leq y \wedge \neg y \leq x$
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*)
show $x \leq y \implies y \leq z \implies x \leq z$ — transitivity
by (*auto simp add: le-acc-def less-acc-def split: acc-modi.split*)
show $x = y$ **if** $x \leq y \wedge y \leq x$ — antisymmetry
proof —
have $\forall x\ y. x < (y :: \text{acc-modi}) \wedge y < x \longrightarrow \text{False}$
by (*auto simp add: less-acc-def split: acc-modi.split*)
with that show ?thesis **by** (*unfold le-acc-def iprover*)
qed
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]*: $\text{Public} \leq a \implies a = \text{Public}$

by (*auto simp add: le-acc-def less-acc-def split: acc-modi.splits*)

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

by (*auto simp add: le-acc-def less-acc-def split: acc-modi.splits*)

lemma *acc-modi-le-Public*:

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

by (*auto simp add: le-acc-def less-acc-def split: acc-modi.splits*)

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

by (*auto simp add: le-acc-def less-acc-def split: acc-modi.splits*)

lemma *acc-modi-Private-le*:

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

by (*auto simp add: le-acc-def less-acc-def split: acc-modi.splits*)

lemma *acc-modi-Package-le*:

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

by (*auto simp add: le-acc-def less-acc-def split: acc-modi.split*)

lemma *acc-modi-le-Package*:

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

by (*auto simp add: le-acc-def less-acc-def split: acc-modi.splits*)

lemma *acc-modi-Protected-le*:

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

lemma *acc-modi-le-Protected*:

$a \leq Protected \implies a=Private \vee a = Package \vee a = Protected$
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 $Package \leq m$
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*) ($\langle access::acc-modi \rangle$)
 (*type*) *decl* <= (*type*) ($\langle access::acc-modi, \dots :: 'a \rangle$)

4 Member (field or method)

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

translations

(*type*) *member* <= (*type*) ($\langle access::acc-modi, static::bool \rangle$)
 (*type*) *member* <= (*type*) ($\langle access::acc-modi, static::bool, \dots :: 'a \rangle$)

5 Field

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

translations

(*type*) *field* <= (*type*) ($\langle access::acc-modi, static::bool, type::ty \rangle$)
 (*type*) *field* <= (*type*) ($\langle access::acc-modi, static::bool, type::ty, \dots :: 'a \rangle$)

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

translations

(*type*) *fdecl* <= (*type*) *vname* \times *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 uniformly about field and method declarations we introduce the notion of a
member declaration (e.g. useful to define accessibility )

datatype memberdecl = fdecl fdecl | mdecl mdecl

datatype memberid = fid vname | mid sig

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

instantiation memberdecl :: has-memberid
begin

definition
  memberdecl-memberid-def:
    memberid m = (case m of
      fdecl (vn,f) ⇒ fid vn

```

| $mdecl\ (sig,m) \Rightarrow mid\ sig$)

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 :: *qname* \times *memberdecl* \Rightarrow *bool*

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

lemma *is-fieldD*: $is-field\ m \Longrightarrow \exists\ declC\ f. m=(declC,fdecl\ f)$
by (*simp add: is-field-def*)

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

definition

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

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

lemma *is-methodD*: $is-method\ membr \Longrightarrow \exists\ declC\ m. membr=(declC,mdecl\ m)$
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* \times *mhead*) *list* — *method heads*

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

translations

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

definition

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

lemma *access-ibody* [*simp*]: (*access* (*ibody i*)) = *access i*
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 :: *qname* — superclass
 superIfs:: *qname list* — implemented interfaces
type-synonym
 cdecl — class declaration, cf. 8.1
 = *qname* × *class*

translations

(*type*) *cbody* ≤ (*type*) (|*access*::*acc-modi*,*cfields*::*fdecl list*,
 methods::*mdecl list*,*init*::*stmt*)
 (*type*) *cbody* ≤ (*type*) (|*access*::*acc-modi*,*cfields*::*fdecl list*,
 methods::*mdecl list*,*init*::*stmt*,...::'*a*)
 (*type*) *class* ≤ (*type*) (|*access*::*acc-modi*,*cfields*::*fdecl list*,
 methods::*mdecl list*,*init*::*stmt*,
 super::*qname*,*superIfs*::*qname list*)
 (*type*) *class* ≤ (*type*) (|*access*::*acc-modi*,*cfields*::*fdecl list*,
 methods::*mdecl list*,*init*::*stmt*,
 super::*qname*,*superIfs*::*qname list*,...::'*a*)
 (*type*) *cdecl* ≤ (*type*) *qname* × *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* \Rightarrow *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* \neq *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 *NullPointerException*, *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* \Rightarrow (*qtname*, *iface*) *table*
where *iface* *G* *I* == *table-of* (*ifaces* *G*) *I*

abbreviation

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

abbreviation

is-iface :: *prog* \Rightarrow *qtname* \Rightarrow *bool*
where *is-iface* *G I* == *iface* *G I* \neq *None*

abbreviation

is-class :: *prog* \Rightarrow *qtname* \Rightarrow *bool*
where *is-class* *G C* == *class* *G C* \neq *None*

is type

primrec *is-type* :: *prog* \Rightarrow *ty* \Rightarrow *bool*
and *isrtype* :: *prog* \Rightarrow *ref-ty* \Rightarrow *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* \Rightarrow (*qtname* \times *qtname*) *set* — direct subinterface
where *subint1* *G* = {(*I*,*J*). $\exists i \in \text{iface } G \text{ } I: J \in \text{set } (\text{isuperIfs } i)$ }

definition

subcls1 :: *prog* \Rightarrow (*qtname* \times *qtname*) *set* — direct subclass
where *subcls1* *G* = {(*C*,*D*). *C* \neq *Object* \wedge ($\exists c \in \text{class } G \text{ } C: \text{super } c = D$)}

abbreviation

subcls1-syntax :: *prog* \Rightarrow [*qtname*, *qtname*] \Rightarrow *bool* ($\text{isuperIfs } [71, 71, 71] \text{ } 70$)
where $G \vdash C \prec_C 1 D$ == (*C*,*D*) \in *subcls1* *G*

abbreviation

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

abbreviation

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

notation (*ASCII*)

subcls1-syntax ($\text{isuperIfs } [71, 71, 71] \text{ } 70$) **and**
subclsseq-syntax ($\text{isuperIfs } [71, 71, 71] \text{ } 70$) **and**
subcls-syntax ($\text{isuperIfs } [71, 71, 71] \text{ } 70$)

lemma *subint1I*: $\llbracket \text{iface } G \text{ } I = \text{Some } i; J \in \text{set } (\text{isuperIfs } i) \rrbracket$
 $\Longrightarrow (I, J) \in \text{subint1 } G$

apply (*simp add: subint1-def*)
done

lemma *subcls1I*: $\llbracket \text{class } G \text{ } C = \text{Some } c; C \neq \text{Object} \rrbracket \Longrightarrow (C, (\text{super } c)) \in \text{subcls1 } G$

apply (*simp add: subcls1-def*)
done

lemma *subint1D*: $(I, J) \in \text{subint1 } G \implies \exists i \in \text{iface } G \text{ } I: J \in \text{set } (\text{isuperIfs } i)$
by (*simp add: subint1-def*)

lemma *subcls1D*:
 $(C, D) \in \text{subcls1 } G \implies C \neq \text{Object} \wedge (\exists c. \text{class } G \text{ } C = \text{Some } c \wedge (\text{super } c = D))$
apply (*simp add: subcls1-def*)
apply *auto*
done

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

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

lemma *subcls-is-class*:
 $\llbracket G \vdash C \prec_C D \rrbracket \implies \exists c. \text{class } G \text{ } C = \text{Some } c$
by (*auto simp add: subcls1-def dest: tranclD*)

lemma *no-subcls1-Object*: $G \vdash \text{Object} \prec_C D \implies P$
by (*auto simp add: subcls1-def*)

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

well-structured programs

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

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

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

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

```

]]  $\implies$  ws-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;  $J \in \text{set } (\text{isuperIfs } i)$ ; ws-prog G ]]  $\implies$ 
  is-iface G J  $\wedge$   $(J, I) \notin (\text{subint1 } G)^+$ 
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;  $C \neq \text{Object}$ ; ws-prog G ]]  $\implies$ 
  is-class G (super c)  $\wedge$   $(\text{super } c, C) \notin (\text{subcls1 } G)^+$ 
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  $\{I. \text{is-iface } G I\}$ 
apply (fold dom-def)
apply (rule-tac finite-dom-map-of)
done

```

```

lemma finite-is-class: finite  $\{C. \text{is-class } G C\}$ 
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  $B = \{\text{super } (\text{the } (\text{class } G C))\}$  in finite-subset)
apply auto
done

```

```

lemma subint1-irrefl-lemma1:
  ws-prog G  $\implies$   $(\text{subint1 } G)^{-1} \cap (\text{subint1 } G)^+ = \{\}$ 
apply (force dest: subint1D ws-prog-ideclD conjunct2)
done

```

```

lemma subcls1-irrefl-lemma1:
  ws-prog G  $\implies$   $(\text{subcls1 } G)^{-1} \cap (\text{subcls1 } G)^+ = \{\}$ 
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*: $\llbracket (x, y) \in \text{subint1 } G; \text{ws-prog } G \rrbracket \implies x \neq y$
 apply (rule *irrefl-trancl-rD*)
 apply (rule *subint1-irrefl-lemma2*)
 apply auto
 done

lemma *subcls1-irrefl*: $\llbracket (x, y) \in \text{subcls1 } G; \text{ws-prog } G \rrbracket \implies x \neq y$
 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*: $\text{ws-prog } G \implies \text{wf } ((\text{subint1 } G)^{-1})$
 by (auto intro: *finite-acyclic-wf-converse finite-subint1 subint1-acyclic*)

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

lemma *subint1-induct*:
 $\llbracket \text{ws-prog } G; \bigwedge x. \forall y. (x, y) \in \text{subint1 } G \longrightarrow P y \implies P x \rrbracket \implies P a$
 apply (frule *wf-subint1*)
 apply (erule *wf-induct*)
 apply (simp (no-asm-use) only: *converse-iff*)
 apply blast
 done

lemma *subcls1-induct* [consumes 1]:
 $\llbracket \text{ws-prog } G; \bigwedge x. \forall y. (x, y) \in \text{subcls1 } G \longrightarrow P y \implies P x \rrbracket \implies P a$
 apply (frule *wf-subcls1*)
 apply (erule *wf-induct*)
 apply (simp (no-asm-use) only: *converse-iff*)
 apply blast
 done

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

$(C \neq \text{Object} \longrightarrow (C, (\text{super } c)) \in \text{subcls1 } G \wedge$
 $P (\text{super } c) \wedge \text{is-class } G (\text{super } c)) \Longrightarrow P C$
 $\Longrightarrow P C$
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*]:
 $\llbracket \text{class } G C = \text{Some } c; \text{ws-prog } G;$
 $\wedge \text{co. class } G \text{Object} = \text{Some } co \Longrightarrow P \text{Object};$
 $\wedge C c. \llbracket \text{class } G C = \text{Some } c; C \neq \text{Object}; P (\text{super } c) \rrbracket \Longrightarrow P C$
 $\rrbracket \Longrightarrow P C$

proof –
assume *clsC*: $\text{class } G C = \text{Some } c$
and *init*: $\wedge \text{co. class } G \text{Object} = \text{Some } co \Longrightarrow P \text{Object}$
and *step*: $\wedge C c. \llbracket \text{class } G C = \text{Some } c; C \neq \text{Object}; P (\text{super } c) \rrbracket \Longrightarrow P C$
assume *ws*: *ws-prog* *G*
then have *is-class* *G C* $\Longrightarrow P C$
proof (*induct rule: subcls1-induct*)
fix *C*
assume *hyp*: $\forall S. G \vdash C \prec_C 1 S \longrightarrow \text{is-class } G S \longrightarrow P S$
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*]:
 $\llbracket \text{is-class } G C; \text{ws-prog } G;$
 $\wedge \text{co. class } G \text{Object} = \text{Some } co \Longrightarrow P \text{Object};$
 $\wedge C c. \llbracket \text{class } G C = \text{Some } c; C \neq \text{Object}; P (\text{super } c) \rrbracket \Longrightarrow P C$
 $\rrbracket \Longrightarrow P C$
by (*auto intro: ws-class-induct*)

lemma *ws-class-induct''* [*consumes 2, case-names Object Subcls*]:
 $\llbracket \text{class } G C = \text{Some } c; \text{ws-prog } G;$
 $\wedge \text{co. class } G \text{Object} = \text{Some } co \Longrightarrow P \text{Object } co;$
 $\wedge C c \text{ sc. } \llbracket \text{class } G C = \text{Some } c; \text{class } G (\text{super } c) = \text{Some } sc;$
 $C \neq \text{Object}; P (\text{super } c) \text{ sc} \rrbracket \Longrightarrow P C c$
 $\rrbracket \Longrightarrow P C c$
proof –
assume *clsC*: $\text{class } G C = \text{Some } c$
and *init*: $\wedge \text{co. class } G \text{Object} = \text{Some } co \Longrightarrow P \text{Object } co$
and *step*: $\wedge C c \text{ sc. } \llbracket \text{class } G C = \text{Some } c; \text{class } G (\text{super } c) = \text{Some } sc;$
 $C \neq \text{Object}; P (\text{super } c) \text{ sc} \rrbracket \Longrightarrow P C c$
assume *ws*: *ws-prog* *G*
then have $\wedge c. \text{class } G C = \text{Some } c \Longrightarrow P C c$
proof (*induct rule: subcls1-induct*)
fix *C c*

```

assume  hyp:  $\forall S. G \vdash C \prec_C 1 S \longrightarrow (\forall s. \text{class } G S = \text{Some } s \longrightarrow P S s)$ 
and iscls:  $\text{class } G C = \text{Some } c$ 
show  $P C c$ 
proof (cases  $C = \text{Object}$ )
  case True with iscls init show  $P C c$  by auto
next
  case False
  with ws iscls obtain sc where
    sc:  $\text{class } G (\text{super } c) = \text{Some } sc$ 
    by (auto dest: ws-prog-cdeclD)
  from iscls False have  $G \vdash C \prec_C 1 (\text{super } c)$  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:  $\text{is-iface } G I$  and
    ws:  $\text{ws-prog } G$  and
    hyp-sub:  $\bigwedge I i. \llbracket \text{iface } G I = \text{Some } i; \forall J \in \text{set } (\text{isuperIfs } i). (I, J) \in \text{subint1 } G \wedge P J \wedge \text{is-iface } G J \rrbracket \Longrightarrow P I$ 
  shows  $P I$ 
proof –
  from is-if-I ws
  show  $P I$ 
proof (rule ws-subint1-induct)
  fix  $I i$ 
  assume hyp:  $\text{iface } G I = \text{Some } i \wedge (\forall J \in \text{set } (\text{isuperIfs } i). (I, J) \in \text{subint1 } G \wedge P J \wedge \text{is-iface } G J)$ 
  then have if-I:  $\text{iface } G I = \text{Some } i$ 
    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 hierarchies

```

function iface-rec ::  $\text{prog} \Rightarrow \text{qname} \Rightarrow (\text{qname} \Rightarrow \text{iface} \Rightarrow 'a \text{ set} \Rightarrow 'a) \Rightarrow 'a$ 
where
  [simp del]: iface-rec  $G I f =$ 
    (case iface  $G I$  of
      None  $\Rightarrow \text{undefined}$ 
    | Some i  $\Rightarrow \text{if } \text{ws-prog } G \text{ then } f I i$ 

```

```

      ((λJ. iface-rec G J f) 'set (isuperIfs i))
    else undefined)

by auto
termination
by (relation inv-image (same-fst ws-prog (λG. (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 ((λJ. 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 (λG. (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
    (λI 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* \Rightarrow (*qname* \times *qname*) *set* — direct implementation
— direct implementation, cf. 8.1.3
where *implmt1* *G* = {(*C*,*I*). *C* \neq *Object* \wedge ($\exists c \in \text{class } G \ C: I \in \text{set } (\text{superIfs } c)$)}

abbreviation

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

abbreviation

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

abbreviation

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

notation (*ASCII*)

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

subclass and subinterface relations

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

lemma *subcls-direct1*:

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

apply (*auto dest: subcls-direct*)

done

lemma *subcls1I1*:

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

apply (*auto dest: subcls1I*)

done

lemma *subcls-direct2*:

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

apply (*auto dest: subcls1I1*)

done

lemma *subclseq-trans*: $\llbracket G \vdash A \preceq_C B; G \vdash B \preceq_C C \rrbracket \implies G \vdash A \preceq_C C$

by (*blast intro: rtrancl-trans*)

lemma *subcls-trans*: $\llbracket G \vdash A \prec_C B; G \vdash B \prec_C C \rrbracket \implies G \vdash A \prec_C C$

by (*blast intro: trancl-trans*)

lemma *SXcpt-subcls-Throwable-lemma*:

$\llbracket \text{class } G \ (\text{SXcpt } xn) = \text{Some } xc;$

$\text{super } xc = (\text{if } xn = \text{Throwable} \text{ then } \text{Object} \text{ else } \text{SXcpt } \text{Throwable}) \rrbracket$

$\implies G \vdash \text{SXcpt } xn \preceq_C \text{SXcpt } \text{Throwable}$

apply (*case-tac xn = Throwable*)

apply *simp-all*

apply (*drule subcls-direct*)

apply (*auto dest: sym*)

done

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

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!*]: $G \vdash \text{Object} \preceq_C C \implies C = \text{Object}$

apply (*erule rtrancl-induct*)

apply (*auto dest: subcls1D*)

done

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

apply (*erule trancl-induct*)

apply (*auto dest: subcls1D*)

done

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

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

apply (*drule (1) subcls-ObjectI*)

apply (*auto intro: rtrancl-into-trancl3*)

done

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

apply (*erule trancl-trans-induct*)
apply (*auto dest!: subcls1D*)
done

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

lemma *single-inheritance*:
 $\llbracket G \vdash A \prec_C 1 \ B; G \vdash A \prec_C 1 \ C \rrbracket \implies B = C$
by (*auto simp add: subcls1-def*)

lemma *subcls-compareable*:
 $\llbracket G \vdash A \preceq_C X; G \vdash A \preceq_C Y \rrbracket \implies G \vdash X \preceq_C Y \vee G \vdash Y \preceq_C X$
by (*rule triangle-lemma*) (*auto intro: single-inheritance*)

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

proof

assume *ws: ws-prog G* **and**
subcls1: G ⊢ C ≺_C 1 D **and**
eq-C-D: C=D
from *subcls1* **obtain** *c*
where
neq-C-Object: C ≠ 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 ⊢ super c ≺_C C*
by *auto*
from *ws clsC neq-C-Object*
have $\neg G \vdash \text{super } c \prec_C C$
by (*auto dest: ws-prog-cdeclD*)
from *this subcls-super-c-C*
show *False*
by (*rule notE*)

qed

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

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

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 ≺_C 1 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  $\neg G \vdash D \prec_C C$ 
  by (auto dest: ws-prog-cdeclD)
next
  fix C Z
  assume subcls1-C-Z:  $G \vdash C \prec_C 1 Z$  and
    subcls-Z-D:  $G \vdash Z \prec_C D$  and
    nsubcls-D-Z:  $\neg G \vdash D \prec_C Z$ 
  show  $\neg G \vdash D \prec_C C$ 
  proof
    assume subcls-D-C:  $G \vdash D \prec_C C$ 
    show False
    proof –
      from subcls-D-C subcls1-C-Z
      have  $G \vdash D \prec_C Z$ 
      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  $G \vdash C \preceq_C D$ 
obtains  $(Eq) C = D \mid (Subcls) G \vdash C \prec_C D$ 
using assms by (blast intro: rtrancl-cases)

```

lemma *subclseq-acyclic*:

```

 $\llbracket G \vdash C \preceq_C D; G \vdash D \preceq_C C; ws\text{-prog } G \rrbracket \implies C = D$ 
by (auto elim: subclseq-cases dest: subcls-acyclic)

```

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

```

proof –
  assume ws: ws-prog G
  assume subcls:  $G \vdash C \prec_C D$ 
  then show ?thesis
  proof (induct rule: converse-trancl-induct)
    fix C
    assume  $G \vdash C \prec_C 1 D$ 
    with ws
    show  $C \neq D$ 
    by (blast dest: subcls1-irrefl)
  next
    fix C Z
    assume subcls1-C-Z:  $G \vdash C \prec_C 1 Z$  and
      subcls-Z-D:  $G \vdash Z \prec_C D$  and
      neq-Z-D:  $Z \neq D$ 
    show  $C \neq D$ 
    proof
      assume eq-C-D:  $C = D$ 
      show False
      proof (rule notE)
        from subcls1-C-Z eq-C-D
        show  $G \vdash D \prec_C Z$ 
        by (auto)
      also

```

```

    from subcls-Z-D ws
    show  $\neg G \vdash D \prec_C Z$ 
    by (rule subcls-acyclic)
  qed
qed
qed
qed

```

lemma *invert-subclseq*:

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

proof –

```

  assume ws: ws-prog G and
    subclseq-C-D:  $G \vdash C \preceq_C D$ 
  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  $G \vdash C \prec_C D$ 
    by (blast intro: rtranc1-into-tranc13)
    with ws
    show ?thesis
    by (blast dest: subcls-acyclic)
  qed
qed

```

lemma *invert-subcls*:

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

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*:

$\llbracket G \vdash C \prec_C D; \text{class } G \ C = \text{Some } c \rrbracket \implies G \vdash (\text{super } c) \preceq_C D$
proof –
assume $\text{cls}C: \text{class } G \ C = \text{Some } c$
assume $\text{subcls-}C\text{-}C: G \vdash C \prec_C D$
then obtain S **where**
 $G \vdash C \prec_C 1 \ S$ **and**
 $\text{subclseq-}S\text{-}D: G \vdash S \preceq_C D$
by (*blast dest: tranclD*)
with $\text{cls}C$
have $S = \text{super } c$
by (*auto dest: subcls1D*)
with $\text{subclseq-}S\text{-}D$ **show** *?thesis* **by** *simp*
qed

lemma *subclseq-superD*:
 $\llbracket G \vdash C \preceq_C D; C \neq D; \text{class } G \ C = \text{Some } c \rrbracket \implies G \vdash (\text{super } c) \preceq_C D$
proof –
assume $\text{neg-}C\text{-}D: C \neq D$
assume $\text{cls}C: \text{class } G \ C = \text{Some } c$
assume $\text{subclseq-}C\text{-}D: G \vdash C \preceq_C D$
then show *?thesis*
proof (*cases rule: subclseq-cases*)
case *Eq* **with** $\text{neg-}C\text{-}D$ **show** *?thesis* **by** *contradiction*
next
case *Subcls*
with $\text{cls}C$ **show** *?thesis* **by** (*blast dest: subcls-superD*)
qed
qed

implementation relation

lemma *implmt1D*: $G \vdash C \rightsquigarrow 1I \implies C \neq \text{Object} \wedge (\exists c \in \text{class } G \ C: I \in \text{set } (\text{superIfs } c))$
apply (*unfold implmt1-def*)
apply *auto*
done

inductive — implementation, cf. 8.1.4

$\text{implmt} :: \text{prog} \Rightarrow \text{qname} \Rightarrow \text{qname} \Rightarrow \text{bool}$ ($\langle \vdash \rightsquigarrow \rightarrow \rangle$ [71,71,71] 70)
for $G :: \text{prog}$

where

$\text{direct}: G \vdash C \rightsquigarrow 1J \implies G \vdash C \rightsquigarrow J$
 $\mid \text{subint}: G \vdash C \rightsquigarrow 1I \implies G \vdash I \preceq_I J \implies G \vdash C \rightsquigarrow J$
 $\mid \text{subcls1}: G \vdash C \prec_C 1D \implies G \vdash D \rightsquigarrow J \implies G \vdash C \rightsquigarrow J$

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

lemma *implmt-ObjectE* [*elim!*]: $G \vdash \text{Object} \rightsquigarrow I \implies R$
by (*auto dest!: implmtD implmt1D subcls1D*)

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

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

```

apply (erule rev-mp, erule implmt.induct)
apply (auto dest: implmt.subint rtranc1-trans implmt.subcls1)
done

```

```

lemma implmt-is-class:  $G \vdash C \rightsquigarrow I \implies \text{is-class } G \ C$ 
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
 $\text{widen} :: \text{prog} \Rightarrow \text{ty} \Rightarrow \text{ty} \Rightarrow \text{bool} \ (\hookrightarrow \vdash \text{--} \preceq \rightarrow) \ [71, 71, 71] \ 70)$

for $G :: \text{prog}$

where

```

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

```

declare widen.refl [intro!]

declare widen.intros [simp]

```

lemma widen-PrimT:  $G \vdash \text{PrimT } x \preceq T \implies (\exists y. T = \text{PrimT } y)$ 
apply (ind-cases  $G \vdash \text{PrimT } x \preceq T$ )
by auto

```

```

lemma widen-PrimT2:  $G \vdash S \preceq \text{PrimT } x \implies \exists y. S = \text{PrimT } y$ 
apply (ind-cases  $G \vdash S \preceq \text{PrimT } x$ )
by auto

```

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

```

lemma widen-PrimT-strong:  $G \vdash \text{PrimT } x \preceq T \implies T = \text{PrimT } x$ 
by (ind-cases  $G \vdash \text{PrimT } x \preceq T$ ) simp-all

```

```

lemma widen-PrimT2-strong:  $G \vdash S \preceq \text{PrimT } x \implies S = \text{PrimT } x$ 
by (ind-cases  $G \vdash S \preceq \text{PrimT } x$ ) simp-all

```

Specialized versions for booleans also would work for real Java

```

lemma widen-Boolean:  $G \vdash \text{PrimT } \text{Boolean} \preceq T \implies T = \text{PrimT } \text{Boolean}$ 
by (ind-cases  $G \vdash \text{PrimT } \text{Boolean} \preceq T$ ) simp-all

```

```

lemma widen-Boolean2:  $G \vdash S \preceq \text{PrimT } \text{Boolean} \implies S = \text{PrimT } \text{Boolean}$ 
by (ind-cases  $G \vdash S \preceq \text{PrimT } \text{Boolean}$ ) simp-all

```

```

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

```

lemma *widen-RefT2*: $G \vdash S \preceq_{\text{RefT}} R \implies \exists t. S =_{\text{RefT}} t$
apply (*ind-cases* $G \vdash S \preceq_{\text{RefT}} R$)
by *auto*

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

lemma *widen-Iface2*: $G \vdash S \preceq \text{Iface } J \implies S = NT \vee (\exists I. S =_{\text{Iface}} I) \vee (\exists D. S =_{\text{Class}} D)$
apply (*ind-cases* $G \vdash S \preceq \text{Iface } J$)
by *auto*

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

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

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

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

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

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

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

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

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

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

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

lemma *widen-ArrayRefT*:
 $G \vdash \text{RefT } t.[] \preceq T \implies T = \text{Class Object} \vee (\exists s. T = \text{RefT } s.[] \wedge G \vdash \text{RefT } t \preceq \text{RefT } s)$
apply (*ind-cases* $G \vdash \text{RefT } t.[] \preceq T$)
by *auto*

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

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

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

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

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

lemma *widen-trans-lemma* [*rule-format* (*no-asm*)]:
 $\llbracket G \vdash S \preceq U; \forall C. \text{is-class } G \ C \longrightarrow G \vdash C \preceq_C \text{Object} \rrbracket \implies \forall T. G \vdash U \preceq T \longrightarrow G \vdash S \preceq T$
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*: $\llbracket G \vdash S \preceq U; G \vdash U \preceq T; \text{ws-prog } G \rrbracket \implies G \vdash S \preceq T$
by (*auto intro: widen-trans-lemma subcls-ObjectI*)

lemma *widen-antisym-lemma* [*rule-format (no-asm)*]: $\llbracket G \vdash S \preceq T;$
 $\forall I J. G \vdash I \preceq I J \wedge G \vdash J \preceq I I \longrightarrow I = J;$
 $\forall C D. G \vdash C \preceq_C D \wedge G \vdash D \preceq_C C \longrightarrow C = D;$
 $\forall I. G \vdash \text{Object} \rightsquigarrow I \longrightarrow \text{False} \rrbracket \implies G \vdash T \preceq S \longrightarrow S = T$
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*: $\llbracket G \vdash S \preceq T; G \vdash T \preceq S; \text{ws-prog } G \rrbracket \implies S = T$
by (*fast elim: widen-antisym-lemma subint-antisym* [*THEN antisymD*]
subcls-antisym [*THEN antisymD*])

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

definition

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

lemma *widens-Nil* [*simp*]: $G \vdash [][\preceq] []$
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* \Rightarrow *ty* \Rightarrow *ty* \Rightarrow *bool* ($\hookrightarrow \vdash \succ \hookrightarrow$) [*71, 71, 71*] *70*)
for *G* :: *prog*

where

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

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

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

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

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

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

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

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

Specialized versions for booleans also would work for real Java

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

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

casting relation

inductive — casting conversion, cf. 5.5
 $\text{cast} :: \text{prog} \Rightarrow \text{ty} \Rightarrow \text{ty} \Rightarrow \text{bool} \ (\hookrightarrow - \preceq ? \rightarrow [71, 71, 71] \ 70)$
for $G :: \text{prog}$
where
 $\text{widen}: G \vdash S \preceq T \implies G \vdash S \preceq ? T$
 $\mid \text{narrow}: G \vdash S \succ T \implies G \vdash S \preceq ? T$

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

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

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

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

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

```

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

lemma cast-Boolean2:
  assumes bool-cast:  $G \vdash S \preceq ? \text{Prim } T \text{ Boolean}$ 
  shows  $S = \text{Prim } T \text{ Boolean}$ 
using bool-cast
proof (cases)
  case widen
  hence  $G \vdash S \preceq \text{Prim } T \text{ Boolean}$ 
    by simp
  thus ?thesis by (rule widen-Boolean2)
next
  case narrow
  hence  $G \vdash S \succ \text{Prim } T \text{ Boolean}$ 
    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* \Rightarrow *qname* \Rightarrow *bool* **where**
is-public *G* *qn* = (case class *G* *qn* of
 None \Rightarrow (case iface *G* *qn* of
 None \Rightarrow *False*
 | Some *i* \Rightarrow *access i = Public*)
 | Some *c* \Rightarrow *access c = Public*)

2 accessibility of types (cf. 6.6.1)

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

primrec
accessible-in :: *prog* \Rightarrow *ty* \Rightarrow *pname* \Rightarrow *bool* ($\lambda \cdot \vdash \cdot$ - *accessible'-in* \rightarrow [61,61,61] 60) **and**
rt-accessible-in :: *prog* \Rightarrow *ref-ty* \Rightarrow *pname* \Rightarrow *bool* ($\lambda \cdot \vdash \cdot$ - *accessible'-in''* \rightarrow [61,61,61] 60)
where
 $G \vdash (PrimT\ p)\ accessible-in\ pack = True$
| *accessible-in-RefT-simp*:
 $G \vdash (RefT\ r)\ accessible-in\ pack = G \vdash r\ accessible-in'\ pack$
| $G \vdash (NullT)\ accessible-in'\ pack = True$
| $G \vdash (IfaceT\ I)\ accessible-in'\ pack = ((pid\ I = pack) \vee is-public\ G\ I)$
| $G \vdash (ClassT\ C)\ accessible-in'\ pack = ((pid\ C = pack) \vee is-public\ G\ C)$
| $G \vdash (ArrayT\ ty)\ accessible-in'\ pack = G \vdash ty\ accessible-in\ pack$

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

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

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

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

definition

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

lemma *is-acc-classD*:

is-acc-class *G P C* \Longrightarrow *is-class* *G C* \wedge *G* \vdash (*Class* *C*) *accessible-in* *P*
by (*simp* *add*: *is-acc-class-def*)

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

by (*auto simp* *add*: *is-acc-class-def*)

lemma *is-acc-ifaceD*:

is-acc-iface *G P I* \Longrightarrow *is-iface* *G I* \wedge *G* \vdash (*Iface* *I*) *accessible-in* *P*
by (*simp* *add*: *is-acc-iface-def*)

lemma *is-acc-typeD*:

is-acc-type *G P T* \Longrightarrow *is-type* *G T* \wedge *G* \vdash *T* *accessible-in* *P*
by (*simp* *add*: *is-acc-type-def*)

lemma *is-acc-reftypeD*:

is-acc-reftype *G P T* \Longrightarrow *isrtype* *G T* \wedge *G* \vdash *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 \Rightarrow *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 \Rightarrow *accmodi f*
 | *mdecl m* \Rightarrow *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 \Rightarrow *qname*

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

definition

declclass q = (\lfloor *pid* = *pid q*, *tid* = *tid q* \rfloor)

instance ..

end

lemma *qname-declclass-def*:
declclass q \equiv (*q*::*qname*)
by (*induct q*) (*simp add: declclass-qname-ext-def*)

lemma *qname-declclass-simp*[simp]: *declclass* (*q*::*qname*) = *q*
by (*simp add: qname-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 \Rightarrow 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*)) \equiv *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

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

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 mnemonic selectors for various pairs

definition

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

definition

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

definition

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

definition

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

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

definition

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

definition

declclassf:: (*vname* × *qname*) ⇒ *qname*
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 mnemonic 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* \times *mdecl* \Rightarrow *qtname* \times *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* \Rightarrow (*qtname* \times *methd*) \Rightarrow (*qtname* \times *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* \Rightarrow (*qtname* \times *field*) \Rightarrow (*qtname* \times *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*,*methd 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 (methd (qmdecl sig new)) = mdecl (sig, methd 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: $\text{resTy } (m::('b \text{ mhead-scheme})) \equiv \text{resT } m$

lemma *mhead-resTy-simp*: $\text{resTy } (m::'a \text{ mhead-scheme}) = \text{resT } m$
by (*simp add: mhead-ext-type-resTy-def*)

lemma *resTy-mhead* [*simp*]: $\text{resTy } (\text{mhead } m) = \text{resTy } m$
by (*simp add: mhead-def mhead-resTy-simp*)

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

definition

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

instance ..

end

lemma *pair-resTy-simp* [*simp*]: $\text{resTy } (x, m) = \text{resTy } m$
by (*simp add: pair-resTy-def*)

lemma *qmdecl-resTy-simp* [*simp*]: $\text{resTy } (\text{qmdecl sig } m) = \text{resTy } m$
by (*cases m*) (*simp*)

lemma *resTy-mthd* [*simp*]: $\text{resTy } (\text{mthd } m) = \text{resTy } m$
by (*cases m*) (*simp add: mthd-def*)

inheritable-in

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

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

definition

inheritable-in :: prog \Rightarrow (qtname \times memberdecl) \Rightarrow pname \Rightarrow bool ($\langle \cdot \vdash \cdot \text{ inheritable'-in } \cdot \rangle$ [61,61,61] 60)

where

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

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

abbreviation

Method-inheritable-in-syntax::

$prog \Rightarrow (qname \times mdecl) \Rightarrow pname \Rightarrow bool$
 $(\vdash \vdash Method - inheritable-in \rightarrow [61,61,61] 60)$
where $G \vdash Method m inheritable-in p == G \vdash methdMembr m inheritable-in p$

abbreviation

Methd-inheritable-in::

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

declared-in/undeclared-in

definition

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

definition

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

definition

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

where

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

abbreviation

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

abbreviation

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

lemma declared-in-classD:

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

by (cases m)

(auto simp add: declared-in-def cdeclaredmethd-def cdeclaredfield-def)

definition

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

where

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

members

inductive

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

$(\hookleftarrow \vdash - \text{member}'\text{-of} \rightarrow [61,61,61] \ 60)$
for $G :: \text{prog}$
where

Immediate: $\llbracket G \vdash \text{mbr } m \text{ declared-in } C; \text{declclass } m = C \rrbracket \Rightarrow G \vdash m \text{ member-of } C$
Inherited: $\llbracket G \vdash m \text{ inheritable-in } (\text{pid } C); G \vdash \text{memberid } m \text{ undeclared-in } C;$
 $G \vdash C \prec_C 1 \ S; G \vdash (\text{Class } S) \text{ accessible-in } (\text{pid } C); G \vdash m \text{ member-of } S$
 $\rrbracket \Rightarrow G \vdash m \text{ member-of } C$

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

abbreviation

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

abbreviation

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

abbreviation

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

definition

$\text{inherits} :: \text{prog} \Rightarrow \text{qname} \Rightarrow (\text{qname} \times \text{memberdecl}) \Rightarrow \text{bool} \ (\hookleftarrow \vdash - \text{inherits} \rightarrow [61,61,61] \ 60)$
where

$G \vdash C \text{ inherits } m =$
 $(G \vdash m \text{ inheritable-in } (\text{pid } C) \wedge G \vdash \text{memberid } m \text{ undeclared-in } C \wedge$
 $(\exists S. G \vdash C \prec_C 1 \ S \wedge G \vdash (\text{Class } S) \text{ accessible-in } (\text{pid } C) \wedge G \vdash m \text{ member-of } S))$

lemma *inherits-member*: $G \vdash C \text{ inherits } m \Rightarrow G \vdash m \text{ member-of } C$

by (*auto simp add: inherits-def intro: members.Inherited*)

definition

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

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

abbreviation

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

abbreviation

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

lemma *member-inD*: $G \vdash m \text{ member-in } C$
 $\implies \exists \text{ provC}. G \vdash C \preceq_C \text{ provC} \wedge G \vdash m \text{ member-of } \text{provC}$
by (*auto simp add: member-in-def*)

lemma *member-inI*: $\llbracket G \vdash m \text{ member-of } \text{provC}; G \vdash C \preceq_C \text{ provC} \rrbracket \implies G \vdash m \text{ member-in } C$
by (*auto simp add: member-in-def*)

lemma *member-of-to-member-in*: $G \vdash m \text{ member-of } C \implies G \vdash m \text{ member-in } C$
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 :: $\text{prog} \Rightarrow (\text{qname} \times \text{mdecl}) \Rightarrow (\text{qname} \times \text{mdecl}) \Rightarrow \text{bool}$
 $(\langle \cdot \vdash - \text{overrides}_S \rightarrow [61, 61, 61] \ 60)$

for $G :: \text{prog}$

where

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

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

Dynamic overriding (used during the typecheck of the compiler)

inductive

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

for $G :: \text{prog}$

where

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

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

abbreviation (input)

sig-stat-overrides ::

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

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

abbreviation (*input*)

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

Hiding**definition**

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

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

abbreviation

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

lemma *hidesI*:

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

lemma *hidesD*:

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

lemma *overrides-commonD*:

$\llbracket G \vdash \text{new overrides old} \rrbracket \implies$
 $\text{declclass new} \neq \text{Object} \wedge \neg \text{is-static new} \wedge \neg \text{is-static old} \wedge$
 $\text{accmodi new} \neq \text{Private} \wedge$
 $\text{msig new} = \text{msig old} \wedge$
 $G \vdash (\text{declclass new}) \prec_C (\text{declclass old}) \wedge$
 $G \vdash \text{Method new declared-in } (\text{declclass new}) \wedge$
 $G \vdash \text{Method old declared-in } (\text{declclass old})$
by (*induct set: overridesR*) (*auto intro: trancl-trans*)

lemma *ws-overrides-commonD*:

$\llbracket G \vdash \text{new overrides old}; \text{ws-prog } G \rrbracket \implies$
 $\text{declclass new} \neq \text{Object} \wedge \neg \text{is-static new} \wedge \neg \text{is-static old} \wedge$
 $\text{accmodi new} \neq \text{Private} \wedge G \vdash \text{resTy new} \preceq \text{resTy old} \wedge$
 $\text{msig new} = \text{msig old} \wedge$
 $G \vdash (\text{declclass new}) \prec_C (\text{declclass old}) \wedge$
 $G \vdash \text{Method new declared-in } (\text{declclass new}) \wedge$
 $G \vdash \text{Method old declared-in } (\text{declclass old})$
by (*induct set: overridesR*) (*auto intro: trancl-trans ws-widen-trans*)

lemma *overrides-eq-sigD*:

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

by (*auto dest: overrides-commonD*)

lemma *hides-eq-sigD*:

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

by (*auto simp add: hides-def*)

permits access

definition

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

where

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

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

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

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

inductive

$\text{accessible-fromR} :: \text{prog} \Rightarrow \text{qname} \Rightarrow (\text{qname} \times \text{memberdecl}) \Rightarrow \text{qname} \Rightarrow \text{bool}$

and $\text{accessible-from} :: \text{prog} \Rightarrow (\text{qname} \times \text{memberdecl}) \Rightarrow \text{qname} \Rightarrow \text{qname} \Rightarrow \text{bool}$

($\langle - \vdash - \text{ of } - \text{ accessible'-from} \rightarrow [61,61,61,61] \text{ 60}$)

and $\text{method-accessible-from} :: \text{prog} \Rightarrow (\text{qname} \times \text{mdecl}) \Rightarrow \text{qname} \Rightarrow \text{qname} \Rightarrow \text{bool}$

($\langle - \vdash \text{Method } - \text{ of } - \text{ accessible'-from} \rightarrow [61,61,61,61] \text{ 60}$)

for $G :: \text{prog}$ **and** $\text{accclass} :: \text{qname}$

where

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

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

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

$\llbracket G \vdash \text{membr member-of class;}$

$G \vdash (\text{Class class}) \text{ accessible-in } (\text{pid accclass});$

$G \vdash \text{membr in class permits-acc-from accclass}$

$\rrbracket \implies G \vdash \text{membr of class accessible-from accclass}$

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

$\llbracket G \vdash \text{membr member-of class;}$

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

abbreviation

methd-accessible-from::

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

where

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

abbreviation

field-accessible-from::

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

where

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

inductive

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

where

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

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

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

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

abbreviation

methd-dyn-accessible-from::

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

where

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

abbreviation

field-dyn-accessible-from::

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

where

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

lemma *accessible-from-commonD*: $G \vdash m \text{ of } C \text{ accessible-from } S$
 $\implies G \vdash m \text{ member-of } C \wedge G \vdash (\text{Class } C) \text{ accessible-in } (pid \ S)$
by (*auto elim: accessible-fromR.induct*)

lemma *unique-declaration*:
 $\llbracket G \vdash m \text{ declared-in } C; \ G \vdash n \text{ declared-in } C; \text{ memberid } m = \text{ memberid } n \rrbracket$
 $\implies m = n$
apply (*cases m*)
apply (*cases n*,
auto simp add: declared-in-def cdeclaredmethd-def cdeclaredfield-def) +
done

lemma *declared-not-undeclared*:
 $G \vdash m \text{ declared-in } C \implies \neg G \vdash \text{ memberid } m \text{ undeclared-in } C$
by (*cases m*) (*auto simp add: declared-in-def undeclared-in-def*)

lemma *undeclared-not-declared*:
 $G \vdash \text{ memberid } m \text{ undeclared-in } C \implies \neg G \vdash m \text{ declared-in } C$
by (*cases m*) (*auto simp add: declared-in-def undeclared-in-def*)

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

proof –
assume *not-undecl*: $\neg G \vdash \text{ membr-id undeclared-in } C$
show *?thesis* (*is ?P membr-id*)
proof (*cases membr-id*)
case (*fid vname*)
with *not-undecl*
obtain fld where
 $G \vdash \text{fdecl } (vname, fld) \text{ declared-in } C$
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
 $G \vdash \text{mdecl } (sig, mthd) \text{ declared-in } C$
by (*auto simp add: undeclared-in-def declared-in-def*
cdeclaredmethd-def)
with mid show *?thesis*
by *auto*
qed
qed

lemma *unique-declared-in*:
 $\llbracket G \vdash m \text{ declared-in } C; \ G \vdash n \text{ declared-in } C; \text{ memberid } m = \text{ memberid } n \rrbracket$
 $\implies m = n$
by (*auto simp add: declared-in-def cdeclaredmethd-def cdeclaredfield-def*
split: memberdecl.splits)

lemma *unique-member-of*:
assumes *n*: $G \vdash n \text{ member-of } C$ **and**

```

    m:  $G \vdash m$  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:  $G \vdash$  mbr  $n$  declared-in  $C$  declclass  $n = C$ 
  assume eqid: memberid  $n =$  memberid  $m$ 
  assume  $G \vdash m$  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:  $G \vdash$  memberid  $n$  undeclared-in  $C$ 
  assume super:  $G \vdash C <_C 1S$ 
  assume hyp:  $\llbracket G \vdash m$  member-of  $S$ ; memberid  $n =$  memberid  $m \rrbracket \implies n = m$ 
  assume eqid: memberid  $n =$  memberid  $m$ 
  assume  $G \vdash m$  member-of  $C$ 
  then show  $n=m$ 
  proof (cases)
    case Immediate
    then have  $G \vdash$  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  $G \vdash m$  member-of  $S$ 
  by (auto dest!: subcls1D)
  with eqid hyp
  show ?thesis
  by blast
qed
qed
qed

lemma member-of-is-classD:  $G \vdash m$  member-of  $C \implies$  is-class  $G$   $C$ 
proof (induct set: members)
  case (Immediate  $m$   $C$ )
  assume  $G \vdash$  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  $m$   $C$   $S$ )
show is-class  $G$   $C$  if  $G \vdash C \prec_C 1S$  and is-class  $G$   $S$ 
  by (rule subcls-is-class2) (use that in auto)
qed

```

```

lemma member-of-declC:
   $G \vdash m$  member-of  $C$ 
   $\implies G \vdash m$  declared-in (declclass  $m$ )
by (induct set: members) auto

```

```

lemma member-of-member-of-declC:
   $G \vdash m$  member-of  $C$ 
   $\implies G \vdash m$  member-of (declclass  $m$ )
by (auto dest: member-of-declC intro: members.Immediate)

```

```

lemma member-of-class-relation:
   $G \vdash m$  member-of  $C \implies G \vdash C \preceq_C$  declclass  $m$ 
proof (induct set: members)
  case (Immediate  $m$   $C$ )
  then show  $G \vdash C \preceq_C$  declclass  $m$  by simp
next
  case (Inherited  $m$   $C$   $S$ )
  then show  $G \vdash C \preceq_C$  declclass  $m$ 
    by (auto dest: r-into-rtrancl intro: rtrancl-trans)
qed

```

```

lemma member-in-class-relation:
   $G \vdash m$  member-in  $C \implies G \vdash C \preceq_C$  declclass  $m$ 
by (auto dest: member-inD member-of-class-relation
  intro: rtrancl-trans)

```

```

lemma stat-override-declclasses-relation:
   $\llbracket G \vdash (\text{declclass } \text{new}) \prec_C 1 \text{ superNew}; G \vdash \text{Method } \text{old} \text{ member-of } \text{superNew} \rrbracket$ 
   $\implies G \vdash (\text{declclass } \text{new}) \prec_C (\text{declclass } \text{old})$ 
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:
   $\llbracket G \vdash \text{new overrides}_S \text{ old} \rrbracket \implies$ 
   $\text{declclass } \text{new} \neq \text{Object} \wedge \neg \text{is-static } \text{new} \wedge \text{msig } \text{new} = \text{msig } \text{old} \wedge$ 
   $G \vdash (\text{declclass } \text{new}) \prec_C (\text{declclass } \text{old}) \wedge$ 
   $G \vdash \text{Method } \text{new} \text{ declared-in } (\text{declclass } \text{new}) \wedge$ 
   $G \vdash \text{Method } \text{old} \text{ declared-in } (\text{declclass } \text{old})$ 
apply (induct set: stat-overridesR)
apply (frule (1) stat-override-declclasses-relation)
apply (auto intro: trancl-trans)
done

```

```

lemma member-of-Package:
  assumes  $G \vdash m$  member-of  $C$ 
  and accmodi  $m = \text{Package}$ 
  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*: $G \vdash m \text{ member-in } C \implies G \vdash m \text{ member-in } (\text{declclass } m)$

proof –

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

from *member-in-C*

obtain *provC* **where**

subclseq-C-provC: $G \vdash C \preceq_C \text{ provC}$ **and**

member-of-provC: $G \vdash m \text{ member-of } \text{provC}$

by (auto simp add: member-in-def)

from *member-of-provC*

have $G \vdash m \text{ member-of } \text{declclass } m$

by (rule member-of-member-of-declC)

moreover

from *member-in-C*

have $G \vdash C \preceq_C \text{ declclass } m$

by (rule member-in-class-relation)

ultimately

show ?thesis

by (auto simp add: member-in-def)

qed

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

$\implies G \vdash m \text{ member-in } C$

by (auto elim: dyn-accessible-fromR.induct)

lemma *no-Private-stat-override*:

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

by (induct set: stat-overridesR) (auto simp add: inheritable-in-def)

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

by (induct set: overridesR) (auto simp add: inheritable-in-def)

lemma *permits-acc-inheritance*:

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

$\implies G \vdash m \text{ in } \text{dynC} \text{ permits-acc-from } \text{accC}$

by (cases accmodi m)

(auto simp add: permits-acc-def

intro: subclseq-trans)

lemma *permits-acc-static-declC*:

$\llbracket G \vdash m \text{ in } C \text{ permits-acc-from } \text{accC}; G \vdash m \text{ member-in } C; \text{is-static } m \rrbracket$

$\implies G \vdash m \text{ in } (\text{declclass } m) \text{ permits-acc-from } \text{accC}$

by (cases accmodi m) (auto simp add: permits-acc-def)

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

assumes *acc-C*: $G \vdash m \text{ in } C \text{ dyn-accessible-from } \text{accC}$ **and**

static: *is-static* m

shows $G \vdash m \text{ in } (\text{declclass } m) \text{ dyn-accessible-from } \text{accC}$

proof –

from *acc-C static*

show $G \vdash m \text{ in } (\text{declclass } m) \text{ dyn-accessible-from } \text{accC}$

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  $\neg$  is-static m
    by (auto dest: overrides-commonD)
  moreover
  assume is-static m
  ultimately show ?case
    by contradiction
qed
qed

lemma field-accessible-fromD:
   $\llbracket G \vdash \text{membr of } C \text{ accessible-from } \text{accC}; \text{is-field membr} \rrbracket$ 
 $\implies G \vdash \text{membr member-of } C \wedge$ 
 $G \vdash (\text{Class } C) \text{ accessible-in } (\text{pid } \text{accC}) \wedge$ 
 $G \vdash \text{membr in } C \text{ permits-acc-from } \text{accC}$ 
by (cases set: accessible-fromR)
  (auto simp add: is-field-def split: memberdecl.splits)

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

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

lemma member-of-Private:
   $\llbracket G \vdash m \text{ member-of } C; \text{accmodi } m = \text{Private} \rrbracket \implies \text{declclass } m = C$ 
by (induct set: members) (auto simp add: inheritable-in-def)

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

lemma member-of-inheritance:
  assumes m:  $G \vdash m \text{ member-of } D$  and
    subclseq-D-C:  $G \vdash D \preceq_C C$  and
    subclseq-C-m:  $G \vdash C \preceq_C \text{declclass } m$  and
    ws: ws-prog G
  shows  $G \vdash m \text{ member-of } C$ 
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  $G \vdash C \preceq_C \text{declclass } m$ 
    with ws have D=C

```

```

  by (auto intro: subclseq-acyclic)
with Immediate
show  $G \vdash m$  member-of  $C$ 
  by (auto intro: members.Immediate)
next
case (Inherited  $m$   $D$   $S$ )
assume member-of- $D$ -props:
   $G \vdash m$  inheritable-in pid  $D$ 
   $G \vdash$  memberid  $m$  undeclared-in  $D$ 
   $G \vdash$  Class  $S$  accessible-in pid  $D$ 
   $G \vdash m$  member-of  $S$ 
assume super:  $G \vdash D \prec_C 1S$ 
assume hyp:  $\llbracket G \vdash S \preceq_C C; G \vdash C \preceq_C \text{declclass } m \rrbracket \implies G \vdash m$  member-of  $C$ 
assume subclseq- $C$ - $m$ :  $G \vdash C \preceq_C \text{declclass } m$ 
assume  $G \vdash D \preceq_C C$ 
then show  $G \vdash m$  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  $G \vdash D \prec_C C$ 
with super
have  $G \vdash S \preceq_C C$ 
  by (auto dest: subcls1D subcls-superD)
with subclseq- $C$ - $m$  hyp show ?thesis
  by blast
qed
qed
qed

```

lemma member-of-subcls:

```

assumes
  old:  $G \vdash$  old member-of  $C$  and
  new:  $G \vdash$  new member-of  $D$  and
  eqid: memberid new = memberid old and
  subclseq- $D$ - $C$ :  $G \vdash D \preceq_C C$  and
  subcls-new-old:  $G \vdash \text{declclass new} \prec_C \text{declclass old}$  and
  ws: ws-prog  $G$ 
shows  $G \vdash D \prec_C C$ 
proof -
  from old
  have subclseq- $C$ -old:  $G \vdash C \preceq_C \text{declclass old}$ 
    by (auto dest: member-of-subclseq-declC)
  from new
  have subclseq- $D$ -new:  $G \vdash D \preceq_C \text{declclass new}$ 
    by (auto dest: member-of-subclseq-declC)
  from subcls-new-old ws
  have new-new-old: new  $\neq$  old
    by (cases new, cases old) (auto dest: subcls-irrefl)
  from subclseq- $D$ -new subclseq- $D$ -C
  have  $G \vdash (\text{declclass new}) \preceq_C C \vee G \vdash C \preceq_C (\text{declclass new})$ 
    by (rule subcls-compareable)
  then have  $G \vdash (\text{declclass new}) \preceq_C C$ 
  proof
    assume  $G \vdash \text{declclass new} \preceq_C C$  then show ?thesis .
  next

```

```

    assume  $G \vdash C \preceq_C (\text{declclass new})$ 
    with  $\text{new subclseq-}D\text{-}C \text{ ws}$ 
    have  $G \vdash \text{new member-of } C$ 
      by (blast intro: member-of-inheritance)
    with  $\text{eqid old}$ 
    have  $\text{new} = \text{old}$ 
      by (blast intro: unique-member-of)
    with  $\text{neg-new-old}$ 
    show ?thesis
      by contradiction
  qed
  then show ?thesis
  proof (cases rule: subclseq-cases)
    case Eq
    assume  $\text{declclass new} = C$ 
    with  $\text{new}$  have  $G \vdash \text{new member-of } C$ 
      by (auto dest: member-of-member-of-declC)
    with  $\text{eqid old}$ 
    have  $\text{new} = \text{old}$ 
      by (blast intro: unique-member-of)
    with  $\text{neg-new-old}$ 
    show ?thesis
      by contradiction
  next
    case Subcls
    assume  $G \vdash \text{declclass new} \prec_C C$ 
    with  $\text{subclseq-}D\text{-new}$ 
    show  $G \vdash D \prec_C C$ 
      by (rule rtrancl-trancl-trancl)
  qed
qed

corollary member-of-overrides-subcls:
 $\llbracket G \vdash \text{Methd sig old member-of } C; G \vdash \text{Methd sig new member-of } D; G \vdash D \preceq_C C; \\ G, \text{sig} \vdash \text{new overrides old}; \text{ws-prog } G \rrbracket \\ \implies G \vdash D \prec_C C$ 
by (drule overrides-commonD) (auto intro: member-of-subcls)

corollary member-of-stat-overrides-subcls:
 $\llbracket G \vdash \text{Methd sig old member-of } C; G \vdash \text{Methd sig new member-of } D; G \vdash D \preceq_C C; \\ G, \text{sig} \vdash \text{new overrides}_S \text{ old}; \text{ws-prog } G \rrbracket \\ \implies G \vdash D \prec_C C$ 
by (drule stat-overrides-commonD) (auto intro: member-of-subcls)

lemma inherited-field-access:
  assumes  $\text{stat-acc: } G \vdash \text{membr of stat}C \text{ accessible-from acc}C$  and
     $\text{is-field: is-field membr}$  and
     $\text{subclseq: } G \vdash \text{dyn}C \preceq_C \text{stat}C$ 
  shows  $G \vdash \text{membr in dyn}C \text{ dyn-accessible-from acc}C$ 
proof –
  from  $\text{stat-acc is-field subclseq}$ 
  show ?thesis
    by (auto dest: accessible-fieldD
      intro: dyn-accessible-fromR.Immediate
      member-inI
      permits-acc-inheritance)
  qed

```


lemma *accessible-inheritance*:

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

proof –

from *stat-acc*
have *member-statC*: $G \vdash m$ *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* \times *qname*

translations

(*type*) *fspec* \leq (*type*) *vname* \times *qname*

definition

imethds :: *prog* \Rightarrow *qname* \Rightarrow (*sig*, *qname* \times *mhead*) *tables* **where**
imethds *G* *I* =
iface-rec *G* *I* (λI *i* *ts*. (*Un-tables* *ts*) $\oplus \oplus$
(*set-option* \circ *table-of* (*map* ($\lambda(s,m)$. (*s*, *I*, *m*)) (*imethds* *i*))))

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

definition

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

only returns *imethds* if the interface is accessible

definition

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

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

definition

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

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 $:: prog \Rightarrow qname \Rightarrow qname \Rightarrow (sig, qname \times methd) table$ **where**
dynmethd $G\ statC\ dynC =$
 $(\lambda sig.$
 $\quad (if\ G \vdash dynC \preceq_C statC$
 $\quad \quad then\ (case\ methd\ G\ statC\ sig\ of$
 $\quad \quad \quad None \Rightarrow None$
 $\quad \quad \quad | Some\ statM$
 $\quad \quad \quad \Rightarrow (class-rec\ G\ dynC\ Map.empty$
 $\quad \quad \quad \quad (\lambda C\ c\ subcls-mthds.$
 $\quad \quad \quad \quad \quad subcls-mthds$
 $\quad \quad \quad \quad ++$
 $\quad \quad \quad \quad (filter-tab$
 $\quad \quad \quad \quad \quad (\lambda - dynM.\ G, sig \vdash dynM\ overrides\ statM \vee dynM = statM)$
 $\quad \quad \quad \quad \quad (methd\ G\ C))$
 $\quad \quad \quad \quad)\ sig$
 $\quad \quad)$
 $\quad else\ None))$

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 $:: prog \Rightarrow qname \Rightarrow qname \Rightarrow (sig, qname \times methd) table$ **where**
dynimethd $G\ I\ dynC =$
 $(\lambda sig.\ if\ imethds\ G\ I\ sig \neq \{\}$
 $\quad then\ methd\ G\ dynC\ sig$
 $\quad else\ dynmethd\ G\ Object\ dynC\ sig)$

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 $:: prog \Rightarrow ref-ty \Rightarrow qname \Rightarrow (sig, qname \times methd) table$ **where**
dynlookup $G\ statT\ dynC =$

```

(case statT of
  NullT      ⇒ Map.empty
| IfaceT I    ⇒ dynimethd G I    dynC
| ClassT statC ⇒ dynmethd G statC dynC
| ArrayT ty   ⇒ dynmethd G Object dynC)

```

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

definition

```

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

```

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

definition

```

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

```

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

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

definition

```

is-methd :: prog ⇒ qname ⇒ sig ⇒ bool
where is-methd G = (λC sig. is-class G C ∧ methd G C sig ≠ None)

```

definition

```

efname :: ((vname × qname) × field) ⇒ (vname × qname)
where efname = fst

```

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

4 imethds

lemma *imethds-rec*: $\llbracket \text{iface } G \text{ } I = \text{Some } i; \text{ws-prog } G \rrbracket \implies$
 $\text{imethds } G \text{ } I = \text{Un-tables } ((\lambda J. \text{imethds } G \text{ } J) \text{'set (isuperIfs } i) \oplus \oplus$
 $(\text{set-option } \circ \text{table-of } (\text{map } (\lambda(s,mh). (s,I,mh)) (\text{imethds } i)))$
apply (*unfold imethds-def*)
apply (*rule iface-rec [THEN trans]*)
apply *auto*
done

lemma *imethds-norec*:

```

 $\llbracket \text{iface } G \text{ } md = \text{Some } i; \text{ws-prog } G; \text{table-of (imethds } i) \text{ sig} = \text{Some } mh \rrbracket \implies$ 
 $(md, mh) \in \text{imethds } G \text{ } md \text{ sig}$ 
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:  $\llbracket m \in \text{imethds } G \ I \ \text{sig}; \text{ws-prog } G; \text{is-iface } G \ I \rrbracket \implies$ 
   $(\exists i. \text{iface } G \ (\text{decliface } m) = \text{Some } i \wedge$ 
   $\text{table-of } (\text{imethds } i) \ \text{sig} = \text{Some } (\text{mthd } m)) \wedge$ 
   $(I, \text{decliface } m) \in (\text{subint1 } G)^* \wedge m \in \text{imethds } G \ (\text{decliface } m) \ \text{sig}$ 
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:  $im \in \text{imethds } G \ I \ \text{sig}$ 
  and ifI:  $\text{iface } G \ I = \text{Some } i$ 
  and ws:  $\text{ws-prog } G$ 
  obtains (NewMethod)  $\text{table-of } (\text{map } (\lambda(s, mh). (s, I, mh)) (\text{imethds } i)) \ \text{sig} = \text{Some } im$ 
   $| (\text{InheritedMethod}) \ J \text{ where } J \in \text{set } (\text{isuperIfs } i) \text{ and } im \in \text{imethds } G \ J \ \text{sig}$ 
using assms by (auto simp add: imethds-rec)

```

5 accimethd

```

lemma accimethds-simp [simp]:
   $G \vdash \text{Iface } I \text{ accessible-in pack} \implies \text{accimethds } G \ \text{pack } I = \text{imethds } G \ I$ 
by (simp add: accimethds-def)

```

```

lemma accimethdsD:
   $im \in \text{accimethds } G \ \text{pack } I \ \text{sig}$ 
   $\implies im \in \text{imethds } G \ I \ \text{sig} \wedge G \vdash \text{Iface } I \text{ accessible-in pack}$ 
by (auto simp add: accimethds-def)

```

```

lemma accimethdsI:
   $\llbracket im \in \text{imethds } G \ I \ \text{sig}; G \vdash \text{Iface } I \text{ accessible-in pack} \rrbracket$ 
   $\implies im \in \text{accimethds } G \ \text{pack } I \ \text{sig}$ 
by simp

```

6 methd

```

lemma methd-rec:  $\llbracket \text{class } G \ C = \text{Some } c; \text{ws-prog } G \rrbracket \implies$ 
   $\text{methd } G \ C$ 
   $= (\text{if } C = \text{Object}$ 
   $\text{then Map.empty}$ 
   $\text{else filter-tab } (\lambda \text{sig } m. G \vdash C \text{ inherits method sig } m)$ 
   $\text{ (methd } G \ (\text{super } c)))$ 
   $++ \text{table-of } (\text{map } (\lambda(s, m). (s, C, m)) (\text{methods } c))$ 
apply (unfold methd-def)
apply (erule class-rec [THEN trans], assumption)
apply (simp)
done

```

```

lemma methd-norec:
   $\llbracket \text{class } G \ \text{declC} = \text{Some } c; \text{ws-prog } G; \text{table-of } (\text{methods } c) \ \text{sig} = \text{Some } m \rrbracket$ 
   $\implies \text{methd } G \ \text{declC} \ \text{sig} = \text{Some } (\text{declC}, m)$ 
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*:

```


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


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


$$G \vdash C \preceq_C (\text{declclass } m) \wedge \text{methd } G \ (\text{declclass } m) \ \text{sig} = \text{Some } m$$

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 ( $\lambda(s, m). (s, Ca, m)$ ) (methods c)) sig)
apply (force intro: rtranc1-into-rtranc2)

```

```

apply (auto intro: methd-norec)
done

```

lemma *methd-inheritedD*:

```


$$\llbracket \text{class } G \ C = \text{Some } c; \text{ws-prog } G; \text{methd } G \ C \ \text{sig} = \text{Some } m \rrbracket$$


$$\implies (\text{declclass } m \neq C \longrightarrow G \vdash C \text{ inherits method sig } m)$$

by (auto simp add: methd-rec)

```

lemma *methd-diff-cls*:

```


$$\llbracket \text{ws-prog } G; \text{is-class } G \ C; \text{is-class } G \ D; \text{methd } G \ C \ \text{sig} = m; \text{methd } G \ D \ \text{sig} = n; m \neq n \rrbracket$$


$$\implies C \neq D$$

by (auto simp add: methd-rec)

```

lemma *method-declared-inI*:

```


$$\llbracket \text{table-of } (\text{methods } c) \ \text{sig} = \text{Some } m; \text{class } G \ C = \text{Some } c \rrbracket$$


$$\implies G \vdash \text{mdecl } (\text{sig}, m) \text{ declared-in } C$$

by (auto simp add: cdeclaredmethd-def declared-in-def)

```

lemma *methd-declared-in-declclass*:

```


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


$$\implies G \vdash \text{Methd sig } m \text{ declared-in } (\text{declclass } m)$$

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

```

lemma *member-methd*:

```

assumes member-of:  $G \vdash \text{Methd sig } m \text{ member-of } C$  and
           ws: ws-prog  $G$ 
shows methd  $G \ C \ \text{sig} = \text{Some } m$ 
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  $G \vdash \text{Methd sig } m \text{ member-of Object}$ 
    then have  $G \vdash \text{Methd sig } m \text{ declared-in Object} \wedge \text{declclass } m = \text{Object}$ 
      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*:
 $\text{accmethd } G \ S \ C \ \text{sig} = \text{Some } m$
 $\implies \text{methd } G \ C \ \text{sig} = \text{Some } m \wedge G \vdash \text{method } \text{sig } m \text{ of } C \text{ accessible-from } S$
by (*auto simp add: accmethd-def*)

lemma *accmethd-SomeI*:
 $\llbracket \text{methd } G \ C \ \text{sig} = \text{Some } m; G \vdash \text{method } \text{sig } m \text{ of } C \text{ accessible-from } S \rrbracket$
 $\implies \text{accmethd } G \ S \ C \ \text{sig} = \text{Some } m$
by (*auto simp add: accmethd-def intro: filter-tab-SomeI*)

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

lemma *finite-dom-accmethd*:
 $\llbracket \text{ws-prog } G; \text{is-class } G \ C \rrbracket \implies \text{finite } (\text{dom } (\text{accmethd } G \ S \ C))$
by (*auto simp add: accmethd-def intro: finite-dom-filter-tab finite-dom-methd*)

8 dynmethd

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

proof –
assume *clsDynC*: $\text{class } G \ \text{dynC} = \text{Some } c$ **and**
 $\text{ws: ws-prog } G$
then show $? \text{Dynmethd-def } \text{dynC} \ \text{sig} = ? \text{Dynmethd-rec } \text{dynC} \ c \ \text{sig}$
proof (*induct rule: ws-class-induct''*)
case (*Object co*)
show $? \text{Dynmethd-def } \text{Object} \ \text{sig} = ? \text{Dynmethd-rec } \text{Object} \ co \ \text{sig}$
proof (*cases* $G \vdash \text{Object} \preceq_C \text{statC}$)
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 =  $G \vdash \text{qmdecl sig dynM overrides qmdecl sig statM} \vee$ 
  dynM = statM
show ?thesis
proof (cases ?filter dynC sig)
  case None
  with dynM
  have no-termination:  $\neg ?Termination$ 
    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*: $\llbracket \text{is-class } G \ C; \text{ws-prog } G \rrbracket$
 $\implies \text{dynmethd } G \ C \ C \ \text{sig} = \text{methd } G \ C \ \text{sig}$
apply (auto simp add: dynmethd-rec)
done

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

lemma *dynmethd-Some-cases*:
assumes *dynM*: $\text{dynmethd } G \ \text{statC} \ \text{dynC} \ \text{sig} = \text{Some dynM}$
and *is-cls-dynC*: $\text{is-class } G \ \text{dynC}$

```

and ws: ws-prog G
obtains (Static) methd G statC sig = Some dynM
  | (Overrides) statM
  where methd G statC sig = Some statM
    and dynM  $\neq$  statM
    and G,sig  $\vdash$  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  $\neq$  statM
shows dynC  $\neq$  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  $\vdash$  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*:

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

$\rrbracket \implies$

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

proof –

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 $G \vdash \text{dynC} \preceq_C \text{statC}$ **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* \implies ?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: $G \vdash \text{dynC} \preceq_C \text{statC}$ **and**

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

by (blast dest: dynmethdSomeD)

from *clsDynC neq-dynC-Obj*

have *subclseq-dynC-super*: $G \vdash \text{dynC} \preceq_C (\text{super } c)$

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**

$subclseq: G \vdash dynC \preceq_C statC$ **and**

$is-cl\text{-}statC: is-class\ G\ statC$ **and**

$ws: ws-prog\ G$

shows $\exists\ dynM. dynmethd\ G\ statC\ dynC\ sig = Some\ dynM$
(is ?P dynC)

proof –

from $subclseq\ is-cl\text{-}statC$

have $is-cl\text{-}dynC: is-class\ G\ dynC$ **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 $neq-dynC-Obj: dynC \neq Object$

assume $hyp: G \vdash super\ dc \preceq_C statC \implies ?P\ (super\ dc)$

assume $subclseq': G \vdash dynC \preceq_C statC$

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 $G \vdash dynC \prec_C statC$

from $this\ clsDynC'$

have $G \vdash super\ dc \preceq_C statC$ **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 $is-cl\text{-}statC: is-class\ G\ statC$

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 $subclseq\ is-cl\text{-}statC$

have $is-cl\text{-}dynC: is-class\ G\ dynC$ **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:  $G \vdash \text{dynC} \preceq_C \text{statC}$  and
    is-cls-statC: is-class G statC and
    ws: ws-prog G
  shows
     $\exists \text{dynM}. \text{dynmethd } G \text{ statC dynC sig} = \text{Some dynM} \wedge$ 
     $\text{is-static dynM} = \text{is-static statM} \wedge G \vdash \text{resTy dynM} \preceq_{\text{resTy}} \text{statM}$ 
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 dynmethd G I dynC sig = x
    | (ClassT) statC where statT = ClassT statC and dynmethd G statC dynC sig = x
    | (ArrayT) ty where statT = ArrayT ty and dynmethd G Object dynC sig = x
  using assms by (cases statT) (auto simp add: dynlookup-def)

```

10 fields

```

lemma fields-rec:  $\llbracket \text{class } G \text{ } C = \text{Some } c; \text{ws-prog } G \rrbracket \implies$ 
   $\text{fields } G \text{ } C = \text{map } (\lambda(fn, ft). ((fn, C), ft)) (cfields \text{ } c) @$ 
   $(\text{if } C = \text{Object then } [] \text{ else } \text{fields } G \text{ (super } c))$ 
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 ⊆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 (λfn. 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 (λfn. 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*: $\bigwedge y. \llbracket ws-prog G; class G C = Some c; cfields c = [] \rrbracket;$
 $C \neq Object \longrightarrow class G (super c) = Some y \wedge fields G (super c) = [] \implies$
 $fields G C = []$
apply (subst fields-rec)
apply assumption
apply auto
done

lemma *fields-mono-lemma*:

```

[[x ∈ set (fields G C); G⊢D ⊆C C; ws-prog G]]
  ==> x ∈ 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*:

```

 $\llbracket (efn, fd) \in \text{set } (fields\ G\ (super\ c)); fc \in \text{set } (cfields\ c); ws\text{-prog}\ G;$ 
 $\quad fname\ efn = fname\ fc; declclassf\ efn = C;$ 
 $\quad class\ G\ C = Some\ c; C \neq Object; class\ G\ (super\ c) = Some\ d \rrbracket \implies R$ 
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*: $\llbracket is\text{-class}\ G\ C; ws\text{-prog}\ G;$

$\quad \bigwedge C\ c. \llbracket class\ G\ C = Some\ c \rrbracket \implies unique\ (cfields\ c) \rrbracket \implies$
 $unique\ (fields\ G\ C)$

```

apply (rule ws-subcls1-induct, assumption, assumption)
apply (subst fields-rec, assumption)
apply (auto intro!: unique-map-inj inj-onI
 $\quad elim!: unique-append\ ws\text{-unique-fields-lemma}\ fields\text{-norec}$ )
done

```

11 accfield

lemma *accfield-fields*:

```

 $accfield\ G\ S\ C\ fn = Some\ f$ 
 $\implies table\text{-of}\ (fields\ G\ C)\ (fn, declclass\ f) = Some\ (fld\ f)$ 
apply (simp only: accfield-def Let-def)
apply (rule table-of-remap-SomeD)
apply auto
done

```

lemma *accfield-declC-is-class*:

```

 $\llbracket is\text{-class}\ G\ C; accfield\ G\ S\ C\ en = Some\ (fd, f); ws\text{-prog}\ G \rrbracket \implies$ 
 $is\text{-class}\ G\ fd$ 
apply (drule accfield-fields)
apply (drule fields-declC [THEN conjunct1], assumption)
apply auto
done

```

lemma *accfield-accessibleD*:

```

 $accfield\ G\ S\ C\ fn = Some\ f \implies G \vdash Field\ fn\ f\ of\ C\ accessible\text{-from}\ S$ 
by (auto simp add: accfield-def Let-def)

```

12 is methd

lemma *is-methdI*:

```

 $\llbracket class\ G\ C = Some\ y; methd\ G\ C\ sig = Some\ b \rrbracket \implies is\text{-methd}\ G\ C\ sig$ 
apply (unfold is-methd-def)
apply auto
done

```

lemma *is-methdD*:

```

 $is\text{-methd}\ G\ C\ sig \implies class\ G\ C \neq None \wedge methd\ G\ C\ sig \neq None$ 
apply (unfold is-methd-def)

```

apply *auto*
done

lemma *finite-is-methd*:
 $ws\text{-}prog\ G \implies finite\ (Collect\ (case\text{-}prod\ (is\text{-}methd\ G)))$
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

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

lemma *superclasses-rec*: $\llbracket class\ G\ C = Some\ c; ws\text{-}prog\ G \rrbracket \implies$
 $superclasses\ G\ C$
 $= (if\ (C=Object)$
 $\quad then\ \{\}$
 $\quad else\ insert\ (super\ c)\ (superclasses\ G\ (super\ c)))$
apply (*unfold superclasses-def*)
apply (*erule class-rec [THEN trans], assumption*)
apply (*simp*)
done

lemma *superclasses-mono*:
assumes *clsrel*: $G \vdash C \prec_C D$
and *ws*: $ws\text{-}prog\ G$
and *cls-C*: $class\ G\ C = Some\ c$
and *wf*: $\bigwedge C\ c. \llbracket class\ G\ C = Some\ c; C \neq Object \rrbracket$
 $\implies \exists sc. class\ G\ (super\ c) = Some\ sc$
and *x*: $x \in superclasses\ G\ D$
shows $x \in superclasses\ G\ C$ **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*
 $simp\ add: superclasses\text{-}rec\ subcls1\text{-}def$)
next
case (*step C S*)
moreover **note** *wf ws*
moreover **from** *calculation*
have $x \in superclasses\ G\ S$
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:  $G \vdash C \prec_C D$ 
  and ws: ws-prog G
  and cls-C: class G C = Some c
  and wf:  $\bigwedge C \ c. \llbracket \text{class } G \ C = \text{Some } c; \ C \neq \text{Object} \rrbracket$ 
     $\implies \exists sc. \text{class } G \ (\text{super } c) = \text{Some } sc$ 
  shows  $D \in \text{superclasses } G \ C$  using clsrel cls-C
proof (induct arbitrary: c rule: converse-trancl-induct)
  case (base C)
  show ?case
    by (use ws wf base in  $\langle \text{auto intro: no-subcls1-Object simp add: superclasses-rec subcls1-def} \rangle$ )
next
  case (step C S)
  show ?case
    by (rule superclasses-mono)
    (use ws wf step in  $\langle \text{auto dest: no-subcls1-Object simp add: subcls1-def} \rangle$ )
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* \Rightarrow *ty option*
(*type*) *env* <= (*type*) (*prg*::*prog*, *cls*::*qtname*, *lcl*::*lenv*)
(*type*) *env* <= (*type*) (*prg*::*prog*, *cls*::*qtname*, *lcl*::*lenv*, . . . :: 'a)

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

Static overloading: maximally specific methods

type-synonym

$emhead = ref\text{-}ty \times mhead$

— Some mnemonic selectors for `emhead`

definition

$declrefT :: emhead \Rightarrow ref\text{-}ty$

where $declrefT = fst$

definition

$mhd :: emhead \Rightarrow mhead$

where $mhd \equiv snd$

lemma $declrefT\text{-}simp[simp]: declrefT (r, m) = r$

by ($simp$ add: $declrefT\text{-}def$)

lemma $mhd\text{-}simp[simp]: mhd (r, m) = m$

by ($simp$ add: $mhd\text{-}def$)

lemma $static\text{-}mhd\text{-}simp[simp]: static (mhd m) = is\text{-}static m$

by ($cases m$) ($simp$ add: $member\text{-}is\text{-}static\text{-}simp$ $mhd\text{-}def$)

lemma $mhd\text{-}resTy\text{-}simp [simp]: resTy (mhd m) = resTy m$

by ($cases m$) $simp$

lemma $mhd\text{-}is\text{-}static\text{-}simp [simp]: is\text{-}static (mhd m) = is\text{-}static m$

by ($cases m$) $simp$

lemma $mhd\text{-}accmodi\text{-}simp [simp]: accmodi (mhd m) = accmodi m$

by ($cases m$) $simp$

definition

$cmheads :: prog \Rightarrow qname \Rightarrow qname \Rightarrow sig \Rightarrow emhead \text{ set}$

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

definition

$Objectmheads :: prog \Rightarrow qname \Rightarrow sig \Rightarrow emhead \text{ set}$ **where**

$Objectmheads G S =$

$(\lambda sig. (\lambda (Cls, mthd). (ClassT Cls, (mhead mthd))))$

$\text{‘ set-option (filter-tab } (\lambda sig m. accmodi m \neq Private) \text{ (accmethd } G S \text{ Object) sig)}$

definition

$accObjectmheads :: prog \Rightarrow qname \Rightarrow ref\text{-}ty \Rightarrow sig \Rightarrow emhead \text{ set}$

where

$accObjectmheads G S T =$

$(if G \vdash RefT T \text{ accessible-in } (pid S)$

$\text{ then } Objectmheads G S$

$\text{ else } (\lambda sig. \{\}))$

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

where

$mheads G S NullT = (\lambda sig. \{\})$

| $mheads G S (IfaceT I) = (\lambda sig. (\lambda (I, h). (IfaceT I, h)))$

$\text{ ‘ accimethds } G (pid S) I sig \cup$

$accObjectmheads G S (IfaceT I) sig)$

| $mheads G S (ClassT C) = cmheads G S C$

| $mheads G S (ArrayT T) = accObjectmheads G S (ArrayT T)$

definition

— applicable methods, cf. 15.11.2.1

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

definition

— more specific methods, cf. 15.11.2.2

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

definition

— maximally specific methods, cf. 15.11.2.2

$max\text{-}spec :: prog \Rightarrow qname \Rightarrow ref\text{-}ty \Rightarrow sig \Rightarrow (emhead \times ty\ list) \text{ set}$ **where**
 $max\text{-}spec\ G\ S\ rt\ sig = \{m.\ m \in appl\text{-}methds\ G\ S\ rt\ sig \wedge$
 $(\forall m' \in appl\text{-}methds\ G\ S\ rt\ sig.\ more\text{-}spec\ G\ m'\ m \longrightarrow m' = m)\}$

lemma $max\text{-}spec2appl\text{-}methds$:

$x \in max\text{-}spec\ G\ S\ T\ sig \implies x \in appl\text{-}methds\ G\ S\ T\ sig$

by ($auto\ simp$: $max\text{-}spec\text{-}def$)

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

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

by ($auto\ simp$: $appl\text{-}methds\text{-}def$)

lemma $max\text{-}spec2mheads$:

$max\text{-}spec\ G\ S\ rt\ (\downarrow name = mn, parTs = pTs) = insert\ (mh, pTs')\ A$

$\implies mh \in mheads\ G\ S\ rt\ (\downarrow name = mn, parTs = pTs') \wedge G \vdash pTs [\preceq] pTs'$

apply ($auto\ dest$: $equalityD2\ subsetD\ max\text{-}spec2appl\text{-}methds\ appl\text{-}methdsD$)

done

definition

$empty\text{-}dt :: dyn\text{-}ty$

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

definition

$invmode :: ('a::type)member\text{-}scheme \Rightarrow expr \Rightarrow inv\text{-}mode$ **where**

$invmode\ m\ e = (if\ is\text{-}static\ m$

$then\ Static$

$else\ if\ e = Super\ then\ SuperM\ else\ IntVir)$

lemma $invmode\text{-}nonstatic\ [simp]$:

$invmode\ (\downarrow access = a, static = False, \dots = x)\ (Acc\ (LVar\ e)) = IntVir$

apply ($unfold\ invmode\text{-}def$)

apply ($simp\ (no\text{-}asm)\ add$: $member\text{-}is\text{-}static\text{-}simp$)

done

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

apply ($unfold\ invmode\text{-}def$)

apply ($simp\ (no\text{-}asm)$)

done

lemma *invmode-IntVir-eq*: $(\text{invmode } m \ e = \text{IntVir}) = (\neg(\text{is-static } m) \wedge e \neq \text{Super})$
apply (*unfold invmode-def*)
apply (*simp (no-asm)*)
done

lemma *Null-staticD*:
 $a' = \text{Null} \longrightarrow (\text{is-static } m) \implies \text{invmode } m \ e = \text{IntVir} \longrightarrow a' \neq \text{Null}$
apply (*clarsimp simp add: invmode-IntVir-eq*)
done

Typing for unary operations

primrec *unop-type* :: *unop* \Rightarrow *prim-ty*
where
 $\text{unop-type } UPlus = \text{Integer}$
 $\text{unop-type } UMinus = \text{Integer}$
 $\text{unop-type } UBitNot = \text{Integer}$
 $\text{unop-type } UNot = \text{Boolean}$

primrec *wt-unop* :: *unop* \Rightarrow *ty* \Rightarrow *bool*
where
 $\text{wt-unop } UPlus \ t = (t = \text{PrimT Integer})$
 $\text{wt-unop } UMinus \ t = (t = \text{PrimT Integer})$
 $\text{wt-unop } UBitNot \ t = (t = \text{PrimT Integer})$
 $\text{wt-unop } UNot \ t = (t = \text{PrimT Boolean})$

Typing for binary operations

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

primrec *wt-binop* :: *prog* \Rightarrow *binop* \Rightarrow *ty* \Rightarrow *ty* \Rightarrow *bool*
where
 $\text{wt-binop } G \ Mul \ t1 \ t2 = ((t1 = \text{PrimT Integer}) \wedge (t2 = \text{PrimT Integer}))$
 $\text{wt-binop } G \ Div \ t1 \ t2 = ((t1 = \text{PrimT Integer}) \wedge (t2 = \text{PrimT Integer}))$
 $\text{wt-binop } G \ Mod \ t1 \ t2 = ((t1 = \text{PrimT Integer}) \wedge (t2 = \text{PrimT Integer}))$
 $\text{wt-binop } G \ Plus \ t1 \ t2 = ((t1 = \text{PrimT Integer}) \wedge (t2 = \text{PrimT Integer}))$

```

| 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

$(type)\ tys \leq (type)\ ty + ty\ list$

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

where

```

E, dt ⊢ s :: √ ≡ E, dt ⊢ In1 r s :: Inl (PrimT Void)
| E, dt ⊢ e :: − T ≡ E, dt ⊢ In1 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 ⊢ e :: √ ⟹ E, dt ⊢ l · c :: √

| Comp: ⊢ E, dt ⊢ c1 :: √; E, dt ⊢ c2 :: √ ⟹ E, dt ⊢ c1 ;; c2 :: √
— cf. 14.8
| If: ⊢ E, dt ⊢ e :: − PrimT Boolean; E, dt ⊢ c1 :: √; E, dt ⊢ c2 :: √ ⟹ E, dt ⊢ If(e) c1 Else c2 :: √

```

- cf. 14.10
- | *Loop*: $\llbracket E, dt \models e :: - \text{PrimT Boolean};$
 $E, dt \models c :: \sqrt{} \rrbracket \implies$
 $E, dt \models l \cdot \text{While}(e) \ c :: \sqrt{}$
- cf. 14.13, 14.15, 14.16
- | *Jmp*: $E, dt \models \text{Jmp jump} :: \sqrt{}$
- cf. 14.16
- | *Throw*: $\llbracket E, dt \models e :: - \text{Class tn};$
 $\text{prg } E \vdash \text{tn} \preceq_C \text{SXcpt Throwable} \rrbracket \implies$
 $E, dt \models \text{Throw } e :: \sqrt{}$
- cf. 14.18
- | *Try*: $\llbracket E, dt \models c1 :: \sqrt{}; \text{prg } E \vdash \text{tn} \preceq_C \text{SXcpt Throwable};$
 $\text{lcl } E \ (V\text{Name } vn) = \text{None}; E \ (\text{lcl } E) \ (V\text{Name } vn \mapsto \text{Class } tn) \rrbracket, dt \models c2 :: \sqrt{} \rrbracket$
 \implies
 $E, dt \models \text{Try } c1 \ \text{Catch}(tn \ vn) \ c2 :: \sqrt{}$
- cf. 14.18
- | *Fin*: $\llbracket E, dt \models c1 :: \sqrt{}; E, dt \models c2 :: \sqrt{} \rrbracket \implies$
 $E, dt \models c1 \ \text{Finally } c2 :: \sqrt{}$
- | *Init*: $\llbracket \text{is-class } (\text{prg } E) \ C \rrbracket \implies$
 $E, dt \models \text{Init } C :: \sqrt{}$
 — *Init* is created on the fly during evaluation (see Eval.thy). The class isn't necessarily accessible from the points *Init* is called. Therefor we only demand *is-class* and not *is-acc-class* here.
- well-typed expressions
- cf. 15.8
- | *NewC*: $\llbracket \text{is-acc-class } (\text{prg } E) \ (\text{pkg } E) \ C \rrbracket \implies$
 $E, dt \models \text{NewC } C :: - \text{Class } C$
- cf. 15.9
- | *NewA*: $\llbracket \text{is-acc-type } (\text{prg } E) \ (\text{pkg } E) \ T;$
 $E, dt \models i :: - \text{PrimT Integer} \rrbracket \implies$
 $E, dt \models \text{New } T[i] :: - T.$
- cf. 15.15
- | *Cast*: $\llbracket E, dt \models e :: - T; \text{is-acc-type } (\text{prg } E) \ (\text{pkg } E) \ T';$
 $\text{prg } E \vdash T \preceq^? T' \rrbracket \implies$
 $E, dt \models \text{Cast } T' \ e :: - T'$
- cf. 15.19.2
- | *Inst*: $\llbracket E, dt \models e :: - \text{RefT } T; \text{is-acc-type } (\text{prg } E) \ (\text{pkg } E) \ (\text{RefT } T');$
 $\text{prg } E \vdash \text{RefT } T \preceq^? \text{RefT } T' \rrbracket \implies$
 $E, dt \models e \ \text{InstOf } T' :: - \text{PrimT Boolean}$
- cf. 15.7.1
- | *Lit*: $\llbracket \text{typeof } dt \ x = \text{Some } T \rrbracket \implies$
 $E, dt \models \text{Lit } x :: - T$
- | *UnOp*: $\llbracket E, dt \models e :: - T_e; \text{wt-unop unop } T_e; T = \text{PrimT } (\text{unop-type unop}) \rrbracket$
 \implies
 $E, dt \models \text{UnOp unop } e :: - T$
- | *BinOp*: $\llbracket E, dt \models e1 :: - T1; E, dt \models e2 :: - T2; \text{wt-binop } (\text{prg } E) \ \text{binop } T1 \ T2;$
 $T = \text{PrimT } (\text{binop-type binop}) \rrbracket$
 \implies
 $E, dt \models \text{BinOp binop } e1 \ e2 :: - T$

- cf. 15.10.2, 15.11.1
- | *Super*: $\llbracket \text{lcl } E \text{ This} = \text{Some } (\text{Class } C); C \neq \text{Object};$
 $\text{class } (\text{prg } E) C = \text{Some } c \rrbracket \implies$
 $E, dt \models \text{Super}::-\text{Class } (\text{super } c)$
- cf. 15.13.1, 15.10.1, 15.12
- | *Acc*: $\llbracket E, dt \models \text{va}::=T \rrbracket \implies$
 $E, dt \models \text{Acc } \text{va}::-T$
- cf. 15.25, 15.25.1
- | *Ass*: $\llbracket E, dt \models \text{va}::=T; \text{va} \neq \text{LVar This};$
 $E, dt \models v::-T';$
 $\text{prg } E \vdash T' \preceq T \rrbracket \implies$
 $E, dt \models \text{va}:=v::-T'$
- cf. 15.24
- | *Cond*: $\llbracket E, dt \models e0::-\text{PrimT Boolean};$
 $E, dt \models e1::-T1; E, dt \models e2::-T2;$
 $\text{prg } E \vdash T1 \preceq T2 \wedge T = T2 \vee \text{prg } E \vdash T2 \preceq T1 \wedge T = T1 \rrbracket \implies$
 $E, dt \models e0 \text{ ? } e1 : e2::-T$
- cf. 15.11.1, 15.11.2, 15.11.3
- | *Call*: $\llbracket E, dt \models e::-\text{RefT statT};$
 $E, dt \models \text{ps}::=pTs;$
 $\text{max-spec } (\text{prg } E) (\text{cls } E) \text{ statT } (\text{name}=\text{mn}, \text{parTs}=pTs)$
 $= \{((\text{statDeclT}, m), pTs')\}$
 $\rrbracket \implies$
 $E, dt \models \{\text{cls } E, \text{statT}, \text{invmode } m \text{ e}\} \text{e.mn}(\{pTs'\}ps)::-(\text{resTy } m)$
- | *Methd*: $\llbracket \text{is-class } (\text{prg } E) C;$
 $\text{methd } (\text{prg } E) C \text{ sig} = \text{Some } m;$
 $E, dt \models \text{Body } (\text{declclass } m) (\text{stmt } (\text{mbody } (\text{methd } m)))::-T \rrbracket \implies$
 $E, dt \models \text{Methd } C \text{ sig}::-T$
- The class C is the dynamic class of the method call (cf. Eval.thy). It hasn't got to be directly accessible from the current package $\text{pkg } E$. Only the static class must be accessible (enshured indirectly by *Call*). Note that l is just a dummy value. It is only used in the smallstep semantics. To proof typesafety directly for the smallstep semantics we would have to assume conformance of l here!
- | *Body*: $\llbracket \text{is-class } (\text{prg } E) D;$
 $E, dt \models \text{blk}::\checkmark;$
 $(\text{lcl } E) \text{ Result} = \text{Some } T;$
 $\text{is-type } (\text{prg } E) T \rrbracket \implies$
 $E, dt \models \text{Body } D \text{ blk}::-T$
- The class D implementing the method must not directly be accessible from the current package $\text{pkg } E$, but can also be indirectly accessible due to inheritance (enshured in *Call*) The result type hasn't got to be accessible in Java! (If it is not accessible you can only assign it to Object). For dummy value l see rule *Methd*.
- well-typed variables
- cf. 15.13.1
- | *LVar*: $\llbracket \text{lcl } E \text{ vn} = \text{Some } T; \text{is-acc-type } (\text{prg } E) (\text{pkg } E) T \rrbracket \implies$
 $E, dt \models \text{LVar } \text{vn}::=T$
- cf. 15.10.1
- | *FVar*: $\llbracket E, dt \models e::-\text{Class } C;$
 $\text{accfield } (\text{prg } E) (\text{cls } E) C \text{ fn} = \text{Some } (\text{statDeclC}, f) \rrbracket \implies$
 $E, dt \models \{\text{cls } E, \text{statDeclC}, \text{is-static } f\} \text{e..fn}::=(\text{type } f)$
- cf. 15.12
- | *AVar*: $\llbracket E, dt \models e::-T.[];$
 $E, dt \models i::-\text{PrimT Integer} \rrbracket \implies$

$$E, dt \models e.[i] :: T$$

— well-typed expression lists

— cf. 15.11.???

$$| \text{Nil:} \quad E, dt \models [] :: \bot$$

— cf. 15.11.???

$$| \text{Cons: } \llbracket E, dt \models e :: T; \\ E, dt \models es :: Ts \rrbracket \implies \\ E, dt \models e \# es :: T \# Ts$$

abbreviation

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

abbreviation

$$\text{wt-stmt-syntax} :: env \Rightarrow stmt \Rightarrow bool \ (\langle \vdash :: \rangle \ [51, 51] \ 50) \\ \text{where } E \vdash s :: \surd == E \vdash \text{In1r } s :: \text{In1 } (PrimT \ Void)$$

abbreviation

$$\text{ty-expr-syntax} :: env \Rightarrow [expr, ty] \Rightarrow bool \ (\langle \vdash :: \rangle \ [51, 51, 51] \ 50) \\ \text{where } E \vdash e :: T == E \vdash \text{In1l } e :: \text{Inl } T$$

abbreviation

$$\text{ty-var-syntax} :: env \Rightarrow [var, ty] \Rightarrow bool \ (\langle \vdash :: \rangle \ [51, 51, 51] \ 50) \\ \text{where } E \vdash e :: T == E \vdash \text{In2 } e :: \text{Inl } T$$

abbreviation

$$\text{ty-exprs-syntax} :: env \Rightarrow [expr \ list, ty \ list] \Rightarrow bool \ (\langle \vdash :: \rangle \ [51, 51, 51] \ 50) \\ \text{where } E \vdash e :: T == E \vdash \text{In3 } e :: \text{Inr } T$$

notation (ASCII)

$$\text{wt-syntax} \ (\langle \vdash :: \rangle \ [51, 51, 51] \ 50) \text{ and} \\ \text{wt-stmt-syntax} \ (\langle \vdash :: \rangle \ [51, 51] \ 50) \text{ and} \\ \text{ty-expr-syntax} \ (\langle \vdash :: \rangle \ [51, 51, 51] \ 50) \text{ and} \\ \text{ty-var-syntax} \ (\langle \vdash :: \rangle \ [51, 51, 51] \ 50) \text{ and} \\ \text{ty-exprs-syntax} \ (\langle \vdash :: \rangle \ [51, 51, 51] \ 50)$$

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 $\langle \text{map-theory-simpset } (fn \text{ ctxt } => \text{ ctxt } |> \text{Simplifier.del-loop split-all-tac}) \rangle$

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

$$\begin{aligned} E, dt \models \text{In2 } (LVar \ vn) &:: T \\ E, dt \models \text{In2 } (\{accC, statDeclC, s\}e..fn) &:: T \\ E, dt \models \text{In2 } (e.[i]) &:: T \\ E, dt \models \text{In1l } (NewC \ C) &:: T \\ E, dt \models \text{In1l } (New \ T'[i]) &:: T \\ E, dt \models \text{In1l } (Cast \ T' \ e) &:: T \\ E, dt \models \text{In1l } (e \ \text{InstOf } T') &:: T \\ E, dt \models \text{In1l } (Lit \ x) &:: T \\ E, dt \models \text{In1l } (UnOp \ unop \ e) &:: T \\ E, dt \models \text{In1l } (BinOp \ binop \ e1 \ e2) &:: T \\ E, dt \models \text{In1l } (Super) &:: T \end{aligned}$$

```

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 |> Simplifier.add-loop (split-all-tac, split-all-tac))>

lemma is-acc-class-is-accessible:

$is-acc-class\ G\ P\ C \implies G \vdash (Class\ C)\ accessible-in\ P$

by (auto simp add: is-acc-class-def)

lemma is-acc-iface-is-iface: $is-acc-iface\ G\ P\ I \implies is-iface\ G\ I$

by (auto simp add: is-acc-iface-def)

lemma is-acc-iface-Iface-is-accessible:

$is-acc-iface\ G\ P\ I \implies G \vdash (Iface\ I)\ accessible-in\ P$

by (auto simp add: is-acc-iface-def)

lemma is-acc-type-is-type: $is-acc-type\ G\ P\ T \implies is-type\ G\ T$

by (auto simp add: is-acc-type-def)

lemma is-acc-iface-is-accessible:

$is-acc-type\ G\ P\ T \implies G \vdash T\ accessible-in\ P$

by (auto simp add: is-acc-type-def)

lemma wt-Methd-is-methd:

$E \vdash In1l\ (Methd\ C\ sig)::T \implies is-methd\ (prg\ E)\ C\ sig$

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:

$\llbracket E,dt|=e::-RefT\ statT; E,dt|=ps::\dot{=}pTs;$

$max-spec\ (prg\ E)\ (cls\ E)\ statT\ (\langle name=mn,parTs=pTs \rangle)$

$= \{((statDeclC,m),pTs')\}; rT=(resTy\ m); accC=cls\ E;$

$mode = invmode\ m\ e \rrbracket \implies E,dt|=\{accC,statT,mode\}e.mn(\{pTs'\}ps)::-rT$

by (auto elim: wt.Call)

lemma *invocationTypeExpr-noClassD*:

$\llbracket E \vdash e :: - \text{RefT } \text{statT} \rrbracket$
 $\implies (\forall \text{ statC}. \text{statT} \neq \text{ClassT } \text{statC}) \longrightarrow \text{invmode } m \ e \neq \text{SuperM}$

proof –

assume *wt*: $E \vdash e :: - \text{RefT } \text{statT}$

show *?thesis*

proof (*cases* $e = \text{Super}$)

case *True*

with *wt* **obtain** *C* **where** $\text{statT} = \text{ClassT } C$ **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*:

$\llbracket \text{lcl } E \text{ This} = \text{Some } (\text{Class } C); C \neq \text{Object}; \text{class } (\text{prg } E) \ C = \text{Some } c; D = \text{super } c \rrbracket$

$\implies E, dt \models \text{Super} :: - \text{Class } D$

by (*auto elim*: *wt.Super*)

lemma *wt-FVar*:

$\llbracket E, dt \models e :: - \text{Class } C; \text{accfield } (\text{prg } E) \ (\text{cls } E) \ C \ \text{fn} = \text{Some } (\text{statDeclC}, f);$

$\text{sf} = \text{is-static } f; fT = (\text{type } f); \text{accC} = \text{cls } E \rrbracket$

$\implies E, dt \models \{ \text{accC}, \text{statDeclC}, \text{sf} \} e.. \text{fn} :: = fT$

by (*auto dest*: *wt.FVar*)

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

by (*auto elim*: *wt-elim-cases intro*: *wt.Init*)

declare *wt.Skip* [*iff*]

lemma *wt-StatRef*:

$\text{is-acc-type } (\text{prg } E) \ (\text{pkg } E) \ (\text{RefT } rt) \implies E \vdash \text{StatRef } rt :: - \text{RefT } rt$

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*:

$\bigwedge E. E, dt \models t :: U \implies \text{case } t \text{ of}$

$\text{In1 } ec \Rightarrow (\text{case } ec \text{ of}$

$\text{Inl } e \Rightarrow \exists T. U = \text{Inl } T$

$\mid \text{Inr } s \Rightarrow U = \text{Inl } (\text{PrimT } \text{Void}))$

$\mid \text{In2 } e \Rightarrow (\exists T. U = \text{Inl } T)$

$\mid \text{In3 } e \Rightarrow (\exists T. U = \text{Inr } T)$

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*: $E, dt \models \text{In1l } t :: U = (\exists T. U = \text{Inl } T \wedge E, dt \models t :: - T)$
by (*auto*, *frule wt-Inj-elim*, *auto*)

lemma *wt-var-eq*: $E, dt \models \text{In2 } t :: U = (\exists T. U = \text{Inl } T \wedge E, dt \models t :: T)$
by (*auto*, *frule wt-Inj-elim*, *auto*)

lemma *wt-exprs-eq*: $E, dt \models \text{In3 } t :: U = (\exists Ts. U = \text{Inr } Ts \wedge E, dt \models t :: Ts)$
by (*auto*, *frule wt-Inj-elim*, *auto*)

lemma *wt-stmt-eq*: $E, dt \models \text{In1r } t :: U = (U = \text{Inl}(\text{PrimT } \text{Void}) \wedge E, dt \models t :: \checkmark)$
by (*auto*, *frule wt-Inj-elim*, *auto*, *frule wt-Inj-elim*, *auto*)

simproc-setup *wt-expr* ($E, dt \models \text{In1l } t :: U$) = \langle
 $K (K (fn ct =>$
 $(case Thm.term-of ct of$
 $(- \$ - \$ - \$ - \$ (Const - \$ -)) => NONE$
 $| - => SOME (mk-meta-eq @\{thm wt-expr-eq\})))) \rangle$

simproc-setup *wt-var* ($E, dt \models \text{In2 } t :: U$) = \langle
 $K (K (fn ct =>$
 $(case Thm.term-of ct of$
 $(- \$ - \$ - \$ - \$ (Const - \$ -)) => NONE$
 $| - => SOME (mk-meta-eq @\{thm wt-var-eq\})))) \rangle$

simproc-setup *wt-exprs* ($E, dt \models \text{In3 } t :: U$) = \langle
 $K (K (fn ct =>$
 $(case Thm.term-of ct of$
 $(- \$ - \$ - \$ - \$ (Const - \$ -)) => NONE$
 $| - => SOME (mk-meta-eq @\{thm wt-exprs-eq\})))) \rangle$

simproc-setup *wt-stmt* ($E, dt \models \text{In1r } t :: U$) = \langle
 $K (K (fn ct =>$
 $(case Thm.term-of ct of$
 $(- \$ - \$ - \$ - \$ (Const - \$ -)) => NONE$
 $| - => SOME (mk-meta-eq @\{thm wt-stmt-eq\})))) \rangle$

lemma *wt-elim-BinOp*:
 $\llbracket E, dt \models \text{In1l } (\text{BinOp } binop \ e1 \ e2) :: T;$
 $\bigwedge T1 \ T2 \ T3.$
 $\llbracket E, dt \models e1 :: - T1; E, dt \models e2 :: - T2; wt-binop (prg \ E) \ binop \ T1 \ T2;$
 $E, dt \models (if \ b \ then \ \text{In1l } e2 \ else \ \text{In1r } Skip) :: T3;$
 $T = \text{Inl } (\text{PrimT } (\text{binop-type } binop)) \rrbracket$
 $\implies P \rrbracket$
 $\implies P$
apply (*erule wt-elim-cases*)
apply (*cases b*)
apply *auto*
done

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

lemma *single-valued-tys-lemma* [*rule-format (no-asm)*]:
 $\forall S \ T. G \vdash S \preceq T \longrightarrow G \vdash T \preceq S \longrightarrow S = T \implies E, dt \models t :: T \implies$
 $G = prg \ E \longrightarrow (\forall T'. E, dt \models t :: T' \longrightarrow T = T')$
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* $\langle ALLGOALS (fn\ i =>$
 if *i* = 11 *then EVERY'*
 [*Rule-Insts.thin-tac context* *E,dt* $\models e0::\neg PrimT\ Boolean\ [(\mathbf{binding}\ \langle E \rangle, NONE, NoSyn)],$
 Rule-Insts.thin-tac context *E,dt* $\models e1::\neg T1\ [(\mathbf{binding}\ \langle E \rangle, NONE, NoSyn), (\mathbf{binding}\ \langle T1 \rangle, NONE,$
 NoSyn)],
 Rule-Insts.thin-tac context *E,dt* $\models e2::\neg T2\ [(\mathbf{binding}\ \langle E \rangle, NONE, NoSyn), (\mathbf{binding}\ \langle T2 \rangle, NONE,$
 NoSyn)]] *i*
 else Rule-Insts.thin-tac context All P $[(\mathbf{binding}\ \langle P \rangle, NONE, NoSyn)]\ i\rangle$)

apply (*tactic* $\langle ALLGOALS (eresolve-tac\ \mathbf{context}\ @\{\mathit{thms}\ wt\ elim\ cases\})\rangle$)
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*) $\implies single-valued\ \{(t,T). E,dt \models t::T\}$
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* $(\lambda a. None)\ v = Some\ T \implies is-type\ G\ T$
by (*induct v*) *auto*

lemma *typeof-is-type*: $(\forall a. v \neq Addr\ a) \implies \exists T. typeof\ dt\ v = Some\ T \wedge is-type\ G\ T$
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 abruptio (exception, break, continue, return) appeared) , the set of local variables calculated by the analysis is a subset of the variables that were actually assigned during evaluation.

To get more precise information about the sets of assigned variables the analysis includes the following optimisations:

- Inside of a while loop we also take care of the variables assigned before break statements, since the break causes the while loop to continue normally.
- For conditional statements we take care of constant conditions to statically determine the path of evaluation.
- Inside a distinct path of a conditional statements we know to which boolean value the condition has evaluated to, and so can retrieve more information about the variables assigned during evaluation of the boolean condition.

Since in our model of Java the return values of methods are stored in a local variable we also ensure that every path of (normal) evaluation will assign the result variable, or in the sense of real Java every path ends up in and return instruction.

Not covered yet:

- analysis of definite unassigned
- special treatment of final fields

Correct nesting of jump statements

For definite assignment it becomes crucial, that jumps (break, continue, return) are nested correctly i.e. a continue jump is nested in a matching while statement, a break jump is nested in a proper label statement, a class initialiser does not terminate abruptly with a return. With this we can for example ensure that evaluation of an expression will never end up with a jump, since no breaks, continues or returns are allowed in an expression.

primrec *jumpNestingOkS* :: *jump set* \Rightarrow *stmt* \Rightarrow *bool*

where

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

definition $\text{jumpNestingOk} :: \text{jump set} \Rightarrow \text{term} \Rightarrow \text{bool}$ where

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

lemma $\text{jumpNestingOk-expr-simp}$ [simp]: $\text{jumpNestingOk } \text{jmps } (\text{In1l } e) = \text{True}$
by (simp add: jumpNestingOk-def)

lemma $\text{jumpNestingOk-expr-simp1}$ [simp]: $\text{jumpNestingOk } \text{jmps } \langle e :: \text{expr} \rangle = \text{True}$
by (simp add: inj-term-simps)

lemma $\text{jumpNestingOk-stmt-simp}$ [simp]:
 $\text{jumpNestingOk } \text{jmps } (\text{In1r } s) = \text{jumpNestingOkS } \text{jmps } s$
by (simp add: jumpNestingOk-def)

lemma $\text{jumpNestingOk-stmt-simp1}$ [simp]:
 $\text{jumpNestingOk } \text{jmps } \langle s :: \text{stmt} \rangle = \text{jumpNestingOkS } \text{jmps } s$
by (simp add: inj-term-simps)

lemma $\text{jumpNestingOk-var-simp}$ [simp]: $\text{jumpNestingOk } \text{jmps } (\text{In2 } v) = \text{True}$
by (simp add: jumpNestingOk-def)

lemma $\text{jumpNestingOk-var-simp1}$ [simp]: $\text{jumpNestingOk } \text{jmps } \langle v :: \text{var} \rangle = \text{True}$
by (simp add: inj-term-simps)

lemma $\text{jumpNestingOk-expr-list-simp}$ [simp]: $\text{jumpNestingOk } \text{jmps } (\text{In3 } es) = \text{True}$
by (simp add: jumpNestingOk-def)

lemma $\text{jumpNestingOk-expr-list-simp1}$ [simp]:
 $\text{jumpNestingOk } \text{jmps } \langle es :: \text{expr list} \rangle = \text{True}$
by (simp add: inj-term-simps)

Calculation of assigned variables for boolean expressions

2 Very restricted calculation fallback calculation

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

primrec *assignsE* :: *expr* \Rightarrow *lname set*
and *assignsV* :: *var* \Rightarrow *lname set*
and *assignsEs* :: *expr list* \Rightarrow *lname set*

where

assignsE (*NewC* *c*) = {}
| *assignsE* (*NewA* *t e*) = *assignsE* *e*
| *assignsE* (*Cast* *t e*) = *assignsE* *e*
| *assignsE* (*e InstOf* *r*) = *assignsE* *e*
| *assignsE* (*Lit* *val*) = {}
| *assignsE* (*UnOp* *unop e*) = *assignsE* *e*
| *assignsE* (*BinOp* *binop e1 e2*) = (if *binop*=*CondAnd* \vee *binop*=*CondOr*
then (*assignsE* *e1*)
else (*assignsE* *e1*) \cup (*assignsE* *e2*))
| *assignsE* (*Super*) = {}
| *assignsE* (*Acc* *v*) = *assignsV* *v*
| *assignsE* (*v:=e*) = (*assignsV* *v*) \cup (*assignsE* *e*) \cup
(if \exists *n*. *v*=(*LVar* *n*) then {*the-LVar-name* *v*}
else {})
| *assignsE* (*b? e1 : e2*) = (*assignsE* *b*) \cup ((*assignsE* *e1*) \cap (*assignsE* *e2*))
| *assignsE* ({*accC*,*statT*,*mode*}*objRef*.*mn*({*pTs*}*args*))
= (*assignsE* *objRef*) \cup (*assignsEs* *args*)

— Only dummy analysis for intermediate expressions *Method*, *Body*, *InsInitE* and *Callee*

| *assignsE* (*Method* *C sig*) = {}
| *assignsE* (*Body* *C s*) = {}
| *assignsE* (*InsInitE* *s e*) = {}
| *assignsE* (*Callee* *l e*) = {}

| *assignsV* (*LVar* *n*) = {}
| *assignsV* ({*accC*,*statDeclC*,*stat*}*objRef*..*fn*) = *assignsE* *objRef*
| *assignsV* (*e1*.[*e2*]) = *assignsE* *e1* \cup *assignsE* *e2*

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

definition *assigns* :: *term* \Rightarrow *lname set* **where**

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

lemma *assigns-expr-simp* [*simp*]: *assigns* (*In1l* *e*) = *assignsE* *e*
by (*simp* add: *assigns-def*)

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

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 ($\langle v \rangle$) = 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 ($\langle es \rangle$) = assignsEs es
by (simp add: inj-term-simps)

3 Analysis of constant expressions

primrec constVal :: expr \Rightarrow val option

where

```

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

```

— Note that `constVal (Cond b e1 e2)` is stricter as it could be. It requires that all tree expressions are constant even if we can decide which branch to choose, provided the constant value of `b`

```

constVal (Call accC statT mode objRef mn pTs args) = None
| constVal (Methd C sig) = None
| constVal (Body C s)    = None
| constVal (InsInitE s e) = None
| constVal (Callee l e)  = None

```

lemma constVal-Some-induct [consumes 1, case-names Lit UnOp BinOp CondL CondR]:

assumes const: constVal e = Some v **and**

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

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

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

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

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

```

shows P e
proof -
  have  $\bigwedge v. \text{constVal } e = \text{Some } v \implies P \ e$ 
  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:  $\text{constVal } (b ? e1 : e2) = \text{Some } v$ 
      by blast
    then obtain bv where bv:  $\text{constVal } b = \text{Some } bv$ 
      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*: $\text{constVal } e = \text{Some } v \implies \text{assignsE } e = \{\}$
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* \Rightarrow *expr* \Rightarrow *lname set*

where

<i>assigns-if</i> <i>b</i> (<i>NewC</i> <i>c</i>)	= UNIV — can never evaluate to Boolean
<i>assigns-if</i> <i>b</i> (<i>NewA</i> <i>t</i> <i>e</i>)	= UNIV — can never evaluate to Boolean
<i>assigns-if</i> <i>b</i> (<i>Cast</i> <i>t</i> <i>e</i>)	= <i>assigns-if</i> <i>b</i> <i>e</i>
<i>assigns-if</i> <i>b</i> (<i>Inst</i> <i>e</i> <i>r</i>)	= <i>assignsE</i> <i>e</i> — Inst has type Boolean but <i>e</i> is a reference type
<i>assigns-if</i> <i>b</i> (<i>Lit</i> <i>val</i>)	= (if <i>val</i> = <i>Bool</i> <i>b</i> then {} else UNIV)
<i>assigns-if</i> <i>b</i> (<i>UnOp</i> <i>unop</i> <i>e</i>)	= (case <i>constVal</i> (<i>UnOp</i> <i>unop</i> <i>e</i>) of
	<i>None</i> \Rightarrow (if <i>unop</i> = <i>UNot</i>
	then <i>assigns-if</i> (\neg <i>b</i>) <i>e</i>
	else UNIV)
	<i>Some</i> <i>v</i> \Rightarrow (if <i>v</i> = <i>Bool</i> <i>b</i>

```

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

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

proof —

have $\bigwedge b. ?Const\ b\ e \implies ?Ass\ b\ e$

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* = $\langle \bigwedge b. ?Const\ b\ c \implies ?Ass\ b\ c \rangle$

note *hyp-e1* = $\langle \bigwedge b. ?Const\ b\ e1 \implies ?Ass\ b\ e1 \rangle$

note *hyp-e2* = $\langle \bigwedge b. ?Const\ b\ e2 \implies ?Ass\ b\ e2 \rangle$

note *const* = $\langle \text{constVal } (c\ ?\ e1 : e2) = \text{Some } (\text{Bool } b) \rangle$

then obtain *bv* **where** *bv*: $\text{constVal } c = \text{Some } bv$

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 ( $\neg$ b) e = UNIV                  (is ?Ass b e)
proof –
  have  $\bigwedge b. ?\text{Const } b \ e \implies ?\text{Ass } b \ e$ 
  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 =  $\langle \bigwedge b. ?\text{Const } b \ c \implies ?\text{Ass } b \ c \rangle$ 
    note hyp-e1 =  $\langle \bigwedge b. ?\text{Const } b \ e1 \implies ?\text{Ass } b \ e1 \rangle$ 
    note hyp-e2 =  $\langle \bigwedge b. ?\text{Const } b \ e2 \implies ?\text{Ass } b \ e2 \rangle$ 
    note const =  $\langle \text{constVal } (c \ ? \ e1 : e2) = \text{Some } (\text{Bool } b) \rangle$ 
    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  $\Rightarrow$  ('a,'b) tables  $\Rightarrow$  ('a,'b) tables ( $\langle - \Rightarrow \cup \rightarrow$  [67,67] 65)
where  $A \Rightarrow \cup B = (\lambda k. A\ k \cup B\ k)$ 

```

definition

```

intersect-ts :: ('a,'b) tables  $\Rightarrow$  ('a,'b) tables  $\Rightarrow$  ('a,'b) tables ( $\langle - \Rightarrow \cap \rightarrow$  [72,72] 71)
where  $A \Rightarrow \cap B = (\lambda k. A\ k \cap B\ k)$ 

```

definition

```

all-union-ts :: ('a,'b) tables  $\Rightarrow$  'b set  $\Rightarrow$  ('a,'b) tables (infixl  $\langle \Rightarrow \cup_{\forall} \rangle$  40)
where  $(A \Rightarrow \cup_{\forall} B) = (\lambda k. A\ k \cup B)$ 

```

Binary union of tables

```

lemma union-ts-iff [simp]:  $(c \in (A \Rightarrow \cup B)\ k) = (c \in A\ k \vee c \in B\ k)$ 
by (unfold union-ts-def) blast

```

```

lemma union-tsI1 [elim?]:  $c \in A\ k \Longrightarrow c \in (A \Rightarrow \cup B)\ k$ 
by simp

```

```

lemma union-tsI2 [elim?]:  $c \in B\ k \Longrightarrow c \in (A \Rightarrow \cup B)\ k$ 
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!]:
 $\llbracket c \in (A \Rightarrow \cup B)\ k; (c \in A\ k \Longrightarrow P); (c \in B\ k \Longrightarrow P) \rrbracket \Longrightarrow P$ 
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!]:  $\llbracket c \in A\ k; c \in B\ k \rrbracket \Longrightarrow c \in (A \Rightarrow \cap B)\ k$ 
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!]:
 $\llbracket c \in (A \Rightarrow_{\cap} B) \ k; \llbracket c \in A \ k; c \in B \ k \rrbracket \Longrightarrow P \rrbracket \Longrightarrow P$
by *simp*

All-Union of tables and set

lemma *all-union-ts-iff* [simp]: $(c \in (A \Rightarrow_{\cup} B) \ k) = (c \in A \ k \vee c \in B)$
by (*unfold all-union-ts-def*) *blast*

lemma *all-union-tsI1* [elim?]: $c \in A \ k \Longrightarrow c \in (A \Rightarrow_{\cup} B) \ k$
by *simp*

lemma *all-union-tsI2* [elim?]: $c \in B \Longrightarrow c \in (A \Rightarrow_{\cup} B) \ k$
by *simp*

lemma *all-union-tsCI* [intro!]: $(c \notin B \Longrightarrow c \in A \ k) \Longrightarrow c \in (A \Rightarrow_{\cup} B) \ k$
by *auto*

lemma *all-union-tsE* [elim!]:
 $\llbracket c \in (A \Rightarrow_{\cup} B) \ k; (c \in A \ k \Longrightarrow P); (c \in B \Longrightarrow P) \rrbracket \Longrightarrow P$
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* = ($\lambda x. \text{if } x=k \text{ then } UNIV \text{ else } A$)

definition

range-inter-ts :: ('*a*, '*b*) *tables* \Rightarrow '*b set* ($\hookRightarrow \bigcap \rightarrow$ 80)
where $\hookRightarrow \bigcap A = \{x \mid x. \forall k. x \in A \ k\}$

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

inductive

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

where

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

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

\Longrightarrow

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

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

\Longrightarrow

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

| *Comp*: $\llbracket Env \vdash B \gg \langle c1 \rangle \gg C1; Env \vdash \text{nrm } C1 \gg \langle c2 \rangle \gg C2; \rrbracket$

$$\begin{aligned}
& nrm\ A = nrm\ C2; \text{brk}\ A = (\text{brk}\ C1) \Rightarrow \cap (\text{brk}\ C2) \\
& \implies \\
& Env \vdash B \gg \langle c1;; c2 \rangle \gg A
\end{aligned}$$

$$\begin{aligned}
| \text{ If: } & \llbracket Env \vdash B \gg \langle e \rangle \gg E; \\
& Env \vdash (B \cup \text{assigns-if True } e) \gg \langle c1 \rangle \gg C1; \\
& Env \vdash (B \cup \text{assigns-if False } e) \gg \langle c2 \rangle \gg C2; \\
& nrm\ A = nrm\ C1 \cap nrm\ C2; \\
& \text{brk}\ A = \text{brk}\ C1 \Rightarrow \cap \text{brk}\ C2 \rrbracket \\
& \implies \\
& Env \vdash B \gg \langle \text{If}(e)\ c1\ \text{Else}\ c2 \rangle \gg A
\end{aligned}$$

— Note that E is not further used, because we take the specialized sets that also consider if the expression evaluates to true or false. Inside of e there is no **break** or **finally**, so the break map of E will be the trivial one. So $Env \vdash B \gg \langle e \rangle \gg E$ is just used to ensure the definite assignment in expression e . Notice the implicit analysis of a constant boolean expression e in this rule. For example, if e is constantly *True* then *assigns-if False* $e = UNIV$ and therefor $nrm\ C2 = UNIV$. So finally $nrm\ A = nrm\ C1$. For the break maps this trick workd too, because the trivial break map will map all labels to *UNIV*. In the example, if no break occurs in $c2$ the break maps will trivially map to *UNIV* and if a break occurs it will map to *UNIV* too, because *assigns-if False* $e = UNIV$. So in the intersection of the break maps the path $c2$ will have no contribution.

$$\begin{aligned}
| \text{ Loop: } & \llbracket Env \vdash B \gg \langle e \rangle \gg E; \\
& Env \vdash (B \cup \text{assigns-if True } e) \gg \langle c \rangle \gg C; \\
& nrm\ A = nrm\ C \cap (B \cup \text{assigns-if False } e); \\
& \text{brk}\ A = \text{brk}\ C \rrbracket \\
& \implies \\
& Env \vdash B \gg \langle l \cdot \text{While}(e)\ c \rangle \gg A
\end{aligned}$$

— The *Loop* rule resembles some of the ideas of the *If* rule. For the $nrm\ A$ the set $B \cup \text{assigns-if False } e$ will be *UNIV* if the condition is constantly true. To normally exit the while loop, we must consider the body c to be completed normally ($nrm\ C$) or with a break. But in this model, the label l of the loop only handles continue labels, not break labels. The break label will be handled by an enclosing *Lab* statement. So we don't have to handle the breaks specially.

$$\begin{aligned}
| \text{ Jmp: } & \llbracket \text{jump} = \text{Ret} \longrightarrow \text{Result} \in B; \\
& nrm\ A = UNIV; \\
& \text{brk}\ A = (\text{case jump of} \\
& \quad \text{Break } l \Rightarrow \lambda k. \text{ if } k=l \text{ then } B \text{ else } UNIV \\
& \quad | \text{Cont } l \Rightarrow \lambda k. UNIV \\
& \quad | \text{Ret} \Rightarrow \lambda k. UNIV) \rrbracket \\
& \implies \\
& Env \vdash B \gg \langle \text{Jmp jump} \rangle \gg A
\end{aligned}$$

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

$$\begin{aligned}
| \text{ Throw: } & \llbracket Env \vdash B \gg \langle e \rangle \gg E; nrm\ A = UNIV; \text{brk}\ A = (\lambda l. UNIV) \rrbracket \\
& \implies Env \vdash B \gg \langle \text{Throw } e \rangle \gg A
\end{aligned}$$

$$\begin{aligned}
| \text{ Try: } & \llbracket Env \vdash B \gg \langle c1 \rangle \gg C1; \\
& Env(\text{lcl} := (\text{lcl } Env)(VName\ vn \mapsto \text{Class } C)) \vdash (B \cup \{VName\ vn\}) \gg \langle c2 \rangle \gg C2; \\
& nrm\ A = nrm\ C1 \cap nrm\ C2; \\
& \text{brk}\ A = \text{brk}\ C1 \Rightarrow \cap \text{brk}\ C2 \rrbracket \\
& \implies Env \vdash B \gg \langle \text{Try } c1\ \text{Catch}(C\ vn)\ c2 \rangle \gg A
\end{aligned}$$

$$\begin{aligned}
| \text{ Fin: } & \llbracket Env \vdash B \gg \langle c1 \rangle \gg C1; \\
& Env \vdash B \gg \langle c2 \rangle \gg C2; \\
& nrm\ A = nrm\ C1 \cup nrm\ C2; \\
& \text{brk}\ A = ((\text{brk}\ C1) \Rightarrow \cup_{\forall} (nrm\ C2)) \Rightarrow \cap (\text{brk}\ C2) \rrbracket
\end{aligned}$$

$$\implies$$

$$Env \vdash B \gg \langle c1 \text{ Finally } c2 \rangle \gg A$$

— The set of assigned variables before execution $c2$ are the same as before execution $c1$, because $c1$ could throw an exception and so we can't guarantee that any variable will be assigned in $c1$. The *Finally* statement completes normally if both $c1$ and $c2$ complete normally. If $c1$ completes abruptly with a break, then $c2$ also will be executed and may terminate normally or with a break. The overall break map then is the intersection of the maps of both paths. If $c2$ terminates normally we have to extend all break sets in $brk \ C1$ with $nrm \ C2$ ($\Rightarrow \cup$). If $c2$ exits with a break this break will appear in the overall result state. We don't know if $c1$ completed normally or abruptly (maybe with an exception not only a break) so $c1$ has no contribution to the break map following this path.

— Evaluation of expressions and the break sets of definite assignment: Thinking of a Java expression we assume that we can never have a break statement inside of a expression. So for all expressions the break sets could be set to the trivial one: $\lambda l. UNIV$. But we can't trivially proof, that evaluating an expression will never result in a break, although Java expressions already syntactically don't allow nested statements in them. The reason are the nested class initialization statements which are inserted by the evaluation rules. So to proof the absence of a break we need to ensure, that the initialization statements will never end up in a break. In a wellformed initialization statement, of course, where breaks are nested correctly inside of *Lab* or *Loop* statements evaluation of the whole initialization statement will never result in a break, because this break will be handled inside of the statement. But for simplicity we haven't added the analysis of the correct nesting of breaks in the typing judgments right now. So we have decided to adjust the rules of definite assignment to fit to these circumstances. If an initialization is involved during evaluation of the expression (evaluation rules *FVar*, *NewC* and *NewA*

| *Init*: $Env \vdash B \gg \langle \text{Init } C \rangle \gg (\text{nrm}=B, \text{brk}=\lambda l. UNIV)$

— Wellformedness of a program will ensure, that every static initialiser is definitely assigned and the jumps are nested correctly. The case here for *Init* is just for convenience, to get a proper precondition for the induction hypothesis in various proofs, so that we don't have to expand the initialisation on every point where it is triggered by the evaluation rules.

| *NewC*: $Env \vdash B \gg \langle \text{NewC } C \rangle \gg (\text{nrm}=B, \text{brk}=\lambda l. UNIV)$

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

$$\implies$$

$$Env \vdash B \gg \langle \text{New } T[e] \rangle \gg A$$

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

$$\implies$$

$$Env \vdash B \gg \langle \text{Cast } T \ e \rangle \gg A$$

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

$$\implies$$

$$Env \vdash B \gg \langle e \text{ InstOf } T \rangle \gg A$$

| *Lit*: $Env \vdash B \gg \langle \text{Lit } v \rangle \gg (\text{nrm}=B, \text{brk}=\lambda l. UNIV)$

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

$$\implies$$

$$Env \vdash B \gg \langle \text{UnOp } unop \ e \rangle \gg A$$

| *CondAnd*: $\llbracket Env \vdash B \gg \langle e1 \rangle \gg E1; Env \vdash (B \cup \text{assigns-if True } e1) \gg \langle e2 \rangle \gg E2;$
 $nrm \ A = B \cup (\text{assigns-if True } (BinOp \ CondAnd \ e1 \ e2) \cap$

$$\text{assigns-if False } (BinOp \ CondAnd \ e1 \ e2));$$

$$brk \ A = (\lambda l. UNIV) \rrbracket$$

$$\implies$$

$$Env \vdash B \gg \langle BinOp \ CondAnd \ e1 \ e2 \rangle \gg A$$

| *CondOr*: $\llbracket Env \vdash B \gg \langle e1 \rangle \gg E1; Env \vdash (B \cup \text{assigns-if False } e1) \gg \langle e2 \rangle \gg E2;$

$$nrm \ A = B \cup (\text{assigns-if True } (BinOp \ CondOr \ e1 \ e2) \cap$$

$$\text{assigns-if False } (BinOp \ CondOr \ e1 \ e2));$$

- $$\begin{array}{l}
\text{brk } A = (\lambda l. \text{ UNIV}) \mathbb{I} \\
\implies \\
\text{Env} \vdash B \gg \langle \text{BinOp CondOr } e1 \ e2 \rangle \gg A \\
\\
| \text{ BinOp: } \mathbb{I} \text{Env} \vdash B \gg \langle e1 \rangle \gg E1; \text{Env} \vdash \text{nrm } E1 \gg \langle e2 \rangle \gg A; \\
\text{binop} \neq \text{CondAnd}; \text{binop} \neq \text{CondOr} \mathbb{I} \\
\implies \\
\text{Env} \vdash B \gg \langle \text{BinOp binop } e1 \ e2 \rangle \gg A \\
\\
| \text{ Super: } \text{This} \in B \\
\implies \\
\text{Env} \vdash B \gg \langle \text{Super} \rangle \gg (\text{nrm} = B, \text{brk} = \lambda l. \text{ UNIV}) \\
\\
| \text{ AccLVar: } \mathbb{I} vn \in B; \\
\text{nrm } A = B; \text{brk } A = (\lambda k. \text{ UNIV}) \mathbb{I} \\
\implies \\
\text{Env} \vdash B \gg \langle \text{Acc (LVar } vn) \rangle \gg A \\
\\
— \text{ To properly access a local variable we have to test the definite assignment here. The variable must occur in the set } B \\
\\
| \text{ Acc: } \mathbb{I} \forall vn. v \neq \text{LVar } vn; \\
\text{Env} \vdash B \gg \langle v \rangle \gg A \mathbb{I} \\
\implies \\
\text{Env} \vdash B \gg \langle \text{Acc } v \rangle \gg A \\
\\
| \text{ AssLVar: } \mathbb{I} \text{Env} \vdash B \gg \langle e \rangle \gg E; \text{nrm } A = \text{nrm } E \cup \{vn\}; \text{brk } A = \text{brk } E \mathbb{I} \\
\implies \\
\text{Env} \vdash B \gg \langle (\text{LVar } vn) := e \rangle \gg A \\
\\
| \text{ Ass: } \mathbb{I} \forall vn. v \neq \text{LVar } vn; \text{Env} \vdash B \gg \langle v \rangle \gg V; \text{Env} \vdash \text{nrm } V \gg \langle e \rangle \gg A \mathbb{I} \\
\implies \\
\text{Env} \vdash B \gg \langle v := e \rangle \gg A \\
\\
| \text{ CondBool: } \mathbb{I} \text{Env} \vdash (c ? e1 : e2) :: \neg (\text{PrimT Boolean}); \\
\text{Env} \vdash B \gg \langle c \rangle \gg C; \\
\text{Env} \vdash (B \cup \text{assigns-if True } c) \gg \langle e1 \rangle \gg E1; \\
\text{Env} \vdash (B \cup \text{assigns-if False } c) \gg \langle e2 \rangle \gg E2; \\
\text{nrm } A = B \cup (\text{assigns-if True } (c ? e1 : e2) \cap \\
\text{assigns-if False } (c ? e1 : e2)); \\
\text{brk } A = (\lambda l. \text{ UNIV}) \mathbb{I} \\
\implies \\
\text{Env} \vdash B \gg \langle c ? e1 : e2 \rangle \gg A \\
\\
| \text{ Cond: } \mathbb{I} \neg \text{Env} \vdash (c ? e1 : e2) :: \neg (\text{PrimT Boolean}); \\
\text{Env} \vdash B \gg \langle c \rangle \gg C; \\
\text{Env} \vdash (B \cup \text{assigns-if True } c) \gg \langle e1 \rangle \gg E1; \\
\text{Env} \vdash (B \cup \text{assigns-if False } c) \gg \langle e2 \rangle \gg E2; \\
\text{nrm } A = \text{nrm } E1 \cap \text{nrm } E2; \text{brk } A = (\lambda l. \text{ UNIV}) \mathbb{I} \\
\implies \\
\text{Env} \vdash B \gg \langle c ? e1 : e2 \rangle \gg A \\
\\
| \text{ Call: } \mathbb{I} \text{Env} \vdash B \gg \langle e \rangle \gg E; \text{Env} \vdash \text{nrm } E \gg \langle \text{args} \rangle \gg A \mathbb{I} \\
\implies \\
\text{Env} \vdash B \gg \langle \{accC, statT, mode\} e \cdot mn(\{pTs\} args) \rangle \gg A
\end{array}$$

— The interplay of *Call*, *Method* and *Body*: Why rules for *Method* and *Body* at all? Note that a Java source program will not include bare *Method* or *Body* terms. These terms are just introduced during evaluation. So definite assignment of *Call* does not consider *Method* or *Body* at all. So for definite assignment alone we could omit the rules for *Method* and *Body*. But since evaluation of the method invocation is split up into three rules

we must ensure that we have enough information about the call even in the *Body* term to make sure that we can proof type safety. Also we must be able transport this information from *Call* to *Method* and then further to *Body* during evaluation to establish the definite assignment of *Method* during evaluation of *Call*, and of *Body* during evaluation of *Method*. This is necessary since definite assignment will be a precondition for each induction hypothesis coming out of the evaluation rules, and 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.

| *Method*: $\llbracket \text{methd } (\text{prg } \text{Env}) \text{ } D \text{ sig} = \text{Some } m; \text{Env} \vdash B \gg \langle \text{Body } (\text{declclass } m) (\text{stmt } (\text{mbody } (\text{methd } m))) \rangle \gg A \rrbracket$
 \implies
 $\text{Env} \vdash B \gg \langle \text{Method } D \text{ sig} \rangle \gg A$

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

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

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

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

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

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

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

declare *inj-term-sym-simps* [*simp*]
declare *assigns-if.simps* [*simp del*]
declare *split-paired-All* [*simp del*] *split-paired-Ex* [*simp del*]
setup $\langle \text{map-theory-simpset } (fn \text{ } \text{ctxt} \Rightarrow \text{ } \text{ctxt} \mid > \text{Simplifier.del-loop split-all-tac}) \rangle$

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

$\text{Env} \vdash B \gg \langle \text{Skip} \rangle \gg A$
 $\text{Env} \vdash B \gg \text{In1r } \text{Skip} \gg A$
 $\text{Env} \vdash B \gg \langle \text{Expr } e \rangle \gg A$
 $\text{Env} \vdash B \gg \text{In1r } (\text{Expr } e) \gg A$
 $\text{Env} \vdash B \gg \langle l \cdot c \rangle \gg A$
 $\text{Env} \vdash B \gg \text{In1r } (l \cdot c) \gg A$
 $\text{Env} \vdash B \gg \langle c1;; c2 \rangle \gg A$
 $\text{Env} \vdash B \gg \text{In1r } (c1;; c2) \gg A$
 $\text{Env} \vdash B \gg \langle \text{If}(e) \text{ } c1 \text{ Else } c2 \rangle \gg A$
 $\text{Env} \vdash B \gg \text{In1r } (\text{If}(e) \text{ } c1 \text{ Else } c2) \gg A$
 $\text{Env} \vdash B \gg \langle l \cdot \text{While}(e) \text{ } c \rangle \gg A$
 $\text{Env} \vdash B \gg \text{In1r } (l \cdot \text{While}(e) \text{ } c) \gg A$

$Env \vdash B \gg \langle \text{Jump } jump \rangle \gg A$
 $Env \vdash B \gg \text{In1r } (\text{Jump } jump) \gg A$
 $Env \vdash B \gg \langle \text{Throw } e \rangle \gg A$
 $Env \vdash B \gg \text{In1r } (\text{Throw } e) \gg A$
 $Env \vdash B \gg \langle \text{Try } c1 \text{ Catch } (C \text{ } vn) \text{ } c2 \rangle \gg A$
 $Env \vdash B \gg \text{In1r } (\text{Try } c1 \text{ Catch } (C \text{ } vn) \text{ } c2) \gg A$
 $Env \vdash B \gg \langle c1 \text{ Finally } c2 \rangle \gg A$
 $Env \vdash B \gg \text{In1r } (c1 \text{ Finally } c2) \gg A$
 $Env \vdash B \gg \langle \text{Init } C \rangle \gg A$
 $Env \vdash B \gg \text{In1r } (\text{Init } C) \gg A$
 $Env \vdash B \gg \langle \text{NewC } C \rangle \gg A$
 $Env \vdash B \gg \text{In1l } (\text{NewC } C) \gg A$
 $Env \vdash B \gg \langle \text{New } T[e] \rangle \gg A$
 $Env \vdash B \gg \text{In1l } (\text{New } T[e]) \gg A$
 $Env \vdash B \gg \langle \text{Cast } T \text{ } e \rangle \gg A$
 $Env \vdash B \gg \text{In1l } (\text{Cast } T \text{ } e) \gg A$
 $Env \vdash B \gg \langle e \text{ InstOf } T \rangle \gg A$
 $Env \vdash B \gg \text{In1l } (e \text{ InstOf } T) \gg A$
 $Env \vdash B \gg \langle \text{Lit } v \rangle \gg A$
 $Env \vdash B \gg \text{In1l } (\text{Lit } v) \gg A$
 $Env \vdash B \gg \langle \text{UnOp } unop \text{ } e \rangle \gg A$
 $Env \vdash B \gg \text{In1l } (\text{UnOp } unop \text{ } e) \gg A$
 $Env \vdash B \gg \langle \text{BinOp } binop \text{ } e1 \text{ } e2 \rangle \gg A$
 $Env \vdash B \gg \text{In1l } (\text{BinOp } binop \text{ } e1 \text{ } e2) \gg A$
 $Env \vdash B \gg \langle \text{Super} \rangle \gg A$
 $Env \vdash B \gg \text{In1l } (\text{Super}) \gg A$
 $Env \vdash B \gg \langle \text{Acc } v \rangle \gg A$
 $Env \vdash B \gg \text{In1l } (\text{Acc } v) \gg A$
 $Env \vdash B \gg \langle v := e \rangle \gg A$
 $Env \vdash B \gg \text{In1l } (v := e) \gg A$
 $Env \vdash B \gg \langle c \text{ ? } e1 : e2 \rangle \gg A$
 $Env \vdash B \gg \text{In1l } (c \text{ ? } e1 : e2) \gg A$
 $Env \vdash B \gg \langle \{accC, statT, mode\} e.mn(\{pTs\} args) \rangle \gg A$
 $Env \vdash B \gg \text{In1l } (\{accC, statT, mode\} e.mn(\{pTs\} args)) \gg A$
 $Env \vdash B \gg \langle \text{Methd } C \text{ } sig \rangle \gg A$
 $Env \vdash B \gg \text{In1l } (\text{Methd } C \text{ } sig) \gg A$
 $Env \vdash B \gg \langle \text{Body } D \text{ } c \rangle \gg A$
 $Env \vdash B \gg \text{In1l } (\text{Body } D \text{ } c) \gg A$
 $Env \vdash B \gg \langle \text{LVar } vn \rangle \gg A$
 $Env \vdash B \gg \text{In2 } (\text{LVar } vn) \gg A$
 $Env \vdash B \gg \langle \{accC, statDeclC, stat\} e..fn \rangle \gg A$
 $Env \vdash B \gg \text{In2 } (\{accC, statDeclC, stat\} e..fn) \gg A$
 $Env \vdash B \gg \langle e1.[e2] \rangle \gg A$
 $Env \vdash B \gg \text{In2 } (e1.[e2]) \gg A$
 $Env \vdash B \gg \langle [] :: \text{expr list} \rangle \gg A$
 $Env \vdash B \gg \text{In3 } ([] :: \text{expr list}) \gg A$
 $Env \vdash B \gg \langle e \# es \rangle \gg A$
 $Env \vdash B \gg \text{In3 } (e \# es) \gg A$
declare *inj-term-sym-simps* [*simp del*]
declare *assigns-if.simps* [*simp*]
declare *split-paired-All* [*simp*] *split-paired-Ex* [*simp*]
setup $\langle \text{map-theory-simpset } (fn \text{ } ctxt \Rightarrow ctxt \mid > \text{Simplifier.add-loop } (split\text{-all-tac}, split\text{-all-tac})) \rangle$

lemma *da-Skip*: $A = (\text{norm} = B, \text{brk} = \lambda l. \text{UNIV}) \implies Env \vdash B \gg \langle \text{Skip} \rangle \gg A$
by (*auto intro: da.Skip*)

lemma *da-NewC*: $A = (\text{norm} = B, \text{brk} = \lambda l. \text{UNIV}) \implies Env \vdash B \gg \langle \text{NewC } C \rangle \gg A$

by (*auto intro: da.NewC*)

lemma *da-Lit*: $A = \langle \text{nrm}=B, \text{brk}=\lambda l. \text{UNIV} \rangle \implies \text{Env} \vdash B \gg \langle \text{Lit } v \rangle \gg A$
by (*auto intro: da.Lit*)

lemma *da-Super*: $\llbracket \text{This} \in B; A = \langle \text{nrm}=B, \text{brk}=\lambda l. \text{UNIV} \rangle \rrbracket \implies \text{Env} \vdash B \gg \langle \text{Super} \rangle \gg A$
by (*auto intro: da.Super*)

lemma *da-Init*: $A = \langle \text{nrm}=B, \text{brk}=\lambda l. \text{UNIV} \rangle \implies \text{Env} \vdash B \gg \langle \text{Init } C \rangle \gg A$
by (*auto intro: da.Init*)

lemma *assignsE-subseteq-assigns-ifs*:

assumes *boolEx*: $E \vdash e :: - \text{PrimT Boolean}$ (**is** *?Boolean e*)

shows *assignsE e* \subseteq *assigns-if True e* \cap *assigns-if False e* (**is** *?Incl e*)

proof –

obtain *vv* **where** *ex-lit*: $E \vdash \text{Lit } vv :: - \text{PrimT Boolean}$

using *typeof.simps(2) wt.Lit* **by** *blast*

have *?Boolean e* \implies *?Incl e*

proof (*induct e*)

case (*Cast T e*)

have $E \vdash e :: - (\text{PrimT Boolean})$

proof –

from $\langle E \vdash (\text{Cast } T e) :: - (\text{PrimT Boolean}) \rangle$

obtain *Te* **where** $E \vdash e :: - \text{Te}$

prg E $\vdash \text{Te} \preceq ? \text{PrimT Boolean}$

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.prem*s **obtain** *e1T e2T*

where $E \vdash e1 :: - e1T$ **and** $E \vdash e2 :: - e2T$ **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* = $\langle ?\text{Boolean } c \implies ?\text{Incl } c \rangle$

note *hyp-e1* = $\langle ?\text{Boolean } e1 \implies ?\text{Incl } e1 \rangle$

note *hyp-e2* = $\langle ?\text{Boolean } e2 \implies ?\text{Incl } e2 \rangle$

note *wt* = $\langle E \vdash (c ? e1 : e2) :: - \text{PrimT Boolean} \rangle$

```

then obtain
  boolean-c:  $E \vdash c :: \neg \text{PrimT Boolean}$  and
  boolean-e1:  $E \vdash e1 :: \neg \text{PrimT Boolean}$  and
  boolean-e2:  $E \vdash e2 :: \neg \text{PrimT Boolean}$ 
  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  $\bigwedge x. E \vdash \text{Lit } vv :: \neg \text{PrimT Boolean}$ 
    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]:  $l=l' \implies (\text{rmlab } l \ A) \ l' = \text{UNIV}$ 
  by (simp add: rmlab-def)

```

```

lemma rmlab-other-label [simp]:  $l \neq l' \implies (\text{rmlab } l \ A) \ l' = A \ l'$ 
  by (auto simp add: rmlab-def)

```

```

lemma range-inter-ts-subseteq [intro]:  $\forall k. A \ k \subseteq B \ k \implies \Rightarrow \bigcap A \subseteq \Rightarrow \bigcap B$ 
  by (auto simp add: range-inter-ts-def)

```

```

lemma range-inter-ts-subseteq':  $\forall k. A \ k \subseteq B \ k \implies x \in \Rightarrow \bigcap A \implies x \in \Rightarrow \bigcap B$ 
  by (auto simp add: range-inter-ts-def)

```

```

lemma da-monotone:
  assumes da:  $\text{Env} \vdash B \gg t \gg A$  and
     $B \subseteq B'$  and
    da':  $\text{Env} \vdash B' \gg t \gg A'$ 
  shows  $(\text{nrm } A \subseteq \text{nrm } A') \wedge (\forall l. (\text{brk } A \ l \subseteq \text{brk } A' \ l))$ 
proof –
  from da
  have  $\bigwedge B' \ A'. \llbracket \text{Env} \vdash B' \gg t \gg A'; B \subseteq B' \rrbracket$ 
     $\implies (\text{nrm } A \subseteq \text{nrm } A') \wedge (\forall l. (\text{brk } A \ l \subseteq \text{brk } A' \ l))$ 

```

```

    (is PROP ?Hyp Env B t A)
  proof (induct)
    case Skip
    then show ?case by cases simp
  next
    case Expr
    from Expr.prem Expr.hyps
    show ?case by cases simp
  next
    case (Lab Env B c C A l B' A')
    note A = ⟨nrm A = nrm C ∩ brk C l⟩ ⟨brk A = rmlab l (brk C)⟩
    note ⟨PROP ?Hyp Env B ⟨c⟩ C⟩
    moreover
    note ⟨B ⊆ 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.prem
      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
    from hyp-brk
    have rmlab l (brk C) l' ⊆ rmlab l (brk C') l' for 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
  have brk A l' ⊆ brk A' l' for 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}

```



```

      » $\langle c2 \rangle$ »  $C2'$  and
       $A': nrm\ A' = nrm\ C1' \cap nrm\ C2'$ 
       $brk\ A' = brk\ C1' \Rightarrow \cap\ brk\ C2'$ 
    by cases auto
  note  $\langle PROP\ ?Hyp\ Env\ B\ \langle c1 \rangle\ C1 \rangle$ 
  moreover note  $B' = \langle B \subseteq B' \rangle$ 
  moreover note  $da-c1'$ 
  ultimately obtain  $C1': nrm\ C1 \subseteq nrm\ C1' (\forall l. brk\ C1\ l \subseteq brk\ C1'\ l)$ 
  by blast
  note  $\langle PROP\ ?Hyp\ (Env(lcl := (lcl\ Env)(VName\ vn \mapsto Class\ C))\ ()$ 
     $(B \cup \{VName\ vn\})\ \langle c2 \rangle\ C2 \rangle$ 
  with  $B'\ da-c2'$ 
  obtain  $nrm\ C2 \subseteq nrm\ C2' (\forall l. brk\ C2\ l \subseteq brk\ C2'\ l)$ 
  by blast
  with  $C1'\ A\ A'$ 
  show ?case
  by auto
next
case  $(Fin\ Env\ B\ c1\ C1\ c2\ C2\ A\ B'\ A')$ 
note  $A = \langle nrm\ A = nrm\ C1 \cup nrm\ C2 \rangle$ 
 $\langle brk\ A = (brk\ C1 \Rightarrow \cup_{\forall} nrm\ C2) \Rightarrow \cap (brk\ C2) \rangle$ 
from  $\langle Env \vdash B' \rangle \langle c1\ Finally\ c2 \rangle \langle A' \rangle$ 
obtain  $C1'\ C2'$ 
  where  $da-c1': Env \vdash B' \rangle \langle c1 \rangle \rangle C1'$  and
         $da-c2': Env \vdash B' \rangle \langle c2 \rangle \rangle C2'$  and
         $A': nrm\ A' = nrm\ C1' \cup nrm\ C2'$ 
         $brk\ A' = (brk\ C1' \Rightarrow \cup_{\forall} nrm\ C2') \Rightarrow \cap (brk\ C2')$ 
  by cases auto
note  $\langle PROP\ ?Hyp\ Env\ B\ \langle c1 \rangle\ C1 \rangle$ 
moreover note  $B' = \langle B \subseteq B' \rangle$ 
moreover note  $da-c1'$ 
ultimately obtain  $C1': nrm\ C1 \subseteq nrm\ C1' (\forall l. brk\ C1\ l \subseteq brk\ C1'\ l)$ 
  by blast
note  $hyp-c2 = \langle PROP\ ?Hyp\ Env\ B\ \langle c2 \rangle\ C2 \rangle$ 
from  $da-c2'\ B'$ 
obtain  $nrm\ C2 \subseteq nrm\ C2' (\forall l. brk\ C2\ l \subseteq brk\ C2'\ l)$ 
  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  $A = \langle nrm\ A = B \cup$ 
   $assigns-if\ True\ (BinOp\ CondAnd\ e1\ e2) \cap$ 
   $assigns-if\ False\ (BinOp\ CondAnd\ e1\ e2) \rangle$ 

```

```

      ⟨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  $E1'$ :  $nrm\ E1 \subseteq nrm\ E1' \ (\forall l. brk\ E1\ l \subseteq brk\ E1'\ l)$ 
  by blast
note  $\langle PROP\ ?Hyp\ Env\ (B \cup assigns\text{-}if\ False\ c) \ \langle e2 \rangle\ E2 \rangle$ 
with  $B'\ da\text{-}e2'$ 
obtain  $nrm\ E2 \subseteq nrm\ E2' \ (\forall l. brk\ E2\ l \subseteq brk\ E2'\ l)$ 
  by blast
with  $E1'\ A\ A'$ 
show ?case
  by auto
next
case Call
from Call.prem 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'  $\langle B \subseteq B' \rangle$ ] show ?thesis .
qed

```

lemma da-weaken:

```

assumes da:  $Env \vdash B \gg t \gg A$  and  $B \subseteq B'$ 
shows  $\exists A'. Env \vdash B' \gg t \gg A'$ 
proof -
  note assigned.select-convs [Pure.intro]
  from da
  have  $\bigwedge B'. B \subseteq B' \implies \exists A'. Env \vdash B' \gg t \gg A'$  (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  $\langle PROP\ ?Hyp\ Env\ B \ \langle c \rangle \rangle$ 
    moreover
    note  $B' = \langle B \subseteq B' \rangle$ 
    ultimately obtain  $C'$  where  $Env \vdash B' \gg \langle c \rangle \gg C'$ 
      by iprover
    then obtain  $A'$  where  $Env \vdash B' \gg \langle Break\ l.\ c \rangle \gg A'$ 
      by (iprover intro: da.Lab)
    thus ?case ..
  next
    case (Comp Env B c1 C1 c2 C2 A B')
    note  $da\text{-}c1 = \langle Env \vdash B \gg \langle c1 \rangle \gg C1 \rangle$ 
    note  $\langle PROP\ ?Hyp\ Env\ B \ \langle c1 \rangle \rangle$ 
    moreover
    note  $B' = \langle B \subseteq B' \rangle$ 
    ultimately obtain  $C1'$  where  $da\text{-}c1': Env \vdash B' \gg \langle c1 \rangle \gg C1'$ 

```

```

  by iprover
with da-c1 B'
have
  nrm C1  $\subseteq$  nrm C1'
  by (rule da-monotone [elim-format]) simp
moreover
note  $\langle PROP \ ?Hyp \ Env \ (nrm \ C1) \ \langle c2 \rangle \rangle$ 
ultimately obtain C2' where  $Env \vdash \ nrm \ C1' \ \gg \ \langle c2 \rangle \gg \ C2'$ 
  by iprover
with da-c1' obtain A' where  $Env \vdash \ B' \ \gg \ \langle c1;; \ c2 \rangle \gg \ A'$ 
  by (iprover intro: da.Comp)
thus ?case ..
next
case (If Env B e E c1 C1 c2 C2 A B')
note B' =  $\langle B \subseteq B' \rangle$ 
obtain E' where  $Env \vdash \ B' \ \gg \ \langle e \rangle \gg \ E'$ 
proof -
  have PROP ?Hyp Env B  $\langle e \rangle$  by (rule If.hyps)
  with B'
  show ?thesis using that by iprover
qed
moreover
obtain C1' where  $Env \vdash \ (B' \cup \text{assigns-if True } e) \ \gg \ \langle c1 \rangle \gg \ C1'$ 
proof -
  from B'
  have  $(B \cup \text{assigns-if True } e) \subseteq (B' \cup \text{assigns-if True } e)$ 
  by blast
  moreover
  have PROP ?Hyp Env  $(B \cup \text{assigns-if True } e) \ \langle c1 \rangle$  by (rule If.hyps)
  ultimately
  show ?thesis using that by iprover
qed
moreover
obtain C2' where  $Env \vdash \ (B' \cup \text{assigns-if False } e) \ \gg \ \langle c2 \rangle \gg \ C2'$ 
proof -
  from B' have  $(B \cup \text{assigns-if False } e) \subseteq (B' \cup \text{assigns-if False } e)$ 
  by blast
  moreover
  have PROP ?Hyp Env  $(B \cup \text{assigns-if False } e) \ \langle c2 \rangle$  by (rule If.hyps)
  ultimately
  show ?thesis using that by iprover
qed
ultimately
obtain A' where  $Env \vdash \ B' \ \gg \ \langle If(e) \ c1 \ Else \ c2 \rangle \gg \ A'$ 
  by (iprover intro: da.If)
thus ?case ..
next
case (Loop Env B e E c C A l B')
note B' =  $\langle B \subseteq B' \rangle$ 
obtain E' where  $Env \vdash \ B' \ \gg \ \langle e \rangle \gg \ E'$ 
proof -
  have PROP ?Hyp Env B  $\langle e \rangle$  by (rule Loop.hyps)
  with B'
  show ?thesis using that by iprover
qed
moreover
obtain C' where  $Env \vdash \ (B' \cup \text{assigns-if True } e) \ \gg \ \langle c \rangle \gg \ C'$ 
proof -
  from B'

```

```

have  $(B \cup \text{assigns-if True } e) \subseteq (B' \cup \text{assigns-if True } e)$ 
  by blast
moreover
have  $PROP \text{ ?Hyp Env } (B \cup \text{assigns-if True } e) \langle c \rangle$  by  $(\text{rule Loop.hyps})$ 
ultimately
show ?thesis using that by iprover
qed
ultimately
obtain  $A'$  where  $\text{Env} \vdash B' \gg \langle l \cdot \text{While}(e) \ c \rangle \gg A'$ 
  by  $(\text{iprover intro: da.Loop})$ 
thus ?case ..
next
case  $(\text{Jmp jump } B \ A \ \text{Env } B')$ 
note  $B' = \langle B \subseteq B' \rangle$ 
with  $\text{Jmp.hyps}$  have  $\text{jump} = \text{Ret} \longrightarrow \text{Result} \in B'$ 
  by auto
moreover
obtain  $A'::\text{assigned}$ 
  where  $\text{nrn } A' = \text{UNIV}$ 
     $\text{brk } A' = (\text{case jump of}$ 
       $\text{Break } l \Rightarrow \lambda k. \text{ if } k = l \text{ then } B' \text{ else UNIV}$ 
       $| \text{Cont } l \Rightarrow \lambda k. \text{ UNIV}$ 
       $| \text{Ret} \Rightarrow \lambda k. \text{ UNIV})$ 
  by iprover
ultimately have  $\text{Env} \vdash B' \gg \langle \text{Jmp jump} \rangle \gg A'$ 
  by  $(\text{rule da.Jmp})$ 
thus ?case ..
next
case Throw thus ?case by  $(\text{iprover intro: da.Throw})$ 
next
case  $(\text{Try Env } B \ c1 \ C1 \ \text{vn } C \ c2 \ C2 \ A \ B')$ 
note  $B' = \langle B \subseteq B' \rangle$ 
obtain  $C1'$  where  $\text{Env} \vdash B' \gg \langle c1 \rangle \gg C1'$ 
proof –
  have  $PROP \text{ ?Hyp Env } B \ \langle c1 \rangle$  by  $(\text{rule Try.hyps})$ 
  with  $B'$ 
  show ?thesis using that by iprover
qed
moreover
obtain  $C2'$  where
   $\text{Env}(\text{lcl} := (\text{lcl Env})(\text{VName vn} \mapsto \text{Class } C)) \vdash B' \cup \{\text{VName vn}\} \gg \langle c2 \rangle \gg C2'$ 
proof –
  from  $B'$  have  $B \cup \{\text{VName vn}\} \subseteq B' \cup \{\text{VName vn}\}$  by blast
moreover
have  $PROP \text{ ?Hyp } (\text{Env}(\text{lcl} := (\text{lcl Env})(\text{VName vn} \mapsto \text{Class } C)))$ 
   $(B \cup \{\text{VName vn}\}) \langle c2 \rangle$ 
  by  $(\text{rule Try.hyps})$ 
ultimately
show ?thesis using that by iprover
qed
ultimately
obtain  $A'$  where  $\text{Env} \vdash B' \gg \langle \text{Try } c1 \ \text{Catch}(C \ \text{vn}) \ c2 \rangle \gg A'$ 
  by  $(\text{iprover intro: da.Try})$ 
thus ?case ..
next
case  $(\text{Fin Env } B \ c1 \ C1 \ c2 \ C2 \ A \ B')$ 
note  $B' = \langle B \subseteq B' \rangle$ 
obtain  $C1'$  where  $C1': \text{Env} \vdash B' \gg \langle c1 \rangle \gg C1'$ 

```

```

proof –
  have  $PROP \ ?Hyp \ Env \ B \ \langle c1 \rangle$  by (rule Fin.hyps)
  with  $B'$ 
  show ?thesis using that by iprover
qed
moreover
obtain  $C2'$  where  $Env \vdash B' \gg \langle c2 \rangle \gg C2'$ 
proof –
  have  $PROP \ ?Hyp \ Env \ B \ \langle c2 \rangle$  by (rule Fin.hyps)
  with  $B'$ 
  show ?thesis using that by iprover
qed
ultimately
obtain  $A'$  where  $Env \vdash B' \gg \langle c1 \text{ Finally } c2 \rangle \gg A'$ 
  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  $B' = \langle B \subseteq B' \rangle$ 
obtain  $E1'$  where  $Env \vdash B' \gg \langle e1 \rangle \gg E1'$ 
proof –
  have  $PROP \ ?Hyp \ Env \ B \ \langle e1 \rangle$  by (rule CondAnd.hyps)
  with  $B'$ 
  show ?thesis using that by iprover
qed
moreover
obtain  $E2'$  where  $Env \vdash B' \cup \text{ assigns-if True } e1 \gg \langle e2 \rangle \gg E2'$ 
proof –
  from  $B'$  have  $B \cup \text{ assigns-if True } e1 \subseteq B' \cup \text{ assigns-if True } e1$ 
    by blast
  moreover
  have  $PROP \ ?Hyp \ Env \ (B \cup \text{ assigns-if True } e1) \ \langle e2 \rangle$  by (rule CondAnd.hyps)
  ultimately show ?thesis using that by iprover
qed
ultimately
obtain  $A'$  where  $Env \vdash B' \gg \langle BinOp \ CondAnd \ e1 \ e2 \rangle \gg A'$ 
  by (iprover intro: da.CondAnd)
thus ?case ..
next
case (CondOr  $Env \ B \ e1 \ E1 \ e2 \ E2 \ A \ B'$ )
note  $B' = \langle B \subseteq B' \rangle$ 
obtain  $E1'$  where  $Env \vdash B' \gg \langle e1 \rangle \gg E1'$ 
proof –
  have  $PROP \ ?Hyp \ Env \ B \ \langle e1 \rangle$  by (rule CondOr.hyps)
  with  $B'$ 

```

```

  show ?thesis using that by iprover
qed
moreover
obtain  $E2'$  where  $Env \vdash B' \cup \text{assigns-if False } e1 \gg \langle e2 \rangle \gg E2'$ 
proof -
  from  $B'$  have  $B \cup \text{assigns-if False } e1 \subseteq B' \cup \text{assigns-if False } e1$ 
  by blast
  moreover
  have  $PROP \text{ ?Hyp } Env (B \cup \text{assigns-if False } e1) \langle e2 \rangle$  by (rule CondOr.hyps)
  ultimately show ?thesis using that by iprover
qed
ultimately
obtain  $A'$  where  $Env \vdash B' \gg \langle BinOp \text{ CondOr } e1 \ e2 \rangle \gg A'$ 
  by (iprover intro: da.CondOr)
thus ?case ..
next
case (BinOp Env B e1 E1 e2 A binop B')
note  $B' = \langle B \subseteq B' \rangle$ 
obtain  $E1'$  where  $E1': Env \vdash B' \gg \langle e1 \rangle \gg E1'$ 
proof -
  have  $PROP \text{ ?Hyp } Env B \langle e1 \rangle$  by (rule BinOp.hyps)
  with  $B'$ 
  show ?thesis using that by iprover
qed
moreover
obtain  $A'$  where  $Env \vdash nrm \ E1' \gg \langle e2 \rangle \gg A'$ 
proof -
  have  $Env \vdash B \gg \langle e1 \rangle \gg E1$  by (rule BinOp.hyps)
  from this  $B' \ E1'$ 
  have  $nrm \ E1 \subseteq nrm \ E1'$ 
  by (rule da-monotone [THEN conjE])
  moreover
  have  $PROP \text{ ?Hyp } Env (nrm \ E1) \langle e2 \rangle$  by (rule BinOp.hyps)
  ultimately show ?thesis using that by iprover
qed
ultimately
have  $Env \vdash B' \gg \langle BinOp \text{ binop } e1 \ e2 \rangle \gg A'$ 
  using BinOp.hyps by (iprover intro: da.BinOp)
thus ?case ..
next
case (Super B Env B')
note  $B' = \langle B \subseteq B' \rangle$ 
with Super.hyps have  $This \in B'$ 
  by auto
thus ?case by (iprover intro: da.Super)
next
case (AccLVar vn B A Env B')
note  $\langle vn \in B \rangle$ 
moreover
note  $\langle B \subseteq B' \rangle$ 
ultimately have  $vn \in B'$  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  $B' = \langle B \subseteq B' \rangle$ 
then obtain  $E'$  where  $Env \vdash B' \gg \langle e \rangle \gg E'$ 
  by (rule AssLVar.hyps [elim-format]) iprover

```

```

then obtain  $A'$  where
   $Env \vdash B' \gg \langle LVar\ vn := e \rangle \gg A'$ 
  by (iprover intro: da.AssLVar)
thus ?case ..
next
case (Ass  $v$  Env  $B$   $V$   $e$   $A$   $B'$ )
note  $B' = \langle B \subseteq B' \rangle$ 
note  $\langle \forall vn. v \neq LVar\ vn \rangle$ 
moreover
obtain  $V'$  where  $V': Env \vdash B' \gg \langle v \rangle \gg V'$ 
proof –
  have PROP ?Hyp Env  $B$   $\langle v \rangle$  by (rule Ass.hyps)
  with  $B'$ 
  show ?thesis using that by iprover
qed
moreover
obtain  $A'$  where  $Env \vdash nrm\ V' \gg \langle e \rangle \gg A'$ 
proof –
  have  $Env \vdash B \gg \langle v \rangle \gg V$  by (rule Ass.hyps)
  from this  $B'$   $V'$ 
  have  $nrm\ V \subseteq nrm\ V'$ 
  by (rule da-monotone [THEN conjE])
  moreover
  have PROP ?Hyp Env ( $nrm\ V$ )  $\langle e \rangle$  by (rule Ass.hyps)
  ultimately show ?thesis using that by iprover
qed
ultimately
have  $Env \vdash B' \gg \langle v := e \rangle \gg A'$ 
  by (iprover intro: da.Ass)
thus ?case ..
next
case (CondBool Env  $c$   $e1$   $e2$   $B$   $C$   $E1$   $E2$   $A$   $B'$ )
note  $B' = \langle B \subseteq B' \rangle$ 
note  $\langle Env \vdash (c\ ?\ e1 : e2) :: \neg (PrimT\ Boolean) \rangle$ 
moreover obtain  $C'$  where  $C': Env \vdash B' \gg \langle c \rangle \gg C'$ 
proof –
  have PROP ?Hyp Env  $B$   $\langle c \rangle$  by (rule CondBool.hyps)
  with  $B'$ 
  show ?thesis using that by iprover
qed
moreover
obtain  $E1'$  where  $Env \vdash B' \cup assigns\text{-}if\ True\ c \gg \langle e1 \rangle \gg E1'$ 
proof –
  from  $B'$ 
  have  $(B \cup assigns\text{-}if\ True\ c) \subseteq (B' \cup assigns\text{-}if\ True\ c)$ 
  by blast
  moreover
  have PROP ?Hyp Env  $(B \cup assigns\text{-}if\ True\ c)$   $\langle e1 \rangle$  by (rule CondBool.hyps)
  ultimately
  show ?thesis using that by iprover
qed
moreover
obtain  $E2'$  where  $Env \vdash B' \cup assigns\text{-}if\ False\ c \gg \langle e2 \rangle \gg E2'$ 
proof –
  from  $B'$ 
  have  $(B \cup assigns\text{-}if\ False\ c) \subseteq (B' \cup assigns\text{-}if\ False\ c)$ 
  by blast
  moreover
  have PROP ?Hyp Env  $(B \cup assigns\text{-}if\ False\ c)$   $\langle e2 \rangle$  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  $PROP \ ?Hyp \ Env \ (nrm \ E) \ \langle args \rangle$  by (rule Call.hyps)
  ultimately show  $?thesis$  using that by iprover
qed
ultimately
  have  $Env \vdash B' \gg \langle \{accC, statT, mode\} e.mn(\{pTs\} args) \rangle \gg A'$ 
    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  $B' = \langle B \subseteq B' \rangle$ 
  obtain  $C'$  where  $C': Env \vdash B' \gg \langle c \rangle \gg C'$  and  $nrm\text{-}C': nrm \ C \subseteq nrm \ C'$ 
  proof –
    have  $Env \vdash B \gg \langle c \rangle \gg C$  by (rule Body.hyps)
    moreover note  $B'$ 
    moreover
      from  $B'$  obtain  $C'$  where  $da\text{-}c: Env \vdash B' \gg \langle c \rangle \gg C'$ 
        by (rule Body.hyps [elim-format]) blast
    ultimately
      have  $nrm \ C \subseteq nrm \ C'$ 
        by (rule da-monotone [THEN conjE])
      with  $da\text{-}c$  that show  $?thesis$  by iprover
  qed
  moreover
    note  $\langle Result \in nrm \ C \rangle$ 
    with  $nrm\text{-}C'$  have  $Result \in nrm \ C'$ 
      by blast
    moreover note  $\langle jumpNestingOkS \ \{Ret\} \ c \rangle$ 
    ultimately obtain  $A'$  where
       $Env \vdash B' \gg \langle Body \ D \ c \rangle \gg A'$ 
      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  $B' = \langle B \subseteq B' \rangle$ 
  obtain  $E1'$  where  $E1': Env \vdash B' \gg \langle e1 \rangle \gg E1'$ 
  proof –
    have  $PROP \ ?Hyp \ Env \ B \ \langle e1 \rangle$  by (rule AVar.hyps)
    with  $B'$ 
    show  $?thesis$  using that by iprover
  qed
  moreover
    obtain  $A'$  where  $Env \vdash nrm \ E1' \gg \langle e2 \rangle \gg A'$ 
  proof –
    have  $Env \vdash B \gg \langle e1 \rangle \gg E1$  by (rule AVar.hyps)
    from  $this \ B' \ E1'$ 
    have  $nrm \ E1 \subseteq nrm \ E1'$ 
      by (rule da-monotone [THEN conjE])
    moreover
      have  $PROP \ ?Hyp \ Env \ (nrm \ E1) \ \langle e2 \rangle$  by (rule AVar.hyps)
      ultimately show  $?thesis$  using that by iprover
  qed
  ultimately

```

```

  have  $Env \vdash B' \gg \langle e1.[e2] \rangle A'$ 
    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  $B' = \langle B \subseteq B' \rangle$ 
  obtain  $E'$  where  $E': Env \vdash B' \gg \langle e \rangle E'$ 
  proof -
    have  $PROP \ ?Hyp \ Env \ B \ \langle e \rangle$  by (rule Cons.hyps)
    with  $B'$ 
    show ?thesis using that by iprover
  qed
  moreover
  obtain  $A'$  where  $Env \vdash nrm \ E' \gg \langle es \rangle A'$ 
  proof -
    have  $Env \vdash B \gg \langle e \rangle E$  by (rule Cons.hyps)
    from this  $B' \ E'$ 
    have  $nrm \ E \subseteq nrm \ E'$ 
      by (rule da-monotone [THEN conjE])
    moreover
    have  $PROP \ ?Hyp \ Env \ (nrm \ E) \ \langle es \rangle$  by (rule Cons.hyps)
    ultimately show ?thesis using that by iprover
  qed
  ultimately
  have  $Env \vdash B' \gg \langle e \# es \rangle A'$ 
    by (iprover intro: da.Cons)
  thus ?case ..
qed
from this [OF  $\langle B \subseteq B' \rangle$ ] show ?thesis .
qed

```

corollary *da-weakenE* [consumes 2]:

assumes $da: Env \vdash B \gg t \gg A$ and $B': B \subseteq B'$ and

$ex\text{-}mono: \bigwedge A'. \llbracket Env \vdash B' \gg t \gg A'; nrm \ A \subseteq nrm \ A'; \bigwedge l. brk \ A \ l \subseteq brk \ A' \ l \rrbracket \implies P$

shows P

proof -

from $da \ B'$

obtain A' where $A': Env \vdash B' \gg t \gg A'$

by (rule *da-weaken* [elim-format]) iprover

with $da \ B'$

have $nrm \ A \subseteq nrm \ A' \wedge (\forall l. brk \ A \ l \subseteq brk \ A' \ l)$

by (rule *da-monotone*)

with $A' \ ex\text{-}mono$

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

$wf_fdecl :: prog \Rightarrow pname \Rightarrow fdecl \Rightarrow bool$
where $wf_fdecl\ G\ P = (\lambda(fn,f). is_acc_type\ G\ P\ (type\ f))$

lemma $wf_fdecl_def2: \bigwedge fd. wf_fdecl\ G\ P\ fd = is_acc_type\ G\ P\ (type\ (snd\ fd))$

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

$wf_mhead :: prog \Rightarrow pname \Rightarrow sig \Rightarrow mhead \Rightarrow bool$ **where**
 $wf_mhead\ G\ P = (\lambda\ sig\ mh. length\ (parTs\ sig) = length\ (pars\ mh) \wedge$
 $(\forall\ T \in set\ (parTs\ sig). is_acc_type\ G\ P\ T) \wedge$
 $is_acc_type\ G\ P\ (resTy\ mh) \wedge$
 $distinct\ (pars\ mh))$

A method declaration is wellformed if:

- the method head is wellformed
- the names of the local variables are unique
- the types of the local variables must be accessible
- the local variables don't shadow the parameters
- the class of the method is defined
- the body statement is welltyped with respect to the modified environment of local names, were the local variables, the parameters the special result variable (Res) and This are assoziated with there types.

definition

$callee_lcl :: qname \Rightarrow sig \Rightarrow methd \Rightarrow lenv$ **where**
 $callee_lcl\ C\ sig\ m =$
 $(\lambda k. (case\ k\ of$
 $\quad EName\ e$
 $\quad \Rightarrow (case\ e\ of$
 $\quad \quad VName\ v$
 $\quad \quad \Rightarrow ((table_of\ (lcls\ (mbody\ m)))(pars\ m\ [\mapsto]\ parTs\ sig))\ v$
 $\quad \quad | Res \Rightarrow Some\ (resTy\ m))$
 $\quad \quad | This \Rightarrow if\ is_static\ m\ then\ None\ else\ Some\ (Class\ C)))$

definition

$parameters :: methd \Rightarrow lname\ set$ **where**
 $parameters\ m = set\ (map\ (EName \circ VName)\ (pars\ m)) \cup (if\ (static\ m)\ then\ \{\}\ else\ \{This\})$

definition

$wf_mdecl :: prog \Rightarrow qname \Rightarrow mdecl \Rightarrow bool$ **where**
 $wf_mdecl\ G\ C =$
 $(\lambda(sig, m).$
 $\quad wf_mhead\ G\ (pid\ C)\ sig\ (mhead\ m) \wedge$
 $\quad unique\ (lcls\ (mbody\ m)) \wedge$
 $\quad (\forall\ (vn, T) \in set\ (lcls\ (mbody\ m)). is_acc_type\ G\ (pid\ C)\ T) \wedge$
 $\quad (\forall\ pn \in set\ (pars\ m). table_of\ (lcls\ (mbody\ m))\ pn = None) \wedge$
 $\quad jumpNestingOkS\ \{Ret\}\ (stmt\ (mbody\ m)) \wedge$
 $\quad is_class\ G\ C \wedge$
 $\quad (\llbracket prg = G, cls = C, lcl = callee_lcl\ C\ sig\ m \rrbracket \vdash (stmt\ (mbody\ m)) :: \surd \wedge$
 $\quad (\exists\ A. (\llbracket prg = G, cls = C, lcl = callee_lcl\ C\ sig\ m \rrbracket$
 $\quad \quad \vdash parameters\ m \gg stmt\ (mbody\ m) \gg A$
 $\quad \quad \wedge Result \in nrm\ A))$

lemma $callee_lcl_VName_simp\ [simp]:$

$callee_lcl\ C\ sig\ m\ (EName\ (VName\ v))$
 $= ((table_of\ (lcls\ (mbody\ m)))(pars\ m\ [\mapsto]\ parTs\ sig))\ v$
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 \implies callee-lcl C sig m (This) = None
by simp

lemma *callee-lcl-This-not-static-simp*:
 \neg *is-static m \implies callee-lcl C sig m (This) = Some (Class C)*
by simp

lemma *wf-mheadI*:
 $\llbracket \text{length (parTs sig)} = \text{length (pars m)}; \forall T \in \text{set (parTs sig)}. \text{is-acc-type } G \ P \ T;$
 $\text{is-acc-type } G \ P \ (\text{resTy } m); \text{distinct (pars m)} \rrbracket \implies$
 $\text{wf-mhead } G \ P \ \text{sig } m$
apply (unfold wf-mhead-def)
apply (simp (no-asm-simp))
done

lemma *wf-mdeclI*: \llbracket
 $\text{wf-mhead } G \ (\text{pid } C) \ \text{sig } (\text{mhead } m); \text{unique (lcls (mbody } m));$
 $(\forall pn \in \text{set (pars } m). \text{table-of (lcls (mbody } m)) } pn = \text{None});$
 $\forall (vn, T) \in \text{set (lcls (mbody } m)). \text{is-acc-type } G \ (\text{pid } C) \ T;$
 $\text{jumpNestingOkS } \{\text{Ret}\} \ (\text{stmt (mbody } m));$
 $\text{is-class } G \ C;$
 $(\llbracket \text{prg} = G, \text{cls} = C, \text{lcl} = \text{callee-lcl } C \ \text{sig } m \rrbracket \vdash (\text{stmt (mbody } m)) :: \checkmark;$
 $(\exists A. (\llbracket \text{prg} = G, \text{cls} = C, \text{lcl} = \text{callee-lcl } C \ \text{sig } m \rrbracket \vdash \text{parameters } m \gg \langle \text{stmt (mbody } m) \rangle) \gg A$
 $\wedge \text{Result} \in \text{nrm } A)$
 $\rrbracket \implies$
 $\text{wf-mdecl } G \ C \ (\text{sig}, m)$
apply (unfold wf-mdecl-def)
apply simp
done

lemma *wf-mdeclE* [consumes 1]:
 $\llbracket \text{wf-mdecl } G \ C \ (\text{sig}, m);$
 $\llbracket \text{wf-mhead } G \ (\text{pid } C) \ \text{sig } (\text{mhead } m); \text{unique (lcls (mbody } m));$
 $\forall pn \in \text{set (pars } m). \text{table-of (lcls (mbody } m)) } pn = \text{None};$
 $\forall (vn, T) \in \text{set (lcls (mbody } m)). \text{is-acc-type } G \ (\text{pid } C) \ T;$
 $\text{jumpNestingOkS } \{\text{Ret}\} \ (\text{stmt (mbody } m));$
 $\text{is-class } G \ C;$
 $(\llbracket \text{prg} = G, \text{cls} = C, \text{lcl} = \text{callee-lcl } C \ \text{sig } m \rrbracket \vdash (\text{stmt (mbody } m)) :: \checkmark;$
 $(\exists A. (\llbracket \text{prg} = G, \text{cls} = C, \text{lcl} = \text{callee-lcl } C \ \text{sig } m \rrbracket \vdash \text{parameters } m \gg \langle \text{stmt (mbody } m) \rangle) \gg A$
 $\wedge \text{Result} \in \text{nrm } A)$
 $\rrbracket \implies P$
 $\rrbracket \implies P$
by (unfold wf-mdecl-def) simp

lemma *wf-mdeclD1*:
 $\text{wf-mdecl } G \ C \ (\text{sig}, m) \implies$
 $\text{wf-mhead } G \ (\text{pid } C) \ \text{sig } (\text{mhead } m) \wedge \text{unique (lcls (mbody } m)) \wedge$
 $(\forall pn \in \text{set (pars } m). \text{table-of (lcls (mbody } m)) } pn = \text{None}) \wedge$

```

  (∀ (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)

```


$$\begin{aligned} & \text{entails } (\lambda \text{new old. } G \vdash \text{resTy new} \preceq \text{resTy old} \wedge \\ & \quad \text{is-static new} = \text{is-static old})) \wedge \\ & (\text{set-option} \circ \text{table-of } (\text{imethods } i)) \\ & \quad \text{hidings } \text{Un-tables}((\lambda J. (\text{imethds } G \ J)) \text{'set } (\text{isuperIfs } i)) \\ & \quad \text{entails } (\lambda \text{new old. } G \vdash \text{resTy new} \preceq \text{resTy old})) \end{aligned}$$

lemma *wf-idecl-mhead*: $\llbracket \text{wf-idecl } G \ (I, i); (\text{sig}, \text{mh}) \in \text{set } (\text{imethods } i) \rrbracket \implies$
 $\text{wf-mhead } G \ (\text{pid } I) \ \text{sig} \ \text{mh} \wedge \neg \text{is-static } \text{mh} \wedge \text{accmodi } \text{mh} = \text{Public}$
apply (*unfold wf-idecl-def*)
apply *auto*
done

lemma *wf-idecl-hidings*:
 $\text{wf-idecl } G \ (I, i) \implies$
 $(\lambda s. \text{set-option } (\text{table-of } (\text{imethods } i) \ s))$
 $\text{hidings } \text{Un-tables}((\lambda J. \text{imethds } G \ J) \text{'set } (\text{isuperIfs } i))$
 $\text{entails } \lambda \text{new old. } G \vdash \text{resTy new} \preceq \text{resTy old}$
apply (*unfold wf-idecl-def o-def*)
apply *simp*
done

lemma *wf-idecl-hiding*:
 $\text{wf-idecl } G \ (I, i) \implies$
 $(\text{table-of } (\text{imethods } i))$
 $\text{hiding } (\text{methd } G \ \text{Object})$
 $\text{under } (\lambda \text{new old. } \text{accmodi } \text{old} \neq \text{Private})$
 $\text{entails } (\lambda \text{new old. } G \vdash \text{resTy new} \preceq \text{resTy old} \wedge$
 $\quad \text{is-static new} = \text{is-static old}))$
apply (*unfold wf-idecl-def*)
apply *simp*
done

lemma *wf-idecl-supD*:
 $\llbracket \text{wf-idecl } G \ (I, i); J \in \text{set } (\text{isuperIfs } i) \rrbracket$
 $\implies \text{is-acc-iface } G \ (\text{pid } I) \ J \wedge (J, I) \notin (\text{subint1 } G)^+$
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

$entails :: ('a, 'b) \text{ table} \Rightarrow ('b \Rightarrow \text{bool}) \Rightarrow \text{bool} \quad (\cdot - entails \rightarrow 20)$
where $(t \text{ entails } P) = (\forall k. \forall x \in t \ k: P \ x)$

lemma *entailsD*:

$\llbracket t \text{ entails } P; t \ k = \text{Some } x \rrbracket \Longrightarrow P \ x$
by (*simp add: entails-def*)

lemma *empty-entails[simp]*: *Map.empty entails P*

by (*simp add: entails-def*)

definition

$wf\text{-}cdecl :: \text{prog} \Rightarrow \text{cdecl} \Rightarrow \text{bool}$ **where**
 $wf\text{-}cdecl \ G =$
 $(\lambda(C, c).$
 $\neg is\text{-}iface \ G \ C \wedge$
 $(\forall I \in set \ (superIfs \ c). \ is\text{-}acc\text{-}iface \ G \ (pid \ C) \ I \wedge$
 $(\forall s. \forall im \in imethds \ G \ I \ s.$
 $(\exists \ cm \in methd \ G \ C \ s: \ G \vdash resTy \ cm \leq resTy \ im \wedge$
 $\neg is\text{-}static \ cm \wedge$
 $accmodi \ im \leq accmodi \ cm))) \wedge$
 $(\forall f \in set \ (cfields \ c). \ wf\text{-}fdecl \ G \ (pid \ C) \ f) \wedge unique \ (cfields \ c) \wedge$
 $(\forall m \in set \ (methods \ c). \ wf\text{-}mdecl \ G \ C \ m) \wedge unique \ (methods \ c) \wedge$
 $jumpNestingOkS \ \{\} \ (init \ c) \wedge$
 $(\exists \ A. \ (\text{prg}=G, \text{cls}=C, \text{lcl}=Map.empty) \vdash \ \{\} \ \gg \langle init \ c \rangle \gg A) \wedge$
 $(\text{prg}=G, \text{cls}=C, \text{lcl}=Map.empty) \vdash (init \ c) :: \checkmark \wedge ws\text{-}cdecl \ G \ C \ (super \ c) \wedge$
 $(C \neq Object \longrightarrow$
 $(is\text{-}acc\text{-}class \ G \ (pid \ C) \ (super \ c) \wedge$
 $(table\text{-}of \ (map \ (\lambda \ (s, m). \ (s, C, m)) \ (methods \ c))$
 $entails \ (\lambda \ new. \forall \ old \ sig.$
 $(G, sig \vdash new \ overrides_S \ old$
 $\longrightarrow (G \vdash resTy \ new \leq resTy \ old \wedge$
 $accmodi \ old \leq accmodi \ new \wedge$
 $\neg is\text{-}static \ old))) \wedge$
 $(G, sig \vdash new \ hides \ old$
 $\longrightarrow (accmodi \ old \leq accmodi \ new \wedge$
 $is\text{-}static \ old))))))$
 $)))$

lemma *wf-cdeclE* [consumes 1]:
 $\llbracket \text{wf-cdecl } G \ (C, c);$
 $\llbracket \neg \text{is-iface } G \ C;$
 $(\forall I \in \text{set } (\text{superIfs } c). \text{is-acc-iface } G \ (\text{pid } C) \ I \wedge$
 $(\forall s. \forall im \in \text{imethds } G \ I \ s.$
 $(\exists cm \in \text{methd } G \ C \ s: G \vdash \text{resTy } cm \preceq \text{resTy } im \wedge$
 $\neg \text{is-static } cm \wedge$
 $\text{accmodi } im \leq \text{accmodi } cm)))$;
 $\forall f \in \text{set } (\text{cfields } c). \text{wf-fdecl } G \ (\text{pid } C) \ f; \text{unique } (\text{cfields } c);$
 $\forall m \in \text{set } (\text{methods } c). \text{wf-mdecl } G \ C \ m; \text{unique } (\text{methods } c);$
 $\text{jumpNestingOkS } \{\} \ (\text{init } c);$
 $\exists A. (\text{prg} = G, \text{cls} = C, \text{lcl} = \text{Map.empty}) \vdash \{\} \gg \langle \text{init } c \rangle \gg A;$
 $(\text{prg} = G, \text{cls} = C, \text{lcl} = \text{Map.empty}) \vdash (\text{init } c) :: \checkmark;$
 $\text{ws-cdecl } G \ C \ (\text{super } c);$
 $(C \neq \text{Object} \longrightarrow$
 $(\text{is-acc-class } G \ (\text{pid } C) \ (\text{super } c) \wedge$
 $(\text{table-of } (\text{map } (\lambda (s, m). (s, C, m)) \ (\text{methods } c))$
 $\text{entails } (\lambda \text{new}. \forall \text{old sig.}$
 $(G, \text{sig} \vdash \text{new overrides } \text{old}$
 $\longrightarrow (G \vdash \text{resTy } \text{new} \preceq \text{resTy } \text{old} \wedge$
 $\text{accmodi } \text{old} \leq \text{accmodi } \text{new} \wedge$
 $\neg \text{is-static } \text{old})) \wedge$
 $(G, \text{sig} \vdash \text{new hides } \text{old}$
 $\longrightarrow (\text{accmodi } \text{old} \leq \text{accmodi } \text{new} \wedge$
 $\text{is-static } \text{old}))))$
 $\rrbracket \implies P$
 $\rrbracket \implies P$
by (*unfold wf-cdecl-def*) *simp*

lemma *wf-cdecl-unique*:
 $\text{wf-cdecl } G \ (C, c) \implies \text{unique } (\text{cfields } c) \wedge \text{unique } (\text{methods } c)$
apply (*unfold wf-cdecl-def*)
apply *auto*
done

lemma *wf-cdecl-fdecl*:
 $\llbracket \text{wf-cdecl } G \ (C, c); f \in \text{set } (\text{cfields } c) \rrbracket \implies \text{wf-fdecl } G \ (\text{pid } C) \ f$
apply (*unfold wf-cdecl-def*)
apply *auto*
done

lemma *wf-cdecl-mdecl*:
 $\llbracket \text{wf-cdecl } G \ (C, c); m \in \text{set } (\text{methods } c) \rrbracket \implies \text{wf-mdecl } G \ C \ m$
apply (*unfold wf-cdecl-def*)
apply *auto*
done

lemma *wf-cdecl-impD*:
 $\llbracket \text{wf-cdecl } G \ (C, c); I \in \text{set } (\text{superIfs } c) \rrbracket$
 $\implies \text{is-acc-iface } G \ (\text{pid } C) \ I \wedge$
 $(\forall s. \forall im \in \text{imethds } G \ I \ s.$
 $(\exists cm \in \text{methd } G \ C \ s: G \vdash \text{resTy } cm \preceq \text{resTy } im \wedge \neg \text{is-static } cm \wedge$
 $\text{accmodi } im \leq \text{accmodi } cm)))$
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 overridesS 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) overridesS 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
]] ⇒ accmodi old ≤ access 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: hides-eq-sigD simp add: msig-def)
done

```

```

lemma wf-cdecl-wt-init:
wf-cdecl G (C, c) ⇒ (|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* \Rightarrow *bool* **where**
wf-prog *G* = (let *is* = *ifaces* *G*; *cs* = *classes* *G* in
 ObjectC \in *set cs* \wedge
 ($\forall m \in \text{set } \text{Object-mdecls}. \text{accmodi } m \neq \text{Package}$) \wedge
 ($\forall xn. \text{SXcptC } xn \in \text{set } cs$) \wedge
 ($\forall i \in \text{set } is. \text{wf-idecl } G \ i$) \wedge *unique is* \wedge
 ($\forall c \in \text{set } cs. \text{wf-cdecl } G \ c$) \wedge *unique cs*)

lemma *wf-prog-idecl*: $\llbracket \text{iface } G \ I = \text{Some } i; \text{wf-prog } G \rrbracket \Longrightarrow \text{wf-idecl } G \ (I, i)$

apply (*unfold wf-prog-def Let-def*)

apply *simp*

apply (*fast dest: map-of-SomeD*)

done

lemma *wf-prog-cdecl*: $\llbracket \text{class } G \ C = \text{Some } c; \text{wf-prog } G \rrbracket \Longrightarrow \text{wf-cdecl } G \ (C, c)$

apply (*unfold wf-prog-def Let-def*)

apply *simp*

apply (*fast dest: map-of-SomeD*)

done

lemma *wf-prog-Object-mdecls*:

wf-prog *G* \Longrightarrow ($\forall m \in \text{set } \text{Object-mdecls}. \text{accmodi } m \neq \text{Package}$)

apply (*unfold wf-prog-def Let-def*)

apply *simp*

done

lemma *wf-prog-acc-superD*:

$\llbracket \text{wf-prog } G; \text{class } G \ C = \text{Some } c; C \neq \text{Object} \rrbracket$

$\Longrightarrow \text{is-acc-class } G \ (\text{pid } C) \ (\text{super } c)$

by (*auto dest: wf-prog-cdecl wf-cdecl-supD*)

lemma *wf-ws-prog* [*elim!*, *simp*]: *wf-prog* *G* \Longrightarrow *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* \Longrightarrow

class *G* *Object* = *Some* ($\llbracket \text{access} = \text{Public}, \text{cfields} = [], \text{methods} = \text{Object-mdecls},$
 init = *Skip*, *super* = *undefined*, *superIfs* = $[] \rrbracket$)

apply (*unfold wf-prog-def Let-def ObjectC-def*)

apply (*fast dest!:: map-of-SomeI*)

done

lemma *methd-Object* [*simp*]: *wf-prog* *G* \Longrightarrow *methd* *G* *Object* =

table-of (*map* ($\lambda(s, m). (s, \text{Object}, m)$) *Object-mdecls*)

apply (*subst methd-rec*)

apply (*auto simp add: Let-def*)

done

lemma *wf-prog-Object-methd*:

$\llbracket wf\text{-prog } G; methd\ G\ Object\ sig = Some\ m \rrbracket \implies accmodi\ m \neq Package$
by (auto dest!: *wf-prog-Object-mdecls*) (auto dest!: *map-of-SomeD*)

lemma *wf-prog-Object-is-public[intro]*:

$wf\text{-prog } G \implies is\text{-public } G\ Object$
by (auto simp add: *is-public-def dest: class-Object*)

lemma *class-SXcpt [simp]*:

$wf\text{-prog } G \implies$
 $class\ G\ (SXcpt\ xn) = Some\ (\text{access} = Public, cfields = [], methods = SXcpt\text{-mdecls},$
 $init = Skip,$
 $super = if\ xn = Throwable\ then\ Object$
 $else\ SXcpt\ Throwable,$
 $superIfs = [])$

apply (*unfold wf-prog-def Let-def SXcptC-def*)

apply (*fast dest!: map-of-SomeI*)

done

lemma *wf-ObjectC [simp]*:

$wf\text{-cdecl } G\ ObjectC = (\neg is\text{-iface } G\ Object \wedge Ball\ (set\ Object\text{-mdecls})$
 $(wf\text{-mdecl } G\ Object) \wedge unique\ Object\text{-mdecls})$
apply (*unfold wf-cdecl-def ws-cdecl-def ObjectC-def*)
apply (*auto intro: da.Skip*)
done

lemma *Object-is-class [simp,elim!]*: $wf\text{-prog } G \implies is\text{-class } G\ Object$

apply (*simp (no-asm-simp)*)

done

lemma *Object-is-acc-class [simp,elim!]*: $wf\text{-prog } G \implies is\text{-acc-class } G\ S\ Object$

apply (*simp (no-asm-simp) add: is-acc-class-def is-public-def*
 $accessible\text{-in-RefT-simp}$)

done

lemma *SXcpt-is-class [simp,elim!]*: $wf\text{-prog } G \implies is\text{-class } G\ (SXcpt\ xn)$

apply (*simp (no-asm-simp)*)

done

lemma *SXcpt-is-acc-class [simp,elim!]*:

$wf\text{-prog } G \implies is\text{-acc-class } G\ S\ (SXcpt\ xn)$
apply (*simp (no-asm-simp) add: is-acc-class-def is-public-def*
 $accessible\text{-in-RefT-simp}$)

done

lemma *fields-Object [simp]*: $wf\text{-prog } G \implies DeclConcepts.fields\ G\ Object = []$

by (*force intro: fields-emptyI*)

lemma *accfield-Object [simp]*:

$wf\text{-prog } G \implies accfield\ G\ S\ Object = Map.empty$
apply (*unfold accfield-def*)
apply (*simp (no-asm-simp) add: Let-def*)
done

lemma *fields-Throwable [simp]*:

$wf\text{-prog } G \implies DeclConcepts.fields\ G\ (SXcpt\ Throwable) = []$
by (*force intro: fields-emptyI*)

```

lemma fields-SXcpt [simp]: wf-prog G  $\implies$  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]:  $\llbracket G \vdash U \preceq T; G \vdash S \preceq U; \text{wf-prog } G \rrbracket \implies G \vdash S \preceq T$ 
apply (erule (2) widen-trans)
done

```

```

lemma Xcpt-subcls-Throwable [simp]:
wf-prog G  $\implies G \vdash \text{SXcpt } xn \preceq_C \text{SXcpt } \text{Throwable}$ 
apply (rule SXcpt-subcls-Throwable-lemma)
apply auto
done

```

```

lemma unique-fields:
 $\llbracket \text{is-class } G \ C; \text{wf-prog } G \rrbracket \implies \text{unique } (\text{DeclConcepts.fields } G \ C)$ 
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:
 $\llbracket \text{table-of } (\text{DeclConcepts.fields } G \ C) \text{ fn} = \text{Some } f; G \vdash D \preceq_C C; \text{is-class } G \ D; \text{wf-prog } G \rrbracket$ 
 $\implies \text{table-of } (\text{DeclConcepts.fields } G \ D) \text{ fn} = \text{Some } f$ 
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]:
 $\llbracket \text{table-of } (\text{DeclConcepts.fields } G \ C) \text{ m} = \text{Some } f; \text{wf-prog } G; \text{is-class } G \ C \rrbracket \implies$ 
 $\text{is-type } G \ (\text{type } f)$ 
apply (frule wf-ws-prog)
apply (force dest: fields-declC [THEN conjunct1])
 $\text{wf-prog-cdecl [THEN wf-cdecl-fdecl]}$ 
 $\text{simp add: wf-fdecl-def2 is-acc-type-def}$ 
done

```

```

lemma imethds-wf-mhead [rule-format (no-asm)]:
 $\llbracket m \in \text{imethds } G \ I \text{ sig}; \text{wf-prog } G; \text{is-iface } G \ I \rrbracket \implies$ 
 $\text{wf-mhead } G \ (\text{pid } (\text{decliface } m)) \text{ sig } (\text{mthd } m) \wedge$ 
 $\neg \text{is-static } m \wedge \text{accmodi } m = \text{Public}$ 
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:
 $\llbracket \text{methd } G \ C \text{ sig} = \text{Some } m; \text{wf-prog } G; \text{class } G \ C = \text{Some } y \rrbracket \implies$ 
 $G \vdash C \preceq_C (\text{declclass } m) \wedge \text{is-class } G \ (\text{declclass } m) \wedge$ 

```

```

wf-mdecl G (declclass m) (sig,(methd 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]]
 $\implies$  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]]
 $\implies$  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]]
 $\implies$  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  $\in$  imethds G I sig;wf-prog G; is-iface G I]]
 $\implies$   $\neg$ is-static im  $\wedge$  accmodi im = Public
proof -
  assume asm: wf-prog G is-iface G I im  $\in$  imethds G I sig

  have wf-prog G  $\longrightarrow$ 
    ( $\forall$  i im. iface G I = Some i  $\longrightarrow$  im  $\in$  imethds G I sig
       $\longrightarrow$   $\neg$ is-static im  $\wedge$  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:  $\bigwedge$  i J. iface G I = Some i  $\implies$  ws-prog G  $\implies$  J  $\in$  set (isuperIfs i)
       $\implies$  ?P G J
    assume wf: wf-prog G and if-I: iface G I = Some i and
      im: im  $\in$  imethds G I sig
    show  $\neg$ is-static im  $\wedge$  accmodi im = Public
    proof -
      let ?inherited = Un-tables (imethds G ' set (isuperIfs i))
      let ?new = (set-option  $\circ$  table-of (map ( $\lambda$ (s, mh). (s, I, mh)) (imethds i)))
      from if-I wf im have imethds:im  $\in$  (?inherited  $\oplus \oplus$  ?new) sig
      by (simp add: imethds-rec)
      from wf if-I have

```



```

  wf-supI:  $\forall J. J \in \text{set } (\text{isuperIfs } i) \longrightarrow (\exists j. \text{iface } G \ J = \text{Some } j)$ 
  by (blast dest: wf-prog-idecl wf-idecl-supD is-acc-ifaceD)
from wf if-I have
   $\forall im \in \text{set } (\text{imethods } i). \neg \text{is-static } im \wedge \text{accmodi } im = \text{Public}$ 
  by (auto dest!: wf-prog-idecl wf-idecl-mhead)
then have new-ok:  $\forall im. \text{table-of } (\text{imethods } i) \ \text{sig} = \text{Some } im$ 
   $\longrightarrow \neg \text{is-static } im \wedge \text{accmodi } im = \text{Public}$ 
  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:  $G \vdash \text{new hides old}$  and wf: wf-prog G
  shows
     $\text{accmodi old} \leq \text{accmodi new} \wedge$ 
     $\text{is-static old}$ 
proof -
  from hides
  obtain c where
    clsNew:  $\text{class } G \ (\text{declclass new}) = \text{Some } c$  and
    neqObj:  $\text{declclass new} \neq \text{Object}$ 
  by (auto dest: hidesD declared-in-classD)
with hides obtain newM oldM where
    newM:  $\text{table-of } (\text{methods } c) \ (\text{msig new}) = \text{Some newM}$  and
    new:  $\text{new} = (\text{declclass new}, (\text{msig new}), \text{newM})$  and
    old:  $\text{old} = (\text{declclass old}, (\text{msig old}), \text{oldM})$  and
    msig new = msig old
  by (cases new, cases old)
  (auto dest: hidesD
    simp add: cdeclaredmethd-def declared-in-def)
with hides
have hides':
   $G, (\text{msig new}) \vdash (\text{declclass new}, \text{newM}) \text{ hides } (\text{declclass old}, \text{oldM})$ 
  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 $G \vdash \text{new overrides}_S \text{old}$ with the definition of dynamic overriding $G \vdash \text{new overrides old}$. Conforming result types and restrictions on the access modifiers of the old and the new method are not part of the predicate for static overriding. But they are enshured in a wellformed program. Dynamic overriding has no restrictions on the access modifiers but enforces conform result types as precondition. But with some effort we can guarantee the access

modifier restriction for dynamic overriding, too. See lemma *wf-prog-dyn-override-prop*.

lemma *wf-prog-stat-overridesD*:

assumes *stat-override*: $G \vdash \text{new overrides}_S \text{ old}$ **and** *wf*: *wf-prog* *G*

shows

$G \vdash \text{resTy new} \preceq \text{resTy old} \wedge$
 $\text{accmodi old} \leq \text{accmodi new} \wedge$
 $\neg \text{is-static old}$

proof –

from *stat-override*

obtain *c* **where**

clsNew: *class* *G* (*declclass new*) = *Some c* **and**

neqObj: *declclass new* \neq *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'*:

$G, (\text{msig new}) \vdash (\text{declclass new}, \text{newM}) \text{ overrides}_S (\text{declclass old}, \text{oldM})$

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*: $G \vdash \text{new overrides}_S \text{ old}$ **and** *wf* : *wf-prog* *G*

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

proof –

from *stat-override*

show *?thesis* (**is** *?Overrides new old*)

proof (*induct*)

case (*Direct new old superNew*)

then have *stat-override*: $G \vdash \text{new overrides}_S \text{ old}$

by (*rule stat-overridesR.Direct*)

from *stat-override wf*

have *resTy-widen*: $G \vdash \text{resTy new} \preceq \text{resTy old}$ **and**

not-static-old: $\neg \text{is-static old}$

by (*auto dest*: *wf-prog-stat-overridesD*)

have *not-private-new*: *accmodi new* \neq *Private*

proof –

from *stat-override*

have *accmodi old* \neq *Private*

by (*rule no-Private-stat-override*)

moreover

from *stat-override wf*

have *accmodi old* \leq *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:  $G \vdash \text{Method old inheritable-in (pid C)}$  and
    accmodi-old:  $\text{accmodi old} \neq \text{Package}$  and
    instance-method:  $\neg \text{is-static old}$  and
    subcls:  $G \vdash C \prec_C \text{declclass old}$  and
    old-declared:  $G \vdash \text{Method old declared-in (declclass old)}$  and
    wf: wf-prog G
  shows  $G \vdash \text{Method old member-of C} \vee$ 
    ( $\exists \text{ new. } G \vdash \text{new overrides}_S \text{ old} \wedge G \vdash \text{Method new member-of C}$ )
proof -
  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  $G \vdash \text{Object} \prec_C \text{declclass old}$ 
  then show ?P Object old
    by blast
next
  case (Subcls C c)
  assume cls-C:  $\text{class G C} = \text{Some c}$  and
    neq-C-Obj:  $C \neq \text{Object}$  and
    hyp:  $\llbracket G \vdash \text{Method old inheritable-in pid (super c); } G \vdash \text{super c} \prec_C \text{declclass old} \rrbracket \implies ?P (\text{super c}) \text{ old}$  and
    inheritable:  $G \vdash \text{Method old inheritable-in pid C}$  and
    subclsC:  $G \vdash C \prec_C \text{declclass old}$ 
  from cls-C neq-C-Obj
  have super:  $G \vdash C \prec_{C1} \text{super c}$ 
    by (rule subcls1I)
  from wf cls-C neq-C-Obj
  have accessible-super:  $G \vdash (\text{Class (super c)}) \text{ accessible-in (pid C)}$ 
    by (auto dest: wf-prog-cdecl wf-cdecl-supD is-acc-classD)
  have hyp-member-super: ?P C old
    if member-super:  $G \vdash \text{Method old member-of (super c)}$ 
    and inheritable:  $G \vdash \text{Method old inheritable-in pid C}$ 
    and instance-method:  $\neg \text{is-static old}$ 
  for old
proof -
  from member-super
  have old-declared:  $G \vdash \text{Method old declared-in (declclass old)}$ 
    by (cases old) (auto dest: member-of-declC)
  show ?thesis
proof (cases  $G \vdash \text{mid (msig old) undeclared-in C}$ )

```

```

case True
with inheritable super accessible-super member-super
have  $G \vdash \text{Method old member-of } C$ 
  by (cases old) (auto intro: members.Inherited)
then show ?thesis
  by auto
next
case False
then obtain new-member where
   $G \vdash \text{new-member declared-in } C$  and
   $\text{mid}(\text{msig old}) = \text{memberid new-member}$ 
  by (auto dest: not-undeclared-declared)
then obtain new where
   $\text{new: } G \vdash \text{Method new declared-in } C$  and
   $\text{eq-sig: msig old} = \text{msig new}$  and
   $\text{declC-new: declclass new} = C$ 
  by (cases new-member) auto
then have member-new:  $G \vdash \text{Method new member-of } C$ 
  by (cases new) (auto intro: members.Immediate)
from declC-new super member-super
have subcls-new-old:  $G \vdash \text{declclass new} \prec_C \text{declclass old}$ 
  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  $G \vdash \text{new overrides}_S \text{ old}$ 
    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  $G \vdash \text{new hides old}$ 
  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
from subclsC cls-C
have  $G \vdash (\text{super } c) \preceq_C \text{declclass old}$ 
  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  $G \vdash \text{Method old member-of } (\text{super } c)$ 
    by (cases old) (auto intro: members.Immediate)
  with inheritable instance-method
  show ?thesis

```

```

    by (blast dest: hyp-member-super)
next
case Subcls
assume  $G \vdash \text{super } c \prec_C \text{ declclass old}$ 
moreover
from inheritable accmodi-old
have  $G \vdash \text{Method old inheritable-in pid (super c)}$ 
  by (cases accmodi old) (auto simp add: inheritable-in-def)
ultimately
have  $?P \text{ (super c) old}$ 
  by (blast dest: hyp)
then show ?thesis
proof
  assume  $G \vdash \text{Method old member-of super c}$ 
  with inheritable instance-method
  show ?thesis
    by (blast dest: hyp-member-super)
next
assume  $\exists \text{new. } G \vdash \text{new overrides}_S \text{ old} \wedge G \vdash \text{Method new member-of super c}$ 
then obtain super-new where
  super-new-override:  $G \vdash \text{super-new overrides}_S \text{ old}$  and
  super-new-member:  $G \vdash \text{Method super-new member-of super c}$ 
  by blast
from super-new-override wf
have  $\text{accmodi old} \leq \text{accmodi super-new}$ 
  by (auto dest: wf-prog-stat-overridesD)
with inheritable accmodi-old
have  $G \vdash \text{Method super-new inheritable-in pid C}$ 
  by (auto simp add: inheritable-in-def
    split: acc-modi.splits
    dest: acc-modi-le-Dests)
moreover
from super-new-override
have  $\neg \text{is-static super-new}$ 
  by (auto dest: stat-overrides-commonD)
moreover
note super-new-member
ultimately have  $?P \text{ C super-new}$ 
  by (auto dest: hyp-member-super)
then show ?thesis
proof
  assume  $G \vdash \text{Method super-new member-of C}$ 
  with super-new-override
  show ?thesis
    by blast
next
assume  $\exists \text{new. } G \vdash \text{new overrides}_S \text{ super-new} \wedge$ 
   $G \vdash \text{Method new member-of C}$ 
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:  $G \vdash \text{Method old inheritable-in (pid C)}$  and
    accmodi-old:  $\text{accmodi old} \neq \text{Package}$  and

```

instance-method: \neg *is-static old* **and**
subcls: $G \vdash C \prec_C \text{declclass old}$ **and**
old-declared: $G \vdash \text{Method old declared-in (declclass old)}$ **and**
wf: *wf-prog G*
obtains (*Inheritance*) $G \vdash \text{Method old member-of } C$
| (*Overriding*) *new* **where** $G \vdash \text{new overrides}_S \text{ old}$ **and** $G \vdash \text{Method new member-of } C$
proof –
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*: $G \vdash \text{new overrides old}$ **and**
accmodi-old: $\text{accmodi old} \neq \text{Package}$ **and**
wf: *wf-prog G*
shows $G \vdash \text{new overrides}_S \text{ old}$
proof –
from *dyn-override accmodi-old*
show *?thesis (is ?Overrides new old)*
proof (*induct rule: overridesR.induct*)
case (*Direct new old*)
assume *new-declared*: $G \vdash \text{Method new declared-in declclass new}$
assume *eq-sig-new-old*: $\text{msig new} = \text{msig old}$
assume *subcls-new-old*: $G \vdash \text{declclass new} \prec_C \text{declclass old}$
assume $G \vdash \text{Method old inheritable-in pid (declclass new)}$ **and**
accmodi old \neq *Package* **and**
 \neg *is-static old* **and**
 $G \vdash \text{declclass new} \prec_C \text{declclass old}$ **and**
 $G \vdash \text{Method old declared-in declclass old}$
from *this wf*
show *?Overrides new old*
proof (*cases rule: non-Package-instance-method-inheritance-cases*)
case *Inheritance*
assume $G \vdash \text{Method old member-of declclass new}$
then have $G \vdash \text{mid (msig old) undeclared-in declclass new}$
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'*: $G \vdash \text{new' overrides}_S \text{ old}$
then have $\text{msig new'} = \text{msig old}$
by (*auto dest: stat-overrides-commonD*)
with *eq-sig-new-old* **have** *eq-sig-new-new'*: $\text{msig new} = \text{msig new'}$
by *simp*
assume $G \vdash \text{Method new' member-of declclass new}$
then show *?thesis*
proof (*cases*)

```

    case Immediate
    then have declC-new: declclass new' = declclass new
      by auto
    from Immediate
    have  $G \vdash \text{Method new' declared-in declclass new}$ 
      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  $G \vdash \text{mid (msig new') undeclared-in declclass new}$ 
    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  $\neq$  Package
assume accmodi old  $\neq$  Package  $\implies G \vdash \text{inter overrides}_S \text{ old}$ 
with accmodi-old
have stat-override-inter-old:  $G \vdash \text{inter overrides}_S \text{ old}$ 
  by blast
moreover
assume hyp-inter: accmodi inter  $\neq$  Package  $\implies G \vdash \text{new overrides}_S \text{ inter}$ 
moreover
have accmodi inter  $\neq$  Package
proof -
  from stat-override-inter-old wf
  have accmodi old  $\leq$  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:  $G \vdash \text{new overrides old}$  and
    wf: wf-prog G
  shows accmodi old  $\leq$  accmodi new
proof (cases accmodi old = Package)
case True
note old-Package = this
show ?thesis
proof (cases accmodi old  $\leq$  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  $G \vdash \text{new overrides}_S \text{ old}$ 
  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:  $G \vdash \text{new overrides old}$  and
    accmodi-new:  $\text{accmodi new} = \text{Package}$  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  $G \vdash \text{new overrides}_S \text{ old}$ 
    by (auto intro: dynamic-to-static-overriding)
  with wf
  have  $\text{accmodi old} \leq \text{accmodi new}$ 
    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  $G \vdash \text{new overrides}_S \text{ old}$ 
    by (auto intro: dynamic-to-static-overriding)
  with wf
  have  $\text{accmodi old} \leq \text{accmodi new}$ 
    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:  $G \vdash \text{new overrides old}$  and
    accmodi-old:  $\text{accmodi old} = \text{Package}$  and
    accmodi-new:  $\text{accmodi new} = \text{Package}$  and
    wf: wf-prog  $G$ 
  shows 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*
 $G \vdash \text{Method old inheritable-in pid (declclass new)}$
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 $G \vdash \text{new overrides inter}$
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: $G \vdash \text{new overrides old}$* **and**
 accmodi-old: accmodi old = Package **and**
 outside-pack: pid (declclass old) \neq pid (declclass new) **and**
 wf: wf-prog G
shows $\exists \text{ inter. } G \vdash \text{new overrides inter} \wedge G \vdash \text{inter overrides old} \wedge$
 $\text{pid (declclass old) = pid (declclass inter)} \wedge$
 $\text{Protected} \leq \text{accmodi inter}$

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 $G \vdash \text{Method old inheritable-in pid (declclass new)}$
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: $G \vdash \text{new overrides inter}$*
assume *override-inter-old: $G \vdash \text{inter overrides old}$*
assume *hyp-new-inter: $\llbracket \text{accmodi inter} = \text{Package};$*
 $\text{pid (declclass inter)} \neq \text{pid (declclass new)} \rrbracket$
 $\implies ?P \text{ new inter}$
assume *hyp-inter-old: $\llbracket \text{accmodi old} = \text{Package};$*
 $\text{pid (declclass old)} \neq \text{pid (declclass inter)} \rrbracket$
 $\implies ?P \text{ inter old}$
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:  $G \vdash \text{new overrides newinter}$  and
      over-newinter-inter:  $G \vdash \text{newinter overrides inter}$  and
      eq-pid:  $\text{pid}(\text{declclass } \text{inter}) = \text{pid}(\text{declclass } \text{newinter})$  and
      accmodi-newinter:  $\text{Protected} \leq \text{accmodi } \text{newinter}$ 
      by auto
    from over-newinter-inter override-inter-old
    have  $G \vdash \text{newinter overrides old}$ 
    by (rule overridesR.Indirect)
    moreover
    from eq-pid same-pack-old-inter
    have  $\text{pid}(\text{declclass } \text{old}) = \text{pid}(\text{declclass } \text{newinter})$ 
    by simp
    moreover
    note over-new-newinter accmodi-newinter
    ultimately show ?thesis
    by blast
  next
  case False
  with override-new-inter
  have  $\text{Protected} \leq \text{accmodi } \text{inter}$ 
  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:  $G \vdash \text{inter overrides newinter}$  and
    over-newinter-old:  $G \vdash \text{newinter overrides old}$  and
    eq-pid:  $\text{pid}(\text{declclass } \text{old}) = \text{pid}(\text{declclass } \text{newinter})$  and
    accmodi-newinter:  $\text{Protected} \leq \text{accmodi } \text{newinter}$ 
    by auto
  from override-new-inter over-inter-newinter
  have  $G \vdash \text{new overrides newinter}$ 
  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
 $\longrightarrow (\forall c\ m. \text{class } G\ C = \text{Some } c \longrightarrow \text{methd } G\ C\ \text{sig} = \text{Some } m$
 $\longrightarrow G \vdash C \preceq_C \text{declclass } m) \text{ (is ?P } G\ C)$
proof (*induct G C rule: class-rec-induct', intro allI impI*)
fix *G C c m*
assume *Hyp*: $\bigwedge c. \text{class } G\ C = \text{Some } c \implies \text{ws-prog } G \implies C \neq \text{Object}$
 $\implies ?P\ G\ (\text{super } c)$
assume *wf*: *wf-prog G* **and** *cls-C*: *class G C = Some c* **and**
m: *methd G C sig = Some m*
show $G \vdash C \preceq_C \text{declclass } m$
proof (*cases C=Object*)
case *True*
with *wf m* **show** *?thesis* **by** (*simp add: methd-Object-SomeD*)
next
let *?filter*=*filter-tab* ($\lambda \text{sig } m. G \vdash C \text{ inherits method sig } m$)
let *?table* = *table-of* (*map* ($\lambda(s, m). (s, C, m)$) (*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** $G \vdash \text{super } c \preceq_C \text{declclass } m$
by (*auto intro: Hyp [rule-format]*)
moreover **from** *cls-C False* **have** $G \vdash C \prec_C 1 \text{ super } c$ **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*:

$\llbracket \text{wf-prog } G; \text{methd } G\ \text{Object } \text{sig} = \text{Some } m \rrbracket \implies \text{declclass } m = \text{Object}$
by *auto*

lemma *methd-declaredD*:

$\llbracket \text{wf-prog } G; \text{is-class } G\ C; \text{methd } G\ C\ \text{sig} = \text{Some } m \rrbracket$
 $\implies G \vdash (\text{mdecl } (\text{sig}, \text{methd } m)) \text{ declared-in } (\text{declclass } m)$
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*
 $\implies G \vdash (\text{mdecl } (\text{sig}, \text{methd } m)) \text{ declared-in } (\text{declclass } m)$

```

proof (induct C rule: ws-class-induct')
  case Object
  show ?thesis if methd G Object sig = Some m
    by (rule method-declared-inI) (use wf that in 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  $\implies$  ?thesis
  let ?newMethods = table-of (map ( $\lambda(s, m).$  (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  $\neq$  Object
  obtains (NewMethod) table-of (map ( $\lambda(s, m).$  (s, C, m)) (methods c)) sig = Some m
    | (InheritedMethod) G  $\vdash$  C inherits (method sig m) and methd G (super c) sig = Some m
proof –
  let ?inherited = filter-tab ( $\lambda sig m.$  G  $\vdash$  C inherits method sig m)
    (methd G (super c))
  let ?new = table-of (map ( $\lambda(s, m).$  (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
     $\llbracket \text{is-class } G \ C; \text{methd } G \ C \text{ sig} = \text{Some } m \rrbracket \implies G \vdash \text{Methd sig } m \text{ member-of } C$ 
    (is ?Class C  $\implies$  ?Method C  $\implies$  ?MemberOf C)
proof –

```

```

from wf have ws: ws-prog G ..
assume defC: is-class G C
from defC ws
show ?Class C  $\implies$  ?Method C  $\implies$  ?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 $\vdash$  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  $\neq$  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  $\neq$  Object;
   methd G (super c) sig = Some old|]
   $\implies$  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 $\vdash$  Methd sig old member-of (super c)
    by (rule methd-member-of)
  from old wf is-cls-super

```

```

have old-declared:  $G \vdash \text{Methd sig old declared-in (declclass old)}$ 
  by (auto dest: methd-declared-in-declclass)
from new clsC
have new-declared:  $G \vdash \text{Methd sig (C,new) declared-in C}$ 
  by (auto intro: method-declared-inI)
note tranc1-rtranc1-tranc = tranc1-rtranc1-tranc1 [trans]
from clsC neq-C-Obj
have subcls1-C-super:  $G \vdash C \prec_C 1 \text{ super } c$ 
  by (rule subcls1I)
then have  $G \vdash C \prec_C \text{ super } c \dots$ 
also from old wf is-cls-super
have  $G \vdash \text{super } c \preceq_C (\text{declclass old})$  by (auto dest: methd-declC)
finally have subcls-C-old:  $G \vdash C \prec_C (\text{declclass old})$  .
from accmodi-old
have inheritable:  $G \vdash \text{Methd sig old inheritable-in pid C}$ 
  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  $G, \text{sig} \vdash (C, \text{new}) \text{ hides old}$ 
      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  $G, \text{sig} \vdash (C, \text{new}) \text{ overrides}_S \text{ old}$ 
      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  $G D \text{ sig} = \text{Some old}$  and
    accmodi-old: Protected  $\leq$  accmodi old and
    not-static-old:  $\neg \text{is-static old}$ 
  shows
     $\exists \text{new. methd } G C \text{ sig} = \text{Some new} \wedge$ 
       $(\text{new} = \text{old} \vee G, \text{sig} \vdash \text{new overrides}_S \text{old})$ 
    (is  $(\exists \text{new. } (?Constraint C \text{new old}))$ )
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  $G \dots$ 
  from is-cls-C ws subclseq-C-D
  show  $\exists \text{new. } ?Constraint C \text{new old}$ 
  proof (induct rule: ws-class-induct')
    case (Object co)
    then have eq-D-Obj:  $D = \text{Object}$  by auto

```

```

with old
have ?Constraint Object old old
  by auto
with eq-D-Obj
show  $\exists$  new. ?Constraint Object new old by auto
next
case (Subcls C c)
assume hyp:  $G \vdash \text{super } c \preceq_C D \implies \exists \text{new. ?Constraint (super } c) \text{ new old}$ 
assume clsC: class G C = Some c
assume neq-C-Obj:  $C \neq \text{Object}$ 
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:  $G \vdash (\text{Class (super } c)) \text{ accessible-in (pid } C) \text{ and}$ 
  subcls1-C-super:  $G \vdash C \prec_C 1 \text{ super } c$ 
  by (auto dest: wf-prog-cdecl wf-cdecl-supD is-acc-classD
    intro: subcls1I)
show  $\exists \text{new. ?Constraint } C \text{ new old}$ 
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  $\exists \text{new. ?Constraint } C \text{ new old}$  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:  $\neg \text{is-static super-method}$ 
  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:  $G \vdash \text{Methd sig super-method inheritable-in (pid } C)$ 
  by (auto dest!: wf-prog-stat-overridesD
    acc-modi-le-Dests
    simp add: inheritable-in-def)
from super wf is-cls-super
have member:  $G \vdash \text{Methd sig super-method member-of (super } c)$ 
  by (auto intro: methd-member-of)
from member
have decl-super-method:
   $G \vdash \text{Methd sig super-method declared-in (declclass super-method)}$ 
  by (auto dest: member-of-declC)
from super subcls1-C-super ws is-cls-super
have subcls-C-super:  $G \vdash C \prec_C (\text{declclass super-method})$ 
  by (auto intro: rtrancl-into-trancl2 dest: methd-declC)
show  $\exists \text{new. ?Constraint } C \text{ new old}$ 
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:  $G \vdash \text{mid } sig \text{ undeclared-in } C$ 
    by (auto simp add: undeclared-in-def cdeclaredmethd-def)
  with inheritable member superAccessible subcls1-C-super
  have inherits:  $G \vdash C \text{ inherits (method } sig \text{ super-method)}$ 
    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 ( $\lambda(s, m). (s, C, m)$ ) (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:  $\neg \text{is-static new}$ 
  by (auto dest: wf-prog-staticD)
from clsC new
have decl-new:  $G \vdash \text{Methd } sig \text{ new declared-in } C$ 
  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  $G, sig \vdash \text{new overrides}_S \text{ super-method}$ 
  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:  $G \vdash C \preceq_C D$  and
  is-cls-D: is-class G D and
  wf: wf-prog G and
  old: methd G D sig = Some old and
  accmodi-old: Protected  $\leq$  accmodi old and

```


not-static-old: \neg *is-static old*
obtains (*Inheritance*) *methd G C sig = Some old*
 | (*Overriding*) *new where methd G C sig = Some new and G, sig \vdash new overrides_S 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*: $G \vdash C \preceq_C D$ **and**
is-cls-D: *is-class G D* **and**
wf: *wf-prog G* **and**
old: *methd G D sig = Some old* **and**
accmodi-old: *Protected* \leq *accmodi old* **and**
not-static-old: \neg *is-static old*
shows
 \exists *new*. *methd G C sig = Some new* \wedge
 \neg *is-static new* \wedge $G \vdash_{\text{resTy}} \text{new} \preceq_{\text{resTy}} \text{old} \wedge \text{accmodi old} \leq \text{accmodi new}$
 (**is** (\exists *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 \neg *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'*: $x \in A \implies P x \implies \exists x \in A. P x$ **by** *blast*
lemma *ballE'*: $\forall x \in A. P x \implies (x \notin A \implies Q) \implies (P x \implies Q) \implies Q$ **by** *blast*

lemma *subint-widen-imethds*:
assumes *irel*: $G \vdash I \preceq I$
and *wf*: *wf-prog G*
and *is-iface*: *is-iface G J*
and *jm*: $\text{jm} \in \text{imethds } G \ J \ \text{sig}$
shows $\exists \text{im} \in \text{imethds } G \ I \ \text{sig. is-static im} = \text{is-static jm} \wedge$
 $\text{accmodi im} = \text{accmodi jm} \wedge$
 $G \vdash_{\text{resTy}} \text{im} \preceq_{\text{resTy}} \text{jm}$
using *irel jm*
proof (*induct rule: converse-rtrancl-induct*)
case *base*
then show ?case **by** (*blast elim: bexI'*)
next
case (*step I SI*)
from $\langle G \vdash I \prec I1 \ SI \rangle$
obtain *i* **where**
ifI: *iface G I = Some i* **and**
SI: $SI \in \text{set (isuperIfs i)}$
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*:

$\llbracket \text{sig. } \llbracket G \vdash C \rightsquigarrow 1I; \text{wf-prog } G; im \in \text{imethds } G \text{ } I \text{ sig} \rrbracket \implies$
 $\exists cm \in \text{methd } G \text{ } C \text{ sig: } \neg \text{is-static } cm \wedge \neg \text{is-static } im \wedge$
 $G \vdash \text{resTy } cm \preceq \text{resTy } im \wedge$
 $\text{accmodi } im = \text{Public} \wedge \text{accmodi } cm = \text{Public}$

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)]*:

$\llbracket \text{wf-prog } G; G \vdash C \rightsquigarrow I \rrbracket \implies \text{is-iface } G \text{ } I \longrightarrow$
 $(\forall im \in \text{imethds } G \text{ } I \text{ sig.}$
 $\exists cm \in \text{methd } G \text{ } C \text{ sig: } \neg \text{is-static } cm \wedge \neg \text{is-static } im \wedge$
 $G \vdash \text{resTy } cm \preceq \text{resTy } im \wedge$
 $\text{accmodi } im = \text{Public} \wedge \text{accmodi } cm = \text{Public})$

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-rtranc1 [THEN inheritable-instance-method-props,*
OF - implmt-is-class])

apply *auto*

done

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

$\text{emh} \in \text{mheads } G \text{ } S \text{ } t \text{ sig} \longrightarrow \text{wf-prog } G \longrightarrow$
 $(\exists C \text{ } D \text{ } m. t = \text{ClassT } C \wedge \text{declrefT } \text{emh} = \text{ClassT } D \wedge$
 $\text{accmethd } G \text{ } S \text{ } C \text{ sig} = \text{Some } m \wedge$
 $(\text{declclass } m = D) \wedge \text{mhead } (\text{mthd } m) = (\text{mhd } \text{emh})) \vee$
 $(\exists I. t = \text{IfaceT } I \wedge ((\exists im. im \in \text{accimethds } G \text{ } (pid \text{ } S) \text{ } I \text{ sig} \wedge$
 $\text{mthd } im = \text{mhd } \text{emh})) \vee$
 $(\exists m. G \vdash \text{Iface } I \text{ accessible-in } (pid \text{ } S) \wedge \text{accmethd } G \text{ } S \text{ } \text{Object sig} = \text{Some } m \wedge$
 $\text{accmodi } m \neq \text{Private} \wedge$
 $\text{declrefT } \text{emh} = \text{ClassT } \text{Object} \wedge \text{mhead } (\text{mthd } m) = \text{mhd } \text{emh}))) \vee$
 $(\exists T \text{ } m. t = \text{ArrayT } T \wedge G \vdash \text{Array } T \text{ accessible-in } (pid \text{ } S) \wedge$
 $\text{accmethd } G \text{ } S \text{ } \text{Object sig} = \text{Some } m \wedge \text{accmodi } m \neq \text{Private} \wedge$

```

    declrefT emh = ClassT Object  $\wedge$  mhead (methd 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  $\in$  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 (methd m) = mhd emh
  | (Iface-methd) I im where t = IfaceT I
    im  $\in$  accimethds G (pid S) I sig methd im = mhd emh
  | (Iface-Object-methd) I m where
    t = IfaceT I G  $\vdash$  Iface I accessible-in (pid S)
    accmethd G S Object sig = Some m accmodi m  $\neq$  Private
    declrefT emh = ClassT Object mhead (methd m) = mhd emh
  | (Array-Object-methd) T m where
    t = ArrayT T G  $\vdash$  Array T accessible-in (pid S)
    accmethd G S Object sig = Some m accmodi m  $\neq$  Private
    declrefT emh = ClassT Object mhead (methd m) = mhd emh
  using assms by (blast dest!: mheadsD)

```

lemma declclassD[rule-format]:

```

   $\llbracket$  wf-prog G; class G C = Some c; methd G C sig = Some m;
  class G (declclass m) = Some d  $\rrbracket$ 
   $\impl$  table-of (methods d) sig = Some (methd m)
  proof -
    assume wf: wf-prog G
    then have ws: ws-prog G ..
    assume clsC: class G C = Some c
    from clsC ws
    show  $\bigwedge m d. \llbracket$  methd G C sig = Some m; class G (declclass m) = Some d  $\rrbracket$ 
       $\impl$  table-of (methods d) sig = Some (methd 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 ( $\lambda(s, m). (s, C, m)$ ) (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:  $G \vdash C \preceq_C \text{Object}$ 
    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  $\neq$  Private and
    new: new  $\in$  imethds G I sig
  shows  $G \vdash \text{resTy new} \preceq \text{resTy old} \wedge \text{is-static new} = \text{is-static old}$  (is ?P new)
proof -
  from wf have ws: ws-prog G by simp
  have hyp-newmethod: ?P (I,new)
    if ifI: iface G I = Some i
    and new: table-of (imethds i) sig = Some new
    for I i new
  using ifI new not-private wf old
    by (auto dest!: wf-prog-idecl wf-idecl-hiding cond-hiding-entailsD
      simp del: methd-Object)
  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  $\in$  imethds G I sig
  from Step
  have hyp:  $\forall J \in \text{set (isuperIfs i)}. (\text{new} \in \text{imethds G J sig} \longrightarrow ?P \text{ new})$ 
    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  $\in$  set (isuperIfs i)
    new  $\in$  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 $G \vdash \text{Class } \text{dynC} \preceq \text{statT}$. Unfortunately this ordinary widening relation isn't enough to describe the valid lookup classes, since we must cope the special cases of arrays and interfaces, too. If we statically expect an array or interface we may lookup a field or a method in Object which isn't covered in the widening relation.

statT	field	instance	method	static (class)	method
NullT	/	/	/	Iface	dynC Object
Class	dynC	dynC	dynC	dynC	Array / Object
Object	Object	Object	Object	Object	Object

In most cases we can lookup the member in the dynamic class. But as an interface can't declare new static methods, nor an array can define new methods at all, we have to lookup methods in the base class Object.

The limitation to classes in the field column is artificial and comes out of the typing rule for the field access (see rule $FVar$ in the welltyping relation wt in theory WellType). It stems out of the fact, that Object indeed has no non private fields. So interfaces and arrays can actually have no fields at all and a field access would be senseless. (In Java interfaces are allowed to declare new fields but in current Bali not!). So there is no principal reason why we should not allow Objects to declare non private fields. Then we would get the following column:

statT	field	NullT	/	Iface	Object	Class	dynC	Array	Object
----------------	-------	-------	---	-------	--------	-------	------	-------	--------

primrec *valid-lookup-cl*:: $\text{prog} \Rightarrow \text{ref-ty} \Rightarrow \text{qname} \Rightarrow \text{bool} \Rightarrow \text{bool}$
 $(\langle -, - \rangle \vdash \text{valid}'\text{-lookup-cl}'\text{-for} \rightarrow [61, 61, 61, 61] \ 60)$

where

```

  G, NullT    ⊢ dynC valid-lookup-cl-for static-membr = False
| G, IfaceT I ⊢ dynC valid-lookup-cl-for static-membr
  = (if static-membr
     then dynC=Object
     else G ⊢ Class dynC ≤ Iface I)
| G, ClassT C ⊢ dynC valid-lookup-cl-for static-membr = G ⊢ Class dynC ≤ Class C
| G, ArrayT T ⊢ dynC valid-lookup-cl-for static-membr = (dynC=Object)

```

lemma *valid-lookup-cl-is-class*:

assumes $\text{dynC}: G, \text{statT} \vdash \text{dynC valid-lookup-cl-for static-membr}$ **and**
 $\text{ty-statT}: \text{isrtype } G \ \text{statT}$ **and**
 $\text{wf}: \text{wf-prog } G$

shows $\text{is-class } G \ \text{dynC}$

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  $\langle \text{map-theory-simpset } (\text{fn } \text{ctxt} \Rightarrow \text{ctxt} |> \text{Simplifier.del-loop split-all-tac}) \rangle$ 
setup  $\langle \text{map-theory-claset } (\text{fn } \text{ctxt} \Rightarrow \text{ctxt delSWrapper split-all-tac}) \rangle$ 

lemma dynamic-mheadsD:
 $\llbracket \text{emh} \in \text{mheads } G \text{ } S \text{ } \text{statT } \text{sig};$ 
 $G, \text{statT} \vdash \text{dynC valid-lookup-cls-for } (\text{is-static emh});$ 
 $\text{isrtype } G \text{ } \text{statT}; \text{wf-prog } G$ 
 $\rrbracket \implies \exists m \in \text{dynlookup } G \text{ } \text{statT } \text{dynC } \text{sig}:$ 
 $\text{is-static } m = \text{is-static emh} \wedge G \vdash \text{resTy } m \preceq \text{resTy emh}$ 
proof –
  assume  $\text{emh}: \text{emh} \in \text{mheads } G \text{ } S \text{ } \text{statT } \text{sig}$ 
  and  $\text{wf}: \text{wf-prog } G$ 
  and  $\text{dynC-Prop}: G, \text{statT} \vdash \text{dynC valid-lookup-cls-for } (\text{is-static emh})$ 
  and  $\text{istype}: \text{isrtype } G \text{ } \text{statT}$ 
  from dynC-Prop istype wf
  obtain y where
    dynC:  $\text{class } G \text{ } \text{dynC} = \text{Some } y$ 
    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  $\text{statC}: \text{statT} = \text{ClassT } \text{statC}$ 
  assume  $\text{accmethd } G \text{ } S \text{ } \text{statC } \text{sig} = \text{Some } \text{sm}$ 
  then have  $\text{sm}: \text{methd } G \text{ } \text{statC } \text{sig} = \text{Some } \text{sm}$ 
    by (blast dest: accmethd-SomeD)
  assume eq-mheads:  $\text{mhead } (\text{methd } \text{sm}) = \text{mhd emh}$ 
  from statC
  have dynlookup:  $\text{dynlookup } G \text{ } \text{statT } \text{dynC } \text{sig} = \text{dynmethd } G \text{ } \text{statC } \text{dynC } \text{sig}$ 
    by (simp add: dynlookup-def)
  from wf statC istype dynC-Prop sm
  obtain dm where
     $\text{dynmethd } G \text{ } \text{statC } \text{dynC } \text{sig} = \text{Some } \text{dm}$ 
     $\text{is-static } \text{dm} = \text{is-static } \text{sm}$ 
     $G \vdash \text{resTy } \text{dm} \preceq \text{resTy } \text{sm}$ 
    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  $\text{statI}: \text{statT} = \text{IfaceT } I$  and
    eq-mheads:  $\text{methd } \text{im} = \text{mhd emh}$  and
     $\text{im} \in \text{accimethds } G \text{ } (\text{pid } S) \text{ } I \text{ } \text{sig}$ 
  then have im:  $\text{im} \in \text{imethds } G \text{ } I \text{ } \text{sig}$ 
    by (blast dest: accimethdsD)
  with istype statI eq-mheads wf
  have not-static-emh:  $\neg \text{is-static emh}$ 
    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 (methd dm) ≤ resTy (methd 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 (methd 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 (methd dm) ≤ resTy (methd 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 (methd im) ≤ resTy (methd 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 (methd dm) ≤ resTy (methd 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 (methd 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 |> Simplifier.add-loop (split-all-tac, split-all-tac))⟩

```

lemma *methd-declclass*:

$\llbracket \text{class } G \ C = \text{Some } c; \text{ wf-prog } G; \text{ methd } G \ C \text{ sig} = \text{Some } m \rrbracket$
 $\implies \text{methd } G \ (\text{declclass } m) \text{ sig} = \text{Some } m$

proof –

assume *asm*: $\text{class } G \ C = \text{Some } c \text{ wf-prog } G \text{ methd } G \ C \text{ sig} = \text{Some } m$

have $\text{wf-prog } G \longrightarrow$

$(\forall \ c \ m. \text{class } G \ C = \text{Some } c \longrightarrow \text{methd } G \ C \text{ sig} = \text{Some } m$
 $\longrightarrow \text{methd } G \ (\text{declclass } m) \text{ sig} = \text{Some } m) \quad (\text{is } ?P \ G \ C)$

proof (*induct* $G \ C$ *rule*: *class-rec-induct'*, *intro* *allI impI*)

fix $G \ C \ c \ m$

assume *hyp*: $\bigwedge c. \text{class } G \ C = \text{Some } c \implies \text{ws-prog } G \implies C \neq \text{Object} \implies$
 $?P \ G \ (\text{super } c)$

assume *wf*: $\text{wf-prog } G$ and *cls-C*: $\text{class } G \ C = \text{Some } c$ and

$m: \text{methd } G \ C \text{ sig} = \text{Some } m$

show $\text{methd } G \ (\text{declclass } m) \text{ sig} = \text{Some } m$

proof (*cases* $C = \text{Object}$)

case *True*

with *wf* *m* show ?thesis by (auto intro: *table-of-map-SomeI*)

next

let *?filter* = *filter-tab* ($\lambda \text{sig } m. G \vdash C \text{ inherits method sig } m$)

let *?table* = *table-of* ($\text{map } (\lambda (s, m). (s, C, m)) \ (\text{methods } c)$)

case *False*

with *cls-C* *wf* *m*

have *methd-C*: $(?filter \ (\text{methd } G \ (\text{super } c))) \ ++ \ ?table \ \text{sig} = \text{Some } m$

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

show ?thesis

proof (*cases* $?table \ \text{sig}$)

case *None*

from *this* *methd-C* have $?filter \ (\text{methd } G \ (\text{super } c)) \ \text{sig} = \text{Some } m$

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:
  [|dynmethd G statC dynC sig = Some m;
   wf-prog G; is-class G statC|]
  ==> methd G (declclass m) sig = Some m
by (auto dest: dynmethd-declC)

```

```

lemma dynlookup-declC:
  [|dynlookup G statT dynC sig = Some m; wf-prog G;
   is-class G dynC; isrtype G statT|]
  ==> G ⊢ dynC ⊆C (declclass m) ∧ is-class G (declclass m)
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 |> Simplifier.del-loop 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 |> Simplifier.add-loop (split-all-tac, split-all-tac)))

lemma ty-expr-is-type:
   $\llbracket E \vdash e :: -T; \text{wf-prog } (\text{prg } E) \rrbracket \implies \text{is-type } (\text{prg } E) \ T$ 
by (auto dest!: wt-is-type)
lemma ty-var-is-type:
   $\llbracket E \vdash v :: T; \text{wf-prog } (\text{prg } E) \rrbracket \implies \text{is-type } (\text{prg } E) \ T$ 
by (auto dest!: wt-is-type)
lemma ty-exprs-is-type:
   $\llbracket E \vdash es :: Ts; \text{wf-prog } (\text{prg } E) \rrbracket \implies \text{Ball } (\text{set } Ts) (\text{is-type } (\text{prg } E))$ 
by (auto dest!: wt-is-type)

lemma static-mheadsD:
   $\llbracket emh \in \text{mheads } G \ S \ t \ \text{sig}; \text{wf-prog } G; E \vdash e :: -\text{RefT } t; \text{prg } E = G ;$ 
   $\text{invmode } (\text{mhd } emh) \ e \neq \text{IntVir}$ 
   $\rrbracket \implies \exists m. ( (\exists C. t = \text{ClassT } C \wedge \text{accmethd } G \ S \ C \ \text{sig} = \text{Some } m)$ 
     $\vee (\forall C. t \neq \text{ClassT } C \wedge \text{accmethd } G \ S \ \text{Object} \ \text{sig} = \text{Some } m) ) \wedge$ 
     $\text{declrefT } emh = \text{ClassT } (\text{declclass } m) \wedge \text{mhead } (\text{mthd } m) = (\text{mhd } emh)$ 
apply (subgoal-tac is-static emh  $\vee e = \text{Super}$ )
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:
   $\llbracket \text{methd } G \ C \ \text{sig} = \text{Some } m; \text{wf-prog } G;$ 
   $\text{class } G \ C = \text{Some } c \rrbracket \implies$ 
   $\exists T. (\text{prg} = G, \text{cls} = (\text{declclass } m),$ 
     $\text{lcl} = \text{callee-lcl } (\text{declclass } m) \ \text{sig } (\text{mthd } m)) \vdash \text{Methd } C \ \text{sig} :: -T \wedge G \vdash T \preceq_{\text{resTy}} m$ 
apply (frule (2) methd-wf-mdecl, clarify)
apply (force dest!: wf-mdecl-bodyD intro!: wt.Methd)
done

```

2 accessibility concerns

lemma mheads-type-accessible:

$\llbracket emh \in mheads\ G\ S\ T\ sig; wf\text{-}prog\ G \rrbracket$
 $\implies G \vdash RefT\ T\ accessible\text{-}in\ (pid\ S)$
by (*erule mheads-cases*)
 (*auto dest: accmethd-SomeD accessible-from-commonD accimethdsD*)

lemma *static-to-dynamic-accessible-from-aux:*

$\llbracket G \vdash m\ of\ C\ accessible\text{-}from\ accC; wf\text{-}prog\ G \rrbracket$

$\implies G \vdash m\ in\ C\ dyn\text{-}accessible\text{-}from\ accC$

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:* $G \vdash m\ of\ statC\ accessible\text{-}from\ accC$ **and**

subclseq: $G \vdash dynC \preceq_C statC$ **and**

wf: *wf-prog G*

shows $G \vdash m\ in\ dynC\ dyn\text{-}accessible\text{-}from\ accC$

proof –

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:* $G \vdash m\ of\ statC\ accessible\text{-}from\ accC$ **and**

static: *is-static m* **and**

wf: *wf-prog G*

shows $G \vdash m\ in\ (declclass\ m)\ dyn\text{-}accessible\text{-}from\ accC$

proof –

from *stat-acc wf*

have $G \vdash m\ in\ statC\ dyn\text{-}accessible\text{-}from\ accC$

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 $G \vdash Methd\ sig\ m\ member\text{-}in\ dynC$

proof –

from *m*

```

have subclseq:  $G \vdash \text{dyn}C \preceq_C \text{stat}C$ 
  by (auto simp add: dynmethd-def)
from subclseq iscls-statC
have iscls-dynC: is-class  $G \text{ dyn}C$ 
  by (rule subcls-is-class2)
from iscls-dynC iscls-statC wf m
have  $G \vdash \text{dyn}C \preceq_C (\text{declclass } m) \wedge \text{is-class } G (\text{declclass } m) \wedge$ 
   $\text{methd } G (\text{declclass } m) \text{ sig} = \text{Some } m$ 
  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:  $\text{methd } G \text{ stat}C \text{ sig} = \text{Some } \text{stat}M$  and
  stat-acc:  $G \vdash \text{Methd } \text{sig } \text{stat}M \text{ of } \text{stat}C \text{ accessible-from } \text{acc}C$  and
  dynM:  $\text{dynmethd } G \text{ stat}C \text{ dyn}C \text{ sig} = \text{Some } \text{dyn}M$  and
  wf: wf-prog  $G$ 
shows  $G \vdash \text{Methd } \text{sig } \text{dyn}M \text{ in } \text{dyn}C \text{ dyn-accessible-from } \text{acc}C$ 
proof -
  from wf have ws: ws-prog  $G \dots$ 
  from dynM
  have subclseq:  $G \vdash \text{dyn}C \preceq_C \text{stat}C$ 
    by (auto simp add: dynmethd-def)
  from stat-acc
  have is-cls-statC: is-class  $G \text{ stat}C$ 
    by (auto dest: accessible-from-commonD member-of-is-classD)
  with subclseq
  have is-cls-dynC: is-class  $G \text{ dyn}C$ 
    by (rule subcls-is-class2)
  from is-cls-statC statM wf
  have member-statC:  $G \vdash \text{Methd } \text{sig } \text{stat}M \text{ member-of } \text{stat}C$ 
    by (auto intro: methd-member-of)
  from stat-acc
  have statC-acc:  $G \vdash \text{Class } \text{stat}C \text{ accessible-in } (\text{pid } \text{acc}C)$ 
    by (auto dest: accessible-from-commonD)
  from statM subclseq is-cls-statC ws
  show ?thesis
  proof (cases rule: dynmethd-cases)
    case Static
    assume dynmethd:  $\text{dynmethd } G \text{ stat}C \text{ dyn}C \text{ sig} = \text{Some } \text{stat}M$ 
    with dynM have eq-dynM-statM:  $\text{dyn}M = \text{stat}M$ 
    by simp
    with stat-acc subclseq wf
    show ?thesis
    by (auto intro: static-to-dynamic-accessible-from)
  next
    case (Overrides newM)
    assume dynmethd:  $\text{dynmethd } G \text{ stat}C \text{ dyn}C \text{ sig} = \text{Some } \text{new}M$ 
    assume override:  $G, \text{sig} \vdash \text{new}M \text{ overrides } \text{stat}M$ 
    assume neq:  $\text{new}M \neq \text{stat}M$ 
    from dynmethd dynM
    have eq-dynM-newM:  $\text{dyn}M = \text{new}M$ 
    by simp
    from dynmethd eq-dynM-newM wf is-cls-statC
    have  $G \vdash \text{Methd } \text{sig } \text{dyn}M \text{ member-in } \text{dyn}C$ 
    by (auto intro: dynmethd-member-in)
  moreover

```

```

from subclseq
have  $G \vdash \text{dyn}C \prec_C \text{stat}C$ 
proof (cases rule: subclseq-cases)
  case Eq
  assume  $\text{dyn}C = \text{stat}C$ 
  moreover
  from is-cls-statC obtain c
    where class G statC = Some c
    by auto
  moreover
  note statM ws dynmethd
  ultimately
  have  $\text{new}M = \text{stat}M$ 
    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  $G \vdash \text{Methd sig statM in statC dyn-accessible-from accC}$ 
  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 $\neq \{\}$ and*

implements: $G \vdash \text{dyn}C \rightsquigarrow I$ and

isif-I: is-iface G I and

wf: wf-prog G

shows $\exists \text{ dyn}M. \text{methd } G \text{ dyn}C \text{ sig} = \text{Some dyn}M \wedge$

$G \vdash \text{Methd sig dyn}M \text{ in dyn}C \text{ dyn-accessible-from acc}C$

proof –

from *implements*

have *iscls-dynC: is-class G dynC* **by** (*rule implmt-is-class*)

from *iface-methd*

obtain *im*

where $im \in \text{imethds } G \text{ I sig}$

by *auto*

with *wf implements isif-I*

obtain *dynM*

where $\text{dyn}M: \text{methd } G \text{ dyn}C \text{ sig} = \text{Some dyn}M$ **and**

pub: accmodi dynM = Public

by (*blast dest: implmt-methd*)

with *iscls-dynC wf*

have $G \vdash \text{Methd sig dyn}M \text{ in dyn}C \text{ dyn-accessible-from acc}C$

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 $\exists \text{ dynM. dynimethd } G \text{ I dynC sig} = \text{Some dynM} \wedge$
 $G \vdash \text{Methd sig dynM in dynC dyn-accessible-from accC}$

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 $G \vdash \text{Methd sig dynM in dynC dyn-accessible-from accC}$

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: methd 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:  $G \vdash \text{Class } \text{dynC} \preceq \text{RefT } \text{statT}$ 
    by simp
  from statT widen-dynC
  have implmnt:  $G \vdash \text{dynC} \rightsquigarrow I$ 
    by auto
  from eq-static False
  have not-static-dynM:  $\neg \text{is-static } \text{dynM}$ 
    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  $\neg \text{is-static } \text{im}$ 
    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:  $\text{statT} = \text{IfaceT } I$ 
assume accmethd G accC Object sig = Some statM
then have statM:  $\text{methd } G \text{ Object } \text{sig} = \text{Some } \text{statM}$  and
  stat-acc:  $G \vdash \text{Methd } \text{sig } \text{statM} \text{ of Object accessible-from } \text{accC}$ 
  by (auto dest: accmethd-SomeD)
assume not-Private-statM:  $\text{accmodi } \text{statM} \neq \text{Private}$ 
assume eq-mhds:  $\text{mhead } (\text{methd } \text{statM}) = \text{mhd } \text{emh}$ 
from iscls-dynC wf
have widen-dynC-Obj:  $G \vdash \text{dynC} \preceq_C \text{Object}$ 
  by (auto intro: subcls-ObjectI)
show ?thesis
proof (cases imethds G I sig = {})
  case True
  from dynM statT True
  have dynM':  $\text{dynmethd } G \text{ Object } \text{dynC } \text{sig} = \text{Some } \text{dynM}$ 
    by (simp add: dynlookup-def dynimethd-def)
  from statT
  have  $G \vdash \text{RefT } \text{statT} \preceq_{\text{Class}} \text{Object}$ 
    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 ∈ imethds G I sig
  by auto
  have not-static-emh: ¬ 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 ¬ 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 implmnt-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 ⊢ new overridesS 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 ⊢ 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  $\langle G \vdash m \text{ in } C \text{ permits-acc-from } accC \rangle$  and  $\langle accmodi m = Private \rangle$ 
    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:
   $\llbracket G \vdash m \text{ in } C \text{ dyn-accessible-from } accC; accmodi m = Package; \rrbracket$ 
  wf-prog G
   $\implies pid accC = pid (declclass m)$ 
proof –
  assume wf: wf-prog G
  assume accessible: G ⊢ m in C dyn-accessible-from accC
  then show accmodi m = Package
     $\implies pid accC = pid (declclass m)$ 
    (is ?Pack m  $\implies$  ?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:  $G \vdash new$  member-in  $C$  and
         new:  $new = (declC, mdecl\ newm)$  and
         override:  $G \vdash (declC, newm)$  overrides  $old$  and
         subcls-C-Sup:  $G \vdash C \prec_C Sup$  and
         acc-old:  $G \vdash methdMembr\ old$  in  $Sup$  dyn-accessible-from  $accC$  and
         hyp:  $?Pack\ (methdMembr\ old) \implies ?P\ (methdMembr\ old)$  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:  $G \vdash f$  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  $\langle G \vdash f$  in  $C$  permits-acc-from  $accC \rangle$  and  $\langle accmodi\ f = Package \rangle$ 
  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:  $G \vdash f$  in  $C$  dyn-accessible-from  $accC$  and
         prot:  $accmodi\ f = Protected$  and
         field: is-field  $f$  and
         instance-field:  $\neg is-static\ f$  and
         outside:  $pid\ (declclass\ f) \neq pid\ accC$ 
shows  $G \vdash C \preceq_C accC$ 

```

proof –

from *dyn-acc prot field instance-field outside*

show *?thesis*

proof (*induct*)

case (*Immediate f C*)

note $\langle G \vdash f \text{ in } C \text{ permits-acc-from } accC \rangle$

moreover

assume *accmodi f = Protected and is-field f and \neg is-static f and*
 $pid(\text{declclass } f) \neq pid \text{ } accC$

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: $G \vdash f \text{ in } C \text{ dyn-accessible-from } accC$ and*

prot: accmodi f = Protected and

field: is-field f and

static-field: is-static f and

outside: $pid(\text{declclass } f) \neq pid \text{ } accC$

shows $G \vdash accC \preceq_C \text{declclass } f \wedge G \vdash C \preceq_C \text{declclass } f$

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*

$pid(\text{declclass } f) \neq pid \text{ } accC$

moreover

note $\langle G \vdash f \text{ in } C \text{ permits-acc-from } accC \rangle$

ultimately

have $G \vdash accC \preceq_C \text{declclass } f$

by (*auto simp add: permits-acc-def*)

moreover

from $\langle G \vdash f \text{ member-in } C \rangle$

have $G \vdash C \preceq_C \text{declclass } f$

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 handled via a general object abstraction
- the heap and the map for class objects are combined into a single table (*recall* $(loc, obj) \text{ table} \times (qname, obj) \text{ table} \sim= (loc + qname, obj) \text{ table}$)

objects

```
datatype obj-tag =      — tag for generic object
  CInst qname          — class instance
  | Arr ty int          — array with component type and length
  — | CStat qname the tag is irrelevant for a class object, i.e. the static fields of a class, since its type is
  given already by the reference to it (see below)
```

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

translations

```
(type) fspec <= (type) vname × qname
(type) vn    <= (type) fspec + int
(type) obj   <= (type) (tag::obj-tag, values::vn ⇒ val option)
(type) obj   <= (type) (tag::obj-tag, values::vn ⇒ val option,...::'a)
```

definition

```
the-Arr :: obj option ⇒ ty × int × (vn, val) table
where the-Arr obj = (SOME (T,k,t). obj = Some (tag=Arr T k,values=t))
```

```
lemma the-Arr-Arr [simp]: the-Arr (Some (tag=Arr T k,values=cs)) = (T,k,cs)
apply (auto simp: the-Arr-def)
done
```

```
lemma the-Arr-Arr1 [simp,intro,dest]:
  [tag obj = Arr T k] ⇒ the-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 $upd\text{-}obj\ n\ v = (\lambda obj. obj\ (\llbracket values := (values\ obj)\ (n \mapsto v) \rrbracket))$

lemma *upd-obj-def2* [*simp*]:

$upd\text{-}obj\ n\ v\ obj = obj\ (\llbracket values := (values\ obj)\ (n \mapsto v) \rrbracket)$

apply (*auto simp: upd-obj-def*)
done

definition

$obj\text{-}ty :: obj \Rightarrow ty$ **where**
 $obj\text{-}ty\ obj = (case\ tag\ obj\ of$
 $\quad CInst\ C \Rightarrow Class\ C$
 $\quad | Arr\ T\ k \Rightarrow T.\ [])$

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

$obj\text{-}ty\ (obj\ (\llbracket values := vs \rrbracket)) = obj\text{-}ty\ obj$

by *auto*

lemma *obj-ty-CInst* [*simp*]:

$obj\text{-}ty\ (\llbracket tag=CInst\ C, values=vs \rrbracket) = Class\ C$

by (*simp add: obj-ty-def*)

lemma *obj-ty-CInst1* [*simp, intro!, dest*]:

$\llbracket tag\ obj = CInst\ C \rrbracket \Longrightarrow obj\text{-}ty\ obj = Class\ C$

by (*simp add: obj-ty-def*)

lemma *obj-ty-Arr* [*simp*]:

$obj\text{-}ty\ (\llbracket tag=Arr\ T\ i, values=vs \rrbracket) = T.\ []$

by (*simp add: obj-ty-def*)

lemma *obj-ty-Arr1* [*simp, intro!, dest*]:

$\llbracket tag\ obj = Arr\ T\ i \rrbracket \Longrightarrow obj\text{-}ty\ obj = T.\ []$

by (*simp add: obj-ty-def*)

lemma *obj-ty-widenD*:

$G \vdash obj\text{-}ty\ obj \preceq RefT\ t \Longrightarrow (\exists C. tag\ obj = CInst\ C) \vee (\exists T\ k. tag\ obj = Arr\ T\ k)$

apply (*unfold obj-ty-def*)

apply (*auto split: obj-tag.split-asm*)

done

definition

$obj\text{-}class :: obj \Rightarrow qname$ **where**
 $obj\text{-}class\ obj = (case\ tag\ obj\ of$
 $\quad CInst\ C \Rightarrow C$
 $\quad | Arr\ T\ k \Rightarrow Object)$

lemma *obj-class-CInst* [simp]: *obj-class* ($\lfloor \text{tag} = \text{CInst } C, \text{values} = \text{vs} \rfloor$) = *C*
by (*auto simp: obj-class-def*)

lemma *obj-class-CInst1* [simp,intro!,dest]:
 $\text{tag } \text{obj} = \text{CInst } C \implies \text{obj-class } \text{obj} = C$
by (*auto simp: obj-class-def*)

lemma *obj-class-Arr* [simp]: *obj-class* ($\lfloor \text{tag} = \text{Arr } T \ k, \text{values} = \text{vs} \rfloor$) = *Object*
by (*auto simp: obj-class-def*)

lemma *obj-class-Arr1* [simp,intro!,dest]:
 $\text{tag } \text{obj} = \text{Arr } T \ k \implies \text{obj-class } \text{obj} = \text{Object}$
by (*auto simp: obj-class-def*)

lemma *obj-ty-obj-class*: $G \vdash \text{obj-ty } \text{obj} \preceq \text{Class } \text{statC} = G \vdash \text{obj-class } \text{obj} \preceq_C \text{statC}$
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* + *qname* — generalized object reference

translations

(*type*) *oref* <= (*type*) *loc* + *qname*

abbreviation (*input*)

Heap :: *loc* \Rightarrow *oref* **where** *Heap* \equiv *Inl*

abbreviation (*input*)

Stat :: *qname* \Rightarrow *oref* **where** *Stat* \equiv *Inr*

definition

fields-table :: *prog* \Rightarrow *qname* \Rightarrow (*fspec* \Rightarrow *field* \Rightarrow *bool*) \Rightarrow (*fspec*, *ty*) *table* **where**
fields-table *G C P* =
 $\text{map-option type} \circ \text{table-of } (\text{filter } (\text{case-prod } P) (\text{DeclConcepts.fields } G \ C))$

lemma *fields-table-SomeI*:

$\llbracket \text{table-of } (\text{DeclConcepts.fields } G \ C) \ n = \text{Some } f; P \ n \ f \rrbracket$
 $\implies \text{fields-table } G \ C \ P \ n = \text{Some } (\text{type } f)$
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* \implies

$\exists f. (fn, f) \in \text{set}(\text{DeclConcepts.fields } G \ C) \wedge \text{type } f = T$
apply (*unfold fields-table-def*)
apply *clarsimp*
apply (*erule map-of-SomeD*)
apply *auto*
done

lemma *fields-table-SomeD*:

$\llbracket \text{fields-table } G \ C \ P \ fn = \text{Some } T; \text{ unique } (\text{DeclConcepts.fields } G \ C) \rrbracket \implies$
 $\exists f. \text{table-of } (\text{DeclConcepts.fields } G \ C) \ fn = \text{Some } f \wedge \text{type } f = T$

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* \Rightarrow *int* \Rightarrow *bool* ($\langle (- / \text{in}'\text{-bounds } -) \rangle [50, 51] \ 50$)

where *i in-bounds k* = $(0 \leq i \wedge i < k)$

definition

arr-comps :: '*a* \Rightarrow *int* \Rightarrow *int* \Rightarrow '*a* *option*

where *arr-comps T k* = $(\lambda i. \text{if } i \text{ in-bounds } k \text{ then } \text{Some } T \text{ else } \text{None})$

definition

var-tys :: *prog* \Rightarrow *obj-tag* \Rightarrow *oref* \Rightarrow (*vn*, *ty*) *table* **where**

var-tys G oi r =

(*case r of*

Heap a \Rightarrow (*case oi of*

CInst C \Rightarrow *fields-table G C* $(\lambda n f. \neg \text{static } f) (+) \text{Map.empty}$

| *Arr T k* \Rightarrow *Map.empty* $(+) \text{arr-comps } T \ k$)

| *Stat C* \Rightarrow *fields-table G C* $(\lambda fn f. \text{declclassf } fn = C \wedge \text{static } f)$

$(+) \text{Map.empty}$)

lemma *var-tys-Some-eq*:

var-tys G oi r n = *Some T*

= (*case r of*

Inl a \Rightarrow (*case oi of*

CInst C \Rightarrow $(\exists nt. n = \text{Inl } nt \wedge \text{fields-table } G \ C \ (\lambda n f. \neg \text{static } f) \ nt = \text{Some } T)$

| *Arr t k* \Rightarrow $(\exists i. n = \text{Inr } i \wedge i \text{ in-bounds } k \wedge t = T)$)

| *Inr C* \Rightarrow $(\exists nt. n = \text{Inl } nt \wedge$

fields-table G C $(\lambda fn f. \text{declclassf } fn = C \wedge \text{static } f) \ nt$

$= \text{Some } T)$)

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* \leq (*type*) (*oref* , *obj*) *table*

(*type*) *heap* \leq (*type*) (*loc* , *obj*) *table*

datatype *st* =

st globs locals

2 access

definition

$globs :: st \Rightarrow globs$
where $globs = case-st (\lambda g l. g)$

definition

$locals :: st \Rightarrow locals$
where $locals = case-st (\lambda g l. l)$

definition $heap :: st \Rightarrow heap$ **where**

$heap s = globs s \circ Heap$

lemma $globs-def2$ [simp]: $globs (st g l) = g$
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 $val-this :: st \Rightarrow val$

where $val-this s == the (locals s This)$

abbreviation $lookup-obj :: st \Rightarrow val \Rightarrow obj$

where $lookup-obj s a' == the (heap s (the-Addr a'))$

3 memory allocation

definition

$new-Addr :: heap \Rightarrow loc\ option$ **where**
 $new-Addr h = (if (\forall a. h a \neq None) then None else Some (SOME a. h a = None))$

lemma $new-AddrD$: $new-Addr h = Some a \implies h a = None$

apply (auto simp add: new-Addr-def)

apply (erule someI)

done

lemma $new-AddrD2$: $new-Addr h = Some a \implies \forall b. h b \neq None \longrightarrow b \neq a$

apply (drule new-AddrD)

apply auto

done

lemma $new-Addr-SomeI$: $h a = None \implies \exists b. new-Addr h = Some b \wedge h b = None$

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]:
 $0 < j \implies \text{init-vals } (\text{arr-comps } T \ j) =$
 $(\text{init-vals } (\text{arr-comps } T \ (j - 1)))(j - 1 \mapsto \text{default-val } T)$
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* ($\langle \text{gupd}'(-\mapsto-) \rangle$ [10, 10] 1000)
where *gupd* *r obj* = *case-st* ($\lambda g \ l. \text{st } (g(r \mapsto \text{obj})) \ l$)

definition

lupd :: *lname* \Rightarrow *val* \Rightarrow *st* \Rightarrow *st* ($\langle \text{lupd}'(-\mapsto-) \rangle$ [10, 10] 1000)
where *lupd* *vn v* = *case-st* ($\lambda g \ l. \text{st } g \ (l(vn \mapsto v))$)

definition

upd-gobj :: *oref* \Rightarrow *vn* \Rightarrow *val* \Rightarrow *st* \Rightarrow *st*
where *upd-gobj* *r n v* = *case-st* ($\lambda g \ l. \text{st } (\text{chg-map } (\text{upd-obj } n \ v) \ r \ g) \ l$)

definition

set-locals :: *locals* \Rightarrow *st* \Rightarrow *st*
where *set-locals* *l* = *case-st* ($\lambda g \ l'. \text{st } g \ l$)

definition

init-obj :: *prog* \Rightarrow *obj-tag* \Rightarrow *oref* \Rightarrow *st* \Rightarrow *st*
where *init-obj* *G oi r* = *gupd*($r \mapsto (\text{tag} = oi, \text{values} = \text{init-vals } (\text{var-tys } G \ oi \ r))$)

abbreviation

init-class-obj :: *prog* \Rightarrow *qtname* \Rightarrow *st* \Rightarrow *st*
where *init-class-obj* *G C* == *init-obj* *G undefined (Inr C)*

lemma *gupd-def2* [simp]: *gupd*($r \mapsto \text{obj}$) (*st g l*) = *st* ($g(r \mapsto \text{obj})$) *l*
apply (*unfold gupd-def*)
apply (*simp (no-asm)*)
done

lemma *lupd-def2* [simp]: *lupd*($vn \mapsto v$) (*st g l*) = *st g* ($l(vn \mapsto v)$)
apply (*unfold lupd-def*)
apply (*simp (no-asm)*)
done

lemma *globs-gupd* [simp]: *globs* (*gupd*($r \mapsto \text{obj}$) *s*) = (*globs s*)($r \mapsto \text{obj}$)
apply (*induct s*)
by (*simp add: gupd-def*)

lemma *globs-lupd* [simp]: *globs* (*lupd*($vn \mapsto v$) *s*) = *globs s*
apply (*induct s*)
by (*simp add: lupd-def*)

lemma *locals-gupd* [simp]: *locals* (*gupd*($r \mapsto \text{obj}$) *s*) = *locals s*
apply (*induct s*)
by (*simp add: gupd-def*)

lemma *locals-lupd* [simp]: *locals* (*lupd*($vn \mapsto v$) *s*) = (*locals s*)($vn \mapsto v$)

apply (*induct s*)
by (*simp add: lupd-def*)

lemma *globs-upd-gobj-new* [*rule-format (no-asm), simp*]:
 $\text{globs } s \ r = \text{None} \longrightarrow \text{globs } (\text{upd-gobj } r \ n \ v \ s) = \text{globs } s$
apply (*unfold upd-gobj-def*)
apply (*induct s*)
apply *auto*
done

lemma *globs-upd-gobj-upd* [*rule-format (no-asm), simp*]:
 $\text{globs } s \ r = \text{Some } \text{obj} \longrightarrow \text{globs } (\text{upd-gobj } r \ n \ v \ s) = (\text{globs } s)(r \mapsto \text{upd-obj } n \ v \ \text{obj})$
apply (*unfold upd-gobj-def*)
apply (*induct s*)
apply *auto*
done

lemma *locals-upd-gobj* [*simp*]: $\text{locals } (\text{upd-gobj } r \ n \ v \ s) = \text{locals } s$
apply (*induct s*)
by (*simp add: upd-gobj-def*)

lemma *globs-init-obj* [*simp*]: $\text{globs } (\text{init-obj } G \ oi \ r \ s) \ t =$
 $(\text{if } t=r \text{ then } \text{Some } (\text{tag}=oi, \text{values}=\text{init-vals } (\text{var-tys } G \ oi \ r)) \text{ else } \text{globs } s \ t)$
apply (*unfold init-obj-def*)
apply (*simp (no-asm)*)
done

lemma *locals-init-obj* [*simp*]: $\text{locals } (\text{init-obj } G \ oi \ r \ s) = \text{locals } s$
by (*simp add: init-obj-def*)

lemma *surjective-st* [*simp*]: $\text{st } (\text{globs } s) (\text{locals } s) = s$
apply (*induct s*)
by *auto*

lemma *surjective-st-init-obj*:
 $\text{st } (\text{globs } (\text{init-obj } G \ oi \ r \ s)) (\text{locals } s) = \text{init-obj } G \ oi \ r \ s$
apply (*subst locals-init-obj [THEN sym]*)
apply (*rule surjective-st*)
done

lemma *heap-heap-upd* [*simp*]:
 $\text{heap } (\text{st } (g(\text{Inl } a \mapsto \text{obj}))) \ l = (\text{heap } (\text{st } g \ l))(a \mapsto \text{obj})$
apply (*rule ext*)
apply (*simp (no-asm)*)
done

lemma *heap-stat-upd* [*simp*]: $\text{heap } (\text{st } (g(\text{Inr } C \mapsto \text{obj}))) \ l = \text{heap } (\text{st } g \ l)$
apply (*rule ext*)
apply (*simp (no-asm)*)
done

lemma *heap-local-upd* [*simp*]: $\text{heap } (\text{st } g \ (l(vn \mapsto v))) = \text{heap } (\text{st } g \ l)$
apply (*rule ext*)
apply (*simp (no-asm)*)
done

lemma *heap-gupd-Heap* [*simp*]: $\text{heap } (\text{gupd}(\text{Heap } a \mapsto \text{obj}) \ s) = (\text{heap } s)(a \mapsto \text{obj})$
apply (*rule ext*)
apply (*simp (no-asm)*)
done

```

lemma heap-gupd-Stat [simp]: heap (gupd(Stat C $\mapsto$ obj) s) = heap s
apply (rule ext)
apply (simp (no-asm))
done
lemma heap-lupd [simp]: heap (lupd(vn $\mapsto$ 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  $\Rightarrow$  xcpt
  where the-Xcpt (Xcpt x) = x

```

```

primrec the-Jump :: abrupt  $\Rightarrow$  jump
  where the-Jump (Jump j) = j

```

primrec *the-Loc* :: *xcpt* \Rightarrow *loc*
where *the-Loc* (*Loc* *a*) = *a*

primrec *the-Std* :: *xcpt* \Rightarrow *xname*
where *the-Std* (*Std* *x*) = *x*

definition

abrupt-if :: *bool* \Rightarrow *abopt* \Rightarrow *abopt* \Rightarrow *abopt*
where *abrupt-if* *c* *x'* *x* = (if *c* \wedge (*x* = *None*) then *x'* else *x*)

lemma *abrupt-if-True-None* [simp]: *abrupt-if* *True* *x* *None* = *x*
by (simp add: *abrupt-if-def*)

lemma *abrupt-if-True-not-None* [simp]: *x* \neq *None* \Longrightarrow *abrupt-if* *True* *x* *y* \neq *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* \neq *None* \Longrightarrow *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* \wedge *x* = *None* \longrightarrow *P* *x'*) \wedge (\neg (*c* \wedge *x* = *None*) \longrightarrow *P* *x*))
apply (unfold *abrupt-if-def*)
apply (split *if-split*)
apply *auto*
done

abbreviation *raise-if* :: *bool* \Rightarrow *xname* \Rightarrow *abopt* \Rightarrow *abopt*
where *raise-if* *c* *xn* == *abrupt-if* *c* (*Some* (*Xcpt* (*Std* *xn*)))

abbreviation *np* :: *val* \Rightarrow *abopt* \Rightarrow *abopt*
where *np* *v* == *raise-if* (*v* = *Null*) *NullPointer*

abbreviation *check-neg* :: *val* \Rightarrow *abopt* \Rightarrow *abopt*
where *check-neg* *i'* == *raise-if* (*the-Intg* *i'* < 0) *NegArrSize*

abbreviation *error-if* :: *bool* \Rightarrow *error* \Rightarrow *abopt* \Rightarrow *abopt*
where *error-if* *c* *e* == *abrupt-if* *c* (*Some* (*Error* *e*))

lemma *raise-if-None* [simp]: (*raise-if* *c* *x* *y* = *None*) = (\neg *c* \wedge *y* = *None*)
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* \longrightarrow *b*) \wedge *y* = *None*)
apply (simp add: *abrupt-if-def*)
apply *auto*
done

lemma *raise-if-SomeD* [*dest!*]:
 $\text{raise-if } c \ x \ y = \text{Some } z \implies c \wedge z = (\text{Xcpt } (\text{Std } x)) \wedge y = \text{None} \vee (y = \text{Some } z)$
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*]: $(\text{error-if } c \ e \ y = \text{None}) = (\neg c \wedge y = \text{None})$
apply (*simp add: abrupt-if-def*)
by *auto*
declare *error-if-None* [*THEN iffD1, dest!*]

lemma *if-error-if-None* [*simp*]:
 $((\text{if } b \text{ then } y \text{ else error-if } c \ e \ y) = \text{None}) = ((c \longrightarrow b) \wedge y = \text{None})$
apply (*simp add: abrupt-if-def*)
apply *auto*
done

lemma *error-if-SomeD* [*dest!*]:
 $\text{error-if } c \ e \ y = \text{Some } z \implies c \wedge z = (\text{Error } e) \wedge y = \text{None} \vee (y = \text{Some } z)$
apply (*case-tac* *y*)
apply (*case-tac* *c*)
apply (*simp add: abrupt-if-def*)
apply (*simp add: abrupt-if-def*)
apply *auto*
done

definition

$\text{absorb} :: \text{jump} \Rightarrow \text{abopt} \Rightarrow \text{abopt}$
where $\text{absorb } j \ a = (\text{if } a = \text{Some } (\text{Jump } j) \text{ then } \text{None} \text{ else } a)$

lemma *absorb-SomeD* [*dest!*]: $\text{absorb } j \ a = \text{Some } x \implies a = \text{Some } x$
by (*auto simp add: absorb-def*)

lemma *absorb-same* [*simp*]: $\text{absorb } j \ (\text{Some } (\text{Jump } j)) = \text{None}$
by (*auto simp add: absorb-def*)

lemma *absorb-other* [*simp*]: $a \neq \text{Some } (\text{Jump } j) \implies \text{absorb } j \ a = a$
by (*auto simp add: absorb-def*)

lemma *absorb-Some-NoneD*: $\text{absorb } j \ (\text{Some } \text{abr}) = \text{None} \implies \text{abr} = \text{Jump } j$
by (*simp add: absorb-def*)

lemma *absorb-Some-JumpD*: $\text{absorb } j \ s = \text{Some } (\text{Jump } j') \implies j' \neq j$
by (*simp add: absorb-def*)

full program state

type-synonym

$\text{state} = \text{abopt} \times \text{st}$ — state including abruption information

translations

$(\text{type}) \ \text{abopt} \leq (\text{type}) \ \text{abrupt option}$
 $(\text{type}) \ \text{state} \leq (\text{type}) \ \text{abopt} \times \text{st}$

abbreviation

$\text{Norm} :: \text{st} \Rightarrow \text{state}$

where $Norm\ s == (None, s)$

abbreviation (*input*)
 $abrupt :: state \Rightarrow abopt$
where $abrupt == fst$

abbreviation (*input*)
 $store :: state \Rightarrow st$
where $store == snd$

lemma *single-stateE*: $\forall Z. Z = (s::state) \implies False$
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*: $All\ ((=)\ (x::state)) \implies R$
apply (*drule-tac* $x = (if\ abrupt\ x = None\ then\ Some\ x'\ else\ None, y)$ **for** $x'\ y$ **in** *spec*)
apply *clarsimp*
done

definition
 $normal :: state \Rightarrow bool$
where $normal = (\lambda s. abrupt\ s = None)$

lemma *normal-def2* [*simp*]: $normal\ s = (abrupt\ s = None)$
apply (*unfold* *normal-def*)
apply (*simp* (*no-asm*))
done

definition
 $heap-free :: nat \Rightarrow state \Rightarrow bool$
where $heap-free\ n = (\lambda s. atleast-free\ (heap\ (store\ s))\ n)$

lemma *heap-free-def2* [*simp*]: $heap-free\ n\ s = atleast-free\ (heap\ (store\ s))\ n$
apply (*unfold* *heap-free-def*)
apply *simp*
done

6 update

definition
 $abupd :: (abopt \Rightarrow abopt) \Rightarrow state \Rightarrow state$
where $abupd\ f = map-prod\ f\ id$

definition
 $supd :: (st \Rightarrow st) \Rightarrow state \Rightarrow state$
where $supd = map-prod\ id$

lemma *abupd-def2* [*simp*]: $abupd\ f\ (x, s) = (f\ x, s)$
by (*simp* *add*: *abupd-def*)

lemma *abupd-abrupt-if-False* [*simp*]: $\bigwedge s. abupd\ (abrupt-if\ False\ xo)\ s = s$
by *simp*

lemma *supd-def2* [*simp*]: $supd\ f\ (x, s) = (x, f\ s)$
by (*simp* *add*: *supd-def*)

lemma *supd-lupd* [*simp*]:

$\bigwedge s. \text{supd } (\text{lupd } vn \ v) \ s = (\text{abrupt } s, \text{lupd } vn \ v \ (\text{store } s))$
apply (*simp* (*no-asm-simp*) *only: split-tupled-all*)
apply (*simp* (*no-asm*))
done

lemma *supd-gupd* [*simp*]:
 $\bigwedge s. \text{supd } (\text{gupd } r \ \text{obj}) \ s = (\text{abrupt } s, \text{gupd } r \ \text{obj} \ (\text{store } s))$
apply (*simp* (*no-asm-simp*) *only: split-tupled-all*)
apply (*simp* (*no-asm*))
done

lemma *supd-init-obj* [*simp*]:
 $\text{supd } (\text{init-obj } G \ \text{oi } r) \ s = (\text{abrupt } s, \text{init-obj } G \ \text{oi } r \ (\text{store } s))$
apply (*unfold init-obj-def*)
apply (*simp* (*no-asm*))
done

lemma *abupd-store-invariant* [*simp*]: $\text{store } (\text{abupd } f \ s) = \text{store } s$
by (*cases s simp*)

lemma *supd-abrupt-invariant* [*simp*]: $\text{abrupt } (\text{supd } f \ s) = \text{abrupt } s$
by (*cases s simp*)

abbreviation *set-lvars* :: *locals* \Rightarrow *state* \Rightarrow *state*
where *set-lvars l == supd (set-locals l)*

abbreviation *restore-lvars* :: *state* \Rightarrow *state* \Rightarrow *state*
where *restore-lvars s' s == set-lvars (locals (store s')) s*

lemma *set-set-lvars* [*simp*]: $\bigwedge s. \text{set-lvars } l \ (\text{set-lvars } l' \ s) = \text{set-lvars } l \ s$
apply (*simp* (*no-asm-simp*) *only: split-tupled-all*)
apply (*simp* (*no-asm*))
done

lemma *set-lvars-id* [*simp*]: $\bigwedge s. \text{set-lvars } (\text{locals } (\text{store } s)) \ s = s$
apply (*simp* (*no-asm-simp*) *only: split-tupled-all*)
apply (*simp* (*no-asm*))
done

initialisation test

definition
initd :: *qtname* \Rightarrow *globs* \Rightarrow *bool*
where *initd C g = (g (Stat C) \neq None)*

definition
initd :: *qtname* \Rightarrow *state* \Rightarrow *bool*
where *initd C = initd C \circ globs \circ store*

lemma *not-initd-empty* [*simp*]: $\neg \text{initd } C \ \text{Map.empty}$
apply (*unfold initd-def*)
apply (*simp* (*no-asm*))
done

lemma *initd-gupdate* [*simp*]: $\text{initd } C \ (g(r \mapsto \text{obj})) = (\text{initd } C \ g \vee r = \text{Stat } C)$
apply (*unfold initd-def*)
apply (*auto split: st.split*)
done


```

lemma initd-init-class-obj [intro!]: initd C (globs (init-class-obj G C s))
apply (unfold initd-def)
apply (simp (no-asm))
done

```

```

lemma not-initdD:  $\neg \text{initd } C \ g \implies g \ (Stat \ C) = None$ 
apply (unfold initd-def)
apply (erule notnotD)
done

```

```

lemma initdD: initd C g  $\implies \exists \text{ obj. } g \ (Stat \ C) = Some \text{ obj}$ 
apply (unfold initd-def)
apply auto
done

```

```

lemma initd-def2 [simp]: initd C s = initd C (globs (store s))
apply (unfold initd-def)
apply (simp (no-asm))
done

```

error-free

definition

```

error-free :: state  $\Rightarrow$  bool
where error-free s = ( $\neg (\exists \text{ err. abrupt } s = Some \ (Error \ err))$ )

```

```

lemma error-free-Norm [simp,intro]: error-free (Norm s)
by (simp add: error-free-def)

```

```

lemma error-free-normal [simp,intro]: normal s  $\implies$  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]:
 $\neg (\exists \text{ err. } x = Error \ err) \implies \text{error-free } ((Some \ x),s)$ 
by (auto simp add: error-free-def)

```

```

lemma error-free-abupd-absorb [simp,intro]:
error-free s  $\implies$  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)  $\implies$  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]:
 $\llbracket \text{error-free } s; \neg (\exists \text{ err. } x = Error \ err) \rrbracket$ 

```

$\implies \text{error-free } (\text{abupd } (\text{abrupt-if } p \text{ (Some } x)) \text{ } s)$
by (cases s)
 (auto simp add: abrupt-if-def
 split: if-split)

lemma error-free-abrupt-if1 [simp,intro]:
 $\llbracket \text{error-free } (a,s); \neg (\exists \text{ err. } x = \text{Error err}) \rrbracket$
 $\implies \text{error-free } (\text{abrupt-if } p \text{ (Some } x) \text{ } a, s)$
by (auto simp add: abrupt-if-def
 split: if-split)

lemma error-free-abrupt-if-Xcpt [simp,intro]:
 error-free s
 $\implies \text{error-free } (\text{abupd } (\text{abrupt-if } p \text{ (Some (Xcpt } x))) \text{ } s)$
by simp

lemma error-free-abrupt-if-Xcpt1 [simp,intro]:
 error-free (a,s)
 $\implies \text{error-free } (\text{abrupt-if } p \text{ (Some (Xcpt } x)) \text{ } a, s)$
by simp

lemma error-free-abrupt-if-Jump [simp,intro]:
 error-free s
 $\implies \text{error-free } (\text{abupd } (\text{abrupt-if } p \text{ (Some (Jump } j))) \text{ } s)$
by simp

lemma error-free-abrupt-if-Jump1 [simp,intro]:
 error-free (a,s)
 $\implies \text{error-free } (\text{abrupt-if } p \text{ (Some (Jump } j)) \text{ } a, s)$
by simp

lemma error-free-raise-if [simp,intro]:
 error-free s $\implies \text{error-free } (\text{abupd } (\text{raise-if } p \text{ } x) \text{ } s)$
by simp

lemma error-free-raise-if1 [simp,intro]:
 error-free (a,s) $\implies \text{error-free } ((\text{raise-if } p \text{ } x \text{ } a), s)$
by simp

lemma error-free-supd [simp,intro]:
 error-free s $\implies \text{error-free } (\text{supd } f \text{ } s)$
by (cases s) (simp add: error-free-def)

lemma error-free-supd1 [simp,intro]:
 error-free (a,s) $\implies \text{error-free } (a, f \text{ } s)$
by (simp add: error-free-def)

lemma error-free-set-lvars [simp,intro]:
 error-free s $\implies \text{error-free } ((\text{set-lvars } l) \text{ } s)$
by (cases s) simp

lemma error-free-set-locals [simp,intro]:
 error-free (x, s)
 $\implies \text{error-free } (x, \text{set-locals } l \text{ } s')$
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 NullPointerException 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.
 - ++ facilitates exception handling
 - + symmetry
 - the rules are defined carefully in order to be applicable even in not type-correct situations (yielding undefined values), e.g. $\text{the-Addr } (\text{Val } (\text{Bool } b)) = \text{undefined}$.
 - ++ fewer rules
 - less readable because of auxiliary functions like *the-Addr*
- Alternative: "defensive" evaluation throwing some `InternalError` exception in case of (impossible, for correct programs) type mismatches
- there is exactly one rule per syntactic construct
 - + no redundancy in case distinctions
 - `halloc` fails iff there is no free heap address. When there is only one free heap address left, it returns an `OutOfMemory` exception. In this way it is guaranteed that when an `OutOfMemory` exception is thrown for the first time, there is a free location on the heap to allocate it.
 - the allocation of objects that represent standard exceptions is deferred until execution of any enclosing catch clause, which is transparent to the program.
 - requires an auxiliary execution relation
 - ++ avoids copies of allocation code and awkward case distinctions (whether there is enough memory to allocate the exception) in evaluation rules
 - unfortunately *new-Addr* is not directly executable because of Hilbert operator.

simplifications:

- local variables are initialized with default values (no definite assignment)
- garbage collection not considered, therefore also no finalizers
- stack overflow and memory overflow during class initialization not modelled
- exceptions in initializations not replaced by `ExceptionInInitializerError`

type-synonym $vvar = val \times (val \Rightarrow state \Rightarrow state)$

type-synonym $vals = (val, vvar, val \text{ list}) \text{ sum3}$

translations

$(type) \ vvar \leq (type) \ val \times (val \Rightarrow state \Rightarrow state)$

$(type) \ vals \leq (type) \ (val, vvar, val \text{ list}) \text{ sum3}$

To avoid redundancy and to reduce the number of rules, there is only one evaluation rule for each syntactic term. This is also true for variables (e.g. see the rules below for *LVar*, *FVar* and *AVar*). So evaluation of a variable must capture both possible further uses: read (rule *Acc*) or write (rule *Ass*) to the variable. Therefore a variable evaluates to a special value *vvar*, which is a pair, consisting of the current value (for later read access) and an update function (for later write access). Because during assignment to an array variable an exception may occur if the types don't match, the update function is very generic: it transforms the full state. This generic update function causes some technical trouble during some proofs (e.g. type safety, correctness of definite assignment). There we need to prove some additional invariant on this update function to prove the assignment correct, since the update function could potentially alter the whole state in an arbitrary manner. This

invariant must be carried around through the whole induction. So for future approaches it may be better not to take such a generic update function, but only to store the address and the kind of variable (array (+ element type), local variable or field) for later assignment.

abbreviation

dummy-res :: *vals* ($\langle \diamond \rangle$)
where $\diamond == \text{In1 Unit}$

abbreviation (*input*)

val-inj-vals ($\langle \lfloor _ \rfloor_e \rangle 1000$)
where $\lfloor e \rfloor_e == \text{In1 } e$

abbreviation (*input*)

var-inj-vals ($\langle \lfloor _ \rfloor_v \rangle 1000$)
where $\lfloor v \rfloor_v == \text{In2 } v$

abbreviation (*input*)

lst-inj-vals ($\langle \lfloor _ \rfloor_l \rangle 1000$)
where $\lfloor es \rfloor_l == \text{In3 } es$

definition *undefined3* :: ('a1 + 'ar, 'b, 'c) *sum3* \Rightarrow *vals* **where**

undefined3 = *case-sum3* (*In1* \circ *case-sum* ($\lambda x. \text{undefined}$) ($\lambda x. \text{Unit}$))
 $(\lambda x. \text{In2 undefined}) (\lambda x. \text{In3 undefined})$

lemma [*simp*]: *undefined3* (*In1l* *x*) = *In1 undefined*

by (*simp add: undefined3-def*)

lemma [*simp*]: *undefined3* (*In1r* *x*) = \diamond

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* \Rightarrow *abopt* \Rightarrow *abopt* **where**
throw *a'* *x* = *abrupt-if* *True* (*Some* (*Xcpt* (*Loc* (*the-Addr* *a'*)))) (*np* *a' x*)

lemma *throw-def2*:

throw *a' x* = *abrupt-if* *True* (*Some* (*Xcpt* (*Loc* (*the-Addr* *a'*)))) (*np* *a' x*)

apply (*unfold throw-def*)

apply (*simp* (*no-asm*))

done

definition

fits :: *prog* \Rightarrow *st* \Rightarrow *val* \Rightarrow *ty* \Rightarrow *bool* ($\langle _, \vdash _ \rangle \text{fits} \rightarrow [61, 61, 61, 61] 60$)
where $G, s \vdash a' \text{fits } T = ((\exists rt. T = \text{RefT } rt) \longrightarrow a' = \text{Null} \vee G \vdash \text{obj-ty}(\text{lookup-obj } s \ a') \preceq T)$

lemma *fits-Null* [*simp*]: $G, s \vdash \text{Null fits } T$

by (*simp add: fits-def*)

lemma *fits-Addr-RefT* [*simp*]:

$G, s \vdash \text{Addr } a \text{fits RefT } t = G \vdash \text{obj-ty}(\text{the}(\text{heap } s \ a)) \preceq \text{RefT } t$

by (*simp add: fits-def*)

lemma *fitsD*: $\bigwedge X. G, s \vdash a' \text{ fits } T \implies (\exists pt. T = \text{PrimT } pt) \vee$
 $(\exists t. T = \text{RefT } t) \wedge a' = \text{Null} \vee$
 $(\exists t. T = \text{RefT } t) \wedge a' \neq \text{Null} \wedge G \vdash \text{obj-ty } (\text{lookup-obj } s \ a') \preceq T$
apply (*unfold fits-def*)
apply (*case-tac* $\exists pt. T = \text{PrimT } pt$)
apply *simp-all*
apply (*case-tac* T)
defer
apply (*case-tac* $a' = \text{Null}$)
apply *simp-all*
done

definition

catch :: *prog* \Rightarrow *state* \Rightarrow *qname* \Rightarrow *bool* ($\langle -, \vdash \text{catch} \rightarrow [61, 61, 61] 60$) **where**
 $G, s \vdash \text{catch } C = (\exists xc. \text{abrupt } s = \text{Some } (\text{Xcpt } xc) \wedge$
 $G, \text{store } s \vdash \text{Addr } (\text{the-Loc } xc) \text{ fits } \text{Class } C)$

lemma *catch-Norm* [*simp*]: $\neg G, \text{Norm } s \vdash \text{catch } tn$
apply (*unfold catch-def*)
apply (*simp* (*no-asm*))
done

lemma *catch-XcptLoc* [*simp*]:
 $G, (\text{Some } (\text{Xcpt } (\text{Loc } a)), s) \vdash \text{catch } C = G, s \vdash \text{Addr } a \text{ fits } \text{Class } C$
apply (*unfold catch-def*)
apply (*simp* (*no-asm*))
done

lemma *catch-Jump* [*simp*]: $\neg G, (\text{Some } (\text{Jump } j), s) \vdash \text{catch } tn$
apply (*unfold catch-def*)
apply (*simp* (*no-asm*))
done

lemma *catch-Error* [*simp*]: $\neg G, (\text{Some } (\text{Error } e), s) \vdash \text{catch } tn$
apply (*unfold catch-def*)
apply (*simp* (*no-asm*))
done

definition

new-xcpt-var :: *vname* \Rightarrow *state* \Rightarrow *state* **where**
 $\text{new-xcpt-var } vn = (\lambda(x, s). \text{Norm } (\text{lupd}(\text{VName } vn \mapsto \text{Addr } (\text{the-Loc } (\text{the-Xcpt } (\text{the } x)))) s))$

lemma *new-xcpt-var-def2* [*simp*]:
 $\text{new-xcpt-var } vn \ (x, s) =$
 $\text{Norm } (\text{lupd}(\text{VName } vn \mapsto \text{Addr } (\text{the-Loc } (\text{the-Xcpt } (\text{the } x)))) s)$
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**
 $\text{assign } f \ v = (\lambda(x, s). \text{let } (x', s') = (\text{if } x = \text{None} \text{ then } f \ v \text{ else } \text{id}) \ (x, s)$
 $\text{in } (x', \text{if } x' = \text{None} \text{ then } s' \text{ else } s))$

lemma *assign-Norm-Norm* [simp]:
 $f\ v\ (Norm\ s) = Norm\ s' \implies assign\ f\ v\ (Norm\ s) = Norm\ s'$
by (*simp add: assign-def Let-def*)

lemma *assign-Norm-Some* [simp]:
 $\llbracket abrupt\ (f\ v\ (Norm\ s)) = Some\ y \rrbracket$
 $\implies assign\ f\ v\ (Norm\ s) = (Some\ y, s)$
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]: $\neg normal\ s \implies assign\ f\ v\ s = s$
by (*auto simp add: assign-def Let-def split-beta*)

lemma *assign-supd* [simp]:
 $assign\ (\lambda v. supd\ (f\ v))\ v\ (x, s)$
 $= (x, \text{if } x = None \text{ then } f\ v\ s \text{ else } s)$
apply *auto*
done

lemma *assign-raise-if* [simp]:
 $assign\ (\lambda v\ (x, s). ((raise\ if\ (b\ s\ v)\ xcpt)\ x, f\ v\ s))\ v\ (x, s) =$
 $(raise\ if\ (b\ s\ v)\ xcpt\ x, \text{if } x = None \wedge \neg b\ s\ v \text{ then } f\ v\ s \text{ else } s)$
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* \Rightarrow *st* \Rightarrow *val* \Rightarrow *ref-ty* \Rightarrow *qname* **where**
invocation-class *m* *s* *a'* *statT* =
 (*case m of*
Static \Rightarrow *if* ($\exists statC. statT = ClassT\ statC$)
 then the-Class (*RefT* *statT*)
 else Object
 | *SuperM* \Rightarrow *the-Class* (*RefT* *statT*)
 | *IntVir* \Rightarrow *obj-class* (*lookup-obj* *s* *a'*)

definition

invocation-declclass :: *prog* \Rightarrow *inv-mode* \Rightarrow *st* \Rightarrow *val* \Rightarrow *ref-ty* \Rightarrow *sig* \Rightarrow *qname* **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* ⇒ *qname* ⇒ *sig* ⇒ *inv-mode* ⇒ *val* ⇒ *val list* ⇒ *state* ⇒ *state*
where
init-lvars G C sig mode a' pvs =
(λ(x,s).
let m = methd (the (methd G C sig));
l = λ k.
(case k of
ENam 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
ENam e
⇒ (case e of
VNam v
⇒ (Map.empty ((pars (methd (the (methd G C sig))))[↦]pvs)) v
| Res ⇒ None)
| This
⇒ (if mode=Static then None else Some a'))
(if mode = Static then x else np a' x,s)

apply (unfold init-lvars-def)
apply (simp (no-asm) add: Let-def)
done

definition

body :: *prog* ⇒ *qname* ⇒ *sig* ⇒ *expr* **where**
body G C sig =
(let m = the (methd G C sig)
in Body (declclass m) (stmt (mbdy (methd 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 :: qname ⇒ bool ⇒ vname ⇒ val ⇒ state ⇒ vvar × state where
fvar C stat fn a' s =
  (let (oref,xf) = if stat then (Stat C,id)
      else (Heap (the-Addr a'),np a');
      n = Inl (fn,C);
      f = (λv. supd (upd-gobj oref n v))
  in ((the (values (the (globs (store s) oref)) n),f),abupd xf s))

```

definition

```

avar :: prog ⇒ val ⇒ val ⇒ state ⇒ vvar × state where
avar G i' a' s =
  (let oref = Heap (the-Addr a');
      i = the-Intg i';
      n = Inr i;
      (T,k,cs) = the-Arr (globs (store s) oref);
      f = (λv (x,s). (raise-if (¬G,s⊢v fits T)
                              ArrStore x
                              ,upd-gobj oref n v s))
  in ((the (cs n),f),abupd (raise-if (¬i in-bounds k) IndOutBound ∘ np a') s))

```

lemma fvar-def2: — better suited for simplification

```

fvar C stat fn a' s =
  ((the
    (values
      (the (globs (store s) (if stat then Stat C else Heap (the-Addr a'))))
      (Inl (fn,C)))
    , (λv. supd (upd-gobj (if stat then Stat C else Heap (the-Addr a'))
                    (Inl (fn,C))
                    v))))
  ,abupd (if stat then id else np a') s)

```

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'))))
    , (λv (x,s'). (raise-if (¬G,s⊢v fits (fst(the-Arr (globs (store s)
                                                         (Heap (the-Addr a'))))))
                        ArrStore x
                        ,upd-gobj (Heap (the-Addr a'))
                        (Inr (the-Intg i')) v s'))
    ,abupd (raise-if (¬(the-Intg i') in-bounds (fst(snd(the-Arr (globs (store s)
                                                                (Heap (the-Addr a'))))))
                    IndOutBound ∘ np a')
    s)

```

apply (*unfold avar-def*)
apply (*simp (no-asm) add: Let-def split-beta*)
done

definition

check-field-access :: *prog* \Rightarrow *qname* \Rightarrow *qname* \Rightarrow *vname* \Rightarrow *bool* \Rightarrow *val* \Rightarrow *state* \Rightarrow *state* **where**
check-field-access *G accC statDeclC fn stat a' s* =
 (let *oref* = if *stat* then *Stat statDeclC*
 else *Heap (the-Addr a')*;
 dynC = case *oref* of
 Heap a \Rightarrow *obj-class (the (globs (store s) oref))*
 | *Stat C* \Rightarrow *C*;
 f = (*the (table-of (DeclConcepts.fields G dynC) (fn,statDeclC))*)
 in *abupd*
 (error-if ($\neg G \vdash \text{Field } fn \text{ (statDeclC,f) in dynC dyn-accessible-from accC}$)
 AccessViolation)
 s)

definition

check-method-access :: *prog* \Rightarrow *qname* \Rightarrow *ref-ty* \Rightarrow *inv-mode* \Rightarrow *sig* \Rightarrow *val* \Rightarrow *state* \Rightarrow *state* **where**
check-method-access *G accC statT mode sig a' s* =
 (let *invC* = *invocation-class mode (store s) a' statT*;
 dynM = *the (dynlookup G statT invC sig)*
 in *abupd*
 (error-if ($\neg G \vdash \text{Methd sig dynM in invC dyn-accessible-from accC}$)
 AccessViolation)
 s)

evaluation judgments

inductive

halloc :: [*prog, state, obj-tag, loc, state*] \Rightarrow *bool* ($\hookleftarrow - \text{halloc} \rightarrow \rightarrow [61,61,61,61,61]60$) **for** *G::prog*
where — allocating objects on the heap, cf. 12.5

Abrupt:

$G \vdash (\text{Some } x, s) \text{ --halloc } oi \rightarrow \text{undefined} \rightarrow (\text{Some } x, s)$

| *New*: $\llbracket \text{new-Addr (heap s) = Some a;} \text{ } (x, oi') = (\text{if atleast-free (heap s) (Suc (Suc 0)) then (None, oi)} \text{ else (Some (Xcpt (Loc a)), CInst (SXcpt OutOfMemory))} \rrbracket$
 \implies
 $G \vdash \text{Norm } s \text{ --halloc } oi \rightarrow a \rightarrow (x, \text{init-obj } G \text{ } oi' \text{ (Heap a) } s)$

inductive *sxalloc* :: [*prog, state, state*] \Rightarrow *bool* ($\hookleftarrow - \text{sxalloc} \rightarrow \rightarrow [61,61,61]60$) **for** *G::prog*
where — allocating exception objects for standard exceptions (other than OutOfMemory)

Norm: $G \vdash \text{Norm} \quad s \text{ --sxalloc} \rightarrow \text{Norm} \quad s$

| *Jmp*: $G \vdash (\text{Some (Jump } j), s) \text{ --sxalloc} \rightarrow (\text{Some (Jump } j), s)$

| *Error*: $G \vdash (\text{Some (Error } e), s) \text{ --sxalloc} \rightarrow (\text{Some (Error } e), s)$

| *XcptL*: $G \vdash (\text{Some (Xcpt (Loc a)), } s) \text{ --sxalloc} \rightarrow (\text{Some (Xcpt (Loc a)), } s)$

| *SXcpt*: $\llbracket G \vdash \text{Norm } s0 \text{ --halloc (CInst (SXcpt } xn)) \rightarrow a \rightarrow (x, s1) \rrbracket \implies$
 $G \vdash (\text{Some (Xcpt (Std } xn)), s0) \text{ --sxalloc} \rightarrow (\text{Some (Xcpt (Loc a)), } s1)$

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*: $\llbracket G⊢Norm\ s0 -e->v→ s1 \rrbracket ⇒$

$G⊢Norm\ s0 -Expr\ e→ s1$

| *Lab*: $\llbracket G⊢Norm\ s0 -c → s1 \rrbracket ⇒$

$G⊢Norm\ s0 -l⋅ c→ abupd\ (absorb\ l)\ s1$

— cf. 14.2

| *Comp*: $\llbracket G⊢Norm\ s0 -c1 → s1;$

$G⊢ s1 -c2 → s2 \rrbracket ⇒$

$G⊢Norm\ s0 -c1;; c2→ s2$

— cf. 14.8.2

| *If*: $\llbracket G⊢Norm\ s0 -e->b→ s1;$

$G⊢ s1-(if\ the-Bool\ b\ then\ c1\ else\ c2)→ s2 \rrbracket ⇒$

$G⊢Norm\ s0 -If(e)\ c1\ Else\ c2 → s2$

— cf. 14.10, 14.10.1

— A continue jump from the while body c is handled by this rule. If a continue jump with the proper label was invoked inside c this label (Cont l) is deleted out of the abrupt component of the state before the iterative evaluation of the while statement. A break jump is handled by the Lab Statement *Lab* l (*while*...).

| *Loop*: $\llbracket G⊢Norm\ s0 -e->b→ s1;$

if the-Bool b

then $(G⊢s1 -c→ s2 \wedge$

$G⊢(abupd\ (absorb\ (Cont\ l))\ s2) -l⋅ While(e)\ c→ s3)$

else $s3 = s1 \rrbracket ⇒$

$G⊢Norm\ s0 -l⋅ While(e)\ c→ s3$

| *Jmp*: $G⊢Norm\ s -Jmp\ j→ (Some\ (Jump\ j), s)$

— cf. 14.16

| *Throw*: $\llbracket G⊢Norm\ s0 -e->a'→ s1 \rrbracket ⇒$

$G⊢Norm\ s0 -Throw\ e→ abupd\ (throw\ a')\ s1$

— cf. 14.18.1

| *Try*: $\llbracket G \vdash \text{Norm } s0 \text{ } -c1 \rightarrow s1; G \vdash s1 \text{ } -xalloc \rightarrow s2;$
 $\text{if } G, s2 \vdash \text{catch } C \text{ then } G \vdash \text{new-xcpt-var } vn \text{ } s2 \text{ } -c2 \rightarrow s3 \text{ else } s3 = s2 \rrbracket \implies$
 $G \vdash \text{Norm } s0 \text{ } -\text{Try } c1 \text{ } \text{Catch}(C \text{ } vn) \text{ } c2 \rightarrow s3$

— cf. 14.18.2

| *Fin*: $\llbracket G \vdash \text{Norm } s0 \text{ } -c1 \rightarrow (x1, s1);$
 $G \vdash \text{Norm } s1 \text{ } -c2 \rightarrow s2;$
 $s3 = (\text{if } (\exists \text{ err. } x1 = \text{Some } (\text{Error } \text{err}))$
 $\text{then } (x1, s1)$
 $\text{else } \text{abupd } (\text{abrupt-if } (x1 \neq \text{None}) \text{ } x1) \text{ } s2) \rrbracket$
 \implies
 $G \vdash \text{Norm } s0 \text{ } -c1 \text{ } \text{Finally } c2 \rightarrow s3$

— cf. 12.4.2, 8.5

| *Init*: $\llbracket \text{the } (\text{class } G \text{ } C) = c;$
 $\text{if } \text{inited } C \text{ } (\text{globs } s0) \text{ then } s3 = \text{Norm } s0$
 $\text{else } (G \vdash \text{Norm } (\text{init-class-obj } G \text{ } C \text{ } s0)$
 $\text{ } -(\text{if } C = \text{Object then Skip else Init } (\text{super } c)) \rightarrow s1 \wedge$
 $G \vdash \text{set-lvars Map.empty } s1 \text{ } -\text{init } c \rightarrow s2 \wedge s3 = \text{restore-lvars } s1 \text{ } s2) \rrbracket$
 \implies
 $G \vdash \text{Norm } s0 \text{ } -\text{Init } C \rightarrow s3$

— This class initialisation rule is a little bit inaccurate. Look at the exact sequence: (1) The current class object (the static fields) are initialised (*init-class-obj*), (2) the superclasses are initialised, (3) the static initialiser of the current class is invoked. More precisely we should expect another ordering, namely 2 1 3. But we can't just naively toggle 1 and 2. By calling *init-class-obj* before initialising the superclasses, we also implicitly record that we have started to initialise the current class (by setting an value for the class object). This becomes crucial for the completeness proof of the axiomatic semantics *AxCompl.thy*. Static initialisation requires an induction on the number of classes not yet initialised (or to be more precise, classes where the initialisation has not yet begun). So we could first assign a dummy value to the class before superclass initialisation and afterwards set the correct values. But as long as we don't take memory overflow into account when allocating class objects, we can leave things as they are for convenience.

— evaluation of expressions

— cf. 15.8.1, 12.4.1

| *NewC*: $\llbracket G \vdash \text{Norm } s0 \text{ } -\text{Init } C \rightarrow s1;$
 $G \vdash \text{ } s1 \text{ } -\text{halloc } (C \text{Inst } C) \succ a \rightarrow s2 \rrbracket \implies$
 $G \vdash \text{Norm } s0 \text{ } -\text{NewC } C \text{ } -\succ \text{Addr } a \rightarrow s2$

— cf. 15.9.1, 12.4.1

| *NewA*: $\llbracket G \vdash \text{Norm } s0 \text{ } -\text{init-comp-ty } T \rightarrow s1; G \vdash s1 \text{ } -e \succ i' \rightarrow s2;$
 $G \vdash \text{abupd } (\text{check-neg } i') \text{ } s2 \text{ } -\text{halloc } (\text{Arr } T \text{ } (\text{the-Intg } i')) \succ a \rightarrow s3 \rrbracket \implies$
 $G \vdash \text{Norm } s0 \text{ } -\text{New } T[e] \text{ } -\succ \text{Addr } a \rightarrow s3$

— cf. 15.15

| *Cast*: $\llbracket G \vdash \text{Norm } s0 \text{ } -e \succ v \rightarrow s1;$
 $s2 = \text{abupd } (\text{raise-if } (\neg G, \text{store } s1 \vdash v \text{ fits } T) \text{ } \text{ClassCast}) \text{ } s1 \rrbracket \implies$
 $G \vdash \text{Norm } s0 \text{ } -\text{Cast } T \text{ } e \succ v \rightarrow s2$

— cf. 15.19.2

| *Inst*: $\llbracket G \vdash \text{Norm } s0 \text{ } -e \succ v \rightarrow s1;$
 $b = (v \neq \text{Null} \wedge G, \text{store } s1 \vdash v \text{ fits } \text{RefT } T) \rrbracket \implies$
 $G \vdash \text{Norm } s0 \text{ } -e \text{ } \text{InstOf } T \text{ } -\succ \text{Bool } b \rightarrow s1$

— cf. 15.7.1

| *Lit*: $G \vdash \text{Norm } s \text{ } -\text{Lit } v \text{ } -\succ v \rightarrow \text{Norm } s$

| *UnOp*: $\llbracket G \vdash \text{Norm } s0 \text{ } -e \succ v \rightarrow s1 \rrbracket$
 $\implies G \vdash \text{Norm } s0 \text{ } -\text{UnOp } \text{unop } e \text{ } -\succ (\text{eval-unop } \text{unop } v) \rightarrow s1$

| *BinOp*: $\llbracket G \vdash \text{Norm } s0 \text{ } -e1-\succ v1 \rightarrow s1;$
 $G \vdash s1 \text{ } -(if \text{ need-second-arg } binop \text{ } v1 \text{ then } (In1l \text{ } e2) \text{ else } (In1r \text{ } Skip))$
 $\succ \rightarrow (In1 \text{ } v2, s2)$
 \rrbracket
 $\implies G \vdash \text{Norm } s0 \text{ } -BinOp \text{ } binop \text{ } e1 \text{ } e2-\succ (eval-binop \text{ } binop \text{ } v1 \text{ } v2) \rightarrow s2$

— cf. 15.10.2

| *Super*: $G \vdash \text{Norm } s \text{ } -Super-\succ val-this \text{ } s \rightarrow \text{Norm } s$

— cf. 15.2

| *Acc*: $\llbracket G \vdash \text{Norm } s0 \text{ } -va=\succ (v,f) \rightarrow s1 \rrbracket \implies$
 $G \vdash \text{Norm } s0 \text{ } -Acc \text{ } va-\succ v \rightarrow s1$

— cf. 15.25.1

| *Ass*: $\llbracket G \vdash \text{Norm } s0 \text{ } -va=\succ (w,f) \rightarrow s1;$
 $G \vdash s1 \text{ } -e-\succ v \rightarrow s2 \rrbracket \implies$
 $G \vdash \text{Norm } s0 \text{ } -va:=e-\succ v \rightarrow assign \text{ } f \text{ } v \text{ } s2$

— cf. 15.24

| *Cond*: $\llbracket G \vdash \text{Norm } s0 \text{ } -e0-\succ b \rightarrow s1;$
 $G \vdash s1 \text{ } -(if \text{ the-Bool } b \text{ then } e1 \text{ else } e2)-\succ v \rightarrow s2 \rrbracket \implies$
 $G \vdash \text{Norm } s0 \text{ } -e0 \text{ } ? \text{ } e1 : e2-\succ v \rightarrow s2$

— The interplay of *Call*, *Method* and *Body*: Method invocation is split up into these three rules:

Call Calculates the target address and evaluates the arguments of the method, and then performs dynamic or static lookup of the method, corresponding to the call mode. Then the *Method* rule is evaluated on the calculated declaration class of the method invocation.

Method A syntactic bridge for the folded method body. It is used by the axiomatic semantics to add the proper hypothesis for recursive calls of the method.

Body An extra syntactic entity for the unfolded method body was introduced to properly trigger class initialisation. Without class initialisation we could just evaluate the body statement.

— cf. 15.11.4.1, 15.11.4.2, 15.11.4.4, 15.11.4.5

| *Call*:

$\llbracket G \vdash \text{Norm } s0 \text{ } -e-\succ a' \rightarrow s1; G \vdash s1 \text{ } -args=\succ vs \rightarrow s2;$
 $D = invocation-declclass \text{ } G \text{ mode } (store \text{ } s2) \text{ } a' \text{ statT } (\llbracket name=mn, parTs=pTs \rrbracket);$
 $s3 = init-lvars \text{ } G \text{ } D (\llbracket name=mn, parTs=pTs \rrbracket) \text{ mode } a' \text{ vs } s2;$
 $s3' = check-method-access \text{ } G \text{ } accC \text{ statT } mode (\llbracket name=mn, parTs=pTs \rrbracket) \text{ } a' \text{ } s3;$
 $G \vdash s3' \text{ } -Method \text{ } D (\llbracket name=mn, parTs=pTs \rrbracket) -\succ v \rightarrow s4 \rrbracket$
 \implies

$G \vdash \text{Norm } s0 \text{ } -\{accC, statT, mode\}e.mn(\{pTs\}args)-\succ v \rightarrow (restore-lvars \text{ } s2 \text{ } s4)$

— 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.

| *Method*: $\llbracket G \vdash \text{Norm } s0 \text{ } -body \text{ } G \text{ } D \text{ } sig-\succ v \rightarrow s1 \rrbracket \implies$
 $G \vdash \text{Norm } s0 \text{ } -Method \text{ } D \text{ } sig-\succ v \rightarrow s1$

| *Body*: $\llbracket G \vdash \text{Norm } s0 \text{ } -Init \text{ } D \rightarrow s1; G \vdash s1 \text{ } -c \rightarrow s2;$
 $s3 = (if (\exists l. abrupt \text{ } s2 = Some (Jump (Break \text{ } l))) \vee$
 $abrupt \text{ } s2 = Some (Jump (Cont \text{ } l)))$
 $then abrupt (\lambda x. Some (Error \text{ } CrossMethodJump)) \text{ } s2$
 $else \text{ } s2 \rrbracket \implies$
 $G \vdash \text{Norm } s0 \text{ } -Body \text{ } D \text{ } c-\succ the \text{ } (locals \text{ } (store \text{ } s2) \text{ } Result)$
 $\rightarrow abrupt \text{ } (absorb \text{ } Ret) \text{ } s3$

— cf. 14.15, 12.4.1

— We filter out a break/continue in $s2$, so that we can proof definite assignment correct, without the need of conformance of the state. By this the different parts of the typesafety proof can be disentangled a little.

— evaluation of variables

— cf. 15.13.1, 15.7.2

| *LVar*: $G \vdash \text{Norm } s \text{ --LVar } vn \Rightarrow \text{lvar } vn \ s \rightarrow \text{Norm } s$

— cf. 15.10.1, 12.4.1

| *FVar*: $\llbracket G \vdash \text{Norm } s0 \text{ --Init statDeclC} \rightarrow s1; G \vdash s1 \text{ --e--}\triangleright a \rightarrow s2; \\ (v, s2') = \text{fvar statDeclC stat fn a } s2; \\ s3 = \text{check-field-access } G \text{ accC statDeclC fn stat a } s2' \rrbracket \implies \\ G \vdash \text{Norm } s0 \text{ --}\{accC, statDeclC, stat\}e..fn \Rightarrow v \rightarrow s3$

— The accessibility check is after *fvar*, to keep it simple. *fvar* already tests for the absence of a null-pointer reference in case of an instance field

— cf. 15.12.1, 15.25.1

| *AVar*: $\llbracket G \vdash \text{Norm } s0 \text{ --e1--}\triangleright a \rightarrow s1; G \vdash s1 \text{ --e2--}\triangleright i \rightarrow s2; \\ (v, s2') = \text{avar } G \ i \ a \ s2 \rrbracket \implies \\ G \vdash \text{Norm } s0 \text{ --e1.[e2]=}\triangleright v \rightarrow s2'$

— evaluation of expression lists

— cf. 15.11.4.2

| *Nil*:

$$G \vdash \text{Norm } s0 \text{ --}[] \dot{=} \triangleright [] \rightarrow \text{Norm } s0$$

— cf. 15.6.4

| *Cons*: $\llbracket G \vdash \text{Norm } s0 \text{ --e--}\triangleright v \rightarrow s1; \\ G \vdash s1 \text{ --es}\dot{=} \triangleright vs \rightarrow s2 \rrbracket \implies \\ G \vdash \text{Norm } s0 \text{ --e\#es}\dot{=} \triangleright v\#vs \rightarrow s2$

ML \langle

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\}

\rangle

declare *if-split* $[split \ del] \text{ if-split-asm } [split \ del]$
 $option.split \ [split \ del] \ option.split-asm \ [split \ del]$

inductive-cases *halloc-elim-cases*:

$G \vdash (\text{Some } xc, s) \text{ --halloc } oi \triangleright a \rightarrow s'$
 $G \vdash (\text{Norm } s) \text{ --halloc } oi \triangleright a \rightarrow s'$

inductive-cases *sxalloc-elim-cases*:

$G \vdash \text{Norm } s \text{ --sxalloc} \rightarrow s'$
 $G \vdash (\text{Some } (Jump \ j), s) \text{ --sxalloc} \rightarrow s'$
 $G \vdash (\text{Some } (Error \ e), s) \text{ --sxalloc} \rightarrow s'$
 $G \vdash (\text{Some } (Xcpt \ (Loc \ a \)), s) \text{ --sxalloc} \rightarrow s'$
 $G \vdash (\text{Some } (Xcpt \ (Std \ xn)), s) \text{ --sxalloc} \rightarrow s'$

inductive-cases *sxalloc-cases*: $G \vdash s \text{ --sxalloc} \rightarrow s'$

lemma *sxalloc-elim-cases2*: $\llbracket G \vdash s \text{ --sxalloc} \rightarrow s' \rrbracket$

$$\begin{aligned} \bigwedge s. \llbracket s' = \text{Norm } s \rrbracket &\implies P; \\ \bigwedge j s. \llbracket s' = (\text{Some } (\text{Jump } j), s) \rrbracket &\implies P; \\ \bigwedge e s. \llbracket s' = (\text{Some } (\text{Error } e), s) \rrbracket &\implies P; \\ \bigwedge a s. \llbracket s' = (\text{Some } (\text{Xcpt } (\text{Loc } a)), s) \rrbracket &\implies P \\ \llbracket &\implies P \end{aligned}$$

apply *cut-tac*

apply (*erule salloc-cases*)

apply *blast+*

done

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

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

setup $\langle \text{map-theory-simpset } (fn \text{ ctxt } \Rightarrow \text{ ctxt } |> \text{Simplifier.del-loop split-all-tac}) \rangle$

inductive-cases *eval-cases*: $G \vdash s - t \succ \rightarrow (v, s')$

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

$G \vdash (\text{Some } xc, s) - t$	$\succ \rightarrow (v, s')$
$G \vdash \text{Norm } s - \text{In1r Skip}$	$\succ \rightarrow (x, s')$
$G \vdash \text{Norm } s - \text{In1r } (\text{Jmp } j)$	$\succ \rightarrow (x, s')$
$G \vdash \text{Norm } s - \text{In1r } (l \cdot c)$	$\succ \rightarrow (x, s')$
$G \vdash \text{Norm } s - \text{In3 } (\llbracket \rrbracket)$	$\succ \rightarrow (v, s')$
$G \vdash \text{Norm } s - \text{In3 } (e \# es)$	$\succ \rightarrow (v, s')$
$G \vdash \text{Norm } s - \text{In1l } (\text{Lit } w)$	$\succ \rightarrow (v, s')$
$G \vdash \text{Norm } s - \text{In1l } (\text{UnOp } unop \ e)$	$\succ \rightarrow (v, s')$
$G \vdash \text{Norm } s - \text{In1l } (\text{BinOp } binop \ e1 \ e2)$	$\succ \rightarrow (v, s')$
$G \vdash \text{Norm } s - \text{In2 } (\text{LVar } vn)$	$\succ \rightarrow (v, s')$
$G \vdash \text{Norm } s - \text{In1l } (\text{Cast } T \ e)$	$\succ \rightarrow (v, s')$
$G \vdash \text{Norm } s - \text{In1l } (e \ \text{InstOf } T)$	$\succ \rightarrow (v, s')$
$G \vdash \text{Norm } s - \text{In1l } (\text{Super})$	$\succ \rightarrow (v, s')$
$G \vdash \text{Norm } s - \text{In1l } (\text{Acc } va)$	$\succ \rightarrow (v, s')$
$G \vdash \text{Norm } s - \text{In1r } (\text{Expr } e)$	$\succ \rightarrow (x, s')$
$G \vdash \text{Norm } s - \text{In1r } (c1;; c2)$	$\succ \rightarrow (x, s')$
$G \vdash \text{Norm } s - \text{In1l } (\text{Methd } C \ sig)$	$\succ \rightarrow (x, s')$
$G \vdash \text{Norm } s - \text{In1l } (\text{Body } D \ c)$	$\succ \rightarrow (x, s')$
$G \vdash \text{Norm } s - \text{In1l } (e0 \ ? \ e1 : e2)$	$\succ \rightarrow (v, s')$
$G \vdash \text{Norm } s - \text{In1r } (\text{If}(e) \ c1 \ \text{Else } c2)$	$\succ \rightarrow (x, s')$
$G \vdash \text{Norm } s - \text{In1r } (l \cdot \text{While}(e) \ c)$	$\succ \rightarrow (x, s')$
$G \vdash \text{Norm } s - \text{In1r } (c1 \ \text{Finally } c2)$	$\succ \rightarrow (x, s')$
$G \vdash \text{Norm } s - \text{In1r } (\text{Throw } e)$	$\succ \rightarrow (x, s')$
$G \vdash \text{Norm } s - \text{In1l } (\text{NewC } C)$	$\succ \rightarrow (v, s')$
$G \vdash \text{Norm } s - \text{In1l } (\text{New } T[e])$	$\succ \rightarrow (v, s')$
$G \vdash \text{Norm } s - \text{In1l } (\text{Ass } va \ e)$	$\succ \rightarrow (v, s')$
$G \vdash \text{Norm } s - \text{In1r } (\text{Try } c1 \ \text{Catch}(tn \ vn) \ c2)$	$\succ \rightarrow (x, s')$
$G \vdash \text{Norm } s - \text{In2 } (\{accC, statDeclC, stat\}e..fn)$	$\succ \rightarrow (v, s')$
$G \vdash \text{Norm } s - \text{In2 } (e1.[e2])$	$\succ \rightarrow (v, s')$
$G \vdash \text{Norm } s - \text{In1l } (\{accC, statT, mode\}e.mn(\{pT\}p))$	$\succ \rightarrow (v, s')$
$G \vdash \text{Norm } s - \text{In1r } (\text{Init } C)$	$\succ \rightarrow (x, s')$

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

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

declaration $\langle K (\text{Simplifier.map-ss } (fn \text{ ss } \Rightarrow \text{ ss } |> \text{Simplifier.add-loop } (\text{split-all-tac}, \text{split-all-tac}))) \rangle$

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

option.split [*split*] *option.split-asm* [*split*]

lemma *eval-Inj-elim*:

$G \vdash s - t \succ \rightarrow (w, s')$

\implies *case t of*

In1 ec \Rightarrow (*case ec of*

In1 e \Rightarrow ($\exists v. w = \text{In1 } v$)

```

      | Inr c  $\Rightarrow$  w =  $\Diamond$ )
    | 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: $G \vdash s - \text{In1l } t \rightarrow (w, s') = (\exists v. w = \text{In1 } v \wedge G \vdash s - t \rightarrow v \rightarrow s')$
by (auto, frule eval-Inj-elim, auto)

lemma eval-var-eq: $G \vdash s - \text{In2 } t \rightarrow (w, s') = (\exists vf. w = \text{In2 } vf \wedge G \vdash s - t \rightarrow vf \rightarrow s')$
by (auto, frule eval-Inj-elim, auto)

lemma eval-exprs-eq: $G \vdash s - \text{In3 } t \rightarrow (w, s') = (\exists vs. w = \text{In3 } vs \wedge G \vdash s - t \rightarrow vs \rightarrow s')$
by (auto, frule eval-Inj-elim, auto)

lemma eval-stmt-eq: $G \vdash s - \text{In1r } t \rightarrow (w, s') = (w = \Diamond \wedge G \vdash s - t \rightarrow s')$
by (auto, frule eval-Inj-elim, auto, frule eval-Inj-elim, auto)

simproc-setup eval-expr ($G \vdash s - \text{In1l } t \rightarrow (w, s') = \langle$
 $K (K (fn ct =>$
 $(case Thm.term-of ct of$
 $(- \$ - \$ - \$ (Const - \$ -) \$ -) => NONE$
 $| - => SOME (mk-meta-eq @\{thm eval-expr-eq\}))))\rangle$

simproc-setup eval-var ($G \vdash s - \text{In2 } t \rightarrow (w, s') = \langle$
 $K (K (fn ct =>$
 $(case Thm.term-of ct of$
 $(- \$ - \$ - \$ (Const - \$ -) \$ -) => NONE$
 $| - => SOME (mk-meta-eq @\{thm eval-var-eq\}))))\rangle$

simproc-setup eval-exprs ($G \vdash s - \text{In3 } t \rightarrow (w, s') = \langle$
 $K (K (fn ct =>$
 $(case Thm.term-of ct of$
 $(- \$ - \$ - \$ (Const - \$ -) \$ -) => NONE$
 $| - => SOME (mk-meta-eq @\{thm eval-exprs-eq\}))))\rangle$

simproc-setup eval-stmt ($G \vdash s - \text{In1r } t \rightarrow (w, s') = \langle$
 $K (K (fn ct =>$
 $(case Thm.term-of ct of$
 $(- \$ - \$ - \$ (Const - \$ -) \$ -) => NONE$
 $| - => SOME (mk-meta-eq @\{thm eval-stmt-eq\}))))\rangle$

ML \langle
 $ML\text{-Thms.bind-thms (AbruptIs, sum3-instantiate context @\{thm eval.Abrupt\})$
 \rangle

declare *halloc.Abrupt* [*intro!*] *eval.Abrupt* [*intro!*] *AbruptIs* [*intro!*]

Callee, *InsInitE*, *InsInitV*, *FinA* are only used in smallstep semantics, not in the bigstep semantics. So there is no valid evaluation of these terms

lemma eval-Callee: $G \vdash \text{Norm } s - \text{Callee } l \text{ e} \rightarrow v \rightarrow s' = \text{False}$

proof –

have *False*
 if *eval*: $G \vdash s \multimap t \rightarrow (v, s')$
 and *normal*: *normal* *s*
 and *callee*: $t = \text{In1l} \text{ (Callee } l \text{ e)}$
 for *s t v s'*
 using *that* **by** *induct auto*
 then show *?thesis*
 by (*cases s'*) *fastforce*
qed

lemma *eval-InsInitE*: $G \vdash \text{Norm } s \text{--InsInitE } c \text{ e} \multimap v \rightarrow s' = \text{False}$

proof –

have *False*
 if *eval*: $G \vdash s \multimap t \rightarrow (v, s')$
 and *normal*: *normal* *s*
 and *callee*: $t = \text{In1l} \text{ (InsInitE } c \text{ e)}$
 for *s t v s'*
 using *that* **by** *induct auto*
 then show *?thesis*
 by (*cases s'*) *fastforce*
qed

lemma *eval-InsInitV*: $G \vdash \text{Norm } s \text{--InsInitV } c \text{ w} \multimap v \rightarrow s' = \text{False}$

proof –

have *False*
 if *eval*: $G \vdash s \multimap t \rightarrow (v, s')$
 and *normal*: *normal* *s*
 and *callee*: $t = \text{In2} \text{ (InsInitV } c \text{ w)}$
 for *s t v s'*
 using *that* **by** *induct auto*
 then show *?thesis*
 by (*cases s'*) *fastforce*
qed

lemma *eval-FinA*: $G \vdash \text{Norm } s \text{--FinA } a \text{ c} \rightarrow s' = \text{False}$

proof –

have *False*
 if *eval*: $G \vdash s \multimap t \rightarrow (v, s')$
 and *normal*: *normal* *s*
 and *callee*: $t = \text{In1r} \text{ (FinA } a \text{ c)}$
 for *s t v s'*
 using *that* **by** *induct auto*
 then show *?thesis*
 by (*cases s'*) *fastforce*
qed

lemma *eval-no-abrupt-lemma*:

$\bigwedge s \text{ s'}. G \vdash s \multimap t \rightarrow (w, s') \implies \text{normal } s' \longrightarrow \text{normal } s$
by (*erule eval-cases, auto*)

lemma *eval-no-abrupt*:

$G \vdash (x, s) \multimap t \rightarrow (w, \text{Norm } s') =$
 $(x = \text{None} \wedge G \vdash \text{Norm } s \multimap t \rightarrow (w, \text{Norm } s'))$

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 initd 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*:

$G \vdash s - \text{In1}(\text{body } G \ C \ \text{sig}) \succ \rightarrow (w, s')$
 $\implies G \vdash s - \text{In1}(\text{Methd } C \ \text{sig}) \succ \rightarrow (w, s')$

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*: $\llbracket G \vdash \text{Norm } s0 - \text{Init } D \rightarrow s1; G \vdash s1 - c \rightarrow s2;$

$\text{res} = \text{the } (\text{locals } (\text{store } s2) \ \text{Result});$

$s3 = (\text{if } (\exists l. \text{abrupt } s2 = \text{Some } (\text{Jump } (\text{Break } l))) \vee$
 $\text{abrupt } s2 = \text{Some } (\text{Jump } (\text{Cont } l)))$

$\text{then } \text{abupd } (\lambda x. \text{Some } (\text{Error } \text{CrossMethodJump})) \ s2$
 $\text{else } s2);$

$s4 = \text{abupd } (\text{absorb } \text{Ret}) \ s3 \rrbracket \implies$

$G \vdash \text{Norm } s0 - \text{Body } D \ c - \succ \text{res} \rightarrow s4$

by (*auto elim: eval.Body*)

lemma *eval-binop-arg2-indep*:

$\neg \text{need-second-arg } \text{binop } v1 \implies \text{eval-binop } \text{binop } v1 \ x = \text{eval-binop } \text{binop } v1 \ y$

by (*cases binop*)

(*simp-all add: need-second-arg-def*)

lemma *eval-BinOp-arg2-indepI*:

assumes *eval-e1*: $G \vdash \text{Norm } s0 - e1 - \succ v1 \rightarrow s1$ **and**

no-need: $\neg \text{need-second-arg } \text{binop } v1$

shows $G \vdash \text{Norm } s0 - \text{BinOp } \text{binop } e1 \ e2 - \succ (\text{eval-binop } \text{binop } v1 \ v2) \rightarrow s1$
 (*is ?EvalBinOp v2*)

proof –

from *eval-e1*

have *?EvalBinOp Unit*

by (*rule eval.BinOp*)

(*simp add: no-need*)

moreover

from *no-need*

have $\text{eval-binop } \text{binop } v1 \ \text{Unit} = \text{eval-binop } \text{binop } v1 \ v2$

by (*simp add: eval-binop-arg2-indep*)

ultimately

show *?thesis*

by *simp*

qed

single valued

lemma *unique-halloc* [*rule-format* (*no-asm*)]:

$G \vdash s - \text{halloc } oi \succ a \rightarrow s' \implies G \vdash s - \text{halloc } oi \succ a' \rightarrow s'' \longrightarrow a' = a \wedge s'' = s'$

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*:
 $\text{single-valued } \{((s, oi), (a, s')). G \vdash s - \text{halloc } oi \succ a \rightarrow s'\}$
apply (*unfold single-valued-def*)
by (*clarsimp, drule (1) unique-halloc, auto*)

lemma *unique-sxalloc* [*rule-format (no-asm)*]:
 $G \vdash s - \text{sxalloc} \rightarrow s' \implies G \vdash s - \text{sxalloc} \rightarrow s'' \longrightarrow s'' = s'$
apply (*erule sxalloc.induct*)
apply (*auto dest: unique-halloc elim!: sxalloc-elim-cases*
 $\text{split del: if-split if-split-asm}$)
done

lemma *single-valued-sxalloc*: $\text{single-valued } \{(s, s'). G \vdash s - \text{sxalloc} \rightarrow s'\}$
apply (*unfold single-valued-def*)
apply (*blast dest: unique-sxalloc*)
done

lemma *split-pairD*: $(x, y) = p \implies x = \text{fst } p \ \& \ y = \text{snd } p$
by *auto*

lemma *unique-eval* [*rule-format (no-asm)*]:
 $G \vdash s - t \succ \rightarrow (w, s') \implies (\forall w' s''. G \vdash s - t \succ \rightarrow (w', s'') \longrightarrow w' = w \wedge s'' = s')$
apply (*erule eval-induct*)
apply (*tactic <ALLGOALS (EVERY'*
 $[\text{strip-tac } \textbf{context}, \text{rotate-tac } \sim 1, \text{eresolve-tac } \textbf{context} @ \{\text{thms eval-elim-cases}\}])>$)

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*:
 $\text{single-valued } \{((s, t), (v, s')). G \vdash s - t \succ \rightarrow (v, s')\}$
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*: $\bigwedge fd. \text{wf-fdecl } G \ P \ fd = \text{is-acc-type } G \ P \ (\text{type } (\text{snd } fd))$
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'* \Rightarrow *tnam* **and**
vnam' :: *vnam'* \Rightarrow *vname* **and**
label' :: *label'* \Rightarrow *label*

where

inj-tnam' [*simp*]: $\bigwedge x \ y. (tnam' \ x = tnam' \ y) = (x = y)$ **and**
inj-vnam' [*simp*]: $\bigwedge x \ y. (vnam' \ x = vnam' \ y) = (x = y)$ **and**
inj-label' [*simp*]: $\bigwedge x \ y. (label' \ x = label' \ y) = (x = y)$ **and**

surj-tnam': $\bigwedge n. \exists m. n = tnam' \ m$ **and**
surj-vnam': $\bigwedge n. \exists m. n = vnam' \ m$ **and**
surj-label': $\bigwedge n. \exists m. n = label' \ m$

abbreviation

HasFoo :: *qname* **where**
HasFoo == ($\langle pid=java-lang, tid=TName \ (tnam' \ HasFoo') \rangle$)

abbreviation

Base :: *qname* **where**
Base == ($\langle pid=java-lang, tid=TName \ (tnam' \ Base') \rangle$)

abbreviation

Ext :: *qname* **where**
Ext == ($\langle pid=java-lang, tid=TName \ (tnam' \ Ext') \rangle$)

abbreviation

Main :: *qname* **where**
Main == ($\langle pid=java-lang, tid=TName \ (tnam' \ Main') \rangle$)

abbreviation

arr :: *vname* **where**

arr == (*vnam'* *arr'*)

abbreviation

vee :: *vname* **where**
vee == (*vnam'* *vee'*)

abbreviation

z :: *vname* **where**
z == (*vnam'* *z'*)

abbreviation

e :: *vname* **where**
e == (*vnam'* *e'*)

abbreviation

lab1:: *label* **where**
lab1 == *label'* *lab1'*

lemma *neq-Base-Object* [*simp*]: *Base*≠*Object*
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,par*s=[*z*],*resT*=*Class Base*)

definition

```

Base-foo :: mdecl
where Base-foo = (foo-sig, (access=Public,static=False,pars=[z],resT=Class Base,
                           mbody=(lcls=[],stmt=Return (!!z)))))

```

```

definition Ext-foo :: mdecl
where Ext-foo = (foo-sig,
                 (access=Public,static=False,pars=[z],resT=Class Ext,
                  mbody=(lcls=[],
                         stmt=Expr({Ext,Ext,False} Cast (Class Ext) (!!z)..vee :=
                                   Lit (Intg 1)) ;;
                         Return (Lit Null)))
                 ))

```

```

definition
arr-viewed-from :: qname ⇒ qname ⇒ var
where arr-viewed-from 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

```

$(Acc \ (arr\text{-viewed-from} \ Main \ Ext).[Lit \ (Intg \ 2)])) \ Skip))$

well-structuredness

lemma *not-Object-subcls-any* [elim!]: $(Object, C) \in (subcls1 \ tprg)^+ \implies R$
apply (auto dest!: tranclD subcls1D)
done

lemma *not-Throwable-subcls-SXcpt* [elim!]:
 $(SXcpt \ Throwable, SXcpt \ xn) \in (subcls1 \ tprg)^+ \implies R$
apply (auto dest!: tranclD subcls1D)
apply (simp add: Object-def SXcpt-def)
done

lemma *not-SXcpt-n-subcls-SXcpt-n* [elim!]:
 $(SXcpt \ xn, SXcpt \ xn) \in (subcls1 \ tprg)^+ \implies R$
apply (auto dest!: tranclD subcls1D)
apply (drule rtranclD)
apply auto
done

lemma *not-Base-subcls-Ext* [elim!]: $(Base, Ext) \in (subcls1 \ tprg)^+ \implies R$
apply (auto dest!: tranclD subcls1D simp add: BaseCl-def)
done

lemma *not-TName-n-subcls-TName-n* [rule-format (no-asm), elim!]:
 $((\langle pid=java\text{-}lang, tid=TName \ tn \rangle, \langle pid=java\text{-}lang, tid=TName \ tn \rangle) \in (subcls1 \ tprg)^+ \implies R$
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*: $ws\text{-}idecl \ tprg \ HasFoo \ []$
apply (unfold ws-idecl-def)
apply (simp (no-asm))
done

lemma *ws-cdecl-Object*: $ws\text{-}cdecl \ tprg \ Object \ any$
apply (unfold ws-cdecl-def)
apply auto
done

lemma *ws-cdecl-Throwable*: $ws\text{-}cdecl \ tprg \ (SXcpt \ Throwable) \ Object$
apply (unfold ws-cdecl-def)
apply auto
done

lemma *ws-cdecl-SXcpt*: $ws\text{-}cdecl \ tprg \ (SXcpt \ xn) \ (SXcpt \ Throwable)$
apply (unfold ws-cdecl-def)
apply auto
done

lemma *ws-cdecl-Base*: $ws\text{-}cdecl \ tprg \ Base \ Object$

```

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  $\implies$  ( $\forall (I,i) \in \text{set Ifaces. ws-idecl } G \ I \ (\text{isuperIfs } i)$ )
apply (simp (no-asm) add: Ifaces-def HasFooInt-def)
apply (auto intro!: ws-idecl-HasFoo)
done

```

```

lemma ws-cdecl-all: G=tprg  $\implies$  ( $\forall (C,c) \in \text{set Classes. ws-cdecl } G \ C \ (\text{super } c)$ )
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]: $\text{tprg} \vdash \text{SXcpt } xn \preceq_C \text{ SXcpt Throwable}$
apply (rule *SXcpt-subcls-Throwable-lemma*)
apply auto
done

lemma *Ext-subclseq-Base* [simp]: $\text{tprg} \vdash \text{Ext} \preceq_C \text{ Base}$
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]: $\text{tprg} \vdash \text{Ext} \prec_C \text{ Base}$
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]: $\text{DeclConcepts.fields tprg Object} = []$
by (rule *ws-tprg [THEN fields-emptyI]*, force+)

lemma *fields-tprg-Throwable* [simp]:
 $\text{DeclConcepts.fields tprg (SXcpt Throwable)} = []$
by (rule *ws-tprg [THEN fields-emptyI]*, force+)

lemma *fields-tprg-SXcpt* [simp]: $\text{DeclConcepts.fields tprg (SXcpt } xn) = []$
apply (case-tac $xn = \text{Throwable}$)
apply (simp (no-asm-simp))
by (rule *ws-tprg [THEN fields-emptyI]*, force+)

lemmas $\text{fields-rec}' = \text{fields-rec} \text{ [OF - ws-tprg]}$

lemma *fields-Base* [simp]:
 $\text{DeclConcepts.fields tprg Base}$
 $= [((\text{arr}, \text{Base}), (\text{access} = \text{Public}, \text{static} = \text{True}, \text{type} = \text{PrimT Boolean}, [])),$
 $((\text{vee}, \text{Base}), (\text{access} = \text{Public}, \text{static} = \text{False}, \text{type} = \text{Iface HasFoo } []))]$
apply (subst *fields-rec'*)
apply (auto simp add: *BaseCl-def*)
done

lemma *fields-Ext* [simp]:
 $\text{DeclConcepts.fields tprg Ext}$
 $= [((\text{vee}, \text{Ext}), (\text{access} = \text{Public}, \text{static} = \text{False}, \text{type} = \text{PrimT Integer}))]$
 $@ \text{DeclConcepts.fields tprg Base}$
apply (rule *trans*)
apply (rule *fields-rec'*)
apply (auto simp add: *ExtCl-def Object-def*)
done

lemmas $\text{imethds-rec}' = \text{imethds-rec} \text{ [OF - ws-tprg]}$

lemmas $\text{methd-rec}' = \text{methd-rec} \text{ [OF - ws-tprg]}$

lemma *imethds-HasFoo* [simp]:
 $\text{imethds tprg HasFoo} = \text{set-option} \circ \text{Map.empty}(\text{foo-sig} \mapsto (\text{HasFoo}, \text{foo-mhead}))$

```

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  $[(\lambda(s,m). (s, Base, m)) Base\text{-}foo]$ 
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:
   $\neg$  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:
   $\neg$  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:
   $\neg$  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  $(\lambda sig\ m. tprg \vdash Ext\ inherits\ method\ sig\ m)$ 
    (Map.empty(fst  $((\lambda(s, m). (s, Base, m)) Base\text{-}foo) \mapsto$ 
      snd  $((\lambda(s, m). (s, Base, m)) Base\text{-}foo)))$ 
    = 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]] ⇒ ∃ 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:*

tp_{rg},sig_l⊢ (Base,(snd Base-foo)) hides old $\implies P$
by (auto dest: hidesD simp add: Base-foo-def member-is-static-simp)

lemma *Ext-foo-no-hide:*

tp_{rg},sig_l⊢ (Ext,(snd Ext-foo)) hides old $\implies P$
by (auto dest: hidesD simp add: Ext-foo-def member-is-static-simp)

lemma *unique-sig-Ext-foo:*

tp_{rg}⊢ mdecl (sig, snd Ext-foo) declared-in Ext $\implies sig=foo-sig$
by (auto simp add: declared-in-def cdeclaredmethd-def
Ext-foo-def ExtCl-def)

lemma *Ext-foo-override:*

tp_{rg},sig_l⊢ (Ext,(snd Ext-foo)) overrides old
 $\implies old = (Base,(snd Base-foo))$
apply (drule overrides-commonD)
apply (clarsimp)
apply (frule subclsEval)
apply (rule ws-tp_{rg})
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:*

tp_{rg},sig_l⊢ (Ext,(snd Ext-foo)) overrides_S old
 $\implies old = (Base,(snd Base-foo))$
apply (drule stat-overrides-commonD)
apply (clarsimp)
apply (frule subclsEval)
apply (rule ws-tp_{rg})
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:*

tp_{rg}⊢ (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  $\implies$  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  $\implies$  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*: $\bigwedge a \text{ obj. } \llbracket \text{the (new-Addr } h) = a; \text{atleast-free } h \text{ (Suc } n) \rrbracket \implies$
 $\text{new-Addr } h = \text{Some } a \wedge \text{atleast-free } (h(a \mapsto \text{obj})) \text{ } n$
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 $\langle \text{ML-Thms.bind-thms (eval-intros, map}$
 $(\text{simplify (context } \triangleright \text{Simplifier.del-simps } @\{\text{thms Skip-eq}\}$
 $\triangleright \text{Simplifier.add-simps } @\{\text{thms lvar-def}\}) \text{ } o$
 $\text{rewrite-rule context } @\{\text{thm assign-def}\}, @\{\text{thm Let-def}\}) @\{\text{thms eval.intros}\}) \rangle$
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 == $(\text{tag} = \text{Arr } (\text{PrimT Boolean}) \text{ } 2$
 $, \text{values} = \text{Map.empty}(\text{Inr } 0 \mapsto \text{Bool False}, \text{Inr } 1 \mapsto \text{Bool False}))$

abbreviation

obj-b == $(\text{tag} = \text{CInst Ext}$
 $, \text{values} = (\text{Map.empty}(\text{Inl } (\text{vee, Base}) \mapsto \text{Null}, \text{Inl } (\text{vee, Ext}) \mapsto \text{Intg } 0)))$

abbreviation

obj-c == $(\text{tag} = \text{CInst } (\text{SXcpt NullPointer}), \text{values} = \text{CONST Map.empty})$

abbreviation *arr-N* == $\text{Map.empty}(\text{Inl } (\text{arr, Base}) \mapsto \text{Null})$
abbreviation *arr-a* == $\text{Map.empty}(\text{Inl } (\text{arr, Base}) \mapsto \text{Addr } a)$

abbreviation

globs1 == $\text{Map.empty}(\text{Inr Ext} \mapsto (\text{tag} = \text{undefined}, \text{values} = \text{Map.empty}),$
 $\text{Inr Base} \mapsto (\text{tag} = \text{undefined}, \text{values} = \text{arr-N}),$
 $\text{Inr Object} \mapsto (\text{tag} = \text{undefined}, \text{values} = \text{Map.empty}))$

abbreviation

globs2 == $\text{Map.empty}(\text{Inr Ext} \mapsto (\text{tag} = \text{undefined}, \text{values} = \text{Map.empty}),$
 $\text{Inr Object} \mapsto (\text{tag} = \text{undefined}, \text{values} = \text{Map.empty}),$
 $\text{Inl } a \mapsto \text{obj-a},$
 $\text{Inr Base} \mapsto (\text{tag} = \text{undefined}, \text{values} = \text{arr-a}))$

abbreviation *globs3* == *globs2*(*Inl b* $\mapsto \text{obj-b}$)
abbreviation *globs8* == *globs3*(*Inl c* $\mapsto \text{obj-c}$)

```

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* × (*lname*, *ty*) *table*

extension of global store

definition *gext* :: *st* ⇒ *st* ⇒ *bool* ($\hookrightarrow \leq$) [71,71] 70) **where**
 $s \leq | s' \equiv \forall r. \forall \text{obj} \in \text{globs } s \ r: \exists \text{obj}' \in \text{globs } s' \ r: \text{tag } \text{obj}' = \text{tag } \text{obj}$

For the the proof of type soundness we will need the property that during execution, objects are not lost and moreover retain the values of their tags. So the object store grows conservatively. Note that if we considered garbage collection, we would have to restrict this property to accessible objects.

lemma *gext-objD*:

$\llbracket s \leq | s'; \text{globs } s \ r = \text{Some } \text{obj} \rrbracket$
 $\implies \exists \text{obj}'. \text{globs } s' \ r = \text{Some } \text{obj}' \wedge \text{tag } \text{obj}' = \text{tag } \text{obj}$
apply (*simp only: gext-def*)
by *force*

lemma *rev-gext-objD*:

$\llbracket \text{globs } s \ r = \text{Some } \text{obj}; s \leq | s' \rrbracket$
 $\implies \exists \text{obj}'. \text{globs } s' \ r = \text{Some } \text{obj}' \wedge \text{tag } \text{obj}' = \text{tag } \text{obj}$
by (*auto elim: gext-objD*)

lemma *init-class-obj-inited*:

$\text{init-class-obj } G \ C \ s1 \leq | s2 \implies \text{inited } C \ (\text{globs } s2)$
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!*]: $\bigwedge s. \text{globs } s \ r = \text{None} \implies s \leq | \text{gupd}(r \mapsto x) s$

by (auto simp: gext-def)

lemma gext-new [simp, elim!]: $\bigwedge s. \text{globs } s \ r = \text{None} \implies s \leq | \text{init-obj } G \text{ oi } r \ s$
apply (simp only: init-obj-def)
apply (erule-tac gext-gupd)
done

lemma gext-trans [elim]: $\bigwedge X. \llbracket s \leq | s'; s' \leq | s'' \rrbracket \implies s \leq | s''$
by (force simp: gext-def)

lemma gext-upd-gobj [intro!]: $s \leq | \text{upd-gobj } r \ n \ v \ s$
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]: $\text{set-locals } l \ s1 \leq | s2 = s1 \leq | s2$
by (auto simp: gext-def)

lemma gext-cong2 [simp]: $s1 \leq | \text{set-locals } l \ s2 = s1 \leq | s2$
by (auto simp: gext-def)

lemma gext-lupd1 [simp]: $\text{lupd}(vn \mapsto v) s1 \leq | s2 = s1 \leq | s2$
by (auto simp: gext-def)

lemma gext-lupd2 [simp]: $s1 \leq | \text{lupd}(vn \mapsto v) s2 = s1 \leq | s2$
by (auto simp: gext-def)

lemma init-ed-gext: $\llbracket \text{init-ed } C \ (\text{globs } s); s \leq | s' \rrbracket \implies \text{init-ed } C \ (\text{globs } s')$
apply (unfold init-ed-def)
apply (auto dest: gext-objD)
done

value conformance

definition conf :: $\text{prog} \Rightarrow \text{st} \Rightarrow \text{val} \Rightarrow \text{ty} \Rightarrow \text{bool}$ ($\hookleftarrow, \vdash, \preceq, \preceq^*$ [71,71,71,71] 70)
where $G, s \vdash v :: \preceq T = (\exists T' \in \text{typeof} \ (\lambda a. \text{map-option obj-ty} \ (\text{heap } s \ a)) \ v: G \vdash T' \preceq T)$

lemma conf-cong [simp]: $G, \text{set-locals } l \ s \vdash v :: \preceq T = G, s \vdash v :: \preceq T$
by (auto simp: conf-def)

lemma conf-lupd [simp]: $G, \text{lupd}(vn \mapsto va) s \vdash v :: \preceq T = G, s \vdash v :: \preceq T$
by (auto simp: conf-def)

lemma conf-PrimT [simp]: $\forall dt. \text{typeof } dt \ v = \text{Some} \ (\text{PrimT } t) \implies G, s \vdash v :: \preceq \text{PrimT } t$
apply (simp add: conf-def)
done

lemma conf-Boolean: $G, s \vdash v :: \preceq \text{PrimT Boolean} \implies \exists b. v = \text{Bool } b$
by (cases v)
 (auto simp: conf-def obj-ty-def
 dest: widen-Boolean2
 split: obj-tag.splits)

lemma *conf-litval* [rule-format (no-asm)]:
 $\text{typeof } (\lambda a. \text{None}) v = \text{Some } T \longrightarrow G, s \vdash v :: \preceq T$
apply (unfold conf-def)
apply (rule val.induct)
apply auto
done

lemma *conf-Null* [simp]: $G, s \vdash \text{Null} :: \preceq T = G \vdash NT \preceq T$
by (simp add: conf-def)

lemma *conf-Addr*:
 $G, s \vdash \text{Addr } a :: \preceq T = (\exists \text{obj. heap } s \ a = \text{Some obj} \wedge G \vdash \text{obj-ty obj} \preceq T)$
by (auto simp: conf-def)

lemma *conf-AddrI*: $\llbracket \text{heap } s \ a = \text{Some obj}; G \vdash \text{obj-ty obj} \preceq T \rrbracket \Longrightarrow G, s \vdash \text{Addr } a :: \preceq T$
apply (rule conf-Addr [THEN iffD2])
by fast

lemma *defval-conf* [rule-format (no-asm), elim]:
 $\text{is-type } G \ T \longrightarrow G, s \vdash \text{default-val } T :: \preceq T$
apply (unfold conf-def)
apply (induct T)
apply (auto intro: prim-ty.induct)
done

lemma *conf-widen* [rule-format (no-asm), elim]:
 $G \vdash T \preceq T' \Longrightarrow G, s \vdash x :: \preceq T \longrightarrow \text{ws-prog } G \longrightarrow G, s \vdash x :: \preceq T'$
apply (unfold conf-def)
apply (rule val.induct)
apply (auto elim: ws-widen-trans)
done

lemma *conf-gext* [rule-format (no-asm), elim]:
 $G, s \vdash v :: \preceq T \longrightarrow s \leq | s' \longrightarrow G, s' \vdash v :: \preceq T$
apply (unfold gext-def conf-def)
apply (rule val.induct)
apply force+
done

lemma *conf-list-widen* [rule-format (no-asm)]:
 $\text{ws-prog } G \Longrightarrow$
 $\forall Ts \ Ts'. \text{list-all2 } (\text{conf } G \ s) \ \text{vs } Ts$
 $\longrightarrow G \vdash Ts [\preceq] \ Ts' \longrightarrow \text{list-all2 } (\text{conf } G \ s) \ \text{vs } Ts'$
apply (unfold widens-def)
apply (rule list-all2-trans)
apply auto
done

lemma *conf-RefTD* [rule-format (no-asm)]:
 $G, s \vdash a' :: \preceq \text{RefT } T$
 $\longrightarrow a' = \text{Null} \vee (\exists a \ \text{obj } T'. a' = \text{Addr } a \wedge \text{heap } s \ a = \text{Some obj} \wedge$
 $\text{obj-ty obj} = T' \wedge G \vdash T' \preceq \text{RefT } T)$
apply (unfold conf-def)
apply (induct-tac a')
apply (auto dest: widen-PrimT)
done

value list conformance

definition

$lconf :: prog \Rightarrow st \Rightarrow ('a, val) table \Rightarrow ('a, ty) table \Rightarrow bool$ ($\langle -, \vdash - \rangle :: \preceq \rightarrow [71, 71, 71, 71]$ 70)
where $G, st \vdash vs :: \preceq Ts = (\forall n. \forall T \in Ts. n: \exists v \in vs. n: G, st \vdash v :: \preceq T)$

lemma $lconfD$: $\llbracket G, st \vdash vs :: \preceq Ts; Ts\ n = Some\ T \rrbracket \Longrightarrow G, st \vdash (the\ (vs\ n)) :: \preceq T$
by (*force simp: lconf-def*)

lemma $lconf-cong$ [*simp*]: $\bigwedge s. G, set-locals\ x\ st \vdash l :: \preceq L = G, st \vdash l :: \preceq L$
by (*auto simp: lconf-def*)

lemma $lconf-lupd$ [*simp*]: $G, lupd(vn \mapsto v) st \vdash l :: \preceq L = G, st \vdash l :: \preceq L$
by (*auto simp: lconf-def*)

lemma $lconf-new$: $\llbracket L\ vn = None; G, st \vdash l :: \preceq L \rrbracket \Longrightarrow G, st \vdash l(vn \mapsto v) :: \preceq L$
by (*auto simp: lconf-def*)

lemma $lconf-upd$: $\llbracket G, st \vdash l :: \preceq L; G, st \vdash v :: \preceq T; L\ vn = Some\ T \rrbracket \Longrightarrow$
 $G, st \vdash l(vn \mapsto v) :: \preceq L$
by (*auto simp: lconf-def*)

lemma $lconf-ext$: $\llbracket G, st \vdash l :: \preceq L; G, st \vdash v :: \preceq T \rrbracket \Longrightarrow G, st \vdash l(vn \mapsto v) :: \preceq L(vn \mapsto T)$
by (*auto simp: lconf-def*)

lemma $lconf-map-sum$ [*simp*]:
 $G, st \vdash l1\ (+)\ l2 :: \preceq L1\ (+)\ L2 = (G, st \vdash l1 :: \preceq L1 \wedge G, st \vdash l2 :: \preceq L2)$
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*)]:
 $\bigwedge X. \llbracket G, st \vdash l :: \preceq L \rrbracket \Longrightarrow$
 $\forall vs\ Ts. distinct\ vns \longrightarrow length\ Ts = length\ vns$
 $\longrightarrow list-all2\ (conf\ G\ s)\ vs\ Ts \longrightarrow G, st \vdash l(vns[\mapsto]vs) :: \preceq L(vns[\mapsto]Ts)$
apply (*unfold lconf-def*)
apply (*induct-tac vns*)
apply *clarsimp*
apply *clarify*
apply (*frule list-all2-lengthD*)
apply (*clarsimp*)
done

lemma $lconf-deallocL$: $\llbracket G, st \vdash l :: \preceq L(vn \mapsto T); L\ vn = None \rrbracket \Longrightarrow G, st \vdash l :: \preceq L$
apply (*simp only: lconf-def*)
apply *safe*
apply (*drule spec*)
apply (*drule ospec*)
apply *auto*
done

lemma $lconf-gext$ [*elim*]: $\llbracket G, st \vdash l :: \preceq L; s \leq |s| \rrbracket \Longrightarrow G, s \upharpoonright l :: \preceq L$
apply (*simp only: lconf-def*)

apply *fast*
done

lemma *lconf-empty* [*simp*, *intro!*]: $G, \text{st-vals}[\sim::\preceq] \text{Map.empty}$
apply (*unfold lconf-def*)
apply *force*
done

lemma *lconf-init-vals* [*intro!*]:
 $\forall n. \forall T \in \text{fs } n:\text{is-type } G \ T \implies G, \text{st-init-vals } \text{fs}[\sim::\preceq] \text{fs}$
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* \Rightarrow *st* \Rightarrow (*a*, *val*) *table* \Rightarrow (*a*, *ty*) *table* \Rightarrow *bool* ($\langle -, \vdash -[\sim::\preceq] \rangle \rightarrow [71, 71, 71, 71]$ 70)
where $G, \text{st-vals}[\sim::\preceq] \text{Ts} = (\forall n. \forall T \in \text{Ts } n: \forall v \in \text{vs } n: G, \text{st-v}::\preceq T)$

lemma *wlconfD*: $\llbracket G, \text{st-vals}[\sim::\preceq] \text{Ts}; \text{Ts } n = \text{Some } T; \text{vs } n = \text{Some } v \rrbracket \implies G, \text{st-v}::\preceq T$
by (*auto simp: wlconf-def*)

lemma *wlconf-cong* [*simp*]: $\bigwedge s. G, \text{set-locals } x \text{ st-l}[\sim::\preceq] L = G, \text{st-l}[\sim::\preceq] L$
by (*auto simp: wlconf-def*)

lemma *wlconf-lupd* [*simp*]: $G, \text{lupd}(vn \mapsto v) \text{st-l}[\sim::\preceq] L = G, \text{st-l}[\sim::\preceq] L$
by (*auto simp: wlconf-def*)

lemma *wlconf-upd*: $\llbracket G, \text{st-l}[\sim::\preceq] L; G, \text{st-v}::\preceq T; L \text{ vn} = \text{Some } T \rrbracket \implies$
 $G, \text{st-l}(vn \mapsto v) [\sim::\preceq] L$
by (*auto simp: wlconf-def*)

lemma *wlconf-ext*: $\llbracket G, \text{st-l}[\sim::\preceq] L; G, \text{st-v}::\preceq T \rrbracket \implies G, \text{st-l}(vn \mapsto v) [\sim::\preceq] L(vn \mapsto T)$
by (*auto simp: wlconf-def*)

lemma *wlconf-map-sum* [*simp*]:
 $G, \text{st-l1 } (+) \text{ l2}[\sim::\preceq] L1 (+) L2 = (G, \text{st-l1}[\sim::\preceq] L1 \wedge G, \text{st-l2}[\sim::\preceq] L2)$
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*)]:
 $\bigwedge X. \llbracket G, \text{st-l}[\sim::\preceq] L \rrbracket \implies$
 $\forall \text{vs } \text{Ts}. \text{distinct } \text{vns} \longrightarrow \text{length } \text{Ts} = \text{length } \text{vns}$
 $\longrightarrow \text{list-all2 } (\text{conf } G \ s) \ \text{vs } \text{Ts} \longrightarrow G, \text{st-l}(\text{vns}[\mapsto] \text{vs}) [\sim::\preceq] L(\text{vns}[\mapsto] \text{Ts})$
apply (*unfold wlconf-def*)
apply (*induct-tac vns*)
apply *clarsimp*

```

apply clarify
apply (frule list-all2-lengthD)
apply clarsimp
done

```

```

lemma wlconf-deallocL:  $\llbracket G, s \vdash l[\sim::\preceq] L (vn \mapsto T); L \text{ } vn = \text{None} \rrbracket \implies G, s \vdash l[\sim::\preceq] L$ 
apply (simp only: wlconf-def)
apply safe
apply (drule spec)
apply (drule ospec)
defer
apply (drule ospec)
apply auto
done

```

```

lemma wlconf-geat [elim]:  $\llbracket G, s \vdash l[\sim::\preceq] L; s \leq |s| \rrbracket \implies G, s \vdash l[\sim::\preceq] L$ 
apply (simp only: wlconf-def)
apply fast
done

```

```

lemma wlconf-empty [simp, intro!]:  $G, s \vdash vs[\sim::\preceq] \text{Map.empty}$ 
apply (unfold wlconf-def)
apply force
done

```

```

lemma wlconf-empty-vals:  $G, s \vdash \text{Map.empty}[\sim::\preceq] ts$ 
by (simp add: wlconf-def)

```

```

lemma wlconf-init-vals [intro!]:
   $\forall n. \forall T \in fs \ n \text{ is-type } G \ T \implies G, s \vdash \text{init-vals } fs[\sim::\preceq] fs$ 
apply (unfold wlconf-def)
apply force
done

```

```

lemma lconf-wlconf:
   $G, s \vdash l[\sim::\preceq] L \implies G, s \vdash l[\sim::\preceq] L$ 
by (force simp add: lconf-def wlconf-def)

```

object conformance

definition

```

oconf :: prog  $\Rightarrow$  st  $\Rightarrow$  obj  $\Rightarrow$  oref  $\Rightarrow$  bool ( $\hookleftarrow, \vdash, \dashv, \preceq, \surd$ ) [71,71,71,71] 70) where
  ( $G, s \vdash obj::\preceq \surd r$ ) = ( $G, s \vdash \text{values } obj[\sim::\preceq] \text{var-tys } G \text{ (tag obj) } r \wedge$ 
    (case r of
      Heap a  $\Rightarrow$  is-type G (obj-ty obj)
      | Stat C  $\Rightarrow$  True))

```

```

lemma oconf-is-type:  $G, s \vdash obj::\preceq \surd \text{Heap } a \implies \text{is-type } G \text{ (obj-ty obj)}$ 
by (auto simp: oconf-def Let-def)

```

```

lemma oconf-lconf:  $G, s \vdash obj::\preceq \surd r \implies G, s \vdash \text{values } obj[\sim::\preceq] \text{var-tys } G \text{ (tag obj) } r$ 
by (simp add: oconf-def)

```

```

lemma oconf-cong [simp]:  $G, \text{set-locals } l \vdash obj::\preceq \surd r = G, s \vdash obj::\preceq \surd r$ 
by (auto simp: oconf-def Let-def)

```

lemma *oconf-init-obj-lemma*:
 $\llbracket \bigwedge C \ c. \text{class } G \ C = \text{Some } c \implies \text{unique } (\text{DeclConcepts.fields } G \ C);$
 $\bigwedge C \ c \ f \text{fld}. \llbracket \text{class } G \ C = \text{Some } c;$
 $\text{table-of } (\text{DeclConcepts.fields } G \ C) \ f = \text{Some fld} \rrbracket$
 $\implies \text{is-type } G \ (\text{type fld});$
 $(\text{case } r \text{ of}$
 $\quad \text{Heap } a \Rightarrow \text{is-type } G \ (\text{obj-ty obj})$
 $\quad | \text{Stat } C \Rightarrow \text{is-class } G \ C)$
 $\rrbracket \implies G, \text{s} \vdash \text{obj} \ (\text{values} = \text{init-vals } (\text{var-tys } G \ (\text{tag obj}) \ r)) :: \preceq \sqrt{r}$
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 :: $\text{state} \Rightarrow \text{env}' \Rightarrow \text{bool}$ ($\hookleftarrow :: \preceq \rightarrow [71, 71] \ 70$) **where**
 $xs :: \preceq E =$
 $(\text{let } (G, L) = E; s = \text{snd } xs; l = \text{locals } s \text{ in}$
 $(\forall r. \forall \text{obj} \in \text{globs } s \ r: G, \text{s} \vdash \text{obj} :: \preceq \sqrt{r}) \wedge G, \text{s} \vdash l \ [\sim :: \preceq] L \wedge$
 $(\forall a. \text{fst } xs = \text{Some}(Xcpt \ (\text{Loc } a)) \longrightarrow G, \text{s} \vdash \text{Addr } a :: \preceq \text{Class } (SXcpt \ \text{Throwable})) \wedge$
 $(\text{fst } xs = \text{Some}(\text{Jump } \text{Ret}) \longrightarrow l \ \text{Result} \neq \text{None}))$

conforms

lemma *conforms-globsD*:

$\llbracket (x, s) :: \preceq (G, L); \text{globs } s \ r = \text{Some obj} \rrbracket \implies G, \text{s} \vdash \text{obj} :: \preceq \sqrt{r}$
by (*auto simp: conforms-def Let-def*)

lemma *conforms-localD*: $(x, s) :: \preceq (G, L) \implies G, \text{s} \vdash \text{locals } s \ [\sim :: \preceq] L$
by (*auto simp: conforms-def Let-def*)

lemma *conforms-XcptLocD*: $\llbracket (x, s) :: \preceq (G, L); x = \text{Some } (Xcpt \ (\text{Loc } a)) \rrbracket \implies$
 $G, \text{s} \vdash \text{Addr } a :: \preceq \text{Class } (SXcpt \ \text{Throwable})$
by (*auto simp: conforms-def Let-def*)

lemma *conforms-RetD*: $\llbracket (x, s) :: \preceq (G, L); x = \text{Some } (\text{Jump } \text{Ret}) \rrbracket \implies$
 $(\text{locals } s) \ \text{Result} \neq \text{None}$
by (*auto simp: conforms-def Let-def*)

lemma *conforms-RefTD*:

$\llbracket G, \text{s} \vdash a' :: \preceq \text{RefT } t; a' \neq \text{Null}; (x, s) :: \preceq (G, L) \rrbracket \implies$
 $\exists a \ \text{obj}. a' = \text{Addr } a \wedge \text{globs } s \ (\text{Inl } a) = \text{Some obj} \wedge$
 $G \vdash \text{obj-ty obj} \preceq \text{RefT } t \wedge \text{is-type } G \ (\text{obj-ty obj})$
apply (*drule-tac conf-RefTD*)
apply *clarsimp*
apply (*rule conforms-globsD [THEN oconf-is-type]*)
apply *auto*
done

lemma *conforms-Jump [iff]*:

$j = \text{Ret} \longrightarrow \text{locals } s \ \text{Result} \neq \text{None}$
 $\implies ((\text{Some } (\text{Jump } j), s) :: \preceq (G, L)) = (\text{Norm } s :: \preceq (G, L))$
by (*auto simp: conforms-def Let-def*)

lemma *conforms-StdXcpt [iff]*:

$((\text{Some } (Xcpt \text{ (Std } xn)), s)::\preceq(G, L)) = (\text{Norm } s::\preceq(G, L))$
by (*auto simp: conforms-def*)

lemma *conforms-Err* [iff]:
 $((\text{Some } (\text{Error } e), s)::\preceq(G, L)) = (\text{Norm } s::\preceq(G, L))$
by (*auto simp: conforms-def*)

lemma *conforms-raise-if* [iff]:
 $((\text{raise-if } c \text{ } xn \text{ } x, s)::\preceq(G, L)) = ((x, s)::\preceq(G, L))$
by (*auto simp: abrupt-if-def*)

lemma *conforms-error-if* [iff]:
 $((\text{error-if } c \text{ } err \text{ } x, s)::\preceq(G, L)) = ((x, s)::\preceq(G, L))$
by (*auto simp: abrupt-if-def*)

lemma *conforms-NormI*: $(x, s)::\preceq(G, L) \implies \text{Norm } s::\preceq(G, L)$
by (*auto simp: conforms-def Let-def*)

lemma *conforms-absorb* [rule-format]:
 $(a, b)::\preceq(G, L) \longrightarrow (\text{absorb } j \text{ } a, b)::\preceq(G, L)$
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*: $\llbracket \forall r. \forall obj \in \text{globs } s \text{ } r: G, s \vdash obj::\preceq \sqrt{r};$
 $G, s \vdash \text{locals } s[\sim::\preceq] L;$
 $\forall a. x = \text{Some } (Xcpt \text{ (Loc } a)) \longrightarrow G, s \vdash \text{Addr } a::\preceq \text{Class } (SXcpt \text{ Throwable});$
 $x = \text{Some } (\text{Jump Ret}) \longrightarrow \text{locals } s \text{ Result} \neq \text{None} \rrbracket \implies$
 $(x, s)::\preceq(G, L)$
by (*auto simp: conforms-def Let-def*)

lemma *conforms-xconf*: $\llbracket (x, s)::\preceq(G, L);$
 $\forall a. x' = \text{Some } (Xcpt \text{ (Loc } a)) \longrightarrow G, s \vdash \text{Addr } a::\preceq \text{Class } (SXcpt \text{ Throwable});$
 $x' = \text{Some } (\text{Jump Ret}) \longrightarrow \text{locals } s \text{ Result} \neq \text{None} \rrbracket \implies$
 $(x', s)::\preceq(G, L)$
by (*fast intro: conformsI elim: conforms-globsD conforms-localD*)

lemma *conforms-lupd*:
 $\llbracket (x, s)::\preceq(G, L); L \text{ } vn = \text{Some } T; G, s \vdash v::\preceq T \rrbracket \implies (x, \text{lupd}(vn \mapsto v) s)::\preceq(G, L)$
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*:
 $\llbracket (x, s)::\preceq(G, L); G, s \vdash v::\preceq T \rrbracket \implies (x, \text{lupd}(vn \mapsto v) s)::\preceq(G, L(vn \mapsto T))$
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*: $\bigwedge s. \llbracket s :: \preceq (G, L(vn \mapsto T)); L\ vn = \text{None} \rrbracket \implies s :: \preceq (G, L)$
by (*fast intro: conformsI dest: conforms-globsD conforms-RetD*
elim: conforms-XcptLocD conforms-deallocL-aux)

lemma *conforms-gext*: $\llbracket (x, s) :: \preceq (G, L); s \leq |s';$
 $\forall r. \forall \text{obj} \in \text{globs } s' \ r: G, s \vdash \text{obj} :: \preceq \sqrt{r};$
 $\text{locals } s' = \text{locals } s \rrbracket \implies (x, s') :: \preceq (G, L)$
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*:
 $\llbracket (x, s) :: \preceq (G, L); (x', s') :: \preceq (G, L); s' \leq |s; \text{dom } (\text{locals } s') \subseteq \text{dom } (\text{locals } s) \rrbracket$
 $\implies (x', s) :: \preceq (G, L)$
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*: $\bigwedge \text{obj}. \llbracket (x, s) :: \preceq (G, L); G, s \vdash \text{obj} :: \preceq \sqrt{r}; s \leq | \text{gupd}(r \mapsto \text{obj}) s \rrbracket$
 $\implies (x, \text{gupd}(r \mapsto \text{obj}) s) :: \preceq (G, L)$
apply (*rule conforms-gext*)
apply *auto*
apply (*force dest: conforms-globsD simp add: oconf-def*)
done

lemma *conforms-upd-gobj*: $\llbracket (x, s) :: \preceq (G, L); \text{globs } s \ r = \text{Some } \text{obj};$
 $\text{var-ty } G \ (\text{tag } \text{obj}) \ r \ n = \text{Some } T; G, s \vdash v :: \preceq T \rrbracket \implies (x, \text{upd-gobj } r \ n \ v \ s) :: \preceq (G, L)$
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*:
 $\llbracket (x, s) :: \preceq (G, L); G, s \vdash l [\sim :: \preceq] L; x = \text{Some } (\text{Jump Ret}) \longrightarrow l \ \text{Result} \neq \text{None} \rrbracket$
 $\implies (x, \text{set-locals } l \ s) :: \preceq (G, L)$
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:
  
$$\llbracket (a,b)::\preceq(G, L); L\ x = \text{Some } T; \text{locals } b\ x \neq \text{None} \rrbracket$$


$$\implies G, b \vdash_{\text{the } (\text{locals } b\ x)}::\preceq T$$

apply (force simp: conforms-def Let-def wlconf-def)
done

```

```

lemma conforms-return:

$$\wedge s'. \llbracket (x,s)::\preceq(G, L); (x',s')::\preceq(G, L'); s \leq |s'; x' \neq \text{Some } (\text{Jump } \text{Ret}) \rrbracket \implies$$


$$(x', \text{set-locals } (\text{locals } s)\ s')::\preceq(G, L)$$

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 $[[\text{simproc } \text{del}: \text{wt-expr wt-var wt-exprs wt-stmt}]]$

lemma *sxalloc-no-jump*:

assumes *sxalloc*: $G \vdash s0 \text{ --sxalloc--} \rightarrow s1$ **and**
no-jmp: $\text{abrupt } s0 \neq \text{Some } (\text{Jump } j)$
shows $\text{abrupt } s1 \neq \text{Some } (\text{Jump } j)$

using *sxalloc no-jmp*
by cases simp-all

lemma *sxalloc-no-jump'*:

assumes *sxalloc*: $G \vdash s0 \text{ --sxalloc--} \rightarrow s1$ **and**
jump: $\text{abrupt } s1 = \text{Some } (\text{Jump } j)$
shows $\text{abrupt } s0 = \text{Some } (\text{Jump } j)$

using *sxalloc jump*
by cases simp-all

lemma *halloc-no-jump*:

assumes *halloc*: $G \vdash s0 \text{ --halloc oi--} \rightarrow s1$ **and**
no-jmp: $\text{abrupt } s0 \neq \text{Some } (\text{Jump } j)$
shows $\text{abrupt } s1 \neq \text{Some } (\text{Jump } j)$

using *halloc no-jmp*
by cases simp-all

lemma *halloc-no-jump'*:

assumes *halloc*: $G \vdash s0 \text{ --halloc oi--} \rightarrow s1$ **and**
jump: $\text{abrupt } s1 = \text{Some } (\text{Jump } j)$
shows $\text{abrupt } s0 = \text{Some } (\text{Jump } j)$

using *halloc jump*
by cases simp-all

lemma *Body-no-jump*:

assumes *eval*: $G \vdash s0 \text{ --Body } D \text{ c--} \rightarrow s1$ **and**
jump: $\text{abrupt } s0 \neq \text{Some } (\text{Jump } j)$
shows $\text{abrupt } s1 \neq \text{Some } (\text{Jump } j)$

proof (cases normal s0)

case *True*

with *eval* **obtain** $s0'$ **where** eval' : $G \vdash \text{Norm } s0' \text{ --Body } D \text{ c--} \rightarrow s1$ **and**
 $s0$: $s0 = \text{Norm } s0'$

by (cases s0) *simp*

from eval' **obtain** $s2$ **where**

```

    s1: s1 = abupd (absorb Ret)
      (if  $\exists l. \text{abrupt } s2 = \text{Some } (\text{Jump } (\text{Break } l)) \vee$ 
         $\text{abrupt } s2 = \text{Some } (\text{Jump } (\text{Cont } l))$ 
        then  $\text{abupd } (\lambda x. \text{Some } (\text{Error CrossMethodJump})) s2$  else  $s2$ )
  by cases simp
show ?thesis
proof (cases  $\exists l. \text{abrupt } s2 = \text{Some } (\text{Jump } (\text{Break } l)) \vee$ 
   $\text{abrupt } s2 = \text{Some } (\text{Jump } (\text{Cont } l))$ )
  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  $G \vdash (\text{Some } \text{abr}, s0') \text{ --Body } D \text{ c} \rightarrow v \rightarrow s1$ 
     $s0 = (\text{Some } \text{abr}, s0')$ 
  by (cases s0) fastforce
  with this jump
  show ?thesis
  by (cases) (simp)
qed

lemma Methd-no-jump:
  assumes eval:  $G \vdash s0 \text{ --Methd } D \text{ sig} \rightarrow v \rightarrow s1$  and
    jump:  $\text{abrupt } s0 \neq \text{Some } (\text{Jump } j)$ 
  shows  $\text{abrupt } s1 \neq \text{Some } (\text{Jump } j)$ 
proof (cases normal s0)
  case True
  with eval obtain s0' where  $G \vdash \text{Norm } s0' \text{ --Methd } D \text{ sig} \rightarrow v \rightarrow s1$ 
     $s0 = \text{Norm } s0'$ 
  by (cases s0) simp
  then obtain D' body where  $G \vdash s0 \text{ --Body } D' \text{ body} \rightarrow v \rightarrow s1$ 
  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  $G \vdash (\text{Some } \text{abr}, s0') \text{ --Methd } D \text{ sig} \rightarrow v \rightarrow s1$ 
     $s0 = (\text{Some } \text{abr}, s0')$ 
  by (cases s0) fastforce
  with this jump
  show ?thesis
  by (cases) (simp)
qed

lemma jumpNestingOkS-mono:
  assumes jumpNestingOk-l':  $\text{jumpNestingOkS } \text{jmps}' \text{ c}$ 
    and subset:  $\text{jmps}' \subseteq \text{jmps}$ 
  shows  $\text{jumpNestingOkS } \text{jmps} \text{ c}$ 
proof -
  have True and True and

```

$\bigwedge jmps' jmps. \llbracket jumpNestingOkS\ jmps'\ c; jmps' \subseteq jmps \rrbracket \implies jumpNestingOkS\ jmps\ c$
proof (induct rule: var.induct expr.induct stmt.induct)
case (Lab j c jmps' jmps)
note jmpOk = $\langle jumpNestingOkS\ jmps'\ (j \cdot c) \rangle$
note jmps = $\langle jmps' \subseteq jmps \rangle$
with jmpOk **have** jumpNestingOkS ($\{j\} \cup jmps'$) c **by** simp
moreover from jmps **have** ($\{j\} \cup jmps'$) \subseteq ($\{j\} \cup jmps$) **by** auto
ultimately
have jumpNestingOkS ($\{j\} \cup jmps$) 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)
have jumpNestingOkS jmps c1 **by** (rule Comp.hyps) (use Comp.prem in auto)
moreover
have jumpNestingOkS jmps c2 **by** (rule Comp.hyps) (use Comp.prem in auto)
ultimately show ?case **by** simp
next
case (If' e c1 c2 jmps' jmps)
have jumpNestingOkS jmps c1 **by** (rule If'.hyps) (use If'.prem in auto)
moreover
have jumpNestingOkS jmps c2 **by** (rule If'.hyps) (use If'.prem in auto)
ultimately show ?case **by** simp
next
case (Loop l e c jmps' jmps)
from $\langle jumpNestingOkS\ jmps'\ (l \cdot While(e)\ c) \rangle$ **have** jumpNestingOkS ($\{Cont\ l\} \cup jmps'$) c
by simp
moreover
from $\langle jmps' \subseteq jmps \rangle$ **have** $\{Cont\ l\} \cup jmps' \subseteq \{Cont\ l\} \cup jmps$
by auto
ultimately have jumpNestingOkS ($\{Cont\ l\} \cup jmps$) c
by (rule Loop.hyps)
thus ?case **by** simp
next
case (TryC c1 C vn c2 jmps' jmps)
have jumpNestingOkS jmps c1 **by** (rule TryC.hyps) (use TryC.prem in auto)
moreover
have jumpNestingOkS jmps c2 **by** (rule TryC.hyps) (use TryC.prem in auto)
ultimately show ?case
by simp
next
case (Fin c1 c2 jmps' jmps)
have jumpNestingOkS jmps c1 **by** (rule Fin.hyps) (use Fin.prem in auto)
moreover
have jumpNestingOkS jmps c2 **by** (rule Fin.hyps) (use Fin.prem in 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 example, a break can't leave its enclosing loop, an return can't leave its enclosing method. To prove this, the method call is critical. Although the wellformedness of the whole program guarantees that the jumps (breaks, continues and returns) are nested correctly in all method bodies, the call rule alone does not guarantee that I will call a method or even a class that is part of the program due to dynamic binding! To be able to ensure this we need a kind of conformance of the state, like in the typesafety proof. But then we will redo the typesafety proof here. It would be nice if we could find an easy precondition that will guarantee that all calls will actually call classes and methods of the current program, which can be instantiated in the typesafety proof later on. To fix this problem, I have instrumented the semantic definition of a call to filter out any breaks in the state and to throw an error instead.

To get an induction hypothesis which is strong enough to perform the proof, we can't just assume *jumpNestingOk* for the empty set and conclude, that no jump at all will be in the resulting state, because the set is altered by the statements *Lab* and *While*.

The wellformedness of the program is used to ensure that for all class initialisations and methods the nesting of jumps is wellformed, too.

```

theorem jumpNestingOk-eval:
  assumes eval:  $G \vdash s0 \multimap \rightarrow (v, s1)$ 
  and jmpOk: jumpNestingOk jmps t
  and wt:  $Env \vdash t :: T$ 
  and wf: wf-prog G
  and G: prg Env = G
  and no-jmp:  $\forall j. \text{abrupt } s0 = \text{Some } (\text{Jump } j) \longrightarrow j \in \text{jmps}$ 
    (is ?Jmp jmps s0)
  shows  $(\forall j. \text{fst } s1 = \text{Some } (\text{Jump } j) \longrightarrow j \in \text{jmps}) \wedge$ 
    (normal s1  $\longrightarrow$ 
       $(\forall w \text{ upd}. v = \text{In2 } (w, \text{upd})$ 
         $\longrightarrow (\forall s \text{ j val}.$ 
           $\text{abrupt } s \neq \text{Some } (\text{Jump } j) \longrightarrow$ 
           $\text{abrupt } (\text{upd val } s) \neq \text{Some } (\text{Jump } j))))$ 
    (is ?Jmp jmps s1  $\wedge$  ?Upd v s1)

```

```

proof —
  let ?HypObj =  $\lambda t \text{ s0 } s1 \text{ v}.$ 
     $(\forall \text{ jmps } T \text{ Env}.$ 
       $?Jmp \text{ jmps } s0 \longrightarrow \text{jumpNestingOk } \text{jmps } t \longrightarrow \text{Env} \vdash t :: T \longrightarrow \text{prg } \text{Env} = G \longrightarrow$ 
       $?Jmp \text{ jmps } s1 \wedge ?Upd \text{ v } s1)$ 

```

— 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  $\bigwedge \text{ jmps } T \text{ Env}. \llbracket ?Jmp \text{ jmps } s0; \text{jumpNestingOk } \text{jmps } t; \text{Env} \vdash t :: T; \text{prg } \text{Env} = G \rrbracket$ 
   $\implies ?Jmp \text{ jmps } s1 \wedge ?Upd \text{ v } s1$ 
(is PROP ?Hyp t s0 s1 v)

```

— We need to abstract over *jumps* since *jumps* 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 jumps T Env)
  note jmpOK =  $\langle \text{jumpNestingOk jumps (In1r (jmp\cdot c))} \rangle$ 
  note G =  $\langle \text{prg Env} = G \rangle$ 
  have wt-c: Env  $\vdash$  c ::  $\checkmark$ 
  using Lab.premis by (elim wt-elim-cases)
  have j  $\in$  jumps if ab-s1: abrupt (abupd (absorb jmp) s1) = Some (Jump j) for j
  proof –
    from ab-s1 have jmp-s1: abrupt s1 = Some (Jump j)
    by (cases s1) (simp add: absorb-def)
    note hyp-c =  $\langle \text{PROP ?Hyp (In1r c) (Norm s0) s1} \Diamond \rangle$ 
    from ab-s1 have j  $\neq$  jmp
    by (cases s1) (simp add: absorb-def)
    moreover have j  $\in$  {jmp}  $\cup$  jumps
    proof –
      from jmpOK
      have jumpNestingOk ({jmp}  $\cup$  jumps) (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 jumps T Env)
  note jmpOk =  $\langle \text{jumpNestingOk jumps (In1r (c1;; c2))} \rangle$ 
  note G =  $\langle \text{prg Env} = G \rangle$ 
  from Comp.premis obtain
    wt-c1: Env  $\vdash$  c1 ::  $\checkmark$  and wt-c2: Env  $\vdash$  c2 ::  $\checkmark$ 
  by (elim wt-elim-cases)
  have j  $\in$  jumps if abr-s2: abrupt s2 = Some (Jump j) for j
  proof –
    have jmp: ?Jump jumps s1
    proof –
      note hyp-c1 =  $\langle \text{PROP ?Hyp (In1r c1) (Norm s0) s1} \Diamond \rangle$ 
      with wt-c1 jmpOk G
      show ?thesis by simp
    qed
    moreover note hyp-c2 =  $\langle \text{PROP ?Hyp (In1r c2) s1 s2} (\Diamond::\text{vals}) \rangle$ 
    have jmpOk': jumpNestingOk jumps (In1r c2) using jmpOk by simp
    moreover note wt-c2 G abr-s2
    ultimately show j  $\in$  jumps
    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 jumps T Env)
  note jmpOk =  $\langle \text{jumpNestingOk jumps (In1r (If(e) c1 Else c2))} \rangle$ 
  note G =  $\langle \text{prg Env} = G \rangle$ 

```

```

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
have j∈jmps if jmp: abrupt s2 = Some (Jump j) for j
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)
have j∈jmps if jmp: abrupt s3 = Some (Jump j) for j
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 −
  have j' ∈ jmps if abs: abrupt (abupd (absorb (Cont l)) s2)=Some (Jump j') for j'
  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.premis obtain Te where
    wt-e: Env ⊢ e :: - Te
    by (elim wt-elim-cases)
have j ∈ jmps if jmp: abrupt (abupd (throw a) s1) = Some (Jump j) for j
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.premis obtain
    wt-c1: Env ⊢ c1 :: √ and
    wt-c2: Env(lcl := (lcl Env)(VName vn ↦ Class C)) ⊢ c2 :: √
    by (elim wt-elim-cases)
have j ∈ jmps if jmp: abrupt s3 = Some (Jump j) for j
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 → 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 s2alloc-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 s2alloc-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.prem obtain
  wt-c1: Env ⊢ c1 :: √ and wt-c2: Env ⊢ c2 :: √
  by (elim wt-elim-cases)
have j ∈ jmps if jmp: abrupt s3 = Some (Jump j) for j
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 abrupt (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.premis have c: class G C = Some c
    by (elim wt-elim-cases) auto
  have j∈jmps if jmp: abrupt s3 = (Some (Jump j)) for j
  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 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 ?Jump jmps s1
      by simp
    hence jmp-s1: ?Jump 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)
have j∈jmps if jmp: abrupt s2 = Some (Jump j) for j
proof -
  note ⟨prg Env = G⟩
  moreover note hyp-init = ⟨PROP ?Hyp (In1r (Init C)) (Norm s0) s1 ◇⟩
  moreover from wf NewC.prem
  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)
have j∈jmps if jmp: abrupt s3 = Some (Jump j) for j
proof -
  note G = ⟨prg Env = G⟩
  from NewA.prem
  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)
have j∈jmps if jmp: abrupt s2 = Some (Jump j) for j
proof -
  note hyp-e = ⟨PROP ?Hyp (In1l e) (Norm s0) s1 (In1 v)⟩
  note ⟨prg Env = G⟩
  moreover from Cast.prem
  obtain eT where Env⊢e::−eT by (elim wt-elim-cases)

```

```

moreover
have abrupt s1 = Some (Jump j)
proof –
  note  $\langle s2 = \text{abupd} (\text{raise-if } (\neg G, \text{snd } s1 \vdash v \text{ fits } cT) \text{ ClassCast}) s1 \rangle$ 
  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)
have j ∈ jmps if jmp: abrupt s1 = Some (Jump j) for j
proof –
  note hyp-e =  $\langle \text{PROP } ?\text{Hyp } (In1l\ e) (Norm\ s0)\ s1\ (In1\ v) \rangle$ 
  note  $\langle \text{prg } Env = G \rangle$ 
  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)
have j ∈ jmps if jmp: abrupt s1 = Some (Jump j) for j
proof –
  note hyp-e =  $\langle \text{PROP } ?\text{Hyp } (In1l\ e) (Norm\ s0)\ s1\ (In1\ v) \rangle$ 
  note  $\langle \text{prg } Env = G \rangle$ 
  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)
have j ∈ jmps if jmp: abrupt s2 = Some (Jump j) for j
proof –
  note G =  $\langle \text{prg } Env = G \rangle$ 
  from BinOp.prems
  obtain e1T e2T where
    wt-e1: Env ⊢ e1 :: - e1T and
    wt-e2: Env ⊢ e2 :: - e2T
    by (elim wt-elim-cases)
  note  $\langle \text{PROP } ?\text{Hyp } (In1l\ e1) (Norm\ s0)\ s1\ (In1\ v1) \rangle$ 
  with G wt-e1 have jmp-s1: ?Jump jmps s1 by simp
  note hyp-e2 =
     $\langle \text{PROP } ?\text{Hyp } (\text{if need-second-arg binop } v1 \text{ then } In1l\ e2 \text{ else } In1r\ \text{Skip})\ s1\ s2\ (In1\ v2) \rangle$ 
  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)
have j∈jmps if jmp: abrupt s1 = Some (Jump j) for j
proof -
  note hyp-va = ⟨PROP ?Hyp (In2 va) (Norm s0) s1 (In2 (v,f))⟩
  note ⟨prg Env = G⟩
  moreover from Acc.premis
  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.premis
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)⟩
have j∈jmps if jmp: abrupt (assign f v s2) = Some (Jump j) for j
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 =  $\langle \text{PROP } ?\text{Hyp } (\text{In1l } e0) (\text{Norm } s0) s1 (\text{In1 } b) \rangle$ 
note hyp-e1-e2 =  $\langle \text{PROP } ?\text{Hyp } (\text{In1l } (\text{if the-Bool } b \text{ then } e1 \text{ else } e2)) s1 s2 (\text{In1 } v) \rangle$ 
from Cond.prems
obtain e1T e2T
  where wt-e0:  $\text{Env} \vdash e0 :: -\text{PrimT Boolean}$ 
  and wt-e1:  $\text{Env} \vdash e1 :: -e1T$ 
  and wt-e2:  $\text{Env} \vdash e2 :: -e2T$ 
  by (elim wt-elim-cases)
have  $j \in \text{jmps}$  if jmp: abrupt s2 = Some (Jump j) for j
proof –
  from wt-e0 G
  have jmp-s1:  $? \text{Jump jmps } s1$ 
    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 =  $\langle \text{prg Env} = G \rangle$ 
from Call.prems
obtain eT argsT
  where wt-e:  $\text{Env} \vdash e :: -eT$  and wt-args:  $\text{Env} \vdash \text{args} :: \dot{=} \text{argsT}$ 
  by (elim wt-elim-cases)
have  $j \in \text{jmps}$  if jmp: abrupt ((set-lvars (locals (store s2))) s4) = Some (Jump j) for j
proof –
  note hyp-e =  $\langle \text{PROP } ?\text{Hyp } (\text{In1l } e) (\text{Norm } s0) s1 (\text{In1 } a) \rangle$ 
from wt-e G
have jmp-s1:  $? \text{Jump jmps } s1$ 
  by – (rule hyp-e [THEN conjunct1],simp-all)
note hyp-args =  $\langle \text{PROP } ?\text{Hyp } (\text{In3 } \text{args}) s1 s2 (\text{In3 } \text{vs}) \rangle$ 
have abrupt s2 = Some (Jump j)
proof –
  note  $\langle G \vdash s3' - \text{Methd } D \langle \text{name} = mn, \text{parTs} = pTs \rangle - \succ v \rightarrow s4 \rangle$ 
  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  $\langle s3' = \text{check-method-access } G \text{ accC statT mode} \langle \text{name} = mn, \text{parTs} = pTs \rangle a s3 \rangle$ 
  ultimately have abrupt s3 = Some (Jump j)
    by (cases s3)
    (simp add: check-method-access-def abrupt-if-def Let-def)
  moreover
  note  $\langle s3 = \text{init-lvars } G D \langle \text{name}=mn, \text{parTs}=pTs \rangle \text{ mode } a \text{ vs } s2 \rangle$ 
  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.prem
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⟩
have j ∈ jmps if jmp: abrupt s3 = Some (Jump j) for j
proof -
note hyp-init = ⟨PROP ?Hyp (In1r (Init statDeclC)) (Norm s0) s1 ◇⟩
have ?Jmp jmps s1
by (rule hyp-init [THEN conjunct1]) (use G wt-init in 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)
  have abrupt (upd val s) ≠ Some (Jump j)
    if abrupt s ≠ Some (Jump j)
    for j val and s::state
    using upd that 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.premis
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⟩
have j ∈ jmps if jmp: abrupt s2' = Some (Jump j) for j
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
  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)
  have abrupt (upd val s) ≠ Some (Jump j) if abrupt s ≠ Some (Jump j)
    for j val and s::state
    using upd that 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.premis obtain eT esT
  where wt-e: Env⊢e::-eT and wt-e2: Env⊢es::≐esT
  by (elim wt-elim-cases) simp
have j∈jumps if jmp: abrupt s2 = Some (Jump j) for j
proof -
  note hyp-e = ⟨PROP ?Hyp (In1 e) (Norm s0) s1 (In1 v)⟩
  from G wt-e
  have ?Jmp jumps 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*: $\text{prg Env} \vdash s0 \dashv e \dashv v \rightarrow s1$ **and**
no-jmp: $\text{abrupt } s0 \neq \text{Some } (\text{Jump } j)$ **and**
wt: $\text{Env} \vdash e :: -T$ **and**
wf: *wf-prog* (*prg Env*)
shows $\text{abrupt } s1 \neq \text{Some } (\text{Jump } j)$
using *eval - no-jmp wt wf*
by (*rule jumpNestingOk-eval-no-jumpE, simp-all*)

corollary *eval-var-no-jump*:
assumes *eval*: $\text{prg Env} \vdash s0 \dashv \text{var} \Rightarrow (w, \text{upd}) \rightarrow s1$ **and**
no-jmp: $\text{abrupt } s0 \neq \text{Some } (\text{Jump } j)$ **and**
wt: $\text{Env} \vdash \text{var} :: =T$ **and**
wf: *wf-prog* (*prg Env*)
shows $\text{abrupt } s1 \neq \text{Some } (\text{Jump } j) \wedge$
 $(\text{normal } s1 \longrightarrow$
 $(\text{abrupt } s \neq \text{Some } (\text{Jump } j'))$
 $\longrightarrow \text{abrupt } (\text{upd val } s) \neq \text{Some } (\text{Jump } j'))$
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*: $\text{prg Env} \vdash s0 \dashv c \rightarrow s1$ **and**
jmpOk: *jumpNestingOkS* {} *c* **and**
no-jmp: $\text{abrupt } s0 \neq \text{Some } (\text{Jump } j)$ **and**
wt: $\text{Env} \vdash c :: \surd$ **and**
wf: *wf-prog* (*prg Env*)
shows $\text{abrupt } s1 \neq \text{Some } (\text{Jump } j)$
using *eval - no-jmp wt wf*
by (*rule jumpNestingOk-eval-no-jumpE*) (*simp-all add: jmpOk*)

corollary *eval-expression-list-no-jump*:
assumes *eval*: $\text{prg Env} \vdash s0 \dashv es \dashv v \rightarrow s1$ **and**
no-jmp: $\text{abrupt } s0 \neq \text{Some } (\text{Jump } j)$ **and**
wt: $\text{Env} \vdash es :: \dot{=}T$ **and**
wf: *wf-prog* (*prg Env*)
shows $\text{abrupt } s1 \neq \text{Some } (\text{Jump } j)$
using *eval - no-jmp wt wf*
by (*rule jumpNestingOk-eval-no-jumpE, simp-all*)

lemma *union-subseteq-elim* [*elim*]: $\llbracket A \cup B \subseteq C; \llbracket A \subseteq C; B \subseteq C \rrbracket \Rightarrow P \rrbracket \Rightarrow P$
by *blast*

lemma *dom-locals-halloc-mono*:
assumes *halloc*: $G \vdash s0 \dashv \text{halloc } oi \dashv a \rightarrow s1$
shows $\text{dom } (\text{locals } (\text{store } s0)) \subseteq \text{dom } (\text{locals } (\text{store } s1))$
proof –
from *halloc* **show** ?thesis
by *cases simp-all*
 qed

```

lemma dom-locals-sxalloc-mono:
  assumes sxalloc:  $G \vdash s0 \rightarrow sxalloc \rightarrow s1$ 
  shows dom (locals (store s0))  $\subseteq$  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))  $\subseteq$  dom (locals (store (f n s)))
  shows dom (locals (store s))  $\subseteq$  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))  $\subseteq$  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))
   $\subseteq$  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))
    ⊆ 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))
    ⊆ 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))
    ⊆ 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. That's why we need the complicated second part of the conjunction in the goal. The reason for the very generic way to treat assignments was the aim to omit redundancy. There is only one evaluation rule for each kind of variable (locals, fields, arrays). These rules are used for both accessing variables and updating variables. That's why the evaluation rules for variables result in a pair consisting of a value and an update function. Of course we could also think of a pair of a value and a reference in the store, instead of the generic update function. But as only array updates can cause a special exception (if the types mismatch) and not array reads we then have to introduce two different rules to handle array reads and updates

```

lemma dom-locals-eval-mono:
  assumes eval:  $G \vdash s0 \multimap \rightarrow (v, s1)$ 
  shows dom (locals (store s0)) ⊆ dom (locals (store s1)) ∧
    (∀ vv. v=In2 vv ∧ normal s1
      → (∀ s val. dom (locals (store s))
        ⊆ dom (locals (store ((snd vv) val s)))))
proof –
  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 ...  $\subseteq$  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)))  $\subseteq$  dom (locals (store s1))
    by simp
  also
  from If.hyps
  have ...  $\subseteq$  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)))  $\subseteq$  dom (locals (store s1)) and
      s1-s2: dom (locals (store s1))  $\subseteq$  dom (locals (store s2)) and
      s2-s3: dom (locals (store s2))  $\subseteq$  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)))
     $\subseteq$  dom (locals (store s1)) by simp
  from  $\langle G \vdash s1 \text{ --sxalloc--> } s2 \rangle$ 
  have s1-s2: dom (locals (store s1))  $\subseteq$  dom (locals (store s2))
    by (rule dom-locals-sxalloc-mono)
  thus ?case
  proof (cases G,s2 $\vdash$ 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)))
       $\subseteq$  dom (locals (store s3))
    by simp
    hence dom (locals (store s2))  $\subseteq$  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  $\exists err. x1 = \text{Some} (\text{Error } err)$ )
  case True
  with Fin.hyps show ?thesis
  by simp
next
case False
from Fin.hyps
have  $\text{dom} (\text{locals} (\text{store} ((\text{Norm } s0)::\text{state})))$ 
 $\subseteq \text{dom} (\text{locals} (\text{store} (x1, s1)))$ 
  by simp
hence  $\text{dom} (\text{locals} (\text{store} ((\text{Norm } s0)::\text{state})))$ 
 $\subseteq \text{dom} (\text{locals} (\text{store} ((\text{Norm } s1)::\text{state})))$ 
  by simp
also
from Fin.hyps
have  $\dots \subseteq \text{dom} (\text{locals} (\text{store } s2))$ 
  by simp
finally show ?thesis
  using Fin.hyps by simp
qed
next
case (Init  $C$   $c$   $s0$   $s3$   $s1$   $s2$ )
show ?case
proof (cases  $\text{inited } C (\text{globs } s0)$ )
  case True
  with Init.hyps show ?thesis by simp
next
case False
with Init.hyps
obtain  $s0-s1$ :  $\text{dom} (\text{locals} (\text{store} (\text{Norm} ((\text{init-class-obj } G \ C) \ s0))))$ 
 $\subseteq \text{dom} (\text{locals} (\text{store } s1))$  and
 $s3: s3 = (\text{set-lvars} (\text{locals} (\text{snd } s1))) \ s2$ 
  by simp
from  $s0-s1$ 
have  $\text{dom} (\text{locals} (\text{store} (\text{Norm } s0))) \subseteq \text{dom} (\text{locals} (\text{store } s1))$ 
  by (cases  $s0$ ) simp
with  $s3$ 
have  $\text{dom} (\text{locals} (\text{store} (\text{Norm } s0))) \subseteq \text{dom} (\text{locals} (\text{store } s3))$ 
  by (cases  $s2$ ) simp
thus ?thesis by simp
qed
next
case (NewC  $s0$   $C$   $s1$   $a$   $s2$ )
note  $\text{halloc} = \langle G \vdash s1 \text{ --halloc } C \text{Inst } C \rangle a \rightarrow s2 \rangle$ 
from NewC.hyps
have  $\text{dom} (\text{locals} (\text{store} ((\text{Norm } s0)::\text{state}))) \subseteq \text{dom} (\text{locals} (\text{store } s1))$ 
  by simp
also
from halloc
have  $\dots \subseteq \text{dom} (\text{locals} (\text{store } s2))$  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  $\langle G \vdash s1 \multimap e \multimap v \rightarrow s2 \rangle$ 
    have  $s2 = s1$ 
      by auto
    with  $s0 \cdot s1$  False
    have  $\text{dom} (\text{locals} (\text{store} ((\text{Norm } s0)::\text{state})))$ 
       $\subseteq \text{dom} (\text{locals} (\text{store} (\text{assign } f \ v \ s2)))$ 
      by simp
    thus ?thesis
      by simp
  qed
next
  case (Cond s0 e0 b s1 e1 e2 v s2)
  from Cond.hyps
  have  $\text{dom} (\text{locals} (\text{store} ((\text{Norm } s0)::\text{state}))) \subseteq \text{dom} (\text{locals} (\text{store } s1))$ 
    by simp
  also
  from Cond.hyps
  have  $\dots \subseteq \text{dom} (\text{locals} (\text{store } s2))$ 
    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  $s3 = \langle s3 = \text{init-lvars } G \ D \ (\text{name} = \text{mn}, \text{parTs} = \text{pTs}) \text{ mode } a' \text{ vs } s2 \rangle$ 
  from Call.hyps
  have  $\text{dom} (\text{locals} (\text{store} ((\text{Norm } s0)::\text{state}))) \subseteq \text{dom} (\text{locals} (\text{store } s1))$ 
    by simp
  also
  from Call.hyps
  have  $\dots \subseteq \text{dom} (\text{locals} (\text{store } s2))$ 
    by simp
  also
  have  $\dots \subseteq \text{dom} (\text{locals} (\text{store} ((\text{set-lvars} (\text{locals} (\text{store } s2)))) \ s4)))$ 
    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  $\text{dom} (\text{locals} (\text{store} ((\text{Norm } s0)::\text{state}))) \subseteq \text{dom} (\text{locals} (\text{store } s1))$ 
    by simp
  also
  from Body.hyps
  have  $\dots \subseteq \text{dom} (\text{locals} (\text{store } s2))$ 
    by simp
  also
  have  $\dots \subseteq \text{dom} (\text{locals} (\text{store} (\text{abupd} (\text{absorb Ret}) \ s2)))$ 
    by simp
  also
  have  $\dots \subseteq \text{dom} (\text{locals} (\text{store} (\text{abupd} (\text{absorb Ret}) \ s3)))$ 
  proof -
    from  $\langle s3 =$ 
      (if  $\exists l. \text{abrupt } s2 = \text{Some} (\text{Jump} (\text{Break } l)) \vee$ 
         $\text{abrupt } s2 = \text{Some} (\text{Jump} (\text{Cont } l))$ 
        then  $\text{abupd} (\lambda x. \text{Some} (\text{Error CrossMethodJump})) \ s2 \text{ else } s2 \rangle$ 
    show ?thesis
      by simp
  qed
  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  $\forall s \text{ val. } \text{dom} (\text{locals} (\text{store } s))$ 
     $\subseteq \text{dom} (\text{locals} (\text{store} (\text{snd } v \text{ val } s)))$  (is ?V-ok)
    by (simp add: dom-locals-fvar-vvar-mono)
  hence v-ok: ( $\forall vv. \text{In2 } v = \text{In2 } vv \wedge \text{normal } s3 \longrightarrow ?V\text{-ok}$ )
    by - (intro strip, simp)
  note s3 =  $\langle s3 = \text{check-field-access } G \text{ accC statDeclC fn stat a s2' \rangle$ 
  from FVar.hyps
  have  $\text{dom} (\text{locals} (\text{store} ((\text{Norm } s0)::\text{state}))) \subseteq \text{dom} (\text{locals} (\text{store } s1))$ 
    by simp
  also
  from FVar.hyps
  have  $\dots \subseteq \text{dom} (\text{locals} (\text{store } s2))$ 
    by simp
  also
  from s2'
  have  $\dots \subseteq \text{dom} (\text{locals} (\text{store } s2'))$ 
    by (simp add: dom-locals-fvar-mono)
  also
  from s3
  have  $\dots \subseteq \text{dom} (\text{locals} (\text{store } s3))$ 
    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  $\forall s \text{ val. } \text{dom} (\text{locals} (\text{store } s))$ 
     $\subseteq \text{dom} (\text{locals} (\text{store} (\text{snd } v \text{ val } s)))$  (is ?V-ok)
    by (simp add: dom-locals-avar-vvar-mono)
  hence v-ok: ( $\forall vv. \text{In2 } v = \text{In2 } vv \wedge \text{normal } s2' \longrightarrow ?V\text{-ok}$ )
    by - (intro strip, simp)
  from AVar.hyps
  have  $\text{dom} (\text{locals} (\text{store} ((\text{Norm } s0)::\text{state}))) \subseteq \text{dom} (\text{locals} (\text{store } s1))$ 
    by simp
  also
  from AVar.hyps
  have  $\dots \subseteq \text{dom} (\text{locals} (\text{store } s2))$ 
    by simp
  also
  from s2'

```

```

  have ...  $\subseteq$  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)))  $\subseteq$  dom (locals (store s1))
    by simp
  also
  from Cons.hyps
  have ...  $\subseteq$  dom (locals (store s2))
    by simp
  finally show ?case by simp
qed
qed

```

lemma dom-locals-eval-mono-elim:

```

  assumes eval:  $G \vdash s0 \multimap \rightarrow (v, s1)$ 
  obtains dom (locals (store s0))  $\subseteq$  dom (locals (store s1)) and
     $\bigwedge v \ s \ \text{val}. \llbracket v = \text{In2 } vv; \text{ normal } s1 \rrbracket$ 
       $\implies$  dom (locals (store s))
       $\subseteq$  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:  $G \vdash s0 \multimap \text{halloc } oi \multimap a \rightarrow s1$  and
    normal: normal s1
  shows normal s0
proof -
  from halloc normal show ?thesis
    by cases simp-all
qed

```

lemma salloc-mono-no-abrupt:

```

  assumes salloc:  $G \vdash s0 \multimap \text{salloc} \rightarrow s1$  and
    normal: normal s1
  shows normal s0
proof -
  from salloc normal show ?thesis
    by cases simp-all
qed

```

lemma union-subseteqI: $\llbracket A \cup B \subseteq C; A' \subseteq A; B' \subseteq B \rrbracket \implies A' \cup B' \subseteq C$
by blast

lemma union-subseteqII: $\llbracket A \cup B \subseteq C; A' \subseteq A \rrbracket \implies A' \cup B \subseteq C$
by blast

lemma union-subseteqIr: $\llbracket A \cup B \subseteq C; B' \subseteq B \rrbracket \implies A \cup B' \subseteq C$
by blast

lemma subseteq-union-transl [trans]: $\llbracket A \subseteq B; B \cup C \subseteq D \rrbracket \implies A \cup C \subseteq D$
by blast

lemma subseteq-union-transr [trans]: $\llbracket A \subseteq B; C \cup B \subseteq D \rrbracket \implies A \cup C \subseteq D$
by blast

lemma *union-subseteq-weaken*: $\llbracket A \cup B \subseteq C; \llbracket A \subseteq C; B \subseteq C \rrbracket \implies P \rrbracket \implies P$
by *blast*

lemma *assigns-good-approx*:

assumes

eval: $G \vdash s0 \multimap \rightarrow (v, s1)$ **and**

normal: *normal* *s1*

shows $\text{assigns } t \subseteq \text{dom } (\text{locals } (\text{store } s1))$

proof –

from *eval normal* **show** *?thesis*

proof (*induct*)

case *Abrupt* **thus** *?case* **by** *simp*

next — For statements its trivial, since then $\text{assigns } t = \{\}$

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 $\text{halloc} = \langle G \vdash \text{abupd } (\text{check-neg } i) \text{ } s2 \multimap \text{halloc } \text{Arr } T \text{ } (\text{the-Intg } i) \multimap a \rightarrow s3 \rangle$

have $\text{assigns } (\text{In1l } e) \subseteq \text{dom } (\text{locals } (\text{store } s2))$

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 $\dots \subseteq \text{dom } (\text{locals } (\text{store } s3))$

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 $\text{assigns } (\text{In1l } e) \subseteq \text{dom } (\text{locals } (\text{store } s1))$

```

    by simp
  also
  from Cast.hyps
  have ...  $\subseteq$  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)  $\subseteq$  dom (locals (store s1))
    by iprover
  also
  have ...  $\subseteq$  dom (locals (store s2))
  proof -
    note  $\langle G \vdash s1 \text{ -- (if need-second-arg binop v1 then In1l e2} \\ \text{else In1r Skip)} \rangle \rightarrow (In1 v2, s2) \rangle$ 
    thus ?thesis
      by (rule dom-locals-eval-mono-elim)
  qed
  finally have s2: assigns (In1l e1)  $\subseteq$  dom (locals (store s2)) .
  show ?case
  proof (cases binop=CondAnd  $\vee$  binop=CondOr)
    case True
    with s2 show ?thesis by simp
  next
    case False
    with BinOp
    have assigns (In1l e2)  $\subseteq$  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 =  $\langle \text{normal (assign f v s2)} \rangle$ 
  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)  $\subseteq$  dom (locals (store s1))
    by iprover
  also
  from Ass.hyps
  have ...  $\subseteq$  dom (locals (store s2))

```

```

    by - (erule dom-locals-eval-mono-elim)
  also
  from nrm-s2 Ass.hyps
  have assigns (In1l e)  $\subseteq$  dom (locals (store s2))
    by iprover
  ultimately
  have assigns (In2 va)  $\cup$  assigns (In1l e)  $\subseteq$  dom (locals (store s2))
    by (rule Un-least)
  also
  from Ass.hyps nrm-s1
  have ...  $\subseteq$  dom (locals (store (f v s2)))
    by - (erule dom-locals-eval-mono-elim, cases s2,simp)
  then
  have dom (locals (store s2))  $\subseteq$  dom (locals (store (assign f v s2)))
    by (rule dom-locals-assign-mono)
  finally
  have va-e: assigns (In2 va)  $\cup$  assigns (In1l e)
     $\subseteq$  dom (locals (snd (assign f v s2))) .
  show ?case
  proof (cases  $\exists$  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  $G \vdash \text{Norm } s0 \text{ } \neg \text{LVar } n \Rightarrow (w,f) \rightarrow s1$ 
      by simp
    hence (w,f) = lvar n s0
      by (rule eval-elim-cases) simp
    with nrm-ass-s2
    have n  $\in$  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)  $\subseteq$  dom (locals (store s1))
    by iprover
  also from Cond.hyps
  have ...  $\subseteq$  dom (locals (store s2))
    by - (erule dom-locals-eval-mono-elim)
  finally have e0: assigns (In1l e0)  $\subseteq$  dom (locals (store s2)) .
  show ?case
  proof (cases the-Bool b)
    case True
    with Cond
    have assigns (In1l e1)  $\subseteq$  dom (locals (store s2))
      by simp
    hence assigns (In1l e1)  $\cap$  assigns (In1l e2)  $\subseteq$  ...
      by blast
    with e0
    have assigns (In1l e0)  $\cup$  assigns (In1l e1)  $\cap$  assigns (In1l e2)

```

```

       $\subseteq \text{dom} (\text{locals} (\text{store } s2))$ 
    by (rule Un-least)
  thus ?thesis using True by simp
next
  case False
  with Cond
  have assigns (In1l e2)  $\subseteq \text{dom} (\text{locals} (\text{store } s2))$ 
    by simp
  hence assigns (In1l e1)  $\cap$  assigns (In1l e2)  $\subseteq \dots$ 
    by blast
  with e0
  have assigns (In1l e0)  $\cup$  assigns (In1l e1)  $\cap$  assigns (In1l e2)
     $\subseteq \text{dom} (\text{locals} (\text{store } s2))$ 
    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  $\langle \text{normal} ((\text{set-lvars} (\text{locals} (\text{snd } s2))) s4) \rangle$ 
    have normal-s4: normal s4 by simp
    hence normal s3' using Call.hyps
      by - (erule eval-no-abrupt-lemma [rule-format])
    moreover note
       $\langle s3' = \text{check-method-access } G \text{ accC statT mode } (\text{name}=mn, \text{parTs}=pTs) a' s3 \rangle$ 
    ultimately have normal s3
      by (cases s3) (simp add: check-method-access-def Let-def)
    moreover
    note  $s3 = \langle s3 = \text{init-lvars } G D (\text{name} = mn, \text{parTs} = pTs) \text{ mode } a' \text{ vs } s2 \rangle$ 
    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)  $\subseteq \text{dom} (\text{locals} (\text{store } s1))$ 
    by iprover
  also from Call.hyps
  have  $\dots \subseteq \text{dom} (\text{locals} (\text{store } s2))$ 
    by - (erule dom-locals-eval-mono-elim)
  also
  from nrm-s2 Call.hyps
  have assigns (In3 args)  $\subseteq \text{dom} (\text{locals} (\text{store } s2))$ 
    by iprover
  ultimately have assigns (In1l e)  $\cup$  assigns (In3 args)  $\subseteq \dots$ 
    by (rule Un-least)
  also
  have  $\dots \subseteq \text{dom} (\text{locals} (\text{store} ((\text{set-lvars} (\text{locals} (\text{store } s2))) s4)))$ 
    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)  $\subseteq$  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 ...  $\subseteq$  dom (locals (store s2))
  by - (erule dom-locals-eval-mono-elim)
also from Cons
have assigns (In3 es)  $\subseteq$  dom (locals (store s2))
  by iprover
ultimately
have assigns (In1l e)  $\cup$  assigns (In3 es)  $\subseteq$  dom (locals (store s2))
  by (rule Un-least)
thus ?case
  by simp
qed
qed

```

corollary *assignsE-good-approx:*

```

assumes
  eval: prg Env $\vdash$  s0  $\rightarrow$  e  $\rightarrow$  v  $\rightarrow$  s1 and
  normal: normal s1
shows assignsE e  $\subseteq$  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 $\vdash$  s0  $\rightarrow$  v  $\rightarrow$  vf  $\rightarrow$  s1 and
  normal: normal s1
shows assignsV v  $\subseteq$  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 $\vdash$  s0  $\rightarrow$  es  $\rightarrow$  vs  $\rightarrow$  s1 and
  normal: normal s1
shows assignsEs es  $\subseteq$  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:  $G \vdash \text{Norm } s0 \multimap e \multimap v \rightarrow s$ 
shows  $v = c \wedge \text{normal } s$ 
proof -
  have True and
     $\bigwedge c \ v \ s0 \ s. \llbracket \text{constVal } e = \text{Some } c; G \vdash \text{Norm } s0 \multimap e \multimap v \rightarrow s \rrbracket$ 
     $\implies v = c \wedge \text{normal } s$ 
  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  $\langle \text{constVal } (\text{Lit } \text{val}) = \text{Some } c \rangle$ 
  moreover
  from  $\langle G \vdash \text{Norm } s0 \multimap \text{Lit } \text{val} \multimap v \rightarrow s \rangle$ 
  obtain  $v = \text{val}$  and normal s
    by cases simp
  ultimately show  $v = c \wedge \text{normal } s$  by simp
next
  case (UnOp unop e c v s0 s)
  note const =  $\langle \text{constVal } (\text{UnOp } \text{unop } e) = \text{Some } c \rangle$ 
  then obtain ce where ce: constVal e = Some ce by simp
  from  $\langle G \vdash \text{Norm } s0 \multimap \text{UnOp } \text{unop } e \multimap v \rightarrow s \rangle$ 
  obtain ve where ve:  $G \vdash \text{Norm } s0 \multimap e \multimap ve \rightarrow s$  and
     $v: v = \text{eval-unop } \text{unop } ve$ 
    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 =  $\langle \text{constVal } (\text{BinOp } \text{binop } e1 \ e2) = \text{Some } c \rangle$ 
  then obtain c1 c2 where c1: constVal e1 = Some c1 and
    c2: constVal e2 = Some c2 and
     $c: c = \text{eval-binop } \text{binop } c1 \ c2$ 
    by simp
  from  $\langle G \vdash \text{Norm } s0 \multimap \text{BinOp } \text{binop } e1 \ e2 \multimap v \rightarrow s \rangle$ 
  obtain v1 s1 v2
    where v1:  $G \vdash \text{Norm } s0 \multimap e1 \multimap v1 \rightarrow s1$  and
    v2:  $G \vdash s1 \multimap (\text{if need-second-arg binop } v1 \text{ then In1l } e2$ 
       $\text{else In1r Skip}) \multimap (In1 \ v2, s)$  and
     $v: v = \text{eval-binop } \text{binop } v1 \ v2$ 
    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  $G \vdash \text{Norm } s1' - e2 \multimap v2 \rightarrow s$ 
      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 =  $\langle \text{constVal } (b \text{ ? } e1 : e2) = \text{Some } c \rangle$ 
    then obtain cb c1 c2 where
      cb:  $\text{constVal } b = \text{Some } cb$  and
      c1:  $\text{constVal } e1 = \text{Some } c1$  and
      c2:  $\text{constVal } e2 = \text{Some } c2$ 
      by (auto split: bool.splits)
    from  $\langle G \vdash \text{Norm } s0 - b \text{ ? } e1 : e2 \multimap v \rightarrow s \rangle$ 
    obtain vb s1
      where  $v b: G \vdash \text{Norm } s0 - b \multimap v b \rightarrow s1$  and
       $\text{eval-v: } G \vdash s1 - (\text{if the-Bool } vb \text{ then } e1 \text{ else } e2) \multimap v \rightarrow s$ 
      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  $G \vdash \text{Norm } s1' - e1 \multimap v \rightarrow s$ 
          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  $G \vdash \text{Norm } s1' - e2 \multimap v \rightarrow s$ 
  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:  $\text{Env} \vdash e :: \neg \text{PrimT Boolean}$ 
  shows typeof empty-dt c = Some (PrimT Boolean)
proof -
  have True and
     $\bigwedge c. [\text{constVal } e = \text{Some } c; \text{Env} \vdash e :: \neg \text{PrimT Boolean}] \implies \text{typeof empty-dt } c = \text{Some } (\text{PrimT Boolean})$ 
  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  $\langle \text{constVal } (\text{Lit } v) = \text{Some } c \rangle$ 
  have  $c = v$  by simp
  moreover
  from  $\langle \text{Env} \vdash \text{Lit } v :: \neg \text{PrimT Boolean} \rangle$ 
  have typeof empty-dt v = Some (PrimT Boolean)
  by cases simp
  ultimately show ?case by simp
next
  case (UnOp unop e c)
  from  $\langle \text{Env} \vdash \text{UnOp unop } e :: \neg \text{PrimT Boolean} \rangle$ 
  have Boolean = unop-type unop by cases simp

```

```

moreover
from  $\langle \text{constVal } (UnOp \text{ unop } e) = \text{Some } c \rangle$ 
obtain  $ce$  where  $c = \text{eval-unop unop } ce$  by auto
ultimately show  $?case$  by (simp add: eval-unop-type)
next
case ( $BinOp \text{ binop } e1 \ e2 \ c$ )
from  $\langle Env \vdash BinOp \text{ binop } e1 \ e2 :: -PrimT \ Boolean \rangle$ 
have  $Boolean = \text{binop-type binop}$  by cases simp
moreover
from  $\langle \text{constVal } (BinOp \text{ binop } e1 \ e2) = \text{Some } c \rangle$ 
obtain  $c1 \ c2$  where  $c = \text{eval-binop binop } c1 \ c2$  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  $c = \langle \text{constVal } (b \ ? \ e1 : e2) = \text{Some } c \rangle$ 
then obtain  $cb \ c1 \ c2$  where
   $cb: \text{constVal } b = \text{Some } cb$  and
   $c1: \text{constVal } e1 = \text{Some } c1$  and
   $c2: \text{constVal } e2 = \text{Some } c2$ 
by (auto split: bool.splits)
note  $wt = \langle Env \vdash b \ ? \ e1 : e2 :: -PrimT \ Boolean \rangle$ 
then
obtain  $T1 \ T2$ 
where  $Env \vdash b :: -PrimT \ Boolean$  and
   $wt-e1: Env \vdash e1 :: -PrimT \ Boolean$  and
   $wt-e2: Env \vdash e2 :: -PrimT \ Boolean$ 
by cases (auto dest: widen-Boolean2)
show  $?case$ 
proof (cases the-Bool cb)
case True
from  $c1 \ wt-e1$ 
have  $\text{typeof empty-dt } c1 = \text{Some } (PrimT \ Boolean)$ 
by (rule Cond.hyps)
with  $True \ c \ cb \ c1$  show  $?thesis$  by simp
next
case False
from  $c2 \ wt-e2$ 
have  $\text{typeof empty-dt } c2 = \text{Some } (PrimT \ Boolean)$ 
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  $\text{const } wt$ 
show  $?thesis$ 
by iprover
qed

```

lemma *assigns-if-good-approx:*

assumes

$\text{eval: prg } Env \vdash s0 \ -e-\> b \rightarrow s1$ **and**

$\text{normal: normal } s1$ **and**

bool: $Env \vdash e :: -PrimT\ Boolean$

shows *assigns-if* (*the-Bool* *b*) $e \subseteq dom\ (locals\ (store\ s1))$

proof –

— To properly perform induction on the evaluation relation we have to generalize the lemma to terms not only expressions.

have *generalized*: *assigns-if* (*the-Bool* (*the-In1* *val*)) (*the-In1* *t*) $\subseteq dom\ (locals\ (store\ s1))$

if *eval'*: $prg\ Env \vdash s0 \multimap t \rightarrow (val, s1)$

and *bool'*: $Env \vdash t :: In1\ (PrimT\ Boolean)$

and *expr*: $\exists\ expr. t = In1\ expr$

for *t val*

using *eval'* *normal bool' expr*

proof (*induct*)

case *Abrupt* **thus** ?*case* **by** *simp*

next

case (*NewC* *s0 C s1 a s2*)

from $\langle Env \vdash NewC\ C :: -PrimT\ Boolean \rangle$

have *False*

by *cases simp*

thus ?*case* ..

next

case (*NewA* *s0 T s1 e i s2 a s3*)

from $\langle Env \vdash New\ T[e] :: -PrimT\ Boolean \rangle$

have *False*

by *cases simp*

thus ?*case* ..

next

case (*Cast* *s0 e b s1 s2 T*)

note $s2 = \langle s2 = abrupt\ (raise\text{-}if\ (\neg\ prg\ Env, snd\ s1 \vdash b\ fits\ T)\ ClassCast)\ s1 \rangle$

have *assigns-if* (*the-Bool* *b*) $e \subseteq dom\ (locals\ (store\ s1))$

proof –

from *s2* **and** $\langle normal\ s2 \rangle$

have *normal s1*

by (*cases s1*) *simp*

moreover

from $\langle Env \vdash Cast\ T\ e :: -PrimT\ Boolean \rangle$

have $Env \vdash e :: -PrimT\ Boolean$

by *cases (auto dest: cast-Boolean2)*

ultimately show ?*thesis*

by (*rule Cast.hyps [elim-format]*) *auto*

qed

also from *s2*

have $\dots \subseteq dom\ (locals\ (store\ s2))$

by *simp*

finally show ?*case* **by** *simp*

next

case (*Inst* *s0 e v s1 b T*)

from $\langle prg\ Env \vdash Norm\ s0 \multimap e \multimap v \rightarrow s1 \rangle$ **and** $\langle normal\ s1 \rangle$

have *assignsE* $e \subseteq dom\ (locals\ (store\ s1))$

by (*rule assignsE-good-approx*)

thus ?*case*

by *simp*

next

case (*Lit* *s v*)

from $\langle Env \vdash Lit\ v :: -PrimT\ Boolean \rangle$

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  $\cup$  assigns-if (the-Bool v2) e2
     $\subseteq$  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  $\subseteq$  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  $\cup$  assigns-if (the-Bool v2) e2
       $\subseteq$  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  $\subseteq$  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  $\cup$  assigns-if (the-Bool v2) e2
         $\subseteq$  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  $\langle \neg (binop = CondAnd \vee binop = CondOr) \rangle$ 
from BinOp.hyps
have
  prg Env $\vdash$ Norm s0  $\neg BinOp\ binop\ e1\ e2 \neg \text{eval-binop}\ binop\ v1\ v2 \rightarrow s2$ 
  by  $\neg$  (rule eval.BinOp)
moreover note  $\langle normal\ s2 \rangle$ 
ultimately
have assignsE (BinOp binop e1 e2)  $\subseteq dom\ (locals\ (store\ s2))$ 
  by (rule assignsE-good-approx)
with False None
show ?thesis
  by simp
qed
qed
next
case Super
note  $\langle Env \vdash Super :: \neg PrimT\ Boolean \rangle$ 
hence False
  by cases simp
thus ?case ..
next
case (Acc s0 va v f s1)
from  $\langle prg\ Env \vdash Norm\ s0 \neg va = \text{eval}\ (v, f) \rightarrow s1 \rangle$  and  $\langle normal\ s1 \rangle$ 
have assignsV va  $\subseteq dom\ (locals\ (store\ s1))$ 
  by (rule assignsV-good-approx)
thus ?case by simp
next
case (Ass s0 va w f s1 e v s2)
hence prg Env $\vdash$ Norm s0  $\neg va := e \neg v \rightarrow assign\ f\ v\ s2$ 
  by  $\neg$  (rule eval.Ass)
moreover note  $\langle normal\ (assign\ f\ v\ s2) \rangle$ 
ultimately
have assignsE (va := e)  $\subseteq dom\ (locals\ (store\ (assign\ f\ v\ s2)))$ 
  by (rule assignsE-good-approx)
thus ?case by simp
next
case (Cond s0 e0 b s1 e1 e2 v s2)
from  $\langle Env \vdash e0\ ?\ e1 : e2 :: \neg PrimT\ Boolean \rangle$ 
obtain wt-e1: Env $\vdash$ e1 ::  $\neg PrimT\ Boolean$  and
  wt-e2: Env $\vdash$ e2 ::  $\neg PrimT\ Boolean$ 
  by cases (auto dest: widen-Boolean2)
note eval-e0 =  $\langle prg\ Env \vdash Norm\ s0 \neg e0 \neg b \rightarrow s1 \rangle$ 
have e0-s2: assignsE e0  $\subseteq dom\ (locals\ (store\ s2))$ 
proof  $\neg$ 
  note eval-e0
  moreover
  from Cond.hyps and  $\langle normal\ s2 \rangle$  have normal s1
    by  $\neg$  (erule eval-no-abrupt-lemma [rule-format], simp)
  ultimately

```

```

have assignsE  $e0 \subseteq \text{dom} (\text{locals} (\text{store } s1))$ 
  by (rule assignsE-good-approx)
also
from Cond
have  $\dots \subseteq \text{dom} (\text{locals} (\text{store } s2))$ 
  by  $-(\text{erule dom-locals-eval-mono } [\text{elim-format}], \text{simp})$ 
finally show ?thesis .
qed
show ?case
proof (cases constVal  $e0$ )
  case None
  have assigns-if (the-Bool  $v$ )  $e1 \cap \text{assigns-if} (\text{the-Bool } v) e2$ 
     $\subseteq \text{dom} (\text{locals} (\text{store } s2))$ 
  proof (cases the-Bool  $b$ )
    case True
    from  $\langle \text{normal } s2 \rangle$ 
    have assigns-if (the-Bool  $v$ )  $e1 \subseteq \text{dom} (\text{locals} (\text{store } s2))$ 
      by (rule Cond.hyps [elim-format]) (simp-all add: wt-e1 True)
    thus ?thesis
      by blast
  next
  case False
  from  $\langle \text{normal } s2 \rangle$ 
  have assigns-if (the-Bool  $v$ )  $e2 \subseteq \text{dom} (\text{locals} (\text{store } s2))$ 
    by (rule Cond.hyps [elim-format]) (simp-all add: wt-e2 False)
  thus ?thesis
    by blast
qed
with  $e0-s2$ 
have assignsE  $e0 \cup$ 
  ( $\text{assigns-if} (\text{the-Bool } v) e1 \cap \text{assigns-if} (\text{the-Bool } v) e2$ )
   $\subseteq \text{dom} (\text{locals} (\text{store } s2))$ 
  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  $\langle \text{normal } s2 \rangle$ 
  have assigns-if (the-Bool  $v$ )  $e1 \subseteq \text{dom} (\text{locals} (\text{store } s2))$ 
    by (rule Cond.hyps [elim-format]) (simp-all add: eq-b-c True wt-e1)
  with  $e0-s2$ 
  have assignsE  $e0 \cup \text{assigns-if} (\text{the-Bool } v) e1 \subseteq \dots$ 
    by (rule Un-least)
  with Some True show ?thesis
    by simp
next
case False
from  $\langle \text{normal } s2 \rangle$ 
have assigns-if (the-Bool  $v$ )  $e2 \subseteq \text{dom} (\text{locals} (\text{store } s2))$ 
  by (rule Cond.hyps [elim-format]) (simp-all add: eq-b-c False wt-e2)
with  $e0-s2$ 
have assignsE  $e0 \cup \text{assigns-if} (\text{the-Bool } v) e2 \subseteq \dots$ 
  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 — all the statements and variables
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 ⊆ C ⟹ A ∩ B ⊆ C
  by blast

lemma subset-Intr: B ⊆ C ⟹ A ∩ B ⊆ 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

```

$\longrightarrow ?NormalAssigned\ s1\ A \wedge ?BreakAssigned\ s0\ s1\ A \wedge ?ResAssigned\ s0\ s1$
 — Goal in object logic variant
let $?Hyp = \lambda t\ s0\ s1. (\bigwedge Env\ T\ A. \llbracket ?Wt\ Env\ t\ T; ?Da\ Env\ s0\ t\ A; prg\ Env = G \rrbracket$
 $\implies ?NormalAssigned\ s1\ A \wedge ?BreakAssigned\ s0\ s1\ A \wedge ?ResAssigned\ s0\ s1)$
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.prem*s **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.prem*s **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.prem*s
 show $?NormalAssigned\ s1\ A \wedge ?BreakAssigned\ (Norm\ s0)\ s1\ A$
 $\wedge ?ResAssigned\ (Norm\ s0)\ s1$
 by (*elim wt-elim-cases da-elim-cases*)
 (*rule Expr.hyps, auto*)
 next
 case (*Lab s0 c s1 j Env T A*)
 note $G = \langle prg\ Env = G \rangle$
 from *Lab.prem*s
 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.prem*s
 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 $nrm\ A \subseteq dom\ (locals\ (store\ (abupd\ (absorb\ j)\ s1)))$

```

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 –
  have (brk A l'  $\subseteq$  dom (locals (store (abupd (absorb j) s1))))
  if break: abrupt (abupd (absorb j) s1) = Some (Jump (Break l')) for l'
  proof –
    from j that have l  $\neq$  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 show ?thesis
    by (cases s1) auto
  qed
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 =  $\langle \text{prg } \text{Env} = G \rangle$ 
from Comp.premis
obtain C1 C2
  where da-c1:  $\text{Env} \vdash \text{dom} (\text{locals} (\text{snd} (\text{Norm } s0))) \gg \langle c1 \rangle \gg C1$  and
    da-c2:  $\text{Env} \vdash \text{nrm } C1 \gg \langle c2 \rangle \gg C2$  and
    A:  $\text{nrm } A = \text{nrm } C2$  brk A = (brk C1)  $\Rightarrow \cap$  (brk C2)
  by (elim da-elim-cases) simp
from Comp.premis
obtain wt-c1:  $\text{Env} \vdash c1 :: \checkmark$  and
  wt-c2:  $\text{Env} \vdash c2 :: \checkmark$ 
  by (elim wt-elim-cases) simp
note  $\langle \text{PROP } ?\text{Hyp } (\text{In1r } c1) (\text{Norm } s0) s1 \rangle$ 
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  $\text{nrm } C1 \subseteq \text{dom} (\text{locals} (\text{snd } s1))$  by iprover
  with da-c2 obtain C2'

```

```

where  $da-c2'$ :  $Env \vdash dom (locals (snd s1)) \gg \langle c2 \rangle \gg C2'$  and
       $nrm-c2$ :  $nrm C2 \subseteq nrm C2'$  and
       $brk-c2$ :  $\forall l. brk C2 l \subseteq brk C2' l$ 
by (rule da-weakenE) iprover
note  $\langle PROP ?Hyp (In1r c2) s1 s2 \rangle$ 
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  $\langle G \vdash s1 -c2 \rightarrow s2 \rangle$ 
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  $\exists l. abrupt s1 = Some (Jump (Break l))$ )
  case  $True$ 
  then obtain  $l$  where  $l$ :  $abrupt s1 = Some (Jump (Break l))$  ..
  with  $brk-c1$ 
  have  $brk C1 l \subseteq dom (locals (store s1))$ 
  by simp
  with  $A$   $eq-s1-s2$ 
  have  $brk A l \subseteq dom (locals (store s2))$ 
  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  $G = \langle prg Env = G \rangle$ 
with  $If.hyps$  have  $eval-e$ :  $prg Env \vdash Norm s0 -e-\succ b \rightarrow s1$  by simp
from  $If.prem$ s
obtain  $E$   $C1$   $C2$  where
   $da-e$ :  $Env \vdash dom (locals (store ((Norm s0)::state))) \gg \langle e \rangle \gg E$  and
   $da-c1$ :  $Env \vdash (dom (locals (store ((Norm s0)::state)))$ 
     $\cup assigns-if True e) \gg \langle c1 \rangle \gg C1$  and
   $da-c2$ :  $Env \vdash (dom (locals (store ((Norm s0)::state)))$ 

```

```

       $\cup \text{assigns-if False } e \rangle \langle c2 \rangle \rangle C2$  and
    A:  $\text{nrm } A = \text{nrm } C1 \cap \text{nrm } C2 \quad \text{brk } A = \text{brk } C1 \Rightarrow \cap \text{brk } C2$ 
    by (elim da-elim-cases)
  from If.prem
obtain
    wt-e:  $\text{Env} \vdash e :: - \text{PrimT Boolean}$  and
    wt-c1:  $\text{Env} \vdash c1 :: \checkmark$  and
    wt-c2:  $\text{Env} \vdash c2 :: \checkmark$ 
    by (elim wt-elim-cases)
from If.hyps have
  s0-s1:  $\text{dom } (\text{locals } (\text{store } ((\text{Norm } s0)::\text{state}))) \subseteq \text{dom } (\text{locals } (\text{store } s1))$ 
    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  $\text{assigns-if True } e \subseteq \text{dom } (\text{locals } (\text{store } s1))$ 
          by (rule assigns-if-good-approx [elim-format]) (simp add: True)
        with s0-s1
        have  $\text{dom } (\text{locals } (\text{store } ((\text{Norm } s0)::\text{state}))) \cup \text{assigns-if True } e \subseteq \dots$ 
          by (rule Un-least)
        with da-c1 obtain C1'
          where da-c1':  $\text{Env} \vdash \text{dom } (\text{locals } (\text{store } s1)) \rangle \langle c1 \rangle \rangle C1'$  and
            nrm-c1:  $\text{nrm } C1 \subseteq \text{nrm } C1'$  and
            brk-c1:  $\forall l. \text{brk } C1 \ l \subseteq \text{brk } C1' \ l$ 
          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  $\text{assigns-if False } e \subseteq \text{dom } (\text{locals } (\text{store } s1))$ 
            by (rule assigns-if-good-approx [elim-format]) (simp add: False)
          with s0-s1
          have  $\text{dom } (\text{locals } (\text{store } ((\text{Norm } s0)::\text{state}))) \cup \text{assigns-if False } e \subseteq \dots$ 
            by (rule Un-least)
          with da-c2 obtain C2'
            where da-c2':  $\text{Env} \vdash \text{dom } (\text{locals } (\text{store } s1)) \rangle \langle c2 \rangle \rangle C2'$  and
              nrm-c2:  $\text{nrm } C2 \subseteq \text{nrm } C2'$  and
              brk-c2:  $\forall l. \text{brk } C2 \ l \subseteq \text{brk } C2' \ l$ 

```

```

    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  $\bigwedge j. \text{abrupt } s1 \neq \text{Some } (\text{Jump } j)$ 
  by (rule eval-expression-no-jump) (simp-all add: G wf)
moreover
have s2 = s1
proof -
  from abr and  $\langle G \vdash s1 \rightarrow (\text{if the-Bool } b \text{ then } c1 \text{ else } c2) \rightarrow s2 \rangle$ 
  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 =  $\langle \text{prg } \text{Env} = G \rangle$ 
with Loop.hyps have eval-e:  $\text{prg } \text{Env} \vdash \text{Norm } s0 \rightarrow e \rightarrow b \rightarrow s1$ 
  by (simp (no-asm-simp))
from Loop.prem
obtain E C where
  da-e:  $\text{Env} \vdash \text{dom } (\text{locals } (\text{store } ((\text{Norm } s0)::\text{state}))) \gg \langle e \rangle \gg E$  and
  da-c:  $\text{Env} \vdash (\text{dom } (\text{locals } (\text{store } ((\text{Norm } s0)::\text{state}))) \cup \text{assigns-if True } e) \gg \langle c \rangle \gg C$  and
  A:  $\text{nrm } A = \text{nrm } C \cap (\text{dom } (\text{locals } (\text{store } ((\text{Norm } s0)::\text{state}))) \cup \text{assigns-if False } e)$ 
  brk A = brk C
  by (elim da-elim-cases)
from Loop.prem
obtain
  wt-e:  $\text{Env} \vdash e::\neg \text{PrimT Boolean}$  and
  wt-c:  $\text{Env} \vdash c::\checkmark$ 
  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:  $\text{dom } (\text{locals } (\text{store } ((\text{Norm } s0)::\text{state}))) \subseteq \text{dom } (\text{locals } (\text{store } s1))$ 

```



```

  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:  $G \vdash s1 \rightarrow s2$  and
    eval-while:  $G \vdash \text{abupd} (\text{absorb} (\text{Cont } l)) s2 \rightarrow l \cdot \text{While}(e) c \rightarrow s3$ 
  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  $\subseteq \text{dom} (\text{locals} (\text{store } s1))$ 
    by (rule assigns-if-good-approx [elim-format]) (simp add: True)
  with s0-s1
  have dom (locals (store ((Norm s0)::state)))  $\cup \text{assigns-if True } e \subseteq \dots$ 
    by (rule Un-least)
  with da-c obtain C'
    where da-c':  $\text{Env} \vdash \text{dom} (\text{locals} (\text{store } s1)) \gg \langle c \rangle \gg C'$  and
      nrm-C-C':  $\text{nrm } C \subseteq \text{nrm } C'$  and
      brk-C-C':  $\forall l. \text{brk } C l \subseteq \text{brk } C' l$ 
    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  $\vee$  abrupt s2 = Some (Jump (Cont l)))
  case True
  from Loop.prem obtain
    wt-while:  $\text{Env} \vdash \text{In1r} (l \cdot \text{While}(e) c) :: T$  and
    da-while:  $\text{Env} \vdash \text{dom} (\text{locals} (\text{store} ((\text{Norm } s0)::\text{state})))$ 
       $\gg \langle l \cdot \text{While}(e) c \rangle \gg A$ 
  by simp
  have dom (locals (store ((Norm s0)::state)))
     $\subseteq \text{dom} (\text{locals} (\text{store} (\text{abupd} (\text{absorb} (\text{Cont } l)) s2)))$ 
  proof -
    note s0-s1
    also from eval-c
    have dom (locals (store s1))  $\subseteq \text{dom} (\text{locals} (\text{store } s2))$ 
      by (rule dom-locals-eval-mono-elim)
    also have  $\dots \subseteq \text{dom} (\text{locals} (\text{store} (\text{abupd} (\text{absorb} (\text{Cont } l)) s2)))$ 
      by simp
    finally show ?thesis .
  qed
  with da-while obtain A'
  where
    da-while':  $\text{Env} \vdash \text{dom} (\text{locals} (\text{store} (\text{abupd} (\text{absorb} (\text{Cont } l)) s2)))$ 
       $\gg \langle l \cdot \text{While}(e) c \rangle \gg A'$ 
    and nrm-A-A':  $\text{nrm } A \subseteq \text{nrm } A'$ 

```

```

    and brk-A-A':  $\forall l. \text{brk } A \ l \subseteq \text{brk } A' \ l$ 
    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  $e \subseteq \text{dom } (\text{locals } (\text{store } s1))$ 
  by (rule assigns-if-good-approx [elim-format]) (simp add: False)
with s0-s1
have  $\text{dom } (\text{locals } (\text{store } ((\text{Norm } s0)::\text{state}))) \cup \text{assigns-if False } e \subseteq \dots$ 
  by (rule Un-least)

```

```

hence  $nrm\ C \sqcap$ 
   $(dom\ (locals\ (store\ ((Norm\ s0)::state))) \cup assigns\text{-}if\ False\ e)$ 
   $\subseteq dom\ (locals\ (store\ s1))$ 
  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:  $\bigwedge j. abrupt\ s1 \neq Some\ (Jump\ j)$ 
  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:  $G \vdash s1 \rightarrow c \rightarrow s2$  and
    eval-while:  $G \vdash abupd\ (absorb\ (Cont\ l))\ s2 \rightarrow l. While(e)\ c \rightarrow s3$ 
  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.premis
obtain ret:  $j = Ret \rightarrow Result \in dom\ (locals\ (store\ (Norm\ s)))$  and
  brk:  $brk\ A = (case\ j\ of$ 
    Break l  $\Rightarrow \lambda k. if\ k=l$ 
      then  $dom\ (locals\ (store\ ((Norm\ s)::state)))$ 
    else UNIV
    | Cont l  $\Rightarrow \lambda k. UNIV$ 
    | Ret  $\Rightarrow \lambda k. UNIV)$ 
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.premis obtain E where
  da-e: Env ⊢ dom (locals (store ((Norm s0)::state))) »⟨e⟩ E
  by (elim da-elim-cases)
from Throw.premis
  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.premis 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.premis 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  $\text{nrm } C1 \cap \text{nrm } C2 \subseteq \text{dom } (\text{locals } (\text{store } s1))$ 
  by auto
moreover
have  $s3=s1$ 
proof –
  from sxalloc True have  $\text{eq-}s2-s1: s2=s1$ 
    by (cases s1) (auto elim: sxalloc-elim-cases)
  with True have  $\neg G, s2 \vdash \text{catch } C$ 
    by (simp add: catch-def)
  with Try.hyps have  $s3=s2$ 
    by simp
  with  $\text{eq-}s2-s1$  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  $\exists l. \text{abrupt } s1 = \text{Some } (\text{Jump } (\text{Break } l))$ )
  case True
  then obtain l where  $l: \text{abrupt } s1 = \text{Some } (\text{Jump } (\text{Break } l))$ 
    by auto
  with brk-C1 have  $(\text{brk } C1 \Rightarrow \cap \text{brk } C2) \ l \subseteq \text{dom } (\text{locals } (\text{store } s1))$ 
    by auto
  moreover have  $s3=s1$ 
  proof –
    from sxalloc l have  $\text{eq-}s2-s1: s2=s1$ 
      by (cases s1) (auto elim: sxalloc-elim-cases)
    with l have  $\neg G, s2 \vdash \text{catch } C$ 
      by (simp add: catch-def)
    with Try.hyps have  $s3=s2$ 
      by simp
    with  $\text{eq-}s2-s1$  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  $G, s2 \vdash \text{catch } C$ )
  case True
  with Try.hyps have  $?HypObj \ (In1r \ c2) \ (\text{new-xcpt-var } vn \ s2) \ s3$ 
    by simp
  note hyp-c2 = this [rule-format]
  have  $(\text{dom } (\text{locals } (\text{store } ((\text{Norm } s0)::\text{state}))) \cup \{VName \ vn\})$ 
     $\subseteq \text{dom } (\text{locals } (\text{store } (\text{new-xcpt-var } vn \ s2)))$ 
  proof –
    from  $\langle G \vdash \text{Norm } s0 -c1 \rightarrow s1 \rangle$ 
    have  $\text{dom } (\text{locals } (\text{store } ((\text{Norm } s0)::\text{state})))$ 
       $\subseteq \text{dom } (\text{locals } (\text{store } s1))$ 
    by (rule dom-locals-eval-mono-elim)
    also
    from sxalloc
    have  $\dots \subseteq \text{dom } (\text{locals } (\text{store } s2))$ 
      by (rule dom-locals-sxalloc-mono)
    also

```

```

have ...  $\subseteq \text{dom} (\text{locals} (\text{store} (\text{new-xcpt-var } vn \ s2)))$ 
  by (cases s2) (simp add: new-xcpt-var-def, blast)
also
have { VName vn }  $\subseteq \dots$ 
  by (cases s2) simp
ultimately show ?thesis
  by (rule Un-least)
qed
with da-c2
obtain C2' where
  da-C2':  $\text{Env}(\text{lcl} := (\text{lcl } \text{Env})(\text{VName } vn \mapsto \text{Class } C))$ 
     $\vdash \text{dom} (\text{locals} (\text{store} (\text{new-xcpt-var } vn \ s2))) \gg \langle c2 \rangle \gg C2'$ 
and nrm-C2':  $\text{nrm } C2 \subseteq \text{nrm } C2'$ 
and brk-C2':  $\forall l. \text{brk } C2 \ l \subseteq \text{brk } C2' \ l$ 
  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 xalloc not-normal-s1 abrupt-no-break
obtain  $\neg \text{normal } s2$ 
   $\forall l. \text{abrupt } s2 \neq \text{Some } (\text{Jump } (\text{Break } l))$ 
  by – (rule xalloc-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 xalloc have  $s2=s1$ 
    by (elim xalloc-cases) auto
  with res-s1 eq-s3-s2 show ?thesis by simp
next
case False
with xalloc
have  $\text{abrupt } s2 \neq \text{Some } (\text{Jump } \text{Ret})$ 
  by (rule xalloc-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.premis 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) ⇒ ∪v (nrm C2)) ⇒ ∩ (brk C2)
  by (elim da-elim-cases) simp
from Fin.premis 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  $\subseteq$  dom (locals (snd s3))
    proof -
      have nrm C1  $\subseteq$  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  $\subseteq$  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  $\cup$  nrm C2  $\subseteq$  ...
        by (rule Un-least)
      with nrm-A show ?thesis
        by simp
    qed
  qed

  moreover
  have brk A l  $\subseteq$  dom (locals (store s3)) if brk-s3: abrupt s3 = Some (Jump (Break l)) for l
  proof (cases normal s2)
    case True
    with brk-s3 s3
    have s2-s3: dom (locals (store s2))  $\subseteq$  dom (locals (store s3))
      by simp
    have brk C1 l  $\subseteq$  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  $\subseteq$  dom (locals (store s3))
      by - (rule subset-trans, simp-all)
    ultimately
    have ((brk C1)  $\Rightarrow_{\cup_V}$  (nrm C2)) l  $\subseteq$  ...
      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  $\subseteq$  dom (locals (store s3))
  by simp
with brk-A show ?thesis
  by simp blast
qed
hence ?BreakAssigned (Norm s0) s3 A
  by simp
moreover
have Result  $\in$  dom (locals (store s3)) if abr-s3: abrupt s3 = Some (Jump Ret)
proof (cases x1 = Some (Jump Ret))
  case True
  note ret-x1 = this
  with resAss-s1 have res-s1: Result  $\in$  dom (locals s1)
    by simp
  moreover have dom (locals (store ((Norm s1)::state)))
     $\subseteq$  dom (locals (store s2))
    by (rule dom-locals-eval-mono-elim) (rule Fin.hyps)
  ultimately have Result  $\in$  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 =  $\langle$ prg Env = G $\rangle$ 
from Init.hyps
have eval: prg Env  $\vdash$  Norm s0  $\rightarrow$  Init C  $\rightarrow$  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.premis and  $\langle$ the (class G C) = c $\rangle$ 
have c: class G C = Some c
  by (elim wt-elim-cases) auto
from Init.premis 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.prem
  obtain A: nrm A = dom (locals (store ((Norm s0)::state)))
    brk A = (λ l. UNIV)
  by (elim da-elim-cases) simp
  from wf NewC.prem
  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
  have abrupt s2 ≠ Some (Jump j) for 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
  have abrupt s3 ≠ Some (Jump j) for j
  proof -
    have eval: prg Env⊢ Norm s0 -New elT[e]⊢ Addr a→ s3
      unfolding G by (rule eval.NewA NewA.hyps)+
    from NewA.premss
    obtain T' where T=Inl T'
      by (elim wt-elim-cases) simp
    with NewA.premss 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.premss obtain
    da-e: Env⊢ dom (locals (store ((Norm s0)::state))) »⟨e⟩ A
    by (elim da-elim-cases)
  from Cast.premss obtain eT where
    wt-e: Env⊢ e⊢- eT
    by (elim wt-elim-cases)
  note ⟨PROP ?Hyp (Inl 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
  have abrupt s2 ≠ Some (Jump j) for j
  proof -
    have eval: prg Env⊢ Norm s0 -Cast cT e⊢- v→ s2
      unfolding G by (rule eval.Cast Cast.hyps)+
    from Cast.premss
    obtain T' where T=Inl T'
      by (elim wt-elim-cases) simp
    with Cast.premss 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.premis obtain
  da-e: Env ⊢ dom (locals (store ((Norm s0)::state))) »⟨e⟩ A
  by (elim da-elim-cases)
from Inst.premis 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.premis
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.premis obtain
  da-e: Env ⊢ dom (locals (store ((Norm s0)::state))) »⟨e⟩ A
  by (elim da-elim-cases)
from UnOp.premis 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.premis 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  $\subseteq$  dom (locals (store s2))
  proof (cases binop=CondAnd)
    case True
    note CondAnd = this
    from BinOp.premis obtain
      nrm-A: nrm A = dom (locals (store ((Norm s0)::state)))
         $\cup$  (assigns-if True (BinOp CondAnd e1 e2)  $\cap$ 
          assigns-if False (BinOp CondAnd e1 e2))
      by (elim da-elim-cases) (simp-all add: CondAnd)
    from T BinOp.premis CondAnd
    have Env $\vdash$ BinOp binop e1 e2:: $\neg$ PrimT Boolean
      by (simp)
    with eval normal-s2
    have ass-if: assigns-if (the-Bool (eval-binop binop v1 v2))
      (BinOp binop e1 e2)
       $\subseteq$  dom (locals (store s2))
      by (rule assigns-if-good-approx)
    have (assigns-if True (BinOp CondAnd e1 e2)  $\cap$ 
      assigns-if False (BinOp CondAnd e1 e2))  $\subseteq$  ...
    proof (cases the-Bool (eval-binop binop v1 v2))
      case True
      with ass-if CondAnd
      have assigns-if True (BinOp CondAnd e1 e2)
         $\subseteq$  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)
         $\subseteq$  dom (locals (store s2))
        by (simp only: False)
      thus ?thesis by blast
    qed
  with s0-s2
  have dom (locals (store ((Norm s0)::state)))
     $\cup$  (assigns-if True (BinOp CondAnd e1 e2)  $\cap$ 
      assigns-if False (BinOp CondAnd e1 e2))  $\subseteq$  ...
    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.premis obtain
      nrm-A: nrm A = dom (locals (store ((Norm s0)::state)))
         $\cup$  (assigns-if True (BinOp CondOr e1 e2)  $\cap$ 
          assigns-if False (BinOp CondOr e1 e2))

```

```

    by (elim da-elim-cases) (simp-all add: CondOr)
  from T BinOp.premis 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.premis 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
  have abrupt s2 ≠ Some (Jump j) for j
  proof -
    from BinOp.premis 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.premis
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.premis
  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.premis
have da-v: Env⊢ dom (locals (store ((Norm s0)::state))) »⟨v⟩» A
  by (elim da-elim-cases) simp-all
from Acc.premis 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.premis 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 $\vdash$ Norm s0 -var= $\succ$ (w, upd) $\rightarrow$  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 =  $\langle \text{PROP } ?\text{Hyp } (\text{In2 } \text{var}) (\text{Norm } s0) s1 \rangle$ 
  note hyp-e =  $\langle \text{PROP } ?\text{Hyp } (\text{In1l } e) s1 s2 \rangle$ 
  show nrm A  $\subseteq$  dom (locals (store (assign upd v s2)))
  proof (cases  $\exists$  vn. var = LVar vn)
    case True
    then obtain vn where vn: var=LVar vn..
    from Ass.premis obtain E where
      da-e: Env $\vdash$  dom (locals (store ((Norm s0)::state)))  $\gg \langle e \rangle$  E and
      nrm-A: nrm A = nrm E  $\cup$  {vn}
    by (elim da-elim-cases) (use vn in auto)
    obtain E' where
      da-e': Env $\vdash$  dom (locals (store s1))  $\gg \langle e \rangle$  E' and
      E-E': nrm E  $\subseteq$  nrm E'
    proof -
      have dom (locals (store ((Norm s0)::state)))
         $\subseteq$  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 $\vdash$ Norm s0 -LVar vn= $\succ$ (w, upd) $\rightarrow$  s1
      by simp
    then have upd: upd = snd (lvar vn (store s1))
      by cases (cases lvar vn (store s1),simp)
    have nrm E  $\subseteq$  dom (locals (store (assign upd v s2)))
    proof -
      from hyp-e wt-e da-e' G normal-s2
      have nrm E'  $\subseteq$  dom (locals (store s2))
      by simp
      also
      from upd
      have dom (locals (store s2))  $\subseteq$  dom (locals (store (upd v s2)))
      by (simp add: lvar-def) blast
      hence dom (locals (store s2))
         $\subseteq$  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}  $\subseteq$  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  $\subseteq$  ...
      by (rule Un-least [elim-format]) (simp add: nrm-A)
  next

```

```

case False
from Ass.prems obtain V where
  da-var:  $\text{Env} \vdash \text{dom} (\text{locals} (\text{store} ((\text{Norm } s0)::\text{state}))) \gg \langle \text{var} \rangle \gg V$  and
  da-e:  $\text{Env} \vdash \text{nrm } V \gg \langle e \rangle \gg A$ 
  by (elim da-elim-cases) (use False in simp-all)
from hyp-var wt-var da-var G normal-s1
have  $\text{nrm } V \subseteq \text{dom} (\text{locals} (\text{store } s1))$ 
  by simp
with da-e obtain A'
  where da-e':  $\text{Env} \vdash \text{dom} (\text{locals} (\text{store } s1)) \gg \langle e \rangle \gg A'$  and
    nrm-A-A':  $\text{nrm } A \subseteq \text{nrm } A'$ 
  by (rule da-weakenE) iprover
from hyp-e wt-e da-e' G normal-s2
obtain  $\text{nrm } A' \subseteq \text{dom} (\text{locals} (\text{store } s2))$ 
  by simp
with nrm-A-A' have  $\text{nrm } A \subseteq \dots$ 
  by blast
also have  $\dots \subseteq \text{dom} (\text{locals} (\text{store} (\text{assign upd } v \ s2)))$ 
proof –
  from eval-var normal-s1
  have  $\text{dom} (\text{locals} (\text{store } s2)) \subseteq \text{dom} (\text{locals} (\text{store} (\text{upd } v \ s2)))$ 
    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
have abrupt (assign upd v s2)  $\neq \text{Some } (\text{Jump } j)$  for j
proof –
  have eval: prg  $\text{Env} \vdash \text{Norm } s0 \text{ --var:=e--} \rightarrow (\text{assign upd } v \ s2)$ 
    by (simp only: G) (rule eval.Ass Ass.hyps) +
from Ass.prems
obtain T' where  $T = \text{Inl } T'$ 
  by (elim wt-elim-cases) simp
with Ass.prems have  $\text{Env} \vdash \text{var} := e :: - T'$  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  $G = \langle \text{prg } \text{Env} = G \rangle$ 
have ?NormalAssigned s2 A
proof
  assume normal-s2: normal s2
  show  $\text{nrm } A \subseteq \text{dom} (\text{locals} (\text{store } s2))$ 
proof (cases  $\text{Env} \vdash (e0 \ ? \ e1 : e2) :: - (\text{PrimT Boolean})$ )
  case True
  with Cond.prems
  have nrm-A:  $\text{nrm } A = \text{dom} (\text{locals} (\text{store} ((\text{Norm } s0)::\text{state})))$ 
     $\cup (\text{assigns-if True } (e0 \ ? \ e1 : e2) \cap$ 
     $\text{assigns-if False } (e0 \ ? \ e1 : e2))$ 

```

```

  by (elim da-elim-cases) simp-all
have eval: prg Env⊢Norm s0  $\neg(e0 \text{ ? } e1 : e2) \multimap v \rightarrow s2$ 
  unfolding G by (rule eval.Cond Cond.hyps)+
from eval
have dom (locals (store ((Norm s0)::state)))  $\subseteq$  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)
   $\subseteq$  dom (locals (store s2))
  by (rule assigns-if-good-approx)
have assigns-if True (e0 ? e1:e2)  $\cap$  assigns-if False (e0 ? e1:e2)
   $\subseteq$  dom (locals (store s2))
proof (cases the-Bool v)
case True
  from ass-if
  have assigns-if True (e0 ? e1:e2)  $\subseteq$  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)  $\subseteq$  dom (locals (store s2))
    by (simp only: False)
  thus ?thesis by blast
qed
ultimately show nrm A  $\subseteq$  dom (locals (store s2))
  by (simp only: nrm-A) (rule Un-least)
next
case False
with Cond.premis obtain E1 E2 where
  da-e1: Env⊢ (dom (locals (store ((Norm s0)::state)))
     $\cup$  assigns-if True e0)  $\gg \langle e1 \rangle \gg$  E1 and
  da-e2: Env⊢ (dom (locals (store ((Norm s0)::state)))
     $\cup$  assigns-if False e0)  $\gg \langle e2 \rangle \gg$  E2 and
  nrm-A: nrm A = nrm E1  $\cap$  nrm E2
  by (elim da-elim-cases) simp-all
from Cond.premis 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)))
   $\subseteq$  dom (locals (store s1))
  by (rule dom-locals-eval-mono-elim) (rule Cond.hyps)
have eval-e0: prg Env⊢Norm s0  $\neg e0 \multimap b \rightarrow s1$ 
  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  $\subseteq$  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)))

```

```

       $\cup \text{ assigns-if True } e0 \subseteq \dots$ 
    by (rule Un-least)
  with da-e1 obtain E1' where
    da-e1':  $\text{Env} \vdash \text{dom} (\text{locals} (\text{store } s1)) \gg \langle e1 \rangle \gg E1'$  and
    nrm-E1-E1':  $\text{nrm } E1 \subseteq \text{nrm } E1'$ 
    by (rule da-weakenE) iprover
  ultimately have  $\text{nrm } E1' \subseteq \text{dom} (\text{locals} (\text{store } s2))$ 
    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  $e0 \subseteq \text{dom} (\text{locals} (\text{store } s1))$ 
    by (rule assigns-if-good-approx [elim-format]) (simp only: False)
  with s0-s1
  have  $\text{dom} (\text{locals} (\text{store} ((\text{Norm } s0)::\text{state})))$ 
     $\cup \text{ assigns-if False } e0 \subseteq \dots$ 
    by (rule Un-least)
  with da-e2 obtain E2' where
    da-e2':  $\text{Env} \vdash \text{dom} (\text{locals} (\text{store } s1)) \gg \langle e2 \rangle \gg E2'$  and
    nrm-E2-E2':  $\text{nrm } E2 \subseteq \text{nrm } E2'$ 
    by (rule da-weakenE) iprover
  ultimately have  $\text{nrm } E2' \subseteq \text{dom} (\text{locals} (\text{store } s2))$ 
    using wt-e2 G normal-s2 by simp
  with nrm-E2-E2' show ?thesis
    by (simp only: nrm-A) blast
qed
qed
qed
moreover
have abrupt s2  $\neq \text{Some} (\text{Jump } j)$  for j
proof -
  have eval:  $\text{prg } \text{Env} \vdash \text{Norm } s0 - e0 \text{ ? } e1 : e2 \multimap v \rightarrow s2$ 
    unfolding G by (rule eval.Cond Cond.hyps)+
  from Cond.prem
  obtain T' where  $T = \text{Inl } T'$ 
    by (elim wt-elim-cases) simp
  with Cond.prem have  $\text{Env} \vdash e0 \text{ ? } e1 : e2 :: -T'$  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  $G = \langle \text{prg } \text{Env} = G \rangle$ 
  have ?NormalAssigned (restore-lvars s2 s4) A
  proof
    assume normal-restore-lvars: normal (restore-lvars s2 s4)
    show  $\text{nrm } A \subseteq \text{dom} (\text{locals} (\text{store} (\text{restore-lvars } s2 \text{ s4})))$ 
    proof -
      from Call.prem obtain E where
        da-e:  $\text{Env} \vdash (\text{dom} (\text{locals} (\text{store} ((\text{Norm } s0)::\text{state})))) \gg \langle e \rangle \gg E$  and

```

```

  da-args: Env ⊢ nrm E »⟨args⟩» A
  by (elim da-elim-cases)
from Call.premis 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
have abrupt (restore-lvars s2 s4) ≠ Some (Jump j) for 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.premis
obtain T' where T=Inl T'
  by (elim wt-elim-cases) simp
with Call.premis 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.premis 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 (mbdy (mthd m)))⟩ » A
  by - (erule da-elim-cases)
from Methd.premis m obtain
  isCls: is-class (prg Env) D and
  wt-body: Env ⊢ In1l (Body (declclass m) (stmt (mbdy (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.premis
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.premis
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.premis
have da-e: Env⊢ (dom (locals (store ((Norm s0)::state)))) »⟨e⟩» A
  by (elim da-elim-cases)
from FVar.premis 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
moreover
have abrupt s3 ≠ Some (Jump j) for 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.premis
  obtain T' where T=Inl T'
    by (elim wt-elim-cases) simp
  with FVar.premis 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.premis 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.premis 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
    qed
    finally show ?thesis .
  qed
qed
moreover
have abrupt s2' ≠ Some (Jump j) for 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.premis
  obtain T' where T = Inl T'
  by (elim wt-elim-cases) simp
  with AVar.premis 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.prem
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.prem 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.prem 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 (In1 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
have abrupt s2 ≠ Some (Jump j) for j
proof -
  have eval: prg Env ⊢ Norm s0 - (e # es) ≐ v # vs → s2
  unfolding G by (rule eval.Cons Cons.hyps) +
  from Cons.prem
  obtain T' where T = Inr T'
  by (elim wt-elim-cases) simp
  with Cons.prem 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
 $prg\ Env \vdash s0 \rightarrow t \rightarrow (v, s1)$ and $Env \vdash t :: T$ and
 $Env \vdash dom\ (locals\ (store\ s0)) \gg t \gg A$ and $wf\text{-}prog\ (prg\ Env)$
obtains
 $normal\ s1 \implies nrm\ A \subseteq dom\ (locals\ (store\ s1))$ and
 $\bigwedge l. \llbracket abrupt\ s1 = Some\ (Jump\ (Break\ l)); normal\ s0 \rrbracket$
 $\implies brk\ A\ l \subseteq dom\ (locals\ (store\ s1))$ and
 $\llbracket abrupt\ s1 = Some\ (Jump\ Ret); normal\ s0 \rrbracket \implies Result \in dom\ (locals\ (store\ s1))$
using *assms* by - (*drule* (3) *da-good-approx, simp*)

lemma *da-good-approxE'*:

assumes $eval: G \vdash s0 \rightarrow t \rightarrow (v, s1)$
and $wt: \llbracket prg = G, cls = C, lcl = L \rrbracket \vdash t :: T$
and $da: \llbracket prg = G, cls = C, lcl = L \rrbracket \vdash dom\ (locals\ (store\ s0)) \gg t \gg A$
and $wf: wf\text{-}prog\ G$
obtains $normal\ s1 \implies nrm\ A \subseteq dom\ (locals\ (store\ s1))$ and
 $\bigwedge l. \llbracket abrupt\ s1 = Some\ (Jump\ (Break\ l)); normal\ s0 \rrbracket$
 $\implies brk\ A\ l \subseteq dom\ (locals\ (store\ s1))$ and
 $\llbracket abrupt\ s1 = Some\ (Jump\ Ret); normal\ s0 \rrbracket$
 $\implies Result \in dom\ (locals\ (store\ s1))$

proof -

from *eval* **have** $prg\ \llbracket prg = G, cls = C, lcl = L \rrbracket \vdash s0 \rightarrow t \rightarrow (v, s1)$ **by** *simp*
moreover **note** *wt da*
moreover **from** *wf* **have** $wf\text{-}prog\ (prg\ \llbracket prg = G, cls = C, lcl = L \rrbracket)$ **by** *simp*
ultimately **show** *thesis*
using *that* **by** (*rule da-good-approxE*) *iprover* +

qed

declare $\llbracket simproc\ add: wt\text{-}expr\ wt\text{-}var\ wt\text{-}exprs\ wt\text{-}stmt \rrbracket$

end

Chapter 19

TypeSafe

1 The type soundness proof for Java

```
theory TypeSafe
imports DefiniteAssignmentCorrect Conform
begin

error free

lemma error-free-halloc:
  assumes halloc:  $G \vdash s0 \text{ --halloc } oi \succ a \rightarrow s1$  and
    error-free-s0: error-free s0
  shows error-free s1
proof -
  from halloc error-free-s0
  obtain abrupt0 store0 abrupt1 store1
    where eqs:  $s0 = (abrupt0, store0)$   $s1 = (abrupt1, store1)$  and
      halloc':  $G \vdash (abrupt0, store0) \text{ --halloc } oi \succ a \rightarrow (abrupt1, store1)$  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:  $G \vdash s0 \text{ --sxalloc } \rightarrow s1$  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':  $G \vdash (abrupt0, store0) \text{ --sxalloc } \rightarrow (abrupt1, store1)$  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)
 $\implies$  (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)
 $\implies$  (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
 $\implies$  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  $\implies$ 
    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  $\implies$  error-free (assign ( $\lambda v. \text{supd lupd}(vn \mapsto v)$ ) w s)
by (cases s) simp

```

```

lemma error-free-throw [simp,intro]:
  error-free s  $\implies$  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 ( $\langle \cdot \leq | \cdot \preceq \cdot \rangle \vdash \cdot \rightarrow \cdot$ ) [71,71,71,71] 70)
where
  s  $\leq | f \preceq T :: \preceq E$  =
    ( $(\forall s' w. \text{Norm } s' :: \preceq E \longrightarrow \text{fst } E, s \vdash w :: \preceq T \longrightarrow s \leq | s' \longrightarrow \text{assign } f w (\text{Norm } s') :: \preceq E) \wedge$ 
     ( $\forall s' w. \text{error-free } s' \longrightarrow (\text{error-free } (\text{assign } f w s'))$ )))

```

definition

```

rconf :: prog  $\Rightarrow$  lenv  $\Rightarrow$  st  $\Rightarrow$  term  $\Rightarrow$  vals  $\Rightarrow$  tys  $\Rightarrow$  bool ( $\langle \cdot, \cdot, \vdash \cdot \rightarrow \cdot \rangle \vdash \cdot$ ) [71,71,71,71,71,71] 70)
where
  G, L, s  $\vdash t \rightarrow v :: \preceq T$  =
    (case T of

```

$$\begin{aligned}
& \text{Inl } T \Rightarrow \text{if } (\exists \text{ var. } t = \text{In2 } \text{var}) \\
& \quad \text{then } (\forall n. (\text{the-In2 } t) = \text{LVar } n \\
& \quad \quad \rightarrow (\text{fst } (\text{the-In2 } v) = \text{the } (\text{locals } s \ n)) \wedge \\
& \quad \quad (\text{locals } s \ n \neq \text{None} \rightarrow G, s \vdash \text{fst } (\text{the-In2 } v) :: \preceq T)) \wedge \\
& \quad \quad (\neg (\exists n. \text{the-In2 } t = \text{LVar } n) \rightarrow (G, s \vdash \text{fst } (\text{the-In2 } v) :: \preceq T)) \wedge \\
& \quad \quad (s \leq | \text{snd } (\text{the-In2 } v) \preceq T :: \preceq (G, L)) \\
& \quad \text{else } G, s \vdash \text{the-In1 } v :: \preceq T \\
& | \text{Inr } Ts \Rightarrow \text{list-all2 } (\text{conf } G \ s) \ (\text{the-In3 } v) \ Ts)
\end{aligned}$$

With *rconf* we describe the conformance of the result value of a term. This definition gets rather complicated because of the relations between the injections of the different terms, types and values. The main case distinction is between single values and value lists. In case of value lists, every value has to conform to its type. For single values we have to do a further case distinction, between values of variables $\exists \text{ var. } t = \text{In2 } \text{var}$ and ordinary values. Values of variables are modelled as pairs consisting of the current value and an update function which will perform an assignment to the variable. This stems from the decision, that we only have one evaluation rule for each kind of variable. The decision if we read or write to the variable is made by syntactic enclosing rules. So conformance of variable-values must ensure that both the current value and an update will conform to the type. With the introduction of definite assignment of local variables we have to do another case distinction. For the notion of conformance local variables are allowed to be *None*, since the definedness is not ensured by conformance but by definite assignment. Field and array variables must contain a value.

lemma *rconf-In1 [simp]*:

$G, L, s \vdash \text{In1 } ec \succ \text{In1 } v :: \preceq \text{Inl } T = G, s \vdash v :: \preceq T$
apply (*unfold rconf-def*)
apply (*simp (no-asm)*)
done

lemma *rconf-In2-no-LVar [simp]*:

$\forall n. va \neq \text{LVar } n \implies$
 $G, L, s \vdash \text{In2 } va \succ \text{In2 } vf :: \preceq \text{Inl } T = (G, s \vdash \text{fst } vf :: \preceq T \wedge s \leq | \text{snd } vf \preceq T :: \preceq (G, L))$
apply (*unfold rconf-def*)
apply *auto*
done

lemma *rconf-In2-LVar [simp]*:

$va = \text{LVar } n \implies$
 $G, L, s \vdash \text{In2 } va \succ \text{In2 } vf :: \preceq \text{Inl } T$
 $= ((\text{fst } vf = \text{the } (\text{locals } s \ n)) \wedge$
 $(\text{locals } s \ n \neq \text{None} \rightarrow G, s \vdash \text{fst } vf :: \preceq T) \wedge s \leq | \text{snd } vf \preceq T :: \preceq (G, L))$
apply (*unfold rconf-def*)
by *simp*

lemma *rconf-In3 [simp]*:

$G, L, s \vdash \text{In3 } es \succ \text{In3 } vs :: \preceq \text{Inr } Ts = \text{list-all2 } (\lambda v \ T. G, s \vdash v :: \preceq T) \ vs \ Ts$
apply (*unfold rconf-def*)
apply (*simp (no-asm)*)
done

fits and conf

lemma *conf-fits*: $G, s \vdash v :: \preceq T \implies G, s \vdash v \text{ fits } T$

apply (*unfold fits-def*)
apply *clarify*
apply (*erule contrapos-np, simp (no-asm-use)*)
apply (*drule conf-RefTD*)
apply *auto*

done

lemma *fits-conf*:

$\llbracket G, s \vdash v :: \preceq T; G \vdash T \preceq ? T'; G, s \vdash v \text{ fits } T'; \text{ws-prog } G \rrbracket \implies G, s \vdash v :: \preceq T'$
apply (auto dest!: fitsD cast-PrimT2 cast-RefT2)
apply (force dest: conf-RefTD intro: conf-AddrI)
 done

lemma *fits-Array*:

$\llbracket G, s \vdash v :: \preceq T; G \vdash T'.[] \preceq T.[]; G, s \vdash v \text{ fits } T'; \text{ws-prog } G \rrbracket \implies G, s \vdash v :: \preceq T'$
apply (auto dest!: fitsD widen-ArrayPrimT widen-ArrayRefT)
apply (force dest: conf-RefTD intro: conf-AddrI)
 done

gext

lemma *halloc-gext*: $\bigwedge s1\ s2. G \vdash s1 \text{ --halloc } oi \succ a \rightarrow s2 \implies \text{snd } s1 \leq | \text{snd } s2$
apply (simp (no-asm-simp) only: split-tupled-all)
apply (erule halloc.induct)
apply (auto dest!: new-AddrD)
 done

lemma *sxalloc-gext*: $\bigwedge s1\ s2. G \vdash s1 \text{ --sxalloc } \rightarrow s2 \implies \text{snd } s1 \leq | \text{snd } s2$
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)]:

$G \vdash s \text{ --t } \rightarrow (w, s') \implies \text{snd } s \leq | \text{snd } s' \wedge (\text{case } w \text{ of}$
 $\text{In1 } v \Rightarrow \text{True}$
 $| \text{In2 } vf \Rightarrow \text{normal } s \longrightarrow (\forall v\ x\ s. s \leq | \text{snd } (\text{assign } (\text{snd } vf)\ v\ (x, s)))$
 $| \text{In3 } vs \Rightarrow \text{True})$

apply (erule eval-induct)

prefer 26

apply (case-tac initd C (globs s0), clarsimp, erule thin-rl)
apply (auto del: conjI dest!: not-initdD 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*:

$G \vdash \text{Norm } s1 \text{ --e } \succ vf \rightarrow s2 \implies s \leq | \text{snd } (\text{assign } (\text{snd } vf)\ v\ (x, s))$
apply (drule eval-gext-lemma [THEN conjunct2])
apply auto
 done

lemmas *eval-gext* = eval-gext-lemma [THEN conjunct1]

lemma *eval-gext'*: $G \vdash (x1, s1) \text{ --t } \rightarrow (w, (x2, s2)) \implies s1 \leq | s2$

apply (drule eval-gext)

apply auto

done

lemma *init-yields-initd*: $G \vdash \text{Norm } s1 \text{ --Init } C \rightarrow s2 \implies \text{initd } C\ s2$

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:
   $\llbracket \text{wf-prog } G; \text{ is-type } G \text{ (obj-ty obj)} \rrbracket \implies \text{is-class } G \text{ (obj-class obj)}$ 
apply (case-tac tag obj)
apply (auto simp add: obj-ty-def obj-class-def)
done

```

```

lemma oconf-init-obj:
   $\llbracket \text{wf-prog } G; \text{ (case } r \text{ of Heap } a \Rightarrow \text{is-type } G \text{ (obj-ty obj)} \mid \text{Stat } C \Rightarrow \text{is-class } G \text{ } C) \rrbracket \implies G, \text{s} \vdash \text{obj} \llbracket \text{values} := \text{init-vals (var-tys } G \text{ (tag obj) } r) \rrbracket :: \preceq \sqrt{r}$ 
apply (auto intro!: oconf-init-obj-lemma unique-fields)
done

```

```

lemma conforms-newG:  $\llbracket \text{globs } s \text{ oref} = \text{None}; (x, s) :: \preceq (G, L); \text{wf-prog } G; \text{ case oref of Heap } a \Rightarrow \text{is-type } G \text{ (obj-ty (tag=oi, values=vs))} \mid \text{Stat } C \Rightarrow \text{is-class } G \text{ } C \rrbracket \implies (x, \text{init-obj } G \text{ oi oref } s) :: \preceq (G, L)$ 
apply (unfold init-obj-def)
apply (auto elim!: conforms-gupd dest!: oconf-init-obj)
done

```

```

lemma conforms-init-class-obj:
   $\llbracket (x, s) :: \preceq (G, L); \text{wf-prog } G; \text{class } G \text{ } C = \text{Some } y; \neg \text{inited } C \text{ (globs } s) \rrbracket \implies (x, \text{init-class-obj } G \text{ } C \text{ } s) :: \preceq (G, L)$ 
apply (rule not-initedD [THEN conforms-newG])
apply (auto)
done

```

```

lemma fst-init-lvars[simp]:
   $\text{fst (init-lvars } G \text{ } C \text{ sig (invmode } m \text{ } e) \text{ } a' \text{ pvs (x,s))} = \text{(if is-static } m \text{ then } x \text{ else (np } a') \text{ } x)$ 
apply (simp (no-asm) add: init-lvars-def2)
done

```

```

lemma halloc-conforms:  $\bigwedge s1. \llbracket G \vdash s1 \text{ -halloc oi} \triangleright a \rightarrow s2; \text{wf-prog } G; s1 :: \preceq (G, L); \text{is-type } G \text{ (obj-ty (tag=oi, values=fs))} \rrbracket \implies s2 :: \preceq (G, L)$ 
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:
   $\bigwedge s1. \llbracket G \vdash s1 \text{ -halloc oi} \triangleright a \rightarrow (x, s); \text{wf-prog } G; s1 :: \preceq (G, L); T = \text{obj-ty (tag=oi, values=fs)}; \text{is-type } G \text{ } T \rrbracket \implies (x, s) :: \preceq (G, L) \wedge (x = \text{None} \longrightarrow G, \text{s} \vdash \text{Addr } a :: \preceq T)$ 

```

```

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*:

```

 $\bigwedge s1\ s2. \llbracket G \vdash s1 \rightarrow sxalloc \rightarrow s2; wf\text{-}prog\ G \rrbracket \implies$ 
  case fst s1 of
    None  $\Rightarrow s2 = s1$ 
  | Some abr  $\Rightarrow$  (case abr of
    Xcpt x  $\Rightarrow (\exists a. fst\ s2 = Some(Xcpt (Loc\ a)) \wedge$ 
       $(\forall L. s1 :: \preceq(G, L) \longrightarrow s2 :: \preceq(G, L)))$ 
    | Jump j  $\Rightarrow s2 = s1$ 
    | Error e  $\Rightarrow s2 = s1$ )
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  $\implies (\text{prg}=G, \text{cls}=C, \text{lcl}=L) \vdash \text{init-comp-ty}\ T :: \checkmark$ 
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]  $\Rightarrow$  bool ( $\hookrightarrow \vdash \rightarrow \rightarrow \preceq \rightarrow$  [71, 71, 71, 71] 70)
where
   $G \vdash \text{mode} \rightarrow D \preceq t = (\text{mode} = \text{IntVir} \longrightarrow \text{is-class}\ G\ D \wedge$ 
     $(\text{if } (\exists T. t = \text{ArrayT}\ T) \text{ then } D = \text{Object} \text{ else } G \vdash \text{Class}\ D \preceq \text{RefT}\ t))$ 

```

lemma *DynT-propI*:

```

 $\llbracket (x, s) :: \preceq(G, L); G, s \vdash a' :: \preceq \text{RefT}\ \text{statT}; wf\text{-}prog\ G; \text{mode} = \text{IntVir} \longrightarrow a' \neq \text{Null} \rrbracket$ 
 $\implies G \vdash \text{mode} \rightarrow \text{invocation-class}\ \text{mode}\ s\ a'\ \text{statT} \preceq \text{statT}$ 

```

proof (unfold DynT-prop-def)

```

assume state-conform:  $(x, s) :: \preceq(G, L)$ 
and statT-a':  $G, s \vdash a' :: \preceq \text{RefT}\ \text{statT}$ 
and wf:  $wf\text{-}prog\ G$ 
and mode:  $\text{mode} = \text{IntVir} \longrightarrow a' \neq \text{Null}$ 
let ?invCls = (invocation-class mode s a' statT)
let ?IntVir = mode = IntVir
let ?Concl =  $\lambda \text{invCls}. \text{is-class}\ G\ \text{invCls} \wedge$ 
   $(\text{if } \exists T. \text{statT} = \text{ArrayT}\ T$ 
     $\text{then invCls} = \text{Object}$ 
     $\text{else } G \vdash \text{Class}\ \text{invCls} \preceq \text{RefT}\ \text{statT})$ 

```

show $?IntVir \longrightarrow ?Concl\ ?invCls$

proof

```

assume modeIntVir: ?IntVir
with mode have not-Null:  $a' \neq \text{Null} \dots$ 
from statT-a' not-Null state-conform

```



```

obtain a obj
  where obj-props:  $a' = \text{Addr } a \text{ globs } s \text{ (Inl } a) = \text{Some } obj$ 
                $G \vdash \text{obj-ty } obj \preceq \text{RefT statT is-type } G \text{ (obj-ty } obj)$ 
  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:
 $\llbracket wf\text{-prog } G; \text{statT} \neq \text{NullT};$ 
 $(\forall \text{statC}. \text{statT} = \text{ClassT statC} \longrightarrow \text{is-class } G \text{ statC});$ 
 $(\forall I. \text{statT} = \text{IfaceT } I \longrightarrow \text{is-iface } G \text{ } I \wedge \text{mode} \neq \text{SuperM});$ 
 $(\forall T. \text{statT} = \text{ArrayT } T \longrightarrow \text{mode} \neq \text{SuperM});$ 
 $G \vdash \text{mode} \rightarrow \text{invocation-class mode } s \text{ } a' \text{ statT} \preceq \text{statT};$ 
 $\text{dynlookup } G \text{ statT (invocation-class mode } s \text{ } a' \text{ statT) sig} = \text{Some } m \rrbracket$ 
 $\implies \text{methd } G \text{ (invocation-declclass } G \text{ mode } s \text{ } a' \text{ statT sig) sig} = \text{Some } m$ 
proof –
  assume  $wf: wf\text{-prog } G$ 
  and not-NullT:  $\text{statT} \neq \text{NullT}$ 
  and statC-prop:  $(\forall \text{statC}. \text{statT} = \text{ClassT statC} \longrightarrow \text{is-class } G \text{ statC})$ 
  and statI-prop:  $(\forall I. \text{statT} = \text{IfaceT } I \longrightarrow \text{is-iface } G \text{ } I \wedge \text{mode} \neq \text{SuperM})$ 
  and statA-prop:  $(\forall T. \text{statT} = \text{ArrayT } T \longrightarrow \text{mode} \neq \text{SuperM})$ 
  and invC-prop:  $G \vdash \text{mode} \rightarrow \text{invocation-class mode } s \text{ } a' \text{ statT} \preceq \text{statT}$ 
  and dynlookup:  $\text{dynlookup } G \text{ statT (invocation-class mode } s \text{ } a' \text{ statT) sig} = \text{Some } m$ 

show ?thesis
proof (cases statT)
  case NullT
    with not-NullT show ?thesis by simp
  next
    case IfaceT
    with statI-prop obtain I
    where statI:  $\text{statT} = \text{IfaceT } I$  and
           is-iface:  $\text{is-iface } G \text{ } I$  and
           not-SuperM:  $\text{mode} \neq \text{SuperM}$  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*:

$\llbracket G \vdash \text{invmode } sm \ e \rightarrow \text{invC} \preceq \text{statT};$
 $\text{wf-prog } G; (\text{prg} = G, \text{cls} = C, \text{lcl} = L) \vdash e :: - \text{RefT } \text{statT};$
 $(\text{statDeclT}, sm) \in \text{mheads } G \ C \ \text{statT} \ \text{sig};$

```

invC = invocation-class (invmode sm e) s a' statT;
declC = invocation-declclass G (invmode sm e) s a' statT sig
 $\mathbb{I} \Rightarrow$ 
 $\exists dm.$ 
methd G declC sig = Some dm  $\wedge$  dynlookup G statT invC sig = Some dm  $\wedge$ 
 $G \vdash \text{resTy } (mthd dm) \preceq_{\text{resTy}} sm \wedge$ 
wf-mdecl G declC (sig, mthd dm)  $\wedge$ 
declC = declclass dm  $\wedge$ 
is-static dm = is-static sm  $\wedge$ 
is-class G invC  $\wedge$  is-class G declC  $\wedge$   $G \vdash \text{invC} \preceq_C \text{declC} \wedge$ 
(if invmode sm e = IntVir
  then ( $\forall \text{statC}. \text{statT} = \text{ClassT statC} \longrightarrow G \vdash \text{invC} \preceq_C \text{statC}$ )
  else ( $(\exists \text{statC}. \text{statT} = \text{ClassT statC} \wedge G \vdash \text{statC} \preceq_C \text{declC})$ 
     $\vee (\forall \text{statC}. \text{statT} \neq \text{ClassT statC} \wedge \text{declC} = \text{Object})) \wedge$ 
    statDeclT = ClassT (declclass dm))
proof -
assume invC-prop:  $G \vdash \text{invmode sm e} \rightarrow \text{invC} \preceq \text{statT}$ 
and wf: wf-prog G
and wt-e: ( $\text{prg} = G, \text{cls} = C, \text{lcl} = L$ )  $\vdash e :: -\text{RefT statT}$ 
and sm: ( $\text{statDeclT}, \text{sm}$ )  $\in \text{mheads } G \ C \ \text{statT sig}$ 
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  $\neq \text{NullT}$  by auto
from type-statT
have wf-C: ( $\forall \text{statC}. \text{statT} = \text{ClassT statC} \longrightarrow \text{is-class } G \ \text{statC}$ )
by (auto)
from type-statT wt-e
have wf-I: ( $\forall I. \text{statT} = \text{IfaceT } I \longrightarrow \text{is-iface } G \ I \wedge$ 
  invmode sm e  $\neq \text{SuperM}$ )
by (auto dest: invocationTypeExpr-noClassD)
from wt-e
have wf-A: ( $\forall T. \text{statT} = \text{ArrayT } T \longrightarrow \text{invmode sm e} \neq \text{SuperM}$ )
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  $\wedge$ 
  (if ( $\exists T. \text{statT} = \text{ArrayT } T$ ) then invC = Object
    else  $G \vdash \text{Class invC} \preceq \text{RefT statT}$ )
by (auto simp add: DynT-prop-def)
from True
have  $\neg$  is-static sm
by (simp add: invmode-IntVir-eq member-is-static-simp)
with invC-prop' not-Null
have G, statT  $\vdash$  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:  $G \vdash \text{resTy } (mthd dm) \preceq_{\text{resTy}} sm$  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:  $G \vdash \text{invC} \preceq_C \text{declC} \wedge \text{is-class } G \text{ declC}$ 
  by (auto dest: dynlookup-declC)
from wf dm' declC-prop declC'
have wf-dm: wf-mdecl G declC (sig,(methd dm))
  by (auto dest: methd-wf-mdecl)
from invC-prop'
have statC-prop:  $(\forall \text{statC}. \text{statT} = \text{ClassT statC} \longrightarrow G \vdash \text{invC} \preceq_C \text{statC})$ 
  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':  $\text{is-class } G \text{ invC} \wedge$ 
   $((\exists \text{statC}. \text{statT} = \text{ClassT statC} \wedge \text{invC} = \text{statC}) \vee$ 
   $(\forall \text{statC}. \text{statT} \neq \text{ClassT statC} \wedge \text{invC} = \text{Object}))$ 
  by (case-tac statT) (auto simp add: invocation-class-def
    split: inv-mode.splits)

with not-Null wf
have dynlookup-static:  $\text{dynlookup } G \text{ statT invC sig} = \text{methd } G \text{ invC sig}$ 
  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:  $\text{methd } G \text{ invC sig} = \text{Some dm}$  and
  eq-declC-sm-dm:  $\text{statDeclT} = \text{ClassT (declclass dm)}$  and
  eq-mheads:  $\text{sm} = \text{mhead (methd dm)}$ 
  by - (drule static-mheadsD, (force dest: accmethd-SomeD)+)
then have static:  $\text{is-static dm} = \text{is-static sm}$  by auto
with declC invC dynlookup-static dm
have declC':  $\text{declC} = (\text{declclass dm})$ 
  by (auto simp add: invocation-declclass-def)
from invC-prop' wf declC' dm
have dm':  $\text{methd } G \text{ declC sig} = \text{Some dm}$ 
  by (auto intro: methd-declclass)
from dynlookup-static dm
have dm'':  $\text{dynlookup } G \text{ statT invC sig} = \text{Some dm}$ 
  by simp
from wf dm invC-prop' declC' type-statT
have declC-prop:  $G \vdash \text{invC} \preceq_C \text{declC} \wedge \text{is-class } G \text{ declC}$ 
  by (auto dest: methd-declC)
then have declC-prop1:  $\text{invC} = \text{Object} \longrightarrow \text{declC} = \text{Object}$  by auto
from wf dm' declC-prop declC'
have wf-dm: wf-mdecl G declC (sig,(methd dm))
  by (auto dest: methd-wf-mdecl)
from invC-prop' declC-prop declC-prop1
have statC-prop:  $(\exists \text{statC}. \text{statT} = \text{ClassT statC} \wedge G \vdash \text{statC} \preceq_C \text{declC})$ 
   $\vee (\forall \text{statC}. \text{statT} \neq \text{ClassT statC} \wedge \text{declC} = \text{Object})$ 
  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*: $G \vdash \text{mode} \rightarrow \text{invC} \preceq \text{statT}$
and *wf*: *wf-prog* *G*
and *wt-e*: $(\text{prg} = G, \text{cls} = C, \text{lcl} = L) \vdash e :: -\text{RefT } \text{statT}$
and *mheads*: $(\text{statDeclT}, \text{sm}) \in \text{mheads } G \ C \ \text{statT } \text{sig}$
and *mode*: $\text{mode} = \text{invmode } \text{sm } e$
and *invC*: $\text{invC} = \text{invocation-class } \text{mode } s \ a' \ \text{statT}$
and *declC*: $\text{declC} = \text{invocation-declclass } G \ \text{mode } s \ a' \ \text{statT } \text{sig}$
and *dm*: $\bigwedge \text{dm}. \llbracket \text{methd } G \ \text{declC } \text{sig} = \text{Some } \text{dm};$
 $\text{dynlookup } G \ \text{statT } \text{invC } \text{sig} = \text{Some } \text{dm};$
 $G \vdash \text{resTy } (\text{methd } \text{dm}) \preceq \text{resTy } \text{sm};$
 $\text{wf-mdecl } G \ \text{declC } (\text{sig}, \text{methd } \text{dm});$
 $\text{declC} = \text{declclass } \text{dm};$
 $\text{is-static } \text{dm} = \text{is-static } \text{sm};$
 $\text{is-class } G \ \text{invC}; \text{is-class } G \ \text{declC}; G \vdash \text{invC} \preceq_C \text{declC};$
 $(\text{if } \text{invmode } \text{sm } e = \text{IntVir}$
 $\text{then } (\forall \text{statC}. \text{statT} = \text{ClassT } \text{statC} \longrightarrow G \vdash \text{invC} \preceq_C \text{statC})$
 $\text{else } ((\exists \text{statC}. \text{statT} = \text{ClassT } \text{statC} \wedge G \vdash \text{statC} \preceq_C \text{declC})$
 $\vee (\forall \text{statC}. \text{statT} \neq \text{ClassT } \text{statC} \wedge \text{declC} = \text{Object})) \wedge$
 $\text{statDeclT} = \text{ClassT } (\text{declclass } \text{dm})) \rrbracket \implies P$
shows *P*
proof —
from *invC-compatible* **mode** **have** $G \vdash \text{invmode } \text{sm } e \rightarrow \text{invC} \preceq \text{statT}$ **by** *simp*
moreover **note** *wf wt-e mheads*
moreover **from** *invC* **mode**
have $\text{invC} = \text{invocation-class } (\text{invmode } \text{sm } e) \ s \ a' \ \text{statT}$ **by** *simp*
moreover **from** *declC* **mode**
have $\text{declC} = \text{invocation-declclass } G \ (\text{invmode } \text{sm } e) \ s \ a' \ \text{statT } \text{sig}$ **by** *simp*
ultimately show *?thesis*
by (*rule DynT-mheadsD [THEN exE,rule-format]*)
(elim conjE,rule dm)
qed

lemma *DynT-conf*: $\llbracket G \vdash \text{invocation-class } \text{mode } s \ a' \ \text{statT} \preceq_C \text{declC}; \text{wf-prog } G;$
 $\text{isrtype } G \ (\text{statT});$
 $G, s \vdash a' :: \preceq \text{RefT } \text{statT}; \text{mode} = \text{IntVir} \longrightarrow a' \neq \text{Null};$
 $\text{mode} \neq \text{IntVir} \longrightarrow ((\exists \text{statC}. \text{statT} = \text{ClassT } \text{statC} \wedge G \vdash \text{statC} \preceq_C \text{declC})$
 $\vee (\forall \text{statC}. \text{statT} \neq \text{ClassT } \text{statC} \wedge \text{declC} = \text{Object})) \rrbracket$
 $\implies G, s \vdash a' :: \preceq \text{Class } \text{declC}$
apply (*case-tac mode = IntVir*)
apply (*drule conf-RefTD*)
apply (*force intro!: conf-AddrI*
 $\text{simp add: obj-class-def split: obj-tag.split-asm}$)
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*:

$\llbracket G \vdash \text{Norm } s0 \text{ --var}=\succ(w, f) \rightarrow \text{Norm } s1; G \vdash \text{Norm } s1 \text{ --e--}\succ v \rightarrow \text{Norm } s2;$
 $G, s2 \vdash v :: \preceq eT; s1 \leq s2 \longrightarrow \text{assign } f \ v \ (\text{Norm } s2) :: \preceq (G, L) \rrbracket$
 $\implies \text{assign } f \ v \ (\text{Norm } s2) :: \preceq (G, L) \wedge$
 $(\text{normal } (\text{assign } f \ v \ (\text{Norm } s2))) \longrightarrow G, \text{store } (\text{assign } f \ v \ (\text{Norm } s2)) \vdash v :: \preceq eT$
apply (*drule-tac x = None and s = s2 and v = v in evar-gezt-f*)

apply (*drule eval-gext'*, *clarsimp*)
apply (*erule conf-gext*)
apply *simp*
done

lemma *Throw-lemma*: $\llbracket G \vdash tn \preceq_C SXcpt \text{ Throwable}; wf\text{-prog } G; (x1, s1) :: \preceq (G, L);$
 $x1 = None \longrightarrow G, s1 \vdash a' :: \preceq Class \text{ } tn \rrbracket \Longrightarrow (throw \text{ } a' \text{ } x1, s1) :: \preceq (G, L)$
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*: $\llbracket G \vdash obj\text{-ty } (the \text{ } (globs \text{ } s1' \text{ } (Heap \text{ } a))) \preceq Class \text{ } tn;$
 $(Some \text{ } (Xcpt \text{ } (Loc \text{ } a)), s1') :: \preceq (G, L); wf\text{-prog } G \rrbracket$
 $\Longrightarrow Norm \text{ } (lupd(vn \mapsto Addr \text{ } a) \text{ } s1') :: \preceq (G, L(vn \mapsto Class \text{ } tn))$
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*:
 $\llbracket G \vdash Norm \text{ } s1 - c2 \rightarrow (x2, s2); wf\text{-prog } G; (Some \text{ } a, s1) :: \preceq (G, L); (x2, s2) :: \preceq (G, L);$
 $dom \text{ } (locals \text{ } s1) \subseteq dom \text{ } (locals \text{ } s2) \rrbracket$
 $\Longrightarrow (abrupt\text{-if } True \text{ } (Some \text{ } a) \text{ } x2, s2) :: \preceq (G, L)$
apply (*auto elim: eval-gext' conforms-xgext split: split-abrupt-if*)
done

lemma *FVar-lemma1*:
 $\llbracket table\text{-of } (DeclConcepts.fields \text{ } G \text{ } statC) \text{ } (fn, statDeclC) = Some \text{ } f ;$
 $x2 = None \longrightarrow G, s2 \vdash a :: \preceq Class \text{ } statC; wf\text{-prog } G; G \vdash statC \preceq_C statDeclC;$
 $statDeclC \neq Object;$
 $class \text{ } G \text{ } statDeclC = Some \text{ } y; (x2, s2) :: \preceq (G, L); s1 \leq s2;$
 $inited \text{ } statDeclC \text{ } (globs \text{ } s1);$
 $(if \text{ } static \text{ } f \text{ } then \text{ } id \text{ } else \text{ } np \text{ } a) \text{ } x2 = None \rrbracket$
 \Longrightarrow
 $\exists obj. globs \text{ } s2 \text{ } (if \text{ } static \text{ } f \text{ } then \text{ } Inr \text{ } statDeclC \text{ } else \text{ } Inl \text{ } (the\text{-Addr } a))$
 $= Some \text{ } obj \wedge$
 $var\text{-tys } G \text{ } (tag \text{ } obj) \text{ } (if \text{ } static \text{ } f \text{ } then \text{ } Inr \text{ } statDeclC \text{ } else \text{ } Inl \text{ } (the\text{-Addr } a))$
 $(Inl(fn, statDeclC)) = Some \text{ } (type \text{ } f)$
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*
 $\Longrightarrow error\text{-free}$

```

      (assign
        (λv. 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 |> Simplifier.del-loop 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 ⊆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 |> Simplifier.add-loop (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: $\exists t$ vs. obj = ($\text{tag}=t, \text{values}=vs$)
by (cases obj) auto

lemma AVar-lemma2: error-free state
 \implies error-free
(assign
(λv (x, s').
((raise-if ($\neg G, s \vdash v$ fits T) ArrStore) x,
upd-gobj (Inl a) (Inr (the-Intg i)) v s'))
w state)

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: $\llbracket \text{wf-prog } G; G \vdash (x1, s1) -e2-\triangleright i \rightarrow (x2, s2);$
 $((v,f), \text{Norm } s2') = \text{avar } G \text{ i a } (x2, s2); x1 = \text{None} \longrightarrow G, s1 \vdash a :: \preceq Ta.[];$
 $(x2, s2) :: \preceq (G, L); s1 \leq |s2| \implies G, s2 \vdash v :: \preceq Ta \wedge s2' \leq |f \preceq Ta :: \preceq (G, L)$
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 $\exists t$ vs. obj = ($\text{tag}=t, \text{values}=vs$))
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: $\llbracket \text{wf-prog } G;$
 $\text{wf-mhead } G \text{ P sig mh};$
 $\text{list-all2 } (\text{conf } G \text{ s}) \text{ pvs } pTsa; G \vdash pTsa[\preceq](\text{parTs sig}) \implies$
 $G, s \vdash \text{Map.empty } (\text{pars mh}[\mapsto] \text{pvs})$
 $[\sim :: \preceq](\text{table-of lvars})(\text{pars mh}[\mapsto] \text{parTs sig})$
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]:
 $G, s \vdash (\text{case-lname } l1 \text{ } l2)[:: \preceq](\text{case-lname } L1 \text{ } L2)$


```

=
(G, s ⊢ l1 [::⊆] L1 ∧ G, s ⊢ (λx::unit . l2) [::⊆] (λx::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 ⊢ (λx::unit . l2) [~::⊆] (λx::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 |> Simplifier.del-loop 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
        VName 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 |> Simplifier.add-loop (split-all-tac, split-all-tac))⟩

```

2 accessibility

theorem *dynamic-field-access-ok*:

assumes *wf*: wf-prog *G* **and**

not-Null: $\neg \text{stat} \longrightarrow a \neq \text{Null}$ **and**

conform-a: $G, (\text{store } s) \vdash a :: \preceq \text{Class statC}$ **and**

conform-s: $s :: \preceq (G, L)$ **and**

normal-s: normal *s* **and**

wt-e: $(\text{prg} = G, \text{cls} = \text{accC}, \text{lcl} = L) \vdash e :: - \text{Class statC}$ **and**

f: $\text{accfield } G \text{ accC statC fn} = \text{Some } f$ **and**

dynC: if *stat* then $\text{dynC} = \text{declclass } f$

else $\text{dynC} = \text{obj-class (lookup-obj (store } s) a)$ **and**

stat: if *stat* then $(\text{is-static } f)$ else $(\neg \text{is-static } f)$

shows $\text{table-of (DeclConcepts.fields } G \text{ dynC) (fn, declclass } f) = \text{Some (fld } f) \wedge$

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

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:  $G \vdash \text{dynC} \preceq_C \text{statC}$  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  $G \vdash \text{Field } \text{fn } f \text{ in } \text{dynC} \text{ dyn-accessible-from } \text{accC}$ 
  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: ( $\langle \text{prg} = G, \text{cls} = \text{accC}, \text{lcl} = L \rangle \vdash e :: \text{Class } \text{statC}$ ) and
  eval-init:  $G \vdash \text{Norm } s0 \text{ --Init } \text{statDeclC} \rightarrow s1$  and
  eval-e:  $G \vdash s1 \text{ --e--} a \rightarrow s2$  and
  conf-s2:  $s2 :: \preceq(G, L)$  and
  conf-a:  $\text{normal } s2 \implies G, \text{store } s2 \vdash a :: \preceq \text{Class } \text{statC}$  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  $s2' = \text{store } s2$ 
  by (cases s2) (simp add: fvar-def2)
with fvar conf-s2
have conf-s2':  $s2' :: \preceq(G, L)$ 
  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: initd-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:  $\neg$  (is-static f)  $\longrightarrow$  a $\neq$ 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':  $G, \text{store } s2 \vdash a :: \preceq \text{Class } \text{statC}$ 
    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!: initdD
        simp add: check-field-access-def Let-def)

qed
qed

```

lemma call-access-ok:

```

assumes invC-prop:  $G \vdash \text{invmode } \text{statM } e \rightarrow \text{invC} \preceq \text{statT}$ 
  and wf: wf-prog G
  and wt-e:  $(\text{prg} = G, \text{cls} = C, \text{lcl} = L) \vdash e :: -\text{RefT } \text{statT}$ 
  and statM:  $(\text{statDeclT}, \text{statM}) \in \text{mheads } G \text{ accC } \text{statT } \text{sig}$ 
  and invC:  $\text{invC} = \text{invocation-class } (\text{invmode } \text{statM } e) \text{ s a } \text{statT}$ 
shows  $\exists \text{ dynM. dynlookup } G \text{ statT } \text{invC } \text{sig} = \text{Some } \text{dynM} \wedge$ 
   $G \vdash \text{Methd } \text{sig } \text{dynM} \text{ in } \text{invC } \text{dyn-accessible-from } \text{accC}$ 
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:  $\text{statT} \neq \text{NullT}$  by auto
  from type-statT wt-e
  have wf-I:  $(\forall I. \text{statT} = \text{IfaceT } I \longrightarrow \text{is-iface } G \text{ } I \wedge$ 
     $\text{invmode } \text{statM } e \neq \text{SuperM})$ 
    by (auto dest: invocationTypeExpr-noClassD)
  from wt-e
  have wf-A:  $(\forall T. \text{statT} = \text{ArrayT } T \longrightarrow \text{invmode } \text{statM } e \neq \text{SuperM})$ 
    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  $\wedge$ 
     $(\text{if } (\exists T. \text{statT} = \text{ArrayT } T) \text{ then } \text{invC} = \text{Object}$ 
     $\text{else } G \vdash \text{Class } \text{invC} \preceq \text{RefT } \text{statT})$ 
    by (auto simp add: DynT-prop-def)
  with True not-Null
  have G, statT  $\vdash \text{invC } \text{valid-lookup-clc-for is-static } \text{statM}$ 
    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  $\wedge$ 
    (( $\exists$  statC. statT=ClassT statC  $\wedge$  invC=statC)  $\vee$ 
     ( $\forall$  statC. statT $\neq$ ClassT statC  $\wedge$  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  $\vdash$  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:  $G \vdash s1 \multimap \text{args} \Rightarrow \text{vs} \rightarrow s2$  and
  wt-e: ( $\text{prg} = G, \text{cls} = \text{accC}, \text{lcl} = L$ )  $\vdash e :: \neg(\text{RefT statT})$  and
  statM:  $\text{max-spec } G \text{ accC statT } (\text{name} = mn, \text{parTs} = pTs)$ 
    =  $\{((\text{statDeclT}, \text{statM}), pTs')\}$  and
  conf-s2:  $s2 :: \preceq(G, L)$  and
  conf-a:  $\text{normal } s1 \implies G, \text{store } s1 \vdash a :: \preceq \text{RefT statT}$  and
  invProp:  $\text{normal } s3 \implies$ 
     $G \vdash \text{invmode statM } e \rightarrow \text{invC} \preceq \text{statT}$  and
  s3:  $s3 = \text{init-lvars } G \text{ invDeclC } (\text{name} = mn, \text{parTs} = pTs')$ 
    ( $\text{invmode statM } e$ ) a vs s2 and
  invC:  $\text{invC} = \text{invocation-class } (\text{invmode statM } e) (\text{store } s2) a \text{ statT}$  and
  invDeclC:  $\text{invDeclC} = \text{invocation-declclass } G (\text{invmode statM } e) (\text{store } s2)$ 
    a statT ( $\text{name} = mn, \text{parTs} = pTs')$  and
  wf: wf-prog G
shows check-method-access G accC statT ( $\text{invmode statM } e$ ) ( $\text{name}=mn, \text{parTs}=pTs'$ ) 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:  $G, \text{store } s2 \vdash a :: \preceq \text{RefT statT}$ 
  by (auto dest: eval-gext intro: conf-gext)
show ?thesis
proof (cases a=Null  $\longrightarrow$  (is-static statM))
  case False
  then obtain  $\neg \text{is-static statM } a = \text{Null}$ 
    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
     $\text{statM}': (\text{statDeclT}, \text{statM}) \in \text{mheads } G \text{ accC statT } (\text{name} = \text{mn}, \text{parTs} = \text{pTs}')$ 
    by (blast dest: max-spec2mheads)
  from True normal-s2 s3
  have normal s3
    by (auto simp add: init-lvars-def2)
  then have  $G \vdash \text{invmode statM } e \rightarrow \text{invC} \preceq \text{statT}$ 
    by (rule invProp)
  with wt-e statM' wf invC
  obtain dynM where
     $\text{dynM}: \text{dynlookup } G \text{ statT invC } (\text{name} = \text{mn}, \text{parTs} = \text{pTs}') = \text{Some dynM}$  and
     $\text{acc-dynM}: G \vdash \text{Methd } (\text{name} = \text{mn}, \text{parTs} = \text{pTs}') \text{ dynM}$ 
     $\text{in invC dyn-accessible-from accC}$ 
    by (force dest!: call-access-ok)
  moreover
  from s3 invC
  have  $\text{invC}': \text{invC} = (\text{invocation-class } (\text{invmode statM } e) (\text{store } s3) a \text{ statT})$ 
    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*:

$$\bigwedge \text{tab } qs. \text{length } ps = \text{length } qs \implies \text{tab}(ps[\mapsto]qs@zs) = \text{tab}(ps[\mapsto]qs)$$

proof (*induct ps*)

case *Nil* **thus** *?case* **by** *simp*

next

case (*Cons p ps tab qs*)

from $\langle \text{length } (p \# ps) = \text{length } qs \rangle$

obtain *q qs'* **where** $qs = q \# qs'$ **and** *eq-length*: $\text{length } ps = \text{length } qs'$

by (*cases qs*) *auto*

from *eq-length* **have** $(\text{tab}(p \mapsto q)) (ps[\mapsto]qs'@zs) = (\text{tab}(p \mapsto q)) (ps[\mapsto]qs')$

by (*rule Cons.hyps*)

with *qs* **show** *?case*

by *simp*

qed

lemma *map-upds-upd-eq-length-simp*:

$$\bigwedge \text{tab } qs \ x \ y. \text{length } ps = \text{length } qs \\ \implies \text{tab}(ps[\mapsto]qs, x \mapsto y) = \text{tab}(ps@[x][\mapsto]qs@[y])$$

```

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@[x][↦]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*:

$\bigwedge \text{tab } qs. \text{length } ps = \text{length } qs$
 $\implies \text{tab}(ps[\mapsto]qs, xs[\mapsto]ys) = \text{tab}(ps@xs[\mapsto]qs@ys)$

proof (*induct* ps)

case Nil **thus** $?case$ **by** $simp$

next

case ($Cons\ p\ ps\ \text{tab}\ qs$)

then obtain $q\ qs'$ **where** $qs = q\#qs'$ **and** *eq-length*: $\text{length } ps = \text{length } qs'$

by (*cases* qs) *auto*

from *eq-length*

have $\text{tab}(p\mapsto q, ps[\mapsto]qs', xs[\mapsto]ys) = \text{tab}(p\mapsto q, ps\ @\ xs[\mapsto](qs'\ @\ ys))$

by (*rule* $Cons.hyps$)

with qs

show $?case$ **by** $simp$

qed

lemma *map-upd-cut-irrelevant*:

$\llbracket (\text{tab}(x\mapsto y))\ vn = \text{Some}\ el; (\text{tab}'(x\mapsto y))\ vn = \text{None} \rrbracket$

$\implies \text{tab}\ vn = \text{Some}\ el$

by (*cases* $\text{tab}'\ vn = \text{None}$) (*simp add: fun-upd-def*) $+$

lemma *map-upd-Some-expand*:

$\llbracket \text{tab}\ vn = \text{Some}\ z \rrbracket$

$\implies \exists z. (\text{tab}(x\mapsto y))\ vn = \text{Some}\ z$

by (*simp add: fun-upd-def*)

lemma *map-upds-Some-expand*:

$\bigwedge \text{tab}\ ys\ z. \llbracket \text{tab}\ vn = \text{Some}\ z \rrbracket$

$\implies \exists z. (\text{tab}(xs[\mapsto]ys))\ vn = \text{Some}\ z$

proof (*induct* xs)

case Nil **thus** $?case$ **by** $simp$

next

case ($Cons\ x\ xs\ \text{tab}\ ys\ z$)

note $z = \langle \text{tab}\ vn = \text{Some}\ z \rangle$

show $?case$

proof (*cases* ys)

case Nil

with z **show** $?thesis$ **by** $simp$

next

case ($Cons\ y\ ys'$)

note $ys = \langle ys = y\#ys' \rangle$

from z **obtain** z' **where** $(\text{tab}(x\mapsto y))\ vn = \text{Some}\ z'$

by (*rule* *map-upd-Some-expand* [*of* $\text{tab}, \text{elim-format}$]) *blast*

hence $\exists z. ((\text{tab}(x\mapsto y))(xs[\mapsto]ys'))\ vn = \text{Some}\ z$

by (*rule* $Cons.hyps$)

with ys **show** $?thesis$

by $simp$

qed

qed

lemma *map-upd-Some-swap*:

$(\text{tab}(r\mapsto w, u\mapsto v))\ vn = \text{Some}\ z \implies \exists z. (\text{tab}(u\mapsto v, r\mapsto w))\ vn = \text{Some}\ z$

by (*simp add: fun-upd-def*)

lemma *map-upd-None-swap*:

$(\text{tab}(r \mapsto w, u \mapsto v)) \text{ vn} = \text{None} \implies (\text{tab}(u \mapsto v, r \mapsto w)) \text{ vn} = \text{None}$
by (*simp add: fun-upd-def*)

lemma *map-eq-upd-eq*: $\text{tab vn} = \text{tab}' \text{ vn} \implies (\text{tab}(x \mapsto y)) \text{ vn} = (\text{tab}'(x \mapsto y)) \text{ vn}$

by (*simp add: fun-upd-def*)

lemma *map-upd-in-expansion-map-swap*:

$\llbracket (\text{tab}(x \mapsto y)) \text{ vn} = \text{Some } z; \text{tab vn} \neq \text{Some } z \rrbracket$
 $\implies (\text{tab}'(x \mapsto y)) \text{ vn} = \text{Some } z$

by (*simp add: fun-upd-def*)

lemma *map-upds-in-expansion-map-swap*:

$\bigwedge \text{tab tab}' \text{ ys } z. \llbracket (\text{tab}(\text{xs}[\mapsto] \text{ys})) \text{ vn} = \text{Some } z; \text{tab vn} \neq \text{Some } z \rrbracket$
 $\implies (\text{tab}'(\text{xs}[\mapsto] \text{ys})) \text{ vn} = \text{Some } z$

proof (*induct xs*)

case Nil thus ?*case* **by** *simp*

next

case (*Cons x xs tab tab' ys z*)

note *some* = $\langle (\text{tab}(x \# \text{xs}[\mapsto] \text{ys})) \text{ vn} = \text{Some } z \rangle$

note *tab-not-z* = $\langle \text{tab vn} \neq \text{Some } z \rangle$

show ?*case*

proof (*cases ys*)

case Nil with *some tab-not-z* **show** ?*thesis* **by** *simp*

next

case (*Cons y tl*)

note *ys* = $\langle \text{ys} = y \# \text{tl} \rangle$

show ?*thesis*

proof (*cases (tab(x ↦ y)) vn ≠ Some z*)

case True

with *some ys* **have** $(\text{tab}'(x \mapsto y, \text{xs}[\mapsto] \text{tl})) \text{ vn} = \text{Some } z$

by (*fastforce intro: Cons.hyps*)

with *ys* **show** ?*thesis*

by *simp*

next

case False

hence *tabx-z*: $(\text{tab}(x \mapsto y)) \text{ vn} = \text{Some } z$ **by** *blast*

moreover

from *tabx-z tab-not-z*

have $(\text{tab}'(x \mapsto y)) \text{ vn} = \text{Some } z$

by (*rule map-upd-in-expansion-map-swap*)

ultimately

have $(\text{tab}(x \mapsto y)) \text{ vn} = (\text{tab}'(x \mapsto y)) \text{ vn}$

by *simp*

hence $(\text{tab}(x \mapsto y, \text{xs}[\mapsto] \text{tl})) \text{ vn} = (\text{tab}'(x \mapsto y, \text{xs}[\mapsto] \text{tl})) \text{ vn}$

by (*rule map-upds-cong-ext*)

with *some ys*

show ?*thesis*

by *simp*

qed

qed

qed

lemma *map-upds-Some-swap*:

assumes *r-u*: $(\text{tab}(r \mapsto w, u \mapsto v, \text{xs}[\mapsto] \text{ys})) \text{ vn} = \text{Some } z$

shows $\exists z. (\text{tab}(u \mapsto v, r \mapsto w, \text{xs}[\mapsto] \text{ys})) \text{ vn} = \text{Some } z$

proof (*cases (tab(r ↦ w, u ↦ v)) vn = Some z*)

```

case True
then obtain  $z'$  where  $(\text{tab}(u \mapsto v, r \mapsto w)) \text{ vn} = \text{Some } z'$ 
  by  $(\text{rule map-upd-Some-swap } [\text{elim-format}]) \text{ blast}$ 
thus  $\exists z. (\text{tab}(u \mapsto v, r \mapsto w, xs \mapsto ys)) \text{ vn} = \text{Some } z$ 
  by  $(\text{rule map-upds-Some-expand})$ 
next
case False
with  $r \sim u$ 
have  $(\text{tab}(u \mapsto v, r \mapsto w, xs \mapsto ys)) \text{ vn} = \text{Some } z$ 
  by  $(\text{rule map-upds-in-expansion-map-swap})$ 
thus ?thesis
  by simp
qed

```

```

lemma map-upds-Some-insert:
  assumes  $z: (\text{tab}(xs \mapsto ys)) \text{ vn} = \text{Some } z$ 
  shows  $\exists z. (\text{tab}(u \mapsto v, xs \mapsto ys)) \text{ vn} = \text{Some } z$ 
proof  $(\text{cases } \exists z. \text{tab } \text{vn} = \text{Some } z)$ 
  case True
  then obtain  $z'$  where  $\text{tab } \text{vn} = \text{Some } z'$  by blast
  then obtain  $z''$  where  $(\text{tab}(u \mapsto v)) \text{ vn} = \text{Some } z''$ 
    by  $(\text{rule map-upd-Some-expand } [\text{elim-format}]) \text{ blast}$ 
  thus ?thesis
    by  $(\text{rule map-upds-Some-expand})$ 
next
case False
hence  $\text{tab } \text{vn} \neq \text{Some } z$  by simp
with  $z$ 
have  $(\text{tab}(u \mapsto v, xs \mapsto ys)) \text{ vn} = \text{Some } z$ 
  by  $(\text{rule map-upds-in-expansion-map-swap})$ 
thus ?thesis ..
qed

```

```

lemma map-upds-None-cut:
assumes expand-None:  $(\text{tab}(xs \mapsto ys)) \text{ vn} = \text{None}$ 
shows  $\text{tab } \text{vn} = \text{None}$ 
proof  $(\text{cases } \text{tab } \text{vn} = \text{None})$ 
  case True thus ?thesis by simp
next
case False then obtain  $z$  where  $\text{tab } \text{vn} = \text{Some } z$  by blast
then obtain  $z'$  where  $(\text{tab}(xs \mapsto ys)) \text{ vn} = \text{Some } z'$ 
  by  $(\text{rule map-upds-Some-expand } [\text{where } ?\text{tab}=\text{tab}, \text{elim-format}]) \text{ blast}$ 
with expand-None show ?thesis
  by simp
qed

```

```

lemma map-upds-cut-irrelevant:
 $\bigwedge \text{tab } \text{tab}' \text{ ys}. \llbracket (\text{tab}(xs \mapsto ys)) \text{ vn} = \text{Some } \text{el}; (\text{tab}'(xs \mapsto ys)) \text{ vn} = \text{None} \rrbracket$ 
 $\implies \text{tab } \text{vn} = \text{Some } \text{el}$ 
proof  $(\text{induct } xs)$ 
  case Nil thus ?case by simp
next
case  $(\text{Cons } x \text{ xs } \text{tab } \text{tab}' \text{ ys})$ 
note  $\text{tab-vn} = \langle (\text{tab}(x \# xs \mapsto ys)) \text{ vn} = \text{Some } \text{el} \rangle$ 
note  $\text{tab}'\text{-vn} = \langle (\text{tab}'(x \# xs \mapsto ys)) \text{ vn} = \text{None} \rangle$ 
show ?case
proof  $(\text{cases } \text{ys})$ 
  case Nil

```

```

  with tab-vn show ?thesis by simp
next
  case (Cons y tl)
  note ys = ⟨ys=y#tl⟩
  have (tab(x↦y)) vn = Some el
    by (rule Cons.hyps) (use tab-vn tab'-vn ys in 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 (VName 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 (VName 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*: $\bigwedge \text{tab}. \text{dom} (\text{tab}(x \mapsto y)) = \text{dom } \text{tab} \cup \{x\}$
by (auto simp add: dom-def fun-upd-def)

lemma *dom-map-upds*: $\bigwedge \text{tab } \text{ys}. \text{length } \text{xs} = \text{length } \text{ys}$
 $\implies \text{dom} (\text{tab}(\text{xs}[\mapsto] \text{ys})) = \text{dom } \text{tab} \cup \text{set } \text{xs}$

```

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)
    have  $\text{dom} (\text{tab}(x \mapsto y, \text{xs}[\mapsto] \text{tl})) = \text{dom} (\text{tab}(x \mapsto y)) \cup \text{set } \text{xs}$ 
      by (rule Hyp) (use len Cons in simp)
    moreover
    have  $\text{dom} (\text{tab}(x \mapsto \text{hd } \text{ys})) = \text{dom } \text{tab} \cup \{x\}$ 
      by (rule dom-map-upd)
    ultimately
    show ?thesis using Cons
      by simp
  qed
qed

```

lemma *dom-case-ename-None-simp*:
 $\text{dom} (\text{case-ename } \text{vname-tab } \text{None}) = \text{VName } \text{' } (\text{dom } \text{vname-tab})$
apply (auto simp add: dom-def image-def)
apply (case-tac x)
apply auto
done

lemma *dom-case-ename-Some-simp*:
 $\text{dom} (\text{case-ename } \text{vname-tab } (\text{Some } a)) = \text{VName } \text{' } (\text{dom } \text{vname-tab}) \cup \{\text{Res}\}$
apply (auto simp add: dom-def image-def)
apply (case-tac x)
apply auto
done

lemma *dom-case-lname-None-simp*:
 $\text{dom} (\text{case-lname } \text{ename-tab } \text{None}) = \text{EName } \text{' } (\text{dom } \text{ename-tab})$
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: ( $\text{prg}=G, \text{cls}=\text{acc}C, \text{lcl}=L$ )  $\vdash \text{dom}(\text{locals}(\text{store } s0)) \gg \langle e1 \rangle_e \gg E1$ 
by cases simp+
obtain E2 where
  ( $\text{prg}=G, \text{cls}=\text{acc}C, \text{lcl}=L$ )  $\vdash \text{dom}(\text{locals}(\text{store } s1))$ 
   $\gg (\text{if need-second-arg binop } v1 \text{ then } \langle e2 \rangle_e \text{ else } \langle \text{Skip} \rangle_s) \gg E2$ 
proof (cases need-second-arg binop v1)
  case False
  obtain S where
    daSkip: ( $\text{prg}=G, \text{cls}=\text{acc}C, \text{lcl}=L$ )
       $\vdash \text{dom}(\text{locals}(\text{store } s1)) \gg \langle \text{Skip} \rangle_s \gg S$ 
    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:  $\text{dom}(\text{locals}(\text{store } s0)) \subseteq \text{dom}(\text{locals}(\text{store } s1))$ 
    by (rule dom-locals-eval-mono-elim)
  consider (condAnd) binop=CondAnd | (condOr) binop=CondOr | (notAndOr) binop≠CondAnd binop≠CondOr
  by (cases binop) auto
  then show ?thesis
  proof cases
    case condAnd
    from da obtain E2' where
      ( $\text{prg}=G, \text{cls}=\text{acc}C, \text{lcl}=L$ )
         $\vdash \text{dom}(\text{locals}(\text{store } s0)) \cup \text{assigns-if True } e1 \gg \langle e2 \rangle_e \gg E2'$ 
      by cases (simp add: condAnd)+
    moreover
    have  $\text{dom}(\text{locals}(\text{store } s0))$ 
       $\cup \text{assigns-if True } e1 \subseteq \text{dom}(\text{locals}(\text{store } s1))$ 
    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  $\text{assigns-if True } e1 \subseteq \text{dom}(\text{locals}(\text{store } s1))$ 
      by (rule assigns-if-good-approx' [elim-format])
      (use wt-e1 in <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)
  next
  case condOr

  from da obtain E2' where
    ( $\text{prg}=G, \text{cls}=\text{acc}C, \text{lcl}=L$ )
       $\vdash \text{dom}(\text{locals}(\text{store } s0)) \cup \text{assigns-if False } e1 \gg \langle e2 \rangle_e \gg E2'$ 
    by cases (simp add: condOr)+
  moreover
  have  $\text{dom}(\text{locals}(\text{store } s0))$ 
     $\cup \text{assigns-if False } e1 \subseteq \text{dom}(\text{locals}(\text{store } s1))$ 
  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  $\subseteq$  dom (locals (store s1))
    by (rule assigns-if-good-approx' [elim-format])
    (use wt-e1 in ⟨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)
next
case notAndOr
from da notAndOr obtain E1' where
  da-e1: (⟦prg=G,cls=accC,lcl=L⟧)
     $\vdash$  dom (locals (store s0))  $\gg \langle e1 \rangle_e \gg E1'$ 
  and da-e2: (⟦prg=G,cls=accC,lcl=L⟧)  $\vdash$  nrm E1'  $\gg$  In1l e2  $\gg A$ 
  by cases simp+
from eval-e1 wt-e1 da-e1 wf normal-s1
have nrm E1'  $\subseteq$  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
qed
thus ?thesis ..
qed

```

main proof of type safety

lemma eval-type-sound:

```

assumes eval:  $G \vdash s0 \multimap \rightarrow (v, s1)$ 
and wt: (⟦prg=G,cls=accC,lcl=L⟧)  $\vdash t :: T$ 
and da: (⟦prg=G,cls=accC,lcl=L⟧)  $\vdash$  dom (locals (store s0))  $\gg t \gg A$ 
and wf: wf-prog G
and conf-s0:  $s0 :: \preceq (G, L)$ 
shows  $s1 :: \preceq (G, L) \wedge$  (normal s1  $\longrightarrow G, L, \text{store } s1 \vdash t \gg v :: \preceq T$ )  $\wedge$ 
  (error-free s0 = error-free s1)

```

proof –

```

note inj-term-simps [simp]
let ?TypeSafeObj =  $\lambda s0 s1 t v.$ 
   $\forall L \text{ accC } T A. s0 :: \preceq (G, L) \longrightarrow$  (⟦prg=G,cls=accC,lcl=L⟧)  $\vdash t :: T$ 
   $\longrightarrow$  (⟦prg=G,cls=accC,lcl=L⟧)  $\vdash$  dom (locals (store s0))  $\gg t \gg A$ 
   $\longrightarrow s1 :: \preceq (G, L) \wedge$  (normal s1  $\longrightarrow G, L, \text{store } s1 \vdash t \gg v :: \preceq T$ )
   $\wedge$  (error-free s0 = error-free s1)

```

from eval

```

have  $\bigwedge L \text{ accC } T A. \llbracket s0 :: \preceq (G, L); \llbracket \text{prg=G,cls=accC,lcl=L} \rrbracket \vdash t :: T;$ 
   $\llbracket \text{prg=G,cls=accC,lcl=L} \rrbracket \vdash \text{dom (locals (store s0))} \gg t \gg A \rrbracket$ 
 $\implies s1 :: \preceq (G, L) \wedge$  (normal s1  $\longrightarrow G, L, \text{store } s1 \vdash t \gg v :: \preceq T$ )
 $\wedge$  (error-free s0 = error-free s1)

```

(is PROP ?TypeSafe s0 s1 t v

```

  is  $\bigwedge L \text{ accC } T A. ?Conform L s0 \implies ?WellTyped L \text{ accC } T t$ 
 $\implies ?DefAss L \text{ accC } s0 t A$ 
 $\implies ?Conform L s1 \wedge ?ValueTyped L T s1 t v \wedge$ 
 $?ErrorFree s0 s1)$ 

```

proof (induct)

```

case (Abrupt xc s t L accC T A)
from  $\langle (Some\ xc, s) :: \preceq(G, L) \rangle$ 
show  $(Some\ xc, s) :: \preceq(G, L) \wedge$ 
   $(normal\ (Some\ xc, s) \longrightarrow G, L, store\ (Some\ xc, s) \vdash t \succ undefined3\ t :: \preceq T) \wedge$ 
   $(error\text{-}free\ (Some\ xc, s) = error\text{-}free\ (Some\ xc, s))$ 
by simp
next
case (Skip s L accC T A)
from  $\langle Norm\ s :: \preceq(G, L) \rangle$  and
   $\langle \langle prg = G, cls = accC, lcl = L \rangle \vdash In1r\ Skip :: T \rangle$ 
show  $Norm\ s :: \preceq(G, L) \wedge$ 
   $(normal\ (Norm\ s) \longrightarrow G, L, store\ (Norm\ s) \vdash In1r\ Skip \succ \Diamond :: \preceq T) \wedge$ 
   $(error\text{-}free\ (Norm\ s) = error\text{-}free\ (Norm\ s))$ 
by simp
next
case (Expr s0 e v s1 L accC T A)
note  $\langle G \vdash Norm\ s0 -e- \succ v \rightarrow s1 \rangle$ 
note hyp =  $\langle PROP\ ?TypeSafe\ (Norm\ s0)\ s1\ (In1l\ e)\ (In1\ v) \rangle$ 
note conf-s0 =  $\langle Norm\ s0 :: \preceq(G, L) \rangle$ 
moreover
note wt =  $\langle \langle prg = G, cls = accC, lcl = L \rangle \vdash In1r\ (Expr\ e) :: T \rangle$ 
then obtain eT
  where  $\langle prg = G, cls = accC, lcl = L \rangle \vdash In1l\ e :: eT$ 
by (rule wt-elim-cases) blast
moreover
from Expr.premis obtain E where
   $\langle prg = G, cls = accC, lcl = L \rangle \vdash dom\ (locals\ (store\ ((Norm\ s0) :: state))) \gg In1l\ e \gg E$ 
by (elim da-elim-cases) simp
ultimately
obtain  $s1 :: \preceq(G, L)$  and error-free s1
by (rule hyp [elim-format]) simp
with wt
show  $s1 :: \preceq(G, L) \wedge$ 
   $(normal\ s1 \longrightarrow G, L, store\ s1 \vdash In1r\ (Expr\ e) \succ \Diamond :: \preceq T) \wedge$ 
   $(error\text{-}free\ (Norm\ s0) = error\text{-}free\ s1)$ 
by (simp)
next
case (Lab s0 c s1 l L accC T A)
note hyp =  $\langle PROP\ ?TypeSafe\ (Norm\ s0)\ s1\ (In1r\ c)\ \Diamond \rangle$ 
note conf-s0 =  $\langle Norm\ s0 :: \preceq(G, L) \rangle$ 
moreover
note wt =  $\langle \langle prg = G, cls = accC, lcl = L \rangle \vdash In1r\ (l \cdot c) :: T \rangle$ 
then have  $\langle prg = G, cls = accC, lcl = L \rangle \vdash c :: \checkmark$ 
by (rule wt-elim-cases) blast
moreover from Lab.premis obtain C where
   $\langle prg = G, cls = accC, lcl = L \rangle \vdash dom\ (locals\ (store\ ((Norm\ s0) :: state))) \gg In1r\ c \gg C$ 
by (elim da-elim-cases) simp
ultimately
obtain conf-s1:  $s1 :: \preceq(G, L)$  and
  error-free-s1: error-free s1
by (rule hyp [elim-format]) simp
from conf-s1 have abupd (absorb l)  $s1 :: \preceq(G, L)$ 
by (cases s1) (auto intro: conforms-absorb)
with wt error-free-s1
show abupd (absorb l)  $s1 :: \preceq(G, L) \wedge$ 
   $(normal\ (abupd\ (absorb\ l)\ s1) \longrightarrow G, L, store\ (abupd\ (absorb\ l)\ s1) \vdash In1r\ (l \cdot c) \succ \Diamond :: \preceq T) \wedge$ 
   $(error\text{-}free\ (Norm\ s0) = error\text{-}free\ (abupd\ (absorb\ l)\ s1))$ 

```



```

  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.premis
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.premis obtain E C where
  da-e:  $\langle \text{prg} = G, \text{cls} = \text{acc}C, \text{lcl} = L \rangle \vdash \text{dom} (\text{locals} (\text{store} ((\text{Norm } s0)::\text{state})))$ 
    » In1l e » E and
  da-then-else:
 $\langle \text{prg} = G, \text{cls} = \text{acc}C, \text{lcl} = L \rangle \vdash$ 
   $(\text{dom} (\text{locals} (\text{store} ((\text{Norm } s0)::\text{state})))) \cup \text{assigns-if} (\text{the-Bool } b) e$ 
    » 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:  $s1::\preceq(G, L)$  and error-free-s1: error-free s1
  by (rule hyp-e [elim-format]) simp
show  $s2::\preceq(G, L) \wedge$ 
   $(\text{normal } s2 \longrightarrow G, L, \text{store } s2 \vdash \text{In1r} (\text{If}(e) \text{ c1 Else } c2) \succ \Diamond::\preceq T) \wedge$ 
   $(\text{error-free} (\text{Norm } s0) = \text{error-free } s2)$ 
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
 $\langle \text{prg} = G, \text{cls} = \text{acc}C, \text{lcl} = L \rangle \vdash$ 
 $(\text{dom} (\text{locals} (\text{store } s1))) \text{ » In1r (if the-Bool b then c1 else c2) » } C'$ 
  proof -
    from eval-e have
 $\text{dom} (\text{locals} (\text{store} ((\text{Norm } s0)::\text{state}))) \subseteq \text{dom} (\text{locals} (\text{store } s1))$ 
    by (rule dom-locals-eval-mono-elim)
    moreover
    from eval-e True wt-e
    have  $\text{assigns-if} (\text{the-Bool } b) e \subseteq \text{dom} (\text{locals} (\text{store } s1))$ 
    by (rule assigns-if-good-approx')
    ultimately
    have  $\text{dom} (\text{locals} (\text{store} ((\text{Norm } s0)::\text{state})))$ 
 $\cup \text{assigns-if} (\text{the-Bool } b) e \subseteq \text{dom} (\text{locals} (\text{store } s1))$ 
    by (rule Un-least)
    with da-then-else show thesis
    by (rule da-weakenE) (rule that)
  qed
  with conf-s1 wt-then-else
  obtain  $s2::\preceq(G, L)$  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 evaluation 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 =  $\langle G \vdash \text{Norm } s0 -e \rightarrow b \rightarrow s1 \rangle$ 
  note hyp-e =  $\langle \text{PROP } ?\text{TypeSafe} (\text{Norm } s0) \text{ s1 (In1l } e) (\text{In1l } b) \rangle$ 
  note conf-s0 =  $\langle \text{Norm } s0::\preceq(G, L) \rangle$ 
  note wt =  $\langle \langle \text{prg} = G, \text{cls} = \text{acc}C, \text{lcl} = L \rangle \vdash \text{In1r} (l \cdot \text{While}(e) \text{ c})::T \rangle$ 

```

```

then obtain  $wt-e: (\text{prg} = G, \text{cls} = \text{acc}C, \text{lcl} = L) \vdash e :: -\text{Prim}T \text{ Boolean}$  and
 $wt-c: (\text{prg} = G, \text{cls} = \text{acc}C, \text{lcl} = L) \vdash c :: \checkmark$ 
by (rule wt-elim-cases) blast
note  $da = \langle (\text{prg}=G, \text{cls}=\text{acc}C, \text{lcl}=L) \vdash \text{dom}(\text{locals}(\text{store}((\text{Norm } s0)::\text{state}))) \gg \text{In}1r(l \cdot \text{While}(e) \ c) \gg A \rangle$ 
then
obtain  $E \ C$  where
 $da-e: (\text{prg}=G, \text{cls}=\text{acc}C, \text{lcl}=L) \vdash \text{dom}(\text{locals}(\text{store}((\text{Norm } s0)::\text{state}))) \gg \text{In}1l \ e \gg E$  and
 $da-c: (\text{prg}=G, \text{cls}=\text{acc}C, \text{lcl}=L) \vdash (\text{dom}(\text{locals}(\text{store}((\text{Norm } s0)::\text{state}))) \cup \text{assigns-if } \text{True } e) \gg \text{In}1r \ c \gg C$ 
by (rule da-elim-cases) simp
from conf-s0 wt-e da-e
obtain conf-s1: s1:: $\preceq(G, L)$  and error-free-s1: error-free s1
by (rule hyp-e [elim-format]) simp
show  $s3::\preceq(G, L) \wedge$ 
 $(\text{normal } s3 \longrightarrow G, L, \text{store } s3 \vdash \text{In}1r(l \cdot \text{While}(e) \ c) \gg \Diamond::\preceq T) \wedge$ 
 $(\text{error-free}(\text{Norm } s0) = \text{error-free } s3)$ 
proof (cases normal s1)
case True
note normal-s1 = this
show ?thesis
proof (cases the-Bool b)
case True
with Loop.hyps obtain
 $eval-c: G \vdash s1 \rightarrow c \rightarrow s2$  and
 $eval-while: G \vdash \text{abupd}(\text{absorb}(\text{Cont } l)) \ s2 \rightarrow l \cdot \text{While}(e) \ c \rightarrow s3$ 
by simp
have ?TypeSafeObj s1 s2 (In1r c)  $\Diamond$ 
using Loop.hyps True by simp
note hyp-c = this [rule-format]
have ?TypeSafeObj (abupd (absorb (Cont l)) s2)
 $s3 \ (\text{In}1r \ (l \cdot \text{While}(e) \ c)) \ \Diamond$ 
using Loop.hyps True by simp
note hyp-w = this [rule-format]
from eval-e have
 $s0-s1: \text{dom}(\text{locals}(\text{store}((\text{Norm } s0)::\text{state}))) \subseteq \text{dom}(\text{locals}(\text{store } s1))$ 
by (rule dom-locals-eval-mono-elim)
obtain  $C'$  where
 $(\text{prg}=G, \text{cls}=\text{acc}C, \text{lcl}=L) \vdash (\text{dom}(\text{locals}(\text{store } s1))) \gg \text{In}1r \ c \gg C'$ 
proof –
note s0-s1
moreover
from eval-e normal-s1 wt-e
have  $\text{assigns-if } \text{True } e \subseteq \text{dom}(\text{locals}(\text{store } s1))$ 
by (rule assigns-if-good-approx' [elim-format]) (simp add: True)
ultimately
have  $\text{dom}(\text{locals}(\text{store}((\text{Norm } s0)::\text{state}))) \cup \text{assigns-if } \text{True } e \subseteq \text{dom}(\text{locals}(\text{store } s1))$ 
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:: $\preceq(G, L)$  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 ::  $\preceq(G, L)$ 
  by (cases s2) (auto intro: conforms-absorb)
moreover note wt
moreover
obtain A' where
  ( $\langle \text{prg} = G, \text{cls} = \text{acc}C, \text{lcl} = L \rangle \vdash$ 
     $\text{dom}(\text{locals}(\text{store}(\text{abupd}(\text{absorb}(\text{Cont } l)) \text{ s2})))$ 
     $\gg \text{In1r } (l \cdot \text{While}(e) \text{ c}) \gg A'$ )
proof –
  note s0-s1
  also from eval-c
  have  $\text{dom}(\text{locals}(\text{store } s1)) \subseteq \text{dom}(\text{locals}(\text{store } s2))$ 
    by (rule dom-locals-eval-mono-elim)
  also have  $\dots \subseteq \text{dom}(\text{locals}(\text{store}(\text{abupd}(\text{absorb}(\text{Cont } l)) \text{ s2})))$ 
    by simp
  finally
  have  $\text{dom}(\text{locals}(\text{store}((\text{Norm } s0)::\text{state}))) \subseteq \dots$ 
  with da show thesis
    by (rule da-weakenE) (rule that)
qed
ultimately obtain s3 ::  $\preceq(G, L)$  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:  $\bigwedge j. \text{abrupt } s1 \neq \text{Some } (\text{Jump } j)$ 
    by (rule eval-expression-no-jump
      [where  $\langle \text{prg} = G, \text{cls} = \text{acc}C, \text{lcl} = L \rangle, \text{simplified}$ ])
      (simp-all add: wf)

  show ?thesis
  proof (cases the-Bool b)
    case True
    with Loop.hyps obtain
      eval-c:  $G \vdash s1 \rightarrow c \rightarrow s2$  and
      eval-while:  $G \vdash \text{abupd}(\text{absorb}(\text{Cont } l)) \text{ s2} \rightarrow l \cdot \text{While}(e) \text{ c} \rightarrow s3$ 
    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.prem
  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≼C SXcpt Throwable
  by (rule wt-elim-cases) (auto)
  from Throw.prem 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.premis 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])
  (use False in ⟨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  $G \vdash \text{obj-ty } (\text{the } (\text{globs } (\text{store } s2) (\text{Heap } a))) \preceq \text{Class } \text{catchC}$ 
  by (cases s2) simp
with xcpt-s2 conf-s2 wf
have  $\text{new-xcpt-var } vn \ s2 :: \preceq (G, L(\text{VName } vn \mapsto \text{Class } \text{catchC}))$ 
  by (auto dest: Try-lemma)
moreover note wt-c2
moreover
obtain  $C2'$  where
   $(\text{prg} = G, \text{cls} = \text{accC}, \text{lcl} = L(\text{VName } vn \mapsto \text{Class } \text{catchC}))$ 
   $\vdash (\text{dom } (\text{locals } (\text{store } (\text{new-xcpt-var } vn \ s2)))) \gg \text{In1r } c2 \gg C2'$ 
proof –
  have  $(\text{dom } (\text{locals } (\text{store } ((\text{Norm } s0)::\text{state}))) \cup \{\text{VName } vn\})$ 
     $\subseteq \text{dom } (\text{locals } (\text{store } (\text{new-xcpt-var } vn \ s2)))$ 
  proof –
    from  $\langle G \vdash \text{Norm } s0 \rightarrow c1 \rightarrow s1 \rangle$ 
    have  $\text{dom } (\text{locals } (\text{store } ((\text{Norm } s0)::\text{state})))$ 
       $\subseteq \text{dom } (\text{locals } (\text{store } s1))$ 
    by (rule dom-locals-eval-mono-elim)
    also
    from sx-alloc
    have  $\dots \subseteq \text{dom } (\text{locals } (\text{store } s2))$ 
    by (rule dom-locals-sxalloc-mono)
    also
    have  $\dots \subseteq \text{dom } (\text{locals } (\text{store } (\text{new-xcpt-var } vn \ s2)))$ 
    by (cases s2) (simp add: new-xcpt-var-def, blast)
    also
    have  $\{\text{VName } vn\} \subseteq \dots$ 
    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  $\text{conf-s3}: s3 :: \preceq (G, L(\text{VName } vn \mapsto \text{Class } \text{catchC}))$  and
   $\text{error-free-s3}: \text{error-free } s3$ 
  by (rule hyp-c2 [elim-format])
  (cases s2, simp add: xcpt-s2 error-free-s2)
from conf-s3 fresh-vn
have  $s3 :: \preceq (G, L)$ 
  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  $\text{eval-c1} = \langle G \vdash \text{Norm } s0 \rightarrow c1 \rightarrow (x1, s1) \rangle$ 
note  $\text{eval-c2} = \langle G \vdash \text{Norm } s1 \rightarrow c2 \rightarrow s2 \rangle$ 
note  $s3 = \langle s3 = (\text{if } \exists \text{err}. x1 = \text{Some } (\text{Error } \text{err}))$ 
   $\text{then } (x1, s1)$ 
   $\text{else abrupt } (\text{abrupt-if } (x1 \neq \text{None}) x1) s2 \rangle$ 
note  $\text{hyp-c1} = \langle \text{PROP } ?\text{TypeSafe } (\text{Norm } s0) (x1, s1) (\text{In1r } c1) \Diamond \rangle$ 
note  $\text{hyp-c2} = \langle \text{PROP } ?\text{TypeSafe } (\text{Norm } s1) s2 (\text{In1r } c2) \Diamond \rangle$ 
note  $\text{conf-s0} = \langle \text{Norm } s0 :: \preceq (G, L) \rangle$ 
note  $\text{wt} = \langle (\text{prg} = G, \text{cls} = \text{accC}, \text{lcl} = L) \vdash \text{In1r } (c1 \text{ Finally } c2) :: T \rangle$ 

```

```

then obtain
  wt-c1: ( $\text{prg}=G, \text{cls}=\text{acc}C, \text{lcl}=L$ ) $\vdash c1::\sqrt{\phantom{x}}$  and
  wt-c2: ( $\text{prg}=G, \text{cls}=\text{acc}C, \text{lcl}=L$ ) $\vdash c2::\sqrt{\phantom{x}}$ 
by (rule wt-elim-cases) blast
from Fin.prem obtain C1 C2 where
  da-c1: ( $\text{prg}=G, \text{cls}=\text{acc}C, \text{lcl}=L$ )
     $\vdash \text{dom}(\text{locals}(\text{store}((\text{Norm } s0)::\text{state}))) \gg \text{In1r } c1 \gg C1$  and
  da-c2: ( $\text{prg}=G, \text{cls}=\text{acc}C, \text{lcl}=L$ )
     $\vdash \text{dom}(\text{locals}(\text{store}((\text{Norm } s0)::\text{state}))) \gg \text{In1r } c2 \gg C2$ 
by (elim da-elim-cases) simp
from conf-s0 wt-c1 da-c1
obtain conf-s1: ( $x1, s1$ ) $::\preceq(G, L)$  and error-free-s1: error-free ( $x1, s1$ )
by (rule hyp-c1 [elim-format]) simp
from conf-s1 have Norm s1 $::\preceq(G, L)$ 
by (rule conforms-NormI)
moreover note wt-c2
moreover obtain C2'
  where ( $\text{prg}=G, \text{cls}=\text{acc}C, \text{lcl}=L$ )
     $\vdash \text{dom}(\text{locals}(\text{store}((\text{Norm } s1)::\text{state}))) \gg \text{In1r } c2 \gg C2'$ 
proof –
  from eval-c1
  have  $\text{dom}(\text{locals}(\text{store}((\text{Norm } s0)::\text{state}))) \subseteq \text{dom}(\text{locals}(\text{store}(x1, s1)))$ 
    by (rule dom-locals-eval-mono-elim)
  hence  $\text{dom}(\text{locals}(\text{store}((\text{Norm } s0)::\text{state}))) \subseteq \text{dom}(\text{locals}(\text{store}((\text{Norm } s1)::\text{state})))$ 
    by simp
  with da-c2 show thesis
    by (rule da-weakenE) (rule that)
qed
ultimately
obtain conf-s2:  $s2::\preceq(G, L)$  and error-free-s2: error-free s2
by (rule hyp-c2 [elim-format]) simp
from error-free-s1 s3
have s3':  $s3=\text{abupd}(\text{abrupt-if } (x1 \neq \text{None}) x1) s2$ 
by simp
show  $s3::\preceq(G, L) \wedge$ 
  ( $\text{normal } s3 \longrightarrow G, L, \text{store } s3 \vdash \text{In1r } (c1 \text{ Finally } c2) \gg \Diamond::\preceq T$ )  $\wedge$ 
  ( $\text{error-free}(\text{Norm } s0) = \text{error-free } s3$ )
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
     $\text{dom}(\text{locals}(\text{store}((\text{Norm } s1)::\text{state}))) \subseteq \text{dom}(\text{locals}(\text{store } s2))$ 
    by (rule dom-locals-eval-mono-elim)
  with Some eval-c2 wf conf-s1 conf-s2
  have conf: ( $\text{abrupt-if } \text{True}(\text{Some } x) (\text{abrupt } s2), \text{store } s2$ ) $::\preceq(G, L)$ 
    by (cases s2) (auto dest: Fin-lemma)
  from Some error-free-s1
  have  $\neg (\exists \text{err. } x=\text{Error err})$ 
    by (simp add: error-free-def)
  with error-free-s2
  have error-free ( $\text{abrupt-if } \text{True}(\text{Some } x) (\text{abrupt } s2), \text{store } s2$ )
    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
  show ?thesis
  by (rule that) (use da-Init False in ⟨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-geat')
  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

```

$\langle \text{prg} = G, \text{cls} = \text{acc}C, \text{lcl} = L \rangle$
 $\vdash \text{dom} (\text{locals} (\text{store} ((\text{Norm } s0)::\text{state}))) \gg \text{In1r} (\text{Init } C) \gg I$
by (auto intro: da-Init [simplified] assigned.select-convs)

ultimately

obtain *conf-s1*: $s1::\preceq(G, L)$ **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*: $s2::\preceq(G, L)$
 $(\text{normal } s2 \longrightarrow G, \text{store } s2 \vdash \text{Addr } a::\preceq \text{Class } C)$
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 $s2::\preceq(G, L) \wedge$
 $(\text{normal } s2 \longrightarrow G, L, \text{store } s2 \vdash \text{In1l} (\text{NewC } C) \succ \text{In1} (\text{Addr } a)::\preceq T) \wedge$
 $(\text{error-free} (\text{Norm } s0) = \text{error-free } s2)$
by auto

next

case (*NewA s0 elT s1 e i s2 a s3 L accC T A*)
note *eval-init* = $\langle G \vdash \text{Norm } s0 \text{ --init-comp-ty } \text{elT} \rightarrow s1 \rangle$
note *eval-e* = $\langle G \vdash s1 \text{ --e--} i \rightarrow s2 \rangle$
note *halloc* = $\langle G \vdash \text{abupd} (\text{check-neg } i) \text{ s2-halloc Arr elT (the-Intg } i) \succ a \rightarrow s3 \rangle$
note *hyp-init* = $\langle \text{PROP ?TypeSafe (Norm } s0) \text{ s1 (In1r (init-comp-ty elT))} \rangle \Diamond$
note *hyp-size* = $\langle \text{PROP ?TypeSafe s1 s2 (In1l e) (In1 i)} \rangle$
note *conf-s0* = $\langle \text{Norm } s0::\preceq(G, L) \rangle$
note *wt* = $\langle \langle \text{prg} = G, \text{cls} = \text{acc}C, \text{lcl} = L \rangle \vdash \text{In1l} (\text{New elT}[e])::T \rangle$

then obtain

wt-init: $\langle \text{prg} = G, \text{cls} = \text{acc}C, \text{lcl} = L \rangle \vdash \text{init-comp-ty } \text{elT}::\checkmark$ **and**
wt-size: $\langle \text{prg} = G, \text{cls} = \text{acc}C, \text{lcl} = L \rangle \vdash e::\text{--PrimT Integer}$ **and**
 $\text{elT}: \text{is-type } G \text{ elT}$ **and**
 $T: T = \text{Inl} (\text{elT}.)$

by (rule wt-elim-cases) (auto intro: wt-init-comp-ty dest: is-acc-typeD)

from *NewA.prem*s

have *da-e*: $\langle \text{prg} = G, \text{cls} = \text{acc}C, \text{lcl} = L \rangle$
 $\vdash \text{dom} (\text{locals} (\text{store} ((\text{Norm } s0)::\text{state}))) \gg \text{In1l } e \gg A$
by (elim da-elim-cases) simp

obtain *conf-s1*: $s1::\preceq(G, L)$ **and** *error-free-s1*: *error-free s1*

proof –

note *conf-s0 wt-init*

moreover obtain *I* **where**

$\langle \text{prg} = G, \text{cls} = \text{acc}C, \text{lcl} = L \rangle$
 $\vdash \text{dom} (\text{locals} (\text{store} ((\text{Norm } s0)::\text{state}))) \gg \text{In1r} (\text{init-comp-ty } \text{elT}) \gg I$

proof (cases $\exists C. \text{elT} = \text{Class } C$)

case *True*

show *?thesis*

by (rule that)

(use *True* in $\langle \text{auto intro: da-Init [simplified] assigned.select-convs}$
 $\text{simp add: init-comp-ty-def} \rangle$)

next

case *False*

show *?thesis*

by (rule that)

(use *False* in $\langle \text{auto intro: da-Skip [simplified] assigned.select-convs}$
 $\text{simp add: init-comp-ty-def} \rangle$)

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.prem
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
have G,L,store s2⊢In1l (Cast castT e)>In1 v::≤T

```

```

    if norm-s2: normal s2
  proof -
    from s2 norm-s2 have normal s1
      by (cases s1) simp
    with v-ok
    have G,store s1 ⊢ v :: ≤ e T
      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.premis obtain eT
  where wt-e: (prg = G, cls = accC, lcl = L) ⊢ e :: - RefT e T and
    T: T = Inl (PrimT Boolean)
  by (elim wt-elim-cases) simp
  from Inst.premis
  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 e T 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.premis 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  $normal \ s1 \longrightarrow G, L, snd \ s1 \vdash In1l \ (UnOp \ unop \ e) \triangleright In1 \ (eval-unop \ unop \ v) :: \preceq T$ 
  by (cases unop) auto
with  $conf-s1 \ error-free-s1$ 
show  $s1 :: \preceq (G, L) \wedge$ 
   $(normal \ s1 \longrightarrow G, L, snd \ s1 \vdash In1l \ (UnOp \ unop \ e) \triangleright In1 \ (eval-unop \ unop \ v) :: \preceq T) \wedge$ 
   $error-free \ (Norm \ s0) = error-free \ s1$ 
  by simp
next
case ( $BinOp \ s0 \ e1 \ v1 \ s1 \ binop \ e2 \ v2 \ s2 \ L \ accC \ T \ A$ )
note  $eval-e1 = \langle G \vdash Norm \ s0 \ -e1 \rightarrow v1 \rightarrow s1 \rangle$ 
note  $eval-e2 = \langle G \vdash s1 \ -(if \ need-second-arg \ binop \ v1 \ then \ In1l \ e2$ 
   $\quad \quad \quad else \ In1r \ Skip) \triangleright \rightarrow (In1 \ v2, \ s2) \rangle$ 
note  $hyp-e1 = \langle PROP \ ?TypeSafe \ (Norm \ s0) \ s1 \ (In1l \ e1) \ (In1 \ v1) \rangle$ 
note  $hyp-e2 = \langle PROP \ ?TypeSafe \quad \quad \quad s1 \ s2$ 
   $\quad \quad \quad (if \ need-second-arg \ binop \ v1 \ then \ In1l \ e2 \ else \ In1r \ Skip)$ 
   $\quad \quad \quad (In1 \ v2) \rangle$ 
note  $conf-s0 = \langle Norm \ s0 :: \preceq (G, L) \rangle$ 
note  $wt = \langle \langle prg = G, \ cls = accC, \ lcl = L \rangle \vdash In1l \ (BinOp \ binop \ e1 \ e2) :: T \rangle$ 
then obtain  $e1T \ e2T$  where
   $wt-e1: \langle prg = G, \ cls = accC, \ lcl = L \rangle \vdash e1 :: -e1T$  and
   $wt-e2: \langle prg = G, \ cls = accC, \ lcl = L \rangle \vdash e2 :: -e2T$  and
   $wt-binop: wt-binop \ G \ binop \ e1T \ e2T$  and
   $T: T = Inl \ (PrimT \ (binop-type \ binop))$ 
  by (elim wt-elim-cases) simp
have  $wt-Skip: \langle prg = G, \ cls = accC, \ lcl = L \rangle \vdash Skip :: \checkmark$ 
  by simp
obtain  $S$  where
   $daSkip: \langle prg = G, \ cls = accC, \ lcl = L \rangle$ 
   $\quad \vdash dom \ (locals \ (store \ s1)) \triangleright In1r \ Skip \triangleright S$ 
  by (auto intro: da-Skip [simplified] assigned.select-convs)
note  $da = \langle \langle prg = G, \ cls = accC, \ lcl = L \rangle \vdash dom \ (locals \ (store \ ((Norm \ s0 :: state))))$ 
   $\quad \triangleright \langle BinOp \ binop \ e1 \ e2 \rangle_e \triangleright A \rangle$ 
then obtain  $E1$  where
   $da-e1: \langle prg = G, \ cls = accC, \ lcl = L \rangle$ 
   $\quad \vdash dom \ (locals \ (store \ ((Norm \ s0 :: state)))) \triangleright In1l \ e1 \triangleright E1$ 
  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  $conf-v:$ 
   $G, L, snd \ s2 \vdash In1l \ (BinOp \ binop \ e1 \ e2) \triangleright In1 \ (eval-binop \ binop \ v1 \ v2) :: \preceq T$ 
  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  $s2 :: \preceq (G, L) \wedge$ 
   $(normal \ s2 \longrightarrow$ 
   $G, L, snd \ s2 \vdash In1l \ (BinOp \ binop \ e1 \ e2) \triangleright In1 \ (eval-binop \ binop \ v1 \ v2) :: \preceq T) \wedge$ 

```

```

    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.prem
      obtain This ∈ dom (locals s)
        by (elim da-elim-cases) simp
      with conf-s C have G, st-val-this s::⊢Class C
        by (auto dest: conforms-localD [THEN wlconfD])
      with neq-Obj cls-C wf
      have G, st-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.premis 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.premis 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) (use da.LVar in ⟨auto elim!: da-elim-cases⟩)
have lvar-in-locals: locals (store s1) n ≠ None
  if lvar: v=LVar n for n
proof –
  from Acc.premis 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
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.premis
  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)::≼In1 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)::≼In1 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.prem
  obtain E where
    da-e: ( $\langle \text{prg}=G, \text{cls}=\text{acc}C, \text{lcl}=L \rangle$ 
       $\vdash \text{dom} (\text{locals} (\text{store} ((\text{Norm } s0)::\text{state}))) \gg \text{In1 } e \gg E$ )
    by (elim da-elim-cases) simp+
  from da.LVar vn obtain V where
    da-var: ( $\langle \text{prg}=G, \text{cls}=\text{acc}C, \text{lcl}=L \rangle$ 
       $\vdash \text{dom} (\text{locals} (\text{store} ((\text{Norm } s0)::\text{state}))) \gg \text{In2 } \text{var} \gg V$ )
    by auto
  obtain E' where
    da-e': ( $\langle \text{prg}=G, \text{cls}=\text{acc}C, \text{lcl}=L \rangle$ 
       $\vdash \text{dom} (\text{locals} (\text{store } s1)) \gg \text{In1 } e \gg E'$ )
  proof -
    have  $\text{dom} (\text{locals} (\text{store} ((\text{Norm } s0)::\text{state})))$ 
       $\subseteq \text{dom} (\text{locals} (\text{store } s1))$ 
      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:  $s1::\preceq(G, L)$ 
    and conf-var:  $\text{normal } s1$ 
       $\longrightarrow G, L, \text{store } s1 \vdash \text{In2 } \text{var} \gg \text{In2 } (w, \text{upd})::\preceq \text{In1 } \text{var} T$ 
    and error-free-s1:  $\text{error-free } s1$ 
    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  $\text{assign upd } v \ s2=s1$ 
      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:  $s2::\preceq(G, L)$  and
      conf-v:  $\text{normal } s2 \longrightarrow G, \text{store } s2 \vdash v::\preceq e T$  and
      error-free-s2:  $\text{error-free } s2$ 
      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  $\text{conf-v-e} T: G, \text{store } s2 \vdash v::\preceq e T$ 
        by simp

```

```

with widen wf
have conf-v-varT:  $G, \text{store } s2 \vdash v :: \preceq \text{varT}$ 
  by (auto intro: conf-widen)
from normal-s1 conf-var
have  $G, L, \text{store } s1 \vdash \text{In2 } \text{var} \succ \text{In2 } (w, \text{upd}) :: \preceq \text{In1 } \text{varT}$ 
  by simp
then
have conf-assign:  $\text{store } s1 \leq | \text{upd} \preceq \text{varT} :: \preceq (G, L)$ 
  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 =  $\langle G \vdash \text{Norm } s0 \text{ --e0--} \succ b \rightarrow s1 \rangle$ 
note eval-e1-e2 =  $\langle G \vdash s1 \text{ --(if the-Bool } b \text{ then } e1 \text{ else } e2) \text{--} \succ v \rightarrow s2 \rangle$ 
note hyp-e0 =  $\langle \text{PROP } ?\text{TypeSafe } (\text{Norm } s0) \text{ } s1 \text{ } (\text{In1l } e0) \text{ } (\text{In1 } b) \rangle$ 
note hyp-if =  $\langle \text{PROP } ?\text{TypeSafe } s1 \text{ } s2$ 
   $(\text{In1l } (\text{if the-Bool } b \text{ then } e1 \text{ else } e2)) \text{ } (\text{In1 } v) \rangle$ 
note conf-s0 =  $\langle \text{Norm } s0 :: \preceq (G, L) \rangle$ 
note wt =  $\langle \langle \text{prg} = G, \text{cls} = \text{accC}, \text{lcl} = L \rangle \vdash \text{In1l } (e0 ? e1 : e2) :: T \rangle$ 
then obtain T1 T2 statT where
  wt-e0:  $\langle \text{prg} = G, \text{cls} = \text{accC}, \text{lcl} = L \rangle \vdash e0 :: \text{--PrimT Boolean and}$ 
  wt-e1:  $\langle \text{prg} = G, \text{cls} = \text{accC}, \text{lcl} = L \rangle \vdash e1 :: \text{--T1 and}$ 
  wt-e2:  $\langle \text{prg} = G, \text{cls} = \text{accC}, \text{lcl} = L \rangle \vdash e2 :: \text{--T2 and}$ 
  statT:  $G \vdash T1 \preceq T2 \wedge \text{statT} = T2 \vee G \vdash T2 \preceq T1 \wedge \text{statT} = T1$  and
  T :  $T = \text{Inl statT}$ 
by (rule wt-elim-cases) auto
with Cond.premis obtain E0 E1 E2 where
  da-e0:  $\langle \text{prg} = G, \text{cls} = \text{accC}, \text{lcl} = L \rangle$ 
     $\vdash \text{dom } (\text{locals } (\text{store } ((\text{Norm } s0) :: \text{state})))$ 
     $\gg \text{In1l } e0 \gg E0$  and
  da-e1:  $\langle \text{prg} = G, \text{cls} = \text{accC}, \text{lcl} = L \rangle$ 
     $\vdash (\text{dom } (\text{locals } (\text{store } ((\text{Norm } s0) :: \text{state}))))$ 
     $\cup \text{assigns-if True } e0 \gg \text{In1l } e1 \gg E1$  and
  da-e2:  $\langle \text{prg} = G, \text{cls} = \text{accC}, \text{lcl} = L \rangle$ 
     $\vdash (\text{dom } (\text{locals } (\text{store } ((\text{Norm } s0) :: \text{state}))))$ 
     $\cup \text{assigns-if False } e0 \gg \text{In1l } e2 \gg E2$ 
by (elim da-elim-cases) simp+
from conf-s0 wt-e0 da-e0
obtain conf-s1:  $s1 :: \preceq (G, L)$  and error-free-s1: error-free s1
  by (rule hyp-e0 [elim-format]) simp
show  $s2 :: \preceq (G, L) \wedge$ 
   $(\text{normal } s2 \rightarrow G, L, \text{store } s2 \vdash \text{In1l } (e0 ? e1 : e2) \succ \text{In1 } v :: \preceq T) \wedge$ 
   $(\text{error-free } (\text{Norm } s0) = \text{error-free } s2)$ 
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  $s0\text{-}s1$ :  $\text{dom } (\text{locals } (\text{store } ((\text{Norm } s0) :: \text{state})))$ 
       $\cup \text{assigns-if } (\text{the-Bool } b) \text{ } e0 \subseteq \text{dom } (\text{locals } (\text{store } s1))$ 

```

```

proof –
  from eval-e0 have
     $\text{dom} (\text{locals} (\text{store} ((\text{Norm } s0)::\text{state}))) \subseteq \text{dom} (\text{locals} (\text{store } s1))$ 
    by (rule dom-locals-eval-mono-elim)
  moreover
    from eval-e0 True wt-e0
    have assigns-if (the-Bool b)  $e0 \subseteq \text{dom} (\text{locals} (\text{store } s1))$ 
      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
       $(\text{prg}=G, \text{cls}=\text{accC}, \text{lcl}=L) \vdash (\text{dom} (\text{locals} (\text{store } s1))) \gg \text{In1l } e1 \gg E1'$ 
      by – (rule da-weakenE, auto iff del: Un-subset-iff sup.bounded-iff)
    with conf-s1 wt-e1
    obtain
       $s2::\preceq(G, L)$ 
       $(\text{normal } s2 \longrightarrow G, L, \text{store } s2 \vdash \text{In1l } e1 \succ \text{In1 } v::\preceq \text{In1 } T1)$ 
      error-free s2
      by (rule hyp-e1 [elim-format]) (simp add: error-free-s1)
    moreover
      from statT
      have  $G \vdash T1 \preceq \text{statT}$ 
      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
         $(\text{prg}=G, \text{cls}=\text{accC}, \text{lcl}=L) \vdash (\text{dom} (\text{locals} (\text{store } s1))) \gg \text{In1l } e2 \gg E2'$ 
        by – (rule da-weakenE, auto iff del: Un-subset-iff sup.bounded-iff)
      with conf-s1 wt-e2
      obtain
         $s2::\preceq(G, L)$ 
         $(\text{normal } s2 \longrightarrow G, L, \text{store } s2 \vdash \text{In1l } e2 \succ \text{In1 } v::\preceq \text{In1 } T2)$ 
        error-free s2
        by (rule hyp-e2 [elim-format]) (simp add: error-free-s1)
      moreover
        from statT
        have  $G \vdash T2 \preceq \text{statT}$ 
        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 =  $\langle G \vdash \text{Norm } s0 \rightarrow e \rightarrow a \rightarrow s1 \rangle$ 
  note eval-args =  $\langle G \vdash s1 \rightarrow \text{args} \rightarrow vs \rightarrow s2 \rangle$ 
  note invDeclC =  $\langle \text{invDeclC}$ 
    = invocation-declclass G mode (store s2) a statT
    (name = mn, parTs = pTs') $\rangle$ 

```

```

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⟧)  $\multimap v \rightarrow s_4$ ⟩
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 ::  $\preceq$  (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 ::  $\neg \text{RefT } \text{statT}$  and
  wt-args: ⟦prg = G, cls = accC, lcl = L⟧ ⊢ args ::  $\doteq pTs$  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.premis 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 ::  $\preceq$  (G, L) and
  conf-a: normal s1  $\implies$  G, store s1 ⊢ a ::  $\preceq \text{RefT } \text{statT}$  and
  error-free-s1: error-free s1
by (rule hyp-e [elim-format]) simp
have propagate-abnormal-s2: set-lvars (locals (store s2)) s4 = s2
if abnormal-s2:  $\neg$  normal 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
show (set-lvars (locals (store s2))) s4 ::  $\preceq$  (G, L)  $\wedge$ 
  (normal ((set-lvars (locals (store s2))) s4)  $\longrightarrow$ 
    G, L, store ((set-lvars (locals (store s2))) s4)
    ⊢ In1l (⟦accC', statT, mode⟧ e.mn (⟦pTs⟧ args))  $\triangleright$  In1 v ::  $\preceq T$ )  $\wedge$ 
  (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
   $(\text{prg}=G, \text{cls}=\text{acc}C, \text{lcl}=L) \vdash \text{dom}(\text{locals}(\text{store } s1)) \gg \text{In3 args} \gg A'$ 
proof –
  from eval-e wt-e da-e wf normal-s1
  have  $\text{nrm } E \subseteq \text{dom}(\text{locals}(\text{store } s1))$ 
    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  $\text{conf-s2: } s2::\preceq(G, L)$  and
   $\text{conf-args: normal } s2$ 
   $\implies \text{list-all2}(\text{conf } G(\text{store } s2)) \text{ vs } pTs$  and
   $\text{error-free-s2: error-free } s2$ 
  by (rule hyp-args [elim-format]) (simp add: error-free-s1)
from error-free-s2 init-lvars
have  $\text{error-free-s3: error-free } s3$ 
  by (auto simp add: init-lvars-def2)
from statM
obtain
   $\text{statM}': (\text{statDecl}T, \text{stat}M) \in \text{mheads } G \text{ acc}C \text{ stat}T$   $(\text{name}=\text{mn}, \text{parTs}=\text{pTs}')$  and
   $\text{pTs-widen: } G \vdash \text{pTs}[\preceq] \text{pTs}'$ 
  by (blast dest: max-spec2mheads)
from check
have  $\text{eq-store-s3'-s3: store } s3' = \text{store } s3$ 
  by (cases s3) (simp add: check-method-access-def Let-def)
obtain invC
  where  $\text{inv}C: \text{inv}C = \text{invocation-class mode}(\text{store } s2) \text{ a } \text{stat}T$ 
  by simp
with init-lvars
have  $\text{inv}C': \text{inv}C = (\text{invocation-class mode}(\text{store } s3) \text{ a } \text{stat}T)$ 
  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  $\text{conf-a-s2: } G, \text{store } s2 \vdash a::\preceq \text{Ref}T \text{ stat}T$ 
  by (auto dest: eval-gext intro: conf-gext)
show ?thesis
proof (cases a=Null  $\implies$  is-static statM)
  case False
  then obtain not-static:  $\neg$  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  $\Rightarrow$  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:: $\preceq$ (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-geat dest: eval-geat)
  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':: $\preceq$ (G,L') by simp
moreover
from is-static-eq wf-dynM L'
obtain mthdT where
  (prg=G,cls=invDeclC,lcl=L')
   $\vdash$  Body invDeclC (stmt (mbody (mthd dynM)))::-mthdT and
  mthdT-widen: G $\vdash$ mthdT $\preceq_{resTy}$  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')
   $\vdash$  (Methd invDeclC (name = mn, parTs = pTs'))::-mthdT
  by (auto intro: wt.Methd)
moreover
obtain M where
  (prg=G,cls=invDeclC,lcl=L')
   $\vdash$  dom (locals (store s3'))
   $\gg$  In1l (Methd invDeclC (name = mn, parTs = pTs'))  $\gg$  M
proof -
  from wf-dynM
  obtain M' where
    da-body:
      (prg=G, cls=invDeclC
        ,lcl=callee-lcl invDeclC (name = mn, parTs = pTs') (mthd dynM)
        )  $\vdash$  parameters (mthd dynM)  $\gg$  (stmt (mbody (mthd dynM)))  $\gg$  M' and
      res: Result  $\in$  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'))
    ⊢ parameters (mthd dynM) »⟨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 (methd m))) :: -bodyT and
  T: T = Inl bodyT
  by (rule wt-elim-cases) auto
moreover
from Methd.premis m have
  da-body: (prg = G, cls = accC, lcl = L)
    ⊢ (dom (locals (store ((Norm s0) :: state))))
      » In1l (Body (declclass m) (stmt (mbody (methd 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 =  $\langle G \vdash \text{Norm } s0 \rightarrow \text{Init } D \rightarrow s1 \rangle$ 
note eval-c =  $\langle G \vdash s1 \rightarrow c \rightarrow s2 \rangle$ 
note hyp-init =  $\langle \text{PROP } ?\text{TypeSafe } (\text{Norm } s0) \ s1 \ (\text{In1r } (\text{Init } D)) \ \Diamond \rangle$ 
note hyp-c =  $\langle \text{PROP } ?\text{TypeSafe } s1 \ s2 \ (\text{In1r } c) \ \Diamond \rangle$ 
note conf-s0 =  $\langle \text{Norm } s0 :: \preceq (G, L) \rangle$ 
note wt =  $\langle \langle \text{prg} = G, \text{cls} = \text{accC}, \text{lcl} = L \rangle \vdash \text{In1l } (\text{Body } D \ c) :: T \rangle$ 
then obtain bodyT where
  iscls-D: is-class G D and
  wt-c:  $\langle \text{prg} = G, \text{cls} = \text{accC}, \text{lcl} = L \rangle \vdash c :: \surd$  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.premis obtain C where
  da-c:  $\langle \text{prg} = G, \text{cls} = \text{accC}, \text{lcl} = L \rangle$ 
     $\vdash (\text{dom } (\text{locals } (\text{store } ((\text{Norm } s0) :: \text{state})))) \gg \text{In1r } c \gg C$  and
  jmpOk: jumpNestingOkS {Ret} c and
  res: Result  $\in$  nrm C
  by (elim da-elim-cases) simp
note conf-s0
moreover from iscls-D
have  $\langle \text{prg} = G, \text{cls} = \text{accC}, \text{lcl} = L \rangle \vdash \text{Init } D :: \surd$  by auto
moreover obtain I where
   $\langle \text{prg} = G, \text{cls} = \text{accC}, \text{lcl} = L \rangle$ 
     $\vdash \text{dom } (\text{locals } (\text{store } ((\text{Norm } s0) :: \text{state})))) \gg \text{In1r } (\text{Init } D) \gg I$ 
  by (auto intro: da-Init [simplified] assigned.select-convs)
ultimately obtain
  conf-s1:  $s1 :: \preceq (G, L)$  and error-free-s1: error-free s1
  by (rule hyp-init [elim-format]) simp
obtain C' where da-C':  $\langle \text{prg} = G, \text{cls} = \text{accC}, \text{lcl} = L \rangle$ 
     $\vdash (\text{dom } (\text{locals } (\text{store } s1)))) \gg \text{In1r } c \gg C'$ 
  and nrm-C': nrm C  $\subseteq$  nrm C'
proof -
  from eval-init
  have  $(\text{dom } (\text{locals } (\text{store } ((\text{Norm } s0) :: \text{state}))))$ 
     $\subseteq (\text{dom } (\text{locals } (\text{store } s1)))$ 
  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:  $s2 :: \preceq (G, L)$  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 ::  $\preceq (G, L)$ 
  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)  $\neq$  Some (Jump Ret)
  by (cases s3) (simp add: absorb-def)
moreover have s3 = s2
proof -
  from iscls-D

```

```

have wt-init: ( $\llbracket \text{prg} = G, \text{cls} = \text{accC}, \text{lcl} = L \rrbracket \vdash (\text{Init } D) :: \checkmark$ )
  by auto
have s1-no-jmp:  $\bigwedge j. \text{abrupt } s1 \neq \text{Some } (\text{Jump } j)$ 
  by (rule eval-statement-no-jump [OF - - wt-init]) (use eval-init wf in auto)
from eval-c - wt-c wf
have  $\bigwedge j. \text{abrupt } s2 = \text{Some } (\text{Jump } j) \implies j = \text{Ret}$ 
  by (rule jmpNestingOk-evalE) (auto intro: jmpOk simp add: s1-no-jmp)
moreover
note  $\langle s3 =$ 
  (if  $\exists l. \text{abrupt } s2 = \text{Some } (\text{Jump } (\text{Break } l)) \vee$ 
     $\text{abrupt } s2 = \text{Some } (\text{Jump } (\text{Cont } l))$ 
    then  $\text{abupd } (\lambda x. \text{Some } (\text{Error CrossMethodJump})) s2$  else  $s2 \rangle$ 
ultimately show ?thesis
  by force
qed
moreover
have  $\text{Result} \in \text{dom } (\text{locals } (\text{store } s2))$ 
  if normal-upd-s2: normal (abupd (absorb Ret) s2)
proof -
  from normal-upd-s2
have  $\text{normal } s2 \vee \text{abrupt } s2 = \text{Some } (\text{Jump Ret})$ 
  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  $\text{abrupt } s2 = \text{Some } (\text{Jump Ret})$ 
  with conf-s2 show ?thesis
  by (cases s2) (auto dest: conforms-RetD simp add: dom-def)
qed
qed
moreover note T resultT
ultimately
show  $\text{abupd } (\text{absorb Ret}) s3 :: \preceq (G, L) \wedge$ 
  (normal (abupd (absorb Ret) s3)  $\longrightarrow$ 
     $G, L, \text{store } (\text{abupd } (\text{absorb Ret}) s3)$ 
     $\vdash \text{In1l } (\text{Body } D \text{ c}) \succ \text{In1 } (\text{the } (\text{locals } (\text{store } s2) \text{ Result})) :: \preceq T) \wedge$ 
    (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 =  $\langle \text{Norm } s :: \preceq (G, L) \rangle$  and
   $\text{wt} = \langle \llbracket \text{prg} = G, \text{cls} = \text{accC}, \text{lcl} = L \rrbracket \vdash \text{In2 } (\text{LVar } vn) :: T \rangle$ 
then obtain vnT where
   $\text{vnT}: L \text{ vn} = \text{Some } \text{vnT}$  and
   $T: T = \text{Inl } \text{vnT}$ 
  by (auto elim!: wt-elim-cases)
from conf-s vnT
have conf-fst:  $\text{locals } s \text{ vn} \neq \text{None} \longrightarrow G, s \vdash \text{fst } (\text{lvar } \text{vn } s) :: \preceq \text{vnT}$ 
  by (auto elim: conforms-localD [THEN wlconfD])
  simp add: lvar-def
moreover
from conf-s conf-fst vnT
have  $s \leq | \text{snd } (\text{lvar } \text{vn } s) \preceq \text{vnT} :: \preceq (G, L)$ 
  by (auto elim: conforms-lupd simp add: assign-conforms-def lvar-def)
moreover note conf-s T

```

ultimately

show $Norm\ s :: \preceq(G, L) \wedge$
 $(normal\ (Norm\ s) \longrightarrow$
 $G, L, store\ (Norm\ s) \vdash In2\ (LVar\ vn) \succ In2\ (lvar\ vn\ s) :: \preceq T) \wedge$
 $(error\text{-}free\ (Norm\ s) = error\text{-}free\ (Norm\ s))$

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 $eval\text{-}init = \langle G \vdash Norm\ s0 \rightarrow Init\ statDeclC \rightarrow s1 \rangle$
note $eval\text{-}e = \langle G \vdash s1 \rightarrow e \rightarrow a \rightarrow s2 \rangle$
note $fvar = \langle (v, s2') = fvar\ statDeclC\ stat\ fn\ a\ s2 \rangle$
note $check = \langle s3 = check\text{-}field\text{-}access\ G\ accC\ statDeclC\ fn\ stat\ a\ s2' \rangle$
note $hyp\text{-}init = \langle PROP\ ?TypeSafe\ (Norm\ s0)\ s1\ (In1r\ (Init\ statDeclC)) \Diamond \rangle$
note $hyp\text{-}e = \langle PROP\ ?TypeSafe\ s1\ s2\ (In1l\ e)\ (In1\ a) \rangle$
note $conf\text{-}s0 = \langle Norm\ s0 :: \preceq(G, L) \rangle$
note $wt = \langle \langle prg = G, cls = accC', lcl = L \rangle \vdash In2\ (\{accC, statDeclC, stat\}e..fn) :: T \rangle$
then obtain $statC\ f$ **where**

$wt\text{-}e: \langle prg = G, cls = accC, lcl = L \rangle \vdash e :: \text{Class}\ statC$ **and**

$accfield: accfield\ G\ accC\ statC\ fn = Some\ (statDeclC.f)$ **and**

$eq\text{-}accC\text{-}accC': accC = accC'$ **and**

$stat: stat = is\text{-}static\ f$ **and**

$T: T = (Inl\ (type\ f))$

by (*rule wt-elim-cases*) (*auto simp add: member-is-static-simp*)

from $FVar.premis\ eq\text{-}accC\text{-}accC'$

have $da\text{-}e: \langle prg = G, cls = accC, lcl = L \rangle$
 $\vdash (dom\ (locals\ (store\ ((Norm\ s0)::state)))) \gg In1l\ e \gg A$

by (*elim da-elim-cases*) *simp*

note $conf\text{-}s0$

moreover

from $wf\ wt\text{-}e$

have $iscls\text{-}statC: is\text{-}class\ G\ statC$

by (*auto dest: ty-expr-is-type type-is-class*)

with $wf\ accfield$

have $iscls\text{-}statDeclC: is\text{-}class\ G\ statDeclC$

by (*auto dest!: accfield-fields dest: fields-declC*)

hence $\langle prg = G, cls = accC, lcl = L \rangle \vdash (Init\ statDeclC) :: \checkmark$

by *simp*

moreover obtain I **where**

$\langle prg = G, cls = accC, lcl = L \rangle$

$\vdash dom\ (locals\ (store\ ((Norm\ s0)::state))) \gg In1r\ (Init\ statDeclC) \gg I$

by (*auto intro: da-Init [simplified] assigned.select-convs*)

ultimately

obtain $conf\text{-}s1: s1 :: \preceq(G, L)$ **and** $error\text{-}free\text{-}s1: error\text{-}free\ s1$

by (*rule hyp-init [elim-format]*) *simp*

obtain A' **where**

$\langle prg = G, cls = accC, lcl = L \rangle \vdash (dom\ (locals\ (store\ s1))) \gg In1l\ e \gg A'$

proof –

from $eval\text{-}init$

have $(dom\ (locals\ (store\ ((Norm\ s0)::state))))$
 $\subseteq (dom\ (locals\ (store\ s1)))$

by (*rule dom-locals-eval-mono-elim*)

with $da\text{-}e$ **show** *thesis*

by (*rule da-weakenE*) (*rule that*)

qed

with $conf\text{-}s1\ wt\text{-}e$

obtain $conf\text{-}s2: s2 :: \preceq(G, L)$ **and**

$conf\text{-}a: normal\ s2 \longrightarrow G, store\ s2 \vdash a :: \preceq Class\ statC$ **and**

$error\text{-}free\text{-}s2: error\text{-}free\ s2$

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' ::  $\leq$ (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'  $\implies$ 
   $G, \text{store } s2 \vdash_{\text{fst}} v :: \leq \text{type } f \wedge \text{store } s2' \leq |\text{snd } v| \leq \text{type } f :: \leq (G, L)$ 
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  $\vdash_{vv} :: \leq \text{type } f \wedge \text{store2}' \leq |vf| \leq \text{type } f :: \leq (G, L)$ 
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  $G \vdash \text{statC} \leq_C \text{statDeclC}$ 
    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  $\longrightarrow G, \text{store2} \vdash a :: \leq \text{Class statC}$ 
    by auto
  from fld wf iscls-statC
  show statDeclC  $\neq$  Object
    by (cases statDeclC = Object) (drule fields-declC, simp+)
  from c show class G statDeclC = Some c .
  from conf-s2 s2 show (x2, store2) ::  $\leq$ (G, L) by simp
  from eval-e s2 show snd s1  $\leq$  |store2| by (auto dest: eval-geat)
  from initd-statDeclC-s1 show initd 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 ::  $\leq$ (G, L)  $\wedge$ 

```

```

    (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.premis 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-geat)
    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.premis 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) ⊢ 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*:

$\llbracket G \vdash s - e \rightarrow v \rightarrow s'; \text{wf-prog } G; s :: \preceq (G, L); (\text{prg} = G, \text{cls} = C, \text{lcl} = L) \vdash e :: - T;$
 $(\text{prg} = G, \text{cls} = C, \text{lcl} = L) \vdash \text{dom } (\text{locals } (\text{store } s)) \gg \text{In1l } e \gg A \rrbracket$
 $\implies s' :: \preceq (G, L) \wedge (\text{normal } s' \longrightarrow G, \text{store } s' \vdash v :: \preceq T) \wedge$
 $(\text{error-free } s = \text{error-free } s')$
apply (*drule* (4) *eval-type-sound*)
apply *clarsimp*
done

corollary *evals-ts*:

$\llbracket G \vdash s - es \rightarrow vs \rightarrow s'; \text{wf-prog } G; s :: \preceq (G, L); (\text{prg} = G, \text{cls} = C, \text{lcl} = L) \vdash es :: Ts;$
 $(\text{prg} = G, \text{cls} = C, \text{lcl} = L) \vdash \text{dom } (\text{locals } (\text{store } s)) \gg \text{In3 } es \gg A \rrbracket$
 $\implies s' :: \preceq (G, L) \wedge (\text{normal } s' \longrightarrow \text{list-all2 } (\text{conf } G (\text{store } s')) \text{ vs } Ts) \wedge$
 $(\text{error-free } s = \text{error-free } s')$
apply (*drule* (4) *eval-type-sound*)
apply *clarsimp*
done

corollary *evar-ts*:

$\llbracket G \vdash s - v \rightarrow vf \rightarrow s'; \text{wf-prog } G; s :: \preceq (G, L); (\text{prg} = G, \text{cls} = C, \text{lcl} = L) \vdash v :: T;$
 $(\text{prg} = G, \text{cls} = C, \text{lcl} = L) \vdash \text{dom } (\text{locals } (\text{store } s)) \gg \text{In2 } v \gg A \rrbracket \implies$
 $s' :: \preceq (G, L) \wedge (\text{normal } s' \longrightarrow G, L, (\text{store } s') \vdash \text{In2 } v \gg \text{In2 } vf :: \preceq \text{Inl } T) \wedge$
 $(\text{error-free } s = \text{error-free } s')$
apply (*drule* (4) *eval-type-sound*)
apply *clarsimp*
done

theorem *exec-ts*:

$\llbracket G \vdash s - c \rightarrow s'; \text{wf-prog } G; s :: \preceq (G, L); (\text{prg} = G, \text{cls} = C, \text{lcl} = L) \vdash c :: \checkmark;$
 $(\text{prg} = G, \text{cls} = C, \text{lcl} = L) \vdash \text{dom } (\text{locals } (\text{store } s)) \gg \text{In1r } c \gg A \rrbracket$
 $\implies s' :: \preceq (G, L) \wedge (\text{error-free } s \longrightarrow \text{error-free } s')$
apply (*drule* (4) *eval-type-sound*)
apply *clarsimp*
done

lemma *wf-eval-Fin*:

assumes *wf*: $\text{wf-prog } G$
and *wt-c1*: $(\text{prg} = G, \text{cls} = C, \text{lcl} = L) \vdash \text{In1r } c1 :: \text{Inl } (\text{PrimT } \text{Void})$
and *da-c1*: $(\text{prg} = G, \text{cls} = C, \text{lcl} = L) \vdash \text{dom } (\text{locals } (\text{store } (\text{Norm } s0))) \gg \text{In1r } c1 \gg A$
and *conf-s0*: $\text{Norm } s0 :: \preceq (G, L)$
and *eval-c1*: $G \vdash \text{Norm } s0 - c1 \rightarrow (x1, s1)$
and *eval-c2*: $G \vdash \text{Norm } s1 - c2 \rightarrow s2$
and *s3*: $s3 = \text{abupd } (\text{abrupt-if } (x1 \neq \text{None}) x1) s2$
shows $G \vdash \text{Norm } s0 - c1 \text{ Finally } c2 \rightarrow s3$
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*: $G \vdash s0 \rightarrow (v, s1)$
and *wt*: $(\langle prg = G, cls = accC, lcl = L \rangle) \vdash t :: T$
and *da*: $(\langle prg = G, cls = accC, lcl = L \rangle) \vdash \text{dom} (\text{locals} (\text{store } s0)) \gg t \gg A$
and *wf*: *wf-prog* G
and *abrupt*: $\bigwedge s \ t \ \text{abr} \ L \ \text{acc}C \ T \ A.$

$$\begin{aligned} & \llbracket (\langle prg = G, cls = accC, lcl = L \rangle) \vdash t :: T; \\ & \quad (\langle prg = G, cls = accC, lcl = L \rangle) \vdash \text{dom} (\text{locals} (\text{store} (\text{Some } \text{abr}, s))) \gg t \gg A \\ & \rrbracket \implies P \ L \ \text{acc}C \ (\text{Some } \text{abr}, s) \ t \ (\text{undefined3 } t) (\text{Some } \text{abr}, s) \end{aligned}$$

and *skip*: $\bigwedge s \ L \ \text{acc}C. P \ L \ \text{acc}C \ (\text{Norm } s) \ \langle \text{Skip} \rangle_s \Diamond (\text{Norm } s)$
and *expr*: $\bigwedge e \ s0 \ s1 \ v \ L \ \text{acc}C \ eT \ E.$

$$\begin{aligned} & \llbracket (\langle prg = G, cls = accC, lcl = L \rangle) \vdash e :: -eT; \\ & \quad (\langle prg = G, cls = accC, lcl = L \rangle) \\ & \quad \vdash \text{dom} (\text{locals} (\text{store} ((\text{Norm } s0)::\text{state}))) \gg \langle e \rangle_e \gg E; \\ & \quad P \ L \ \text{acc}C \ (\text{Norm } s0) \ \langle e \rangle_e \ [v]_e \ s1 \rrbracket \\ & \implies P \ L \ \text{acc}C \ (\text{Norm } s0) \ \langle \text{Expr } e \rangle_s \Diamond s1 \end{aligned}$$

and *lab*: $\bigwedge c \ l \ s0 \ s1 \ L \ \text{acc}C \ C.$

$$\begin{aligned} & \llbracket (\langle prg = G, cls = accC, lcl = L \rangle) \vdash c :: \sqrt{}; \\ & \quad (\langle prg = G, cls = accC, lcl = L \rangle) \\ & \quad \vdash \text{dom} (\text{locals} (\text{store} ((\text{Norm } s0)::\text{state}))) \gg \langle c \rangle_s \gg C; \\ & \quad P \ L \ \text{acc}C \ (\text{Norm } s0) \ \langle c \rangle_s \Diamond s1 \rrbracket \\ & \implies P \ L \ \text{acc}C \ (\text{Norm } s0) \ \langle l \cdot c \rangle_s \Diamond (\text{abupd } (\text{absorb } l) \ s1) \end{aligned}$$

and *comp*: $\bigwedge c1 \ c2 \ s0 \ s1 \ s2 \ L \ \text{acc}C \ C1.$

$$\begin{aligned} & \llbracket G \vdash \text{Norm } s0 \rightarrow c1 \rightarrow s1; G \vdash s1 \rightarrow c2 \rightarrow s2; \\ & \quad (\langle prg = G, cls = accC, lcl = L \rangle) \vdash c1 :: \sqrt{}; \\ & \quad (\langle prg = G, cls = accC, lcl = L \rangle) \vdash c2 :: \sqrt{}; \\ & \quad (\langle prg = G, cls = accC, lcl = L \rangle) \vdash \\ & \quad \text{dom} (\text{locals} (\text{store} ((\text{Norm } s0)::\text{state}))) \gg \langle c1 \rangle_s \gg C1; \\ & \quad P \ L \ \text{acc}C \ (\text{Norm } s0) \ \langle c1 \rangle_s \Diamond s1; \\ & \quad \bigwedge Q. \llbracket \text{normal } s1; \\ & \quad \quad \bigwedge C2. \llbracket (\langle prg = G, cls = accC, lcl = L \rangle) \\ & \quad \quad \vdash \text{dom} (\text{locals} (\text{store } s1)) \gg \langle c2 \rangle_s \gg C2; \\ & \quad \quad P \ L \ \text{acc}C \ s1 \ \langle c2 \rangle_s \Diamond s2 \rrbracket \implies Q \\ & \rrbracket \implies Q \\ & \implies P \ L \ \text{acc}C \ (\text{Norm } s0) \ \langle c1;; c2 \rangle_s \Diamond s2 \end{aligned}$$

and *if*: $\bigwedge b \ c1 \ c2 \ e \ s0 \ s1 \ s2 \ L \ \text{acc}C \ E.$

$$\begin{aligned} & \llbracket G \vdash \text{Norm } s0 \rightarrow e \rightarrow b \rightarrow s1; \\ & \quad G \vdash s1 \rightarrow (\text{if the-Bool } b \text{ then } c1 \text{ else } c2) \rightarrow s2; \\ & \quad (\langle prg = G, cls = accC, lcl = L \rangle) \vdash e :: -\text{PrimT Boolean}; \\ & \quad (\langle prg = G, cls = accC, lcl = L \rangle) \vdash (\text{if the-Bool } b \text{ then } c1 \text{ else } c2) :: \sqrt{}; \\ & \quad (\langle prg = G, cls = accC, lcl = L \rangle) \vdash \\ & \quad \text{dom} (\text{locals} (\text{store} ((\text{Norm } s0)::\text{state}))) \gg \langle e \rangle_e \gg E; \\ & \quad P \ L \ \text{acc}C \ (\text{Norm } s0) \ \langle e \rangle_e \ [b]_e \ s1; \\ & \quad \bigwedge Q. \llbracket \text{normal } s1; \\ & \quad \quad \bigwedge C. \llbracket (\langle prg = G, cls = accC, lcl = L \rangle) \vdash (\text{dom} (\text{locals} (\text{store } s1))) \\ & \quad \quad \gg \langle \text{if the-Bool } b \text{ then } c1 \text{ else } c2 \rangle_s \gg C; \\ & \quad \quad P \ L \ \text{acc}C \ s1 \ \langle \text{if the-Bool } b \text{ then } c1 \text{ else } c2 \rangle_s \Diamond s2 \\ & \quad \rrbracket \implies Q \\ & \rrbracket \implies Q \\ & \implies P \ L \ \text{acc}C \ (\text{Norm } s0) \ \langle \text{If}(e) \ c1 \ \text{Else } c2 \rangle_s \Diamond s2 \end{aligned}$$

shows $P \ L \ \text{acc}C \ s0 \ t \ v \ s1$

proof –

```

note inj-term-simps [simp]
from eval
have  $\bigwedge L \text{ acc} C \ T \ A. \llbracket (\text{prg} = G, \text{cls} = \text{acc} C, \text{lcl} = L) \vdash t :: T; \quad$ 
 $\quad (\text{prg} = G, \text{cls} = \text{acc} C, \text{lcl} = L) \vdash \text{dom} (\text{locals} (\text{store } s0)) \gg t \gg A \rrbracket$ 
 $\implies P \ L \ \text{acc} C \ s0 \ t \ v \ s1 \ (\text{is } PROP \ ?Hyp \ s0 \ t \ v \ s1)$ 
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 acc C T A)
  from Expr.prems obtain eT where
     $\llbracket \text{prg} = G, \text{cls} = \text{acc} C, \text{lcl} = L \rrbracket \vdash e :: -eT$ 
    by (elim wt-elim-cases)
  moreover
  from Expr.prems obtain E where
     $\llbracket \text{prg} = G, \text{cls} = \text{acc} C, \text{lcl} = L \rrbracket \vdash \text{dom} (\text{locals} (\text{store} ((\text{Norm } s0) :: \text{state}))) \gg \langle e \rangle_e \gg E$ 
    by (elim da-elim-cases) simp
  moreover from calculation
  have  $P \ L \ \text{acc} C \ (\text{Norm } s0) \ \langle e \rangle_e \ [v]_e \ s1$ 
    by (rule Expr.hyps)
  ultimately show ?case
    by (rule expr)
next
  case (Lab s0 c s1 l L acc C T A)
  from Lab.prems
  have  $\llbracket \text{prg} = G, \text{cls} = \text{acc} C, \text{lcl} = L \rrbracket \vdash c :: \surd$ 
    by (elim wt-elim-cases)
  moreover
  from Lab.prems obtain C where
     $\llbracket \text{prg} = G, \text{cls} = \text{acc} C, \text{lcl} = L \rrbracket \vdash \text{dom} (\text{locals} (\text{store} ((\text{Norm } s0) :: \text{state}))) \gg \langle c \rangle_s \gg C$ 
    by (elim da-elim-cases) simp
  moreover from calculation
  have  $P \ L \ \text{acc} C \ (\text{Norm } s0) \ \langle c \rangle_s \ \Diamond \ s1$ 
    by (rule Lab.hyps)
  ultimately show ?case
    by (rule lab)
next
  case (Comp s0 c1 s1 c2 s2 L acc C T A)
  note eval-c1 =  $\langle G \vdash \text{Norm } s0 \ -c1 \rightarrow s1 \rangle$ 
  note eval-c2 =  $\langle G \vdash s1 \ -c2 \rightarrow s2 \rangle$ 
  from Comp.prems obtain
    wt-c1:  $\llbracket \text{prg} = G, \text{cls} = \text{acc} C, \text{lcl} = L \rrbracket \vdash c1 :: \surd$  and
    wt-c2:  $\llbracket \text{prg} = G, \text{cls} = \text{acc} C, \text{lcl} = L \rrbracket \vdash c2 :: \surd$ 
    by (elim wt-elim-cases)
  from Comp.prems
  obtain C1 C2
    where da-c1:  $\llbracket \text{prg} = G, \text{cls} = \text{acc} C, \text{lcl} = L \rrbracket \vdash$ 
 $\quad \text{dom} (\text{locals} (\text{store} ((\text{Norm } s0) :: \text{state}))) \gg \langle c1 \rangle_s \gg C1$  and
    da-c2:  $\llbracket \text{prg} = G, \text{cls} = \text{acc} C, \text{lcl} = L \rrbracket \vdash \text{norm } C1 \gg \langle c2 \rangle_s \gg C2$ 
    by (elim da-elim-cases) simp
  from wt-c1 da-c1
  have P-c1:  $P \ L \ \text{acc} C \ (\text{Norm } s0) \ \langle c1 \rangle_s \ \Diamond \ s1$ 
    by (rule Comp.hyps)
  have thesis
    if normal-s1: normal s1
    and elim:  $\bigwedge C2'. \llbracket \text{prg} = G, \text{cls} = \text{acc} C, \text{lcl} = L \rrbracket \vdash \text{dom} (\text{locals} (\text{store } s1)) \gg \langle c2 \rangle_s \gg C2';$ 

```

```

    P L accC s1 ⟨c2⟩s ◇ s2]] ⇒ thesis
  for thesis
  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 ⟨c2⟩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.prem
    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.prem obtain E C where
      da-e: (⟦prg=G, cls=accC, lcl=L⟧) ⊢ dom (locals (store ((Norm s0)::state)))
        » ⟨e⟩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) ⟨e⟩e [b]e s1
      by (rule If.hyps)
    have thesis
      if normal-s1: normal s1
      and 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
        ] ⇒ thesis
    for thesis
    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')
      qed
    qed
  qed

```

```

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 ⟨if the-Bool b then c1 else c2⟩s ◇ 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
  (⟦-⟧0 -> -> ⟦(-, -')⟧0 [61,61,80,61,0,0] 60)
and evaln :: [prog, state, var, vvar, nat, state] ⇒ bool
  (⟦-⟧0 -> -> ⟦-⟧0 [61,61,90,61,61,61] 60)
and eval-n :: [prog, state, expr, val, nat, state] ⇒ bool
  (⟦-⟧0 -> -> ⟦-⟧0 [61,61,80,61,61,61] 60)
and evalsn :: [prog, state, expr list, val list, nat, state] ⇒ bool
  (⟦-⟧0 -> -> ⟦-⟧0 [61,61,61,61,61,61] 60)
and execn :: [prog, state, stmt, nat, state] ⇒ bool
  (⟦-⟧0 -> -> ⟦-⟧0 [61,61,65, 61,61] 60)
for G :: prog
where

```

```

  G ⊢ s - c - n → s' ≡ G ⊢ s - In1r c > - n → (◇ c, s')
| G ⊢ s - e > v - n → s' ≡ G ⊢ s - In1l e > - n → (In1 v, s')
| G ⊢ s - e > vf - n → s' ≡ G ⊢ s - In2 e > - n → (In2 vf, s')
| G ⊢ s - e > v - n → s' ≡ G ⊢ s - In3 e > - n → (In3 v, s')

```

— propagation of abrupt completion

```

| Abrupt: G ⊢ (Some xc, s) - t > - n → (undefined3 t, (Some xc, s))

```

— evaluation of variables

```

| LVar: G ⊢ Norm s - LVar vn => lvar vn s - n → Norm s

```

| *FVar*: $\llbracket G \vdash \text{Norm } s0 \text{ -Init statDeclC -} n \rightarrow s1; G \vdash s1 \text{ -} e \text{ -} \succ a \text{ -} n \rightarrow s2;$
 $(v, s2') = \text{fvar statDeclC stat fn } a \text{ } s2;$
 $s3 = \text{check-field-access } G \text{ accC statDeclC fn stat } a \text{ } s2 \rrbracket \implies$
 $G \vdash \text{Norm } s0 \text{ -}\{accC, statDeclC, stat\}e..fn \text{ -} \succ v \text{ -} n \rightarrow s3$

| *AVar*: $\llbracket G \vdash \text{Norm } s0 \text{ -} e1 \text{ -} \succ a \text{ -} n \rightarrow s1; G \vdash s1 \text{ -} e2 \text{ -} \succ i \text{ -} n \rightarrow s2;$
 $(v, s2') = \text{avar } G \text{ } i \text{ } a \text{ } s2 \rrbracket \implies$
 $G \vdash \text{Norm } s0 \text{ -} e1.[e2] \text{ -} \succ v \text{ -} n \rightarrow s2'$

— evaluation of expressions

| *NewC*: $\llbracket G \vdash \text{Norm } s0 \text{ -Init } C \text{ -} n \rightarrow s1;$
 $G \vdash s1 \text{ -} \text{halloc } (C \text{Inst } C) \text{ -} \succ a \rightarrow s2 \rrbracket \implies$
 $G \vdash \text{Norm } s0 \text{ -NewC } C \text{ -} \succ \text{Addr } a \text{ -} n \rightarrow s2$

| *NewA*: $\llbracket G \vdash \text{Norm } s0 \text{ -init-comp-ty } T \text{ -} n \rightarrow s1; G \vdash s1 \text{ -} e \text{ -} \succ i' \text{ -} n \rightarrow s2;$
 $G \vdash \text{abupd } (\text{check-neg } i') \text{ } s2 \text{ -halloc } (\text{Arr } T \text{ } (\text{the-Intg } i')) \text{ -} \succ a \rightarrow s3 \rrbracket \implies$
 $G \vdash \text{Norm } s0 \text{ -New } T[e] \text{ -} \succ \text{Addr } a \text{ -} n \rightarrow s3$

| *Cast*: $\llbracket G \vdash \text{Norm } s0 \text{ -} e \text{ -} \succ v \text{ -} n \rightarrow s1;$
 $s2 = \text{abupd } (\text{raise-if } (\neg G, \text{snd } s1 \vdash v \text{ fits } T) \text{ } \text{ClassCast}) \text{ } s1 \rrbracket \implies$
 $G \vdash \text{Norm } s0 \text{ -Cast } T \text{ } e \text{ -} \succ v \text{ -} n \rightarrow s2$

| *Inst*: $\llbracket G \vdash \text{Norm } s0 \text{ -} e \text{ -} \succ v \text{ -} n \rightarrow s1;$
 $b = (v \neq \text{Null} \wedge G, \text{store } s1 \vdash v \text{ fits } \text{RefT } T) \rrbracket \implies$
 $G \vdash \text{Norm } s0 \text{ -} e \text{ InstOf } T \text{ -} \succ \text{Bool } b \text{ -} n \rightarrow s1$

| *Lit*: $G \vdash \text{Norm } s \text{ -Lit } v \text{ -} \succ v \text{ -} n \rightarrow \text{Norm } s$

| *UnOp*: $\llbracket G \vdash \text{Norm } s0 \text{ -} e \text{ -} \succ v \text{ -} n \rightarrow s1 \rrbracket$
 $\implies G \vdash \text{Norm } s0 \text{ -UnOp } \text{unop } e \text{ -} \succ (\text{eval-unop } \text{unop } v) \text{ -} n \rightarrow s1$

| *BinOp*: $\llbracket G \vdash \text{Norm } s0 \text{ -} e1 \text{ -} \succ v1 \text{ -} n \rightarrow s1;$
 $G \vdash s1 \text{ -} (\text{if need-second-arg binop } v1 \text{ then } (\text{In1l } e2) \text{ else } (\text{In1r Skip}))$
 $\text{ -} \succ \text{ -} n \rightarrow (\text{In1 } v2, s2) \rrbracket$
 $\implies G \vdash \text{Norm } s0 \text{ -BinOp } \text{binop } e1 \text{ } e2 \text{ -} \succ (\text{eval-binop } \text{binop } v1 \text{ } v2) \text{ -} n \rightarrow s2$

| *Super*: $G \vdash \text{Norm } s \text{ -Super -} \succ \text{val-this } s \text{ -} n \rightarrow \text{Norm } s$

| *Acc*: $\llbracket G \vdash \text{Norm } s0 \text{ -} va \text{ -} \succ (v, f) \text{ -} n \rightarrow s1 \rrbracket \implies$
 $G \vdash \text{Norm } s0 \text{ -Acc } va \text{ -} \succ v \text{ -} n \rightarrow s1$

| *Ass*: $\llbracket G \vdash \text{Norm } s0 \text{ -} va \text{ -} \succ (w, f) \text{ -} n \rightarrow s1;$
 $G \vdash s1 \text{ -} e \text{ -} \succ v \text{ -} n \rightarrow s2 \rrbracket \implies$
 $G \vdash \text{Norm } s0 \text{ -} va \text{ -} \text{:=} e \text{ -} \succ v \text{ -} n \rightarrow \text{assign } f \text{ } v \text{ } s2$

| *Cond*: $\llbracket G \vdash \text{Norm } s0 \text{ -} e0 \text{ -} \succ b \text{ -} n \rightarrow s1;$
 $G \vdash s1 \text{ -} (\text{if the-Bool } b \text{ then } e1 \text{ else } e2) \text{ -} \succ v \text{ -} n \rightarrow s2 \rrbracket \implies$
 $G \vdash \text{Norm } s0 \text{ -} e0 \text{ ? } e1 : e2 \text{ -} \succ v \text{ -} n \rightarrow s2$

| *Call*:
 $\llbracket G \vdash \text{Norm } s0 \text{ -} e \text{ -} \succ a' \text{ -} n \rightarrow s1; G \vdash s1 \text{ -} \text{args} \text{ -} \succ vs \text{ -} n \rightarrow s2;$
 $D = \text{invocation-declclass } G \text{ mode } (\text{store } s2) \text{ } a' \text{ } \text{statT } (\text{name} = mn, \text{parTs} = pTs);$
 $s3 = \text{init-lvars } G \text{ } D \text{ } (\text{name} = mn, \text{parTs} = pTs) \text{ mode } a' \text{ } vs \text{ } s2;$
 $s3' = \text{check-method-access } G \text{ accC statT mode } (\text{name} = mn, \text{parTs} = pTs) \text{ } a' \text{ } s3;$

$$\begin{array}{l}
G \vdash s3' - \text{Methd } D \ (\llbracket \text{name} = mn, \text{parTs} = pTs \rrbracket) - \succ v - n \rightarrow s4 \\
\rrbracket \\
\implies \\
G \vdash \text{Norm } s0 - \{accC, statT, mode\} e \cdot mn(\{pTs\} args) - \succ v - n \rightarrow (\text{restore-lvars } s2 \ s4) \\
| \text{Methd:} \llbracket G \vdash \text{Norm } s0 - \text{body } G \ D \ \text{sig} - \succ v - n \rightarrow s1 \rrbracket \implies \\
\quad G \vdash \text{Norm } s0 - \text{Methd } D \ \text{sig} - \succ v - \text{Suc } n \rightarrow s1 \\
| \text{Body:} \llbracket G \vdash \text{Norm } s0 - \text{Init } D - n \rightarrow s1; G \vdash s1 - c - n \rightarrow s2; \\
\quad s3 = (\text{if } (\exists l. \text{abrupt } s2 = \text{Some } (\text{Jump } (\text{Break } l))) \vee \\
\quad \quad \text{abrupt } s2 = \text{Some } (\text{Jump } (\text{Cont } l))) \\
\quad \text{then } \text{abupd } (\lambda x. \text{Some } (\text{Error CrossMethodJump})) \ s2 \\
\quad \text{else } s2 \rrbracket \implies \\
G \vdash \text{Norm } s0 - \text{Body } D \ c \\
\quad - \succ \text{the } (\text{locals } (\text{store } s2) \ \text{Result}) - n \rightarrow \text{abupd } (\text{absorb Ret}) \ s3 \\
- \text{ evaluation of expression lists} \\
| \text{Nil:} \\
\quad G \vdash \text{Norm } s0 - [] \dot{=} \succ [] - n \rightarrow \text{Norm } s0 \\
| \text{Cons:} \llbracket G \vdash \text{Norm } s0 - e - \succ v - n \rightarrow s1; \\
\quad G \vdash \quad s1 - es \dot{=} \succ vs - n \rightarrow s2 \rrbracket \implies \\
\quad G \vdash \text{Norm } s0 - e \# es \dot{=} \succ v \# vs - n \rightarrow s2 \\
- \text{ execution of statements} \\
| \text{Skip:} \quad G \vdash \text{Norm } s - \text{Skip} - n \rightarrow \text{Norm } s \\
| \text{Expr:} \llbracket G \vdash \text{Norm } s0 - e - \succ v - n \rightarrow s1 \rrbracket \implies \\
\quad G \vdash \text{Norm } s0 - \text{Expr } e - n \rightarrow s1 \\
| \text{Lab:} \llbracket G \vdash \text{Norm } s0 - c - n \rightarrow s1 \rrbracket \implies \\
\quad G \vdash \text{Norm } s0 - l \cdot c - n \rightarrow \text{abupd } (\text{absorb } l) \ s1 \\
| \text{Comp:} \llbracket G \vdash \text{Norm } s0 - c1 - n \rightarrow s1; \\
\quad G \vdash \quad s1 - c2 - n \rightarrow s2 \rrbracket \implies \\
\quad G \vdash \text{Norm } s0 - c1;; c2 - n \rightarrow s2 \\
| \text{If:} \llbracket G \vdash \text{Norm } s0 - e - \succ b - n \rightarrow s1; \\
\quad G \vdash \quad s1 - (\text{if the-Bool } b \text{ then } c1 \text{ else } c2) - n \rightarrow s2 \rrbracket \implies \\
\quad G \vdash \text{Norm } s0 - \text{If}(e) \ c1 \ \text{Else } c2 - n \rightarrow s2 \\
| \text{Loop:} \llbracket G \vdash \text{Norm } s0 - e - \succ b - n \rightarrow s1; \\
\quad \text{if the-Bool } b \\
\quad \text{then } (G \vdash s1 - c - n \rightarrow s2 \wedge \\
\quad \quad G \vdash (\text{abupd } (\text{absorb } (\text{Cont } l)) \ s2) - l \cdot \text{While}(e) \ c - n \rightarrow s3) \\
\quad \text{else } s3 = s1 \rrbracket \implies \\
\quad G \vdash \text{Norm } s0 - l \cdot \text{While}(e) \ c - n \rightarrow s3 \\
| \text{Jmp:} \ G \vdash \text{Norm } s - \text{Jmp } j - n \rightarrow (\text{Some } (\text{Jump } j), \ s) \\
| \text{Throw:} \llbracket G \vdash \text{Norm } s0 - e - \succ a' - n \rightarrow s1 \rrbracket \implies \\
\quad G \vdash \text{Norm } s0 - \text{Throw } e - n \rightarrow \text{abupd } (\text{throw } a') \ s1 \\
| \text{Try:} \llbracket G \vdash \text{Norm } s0 - c1 - n \rightarrow s1; G \vdash s1 - s\text{xalloc} \rightarrow s2; \\
\quad \text{if } G, s2 \vdash \text{catch } tn \text{ then } G \vdash \text{new-xcpt-var } vn \ s2 - c2 - n \rightarrow s3 \text{ else } s3 = s2 \rrbracket \\
\implies
\end{array}$$

$$G \vdash \text{Norm } s0 \text{ --Try } c1 \text{ Catch}(tn \text{ } vn) \text{ } c2 \text{ --} n \rightarrow s3$$

| *Fin*: $\llbracket G \vdash \text{Norm } s0 \text{ --} c1 \text{ --} n \rightarrow (x1, s1);$
 $G \vdash \text{Norm } s1 \text{ --} c2 \text{ --} n \rightarrow s2;$
 $s3 = (\text{if } (\exists \text{ err. } x1 = \text{Some } (\text{Error err}))$
 $\text{then } (x1, s1)$
 $\text{else } \text{abupd } (\text{abrupt-if } (x1 \neq \text{None}) \text{ } x1) \text{ } s2) \rrbracket \implies$
 $G \vdash \text{Norm } s0 \text{ --} c1 \text{ Finally } c2 \text{ --} n \rightarrow s3$

| *Init*: $\llbracket \text{the } (\text{class } G \text{ } C) = c;$
 $\text{if } \text{inited } C \text{ (globs } s0) \text{ then } s3 = \text{Norm } s0$
 $\text{else } (G \vdash \text{Norm } (\text{init-class-obj } G \text{ } C \text{ } s0)$
 $\text{--} (\text{if } C = \text{Object then Skip else Init (super } c)) \text{ --} n \rightarrow s1 \wedge$
 $G \vdash \text{set-lvars Map.empty } s1 \text{ --init } c \text{ --} n \rightarrow s2 \wedge$
 $s3 = \text{restore-lvars } s1 \text{ } s2) \rrbracket$
 \implies
 $G \vdash \text{Norm } s0 \text{ --Init } C \text{ --} n \rightarrow s3$

monos

if-bool-eq-conj

declare *if-split* $[split \text{ del}]$ *if-split-asm* $[split \text{ del}]$
 $\text{option.split } [split \text{ del}] \text{ option.split-asm } [split \text{ del}]$
 $\text{not-None-eq } [simp \text{ del}]$
 $\text{split-paired-All } [simp \text{ del}] \text{ split-paired-Ex } [simp \text{ del}]$
setup $\langle \text{map-theory-simpset } (fn \text{ ctxt } => \text{ctxt } |> \text{Simplifier.del-loop split-all-tac}) \rangle$

inductive-cases *evaln-cases*: $G \vdash s \text{ --} t \text{ --} n \rightarrow (v, s')$

inductive-cases *evaln-elim-cases*:

$G \vdash (\text{Some } xc, s) \text{ --} t$	$\succ \text{--} n \rightarrow (v, s')$
$G \vdash \text{Norm } s \text{ --In1r Skip}$	$\succ \text{--} n \rightarrow (x, s')$
$G \vdash \text{Norm } s \text{ --In1r (Jmp } j)$	$\succ \text{--} n \rightarrow (x, s')$
$G \vdash \text{Norm } s \text{ --In1r } (l \cdot c)$	$\succ \text{--} n \rightarrow (x, s')$
$G \vdash \text{Norm } s \text{ --In3 } (\llbracket \rrbracket)$	$\succ \text{--} n \rightarrow (v, s')$
$G \vdash \text{Norm } s \text{ --In3 } (e \# es)$	$\succ \text{--} n \rightarrow (v, s')$
$G \vdash \text{Norm } s \text{ --In1l (Lit } w)$	$\succ \text{--} n \rightarrow (v, s')$
$G \vdash \text{Norm } s \text{ --In1l (UnOp unop } e)$	$\succ \text{--} n \rightarrow (v, s')$
$G \vdash \text{Norm } s \text{ --In1l (BinOp binop } e1 \text{ } e2)$	$\succ \text{--} n \rightarrow (v, s')$
$G \vdash \text{Norm } s \text{ --In2 (LVar } vn)$	$\succ \text{--} n \rightarrow (v, s')$
$G \vdash \text{Norm } s \text{ --In1l (Cast } T \text{ } e)$	$\succ \text{--} n \rightarrow (v, s')$
$G \vdash \text{Norm } s \text{ --In1l } (e \text{ InstOf } T)$	$\succ \text{--} n \rightarrow (v, s')$
$G \vdash \text{Norm } s \text{ --In1l (Super)}$	$\succ \text{--} n \rightarrow (v, s')$
$G \vdash \text{Norm } s \text{ --In1l (Acc } va)$	$\succ \text{--} n \rightarrow (v, s')$
$G \vdash \text{Norm } s \text{ --In1r (Expr } e)$	$\succ \text{--} n \rightarrow (x, s')$
$G \vdash \text{Norm } s \text{ --In1r } (c1;; c2)$	$\succ \text{--} n \rightarrow (x, s')$
$G \vdash \text{Norm } s \text{ --In1l (Methd } C \text{ sig)}$	$\succ \text{--} n \rightarrow (x, s')$
$G \vdash \text{Norm } s \text{ --In1l (Body } D \text{ } c)$	$\succ \text{--} n \rightarrow (x, s')$
$G \vdash \text{Norm } s \text{ --In1l } (e0 \text{ ? } e1 : e2)$	$\succ \text{--} n \rightarrow (v, s')$
$G \vdash \text{Norm } s \text{ --In1r (If(e) } c1 \text{ Else } c2)$	$\succ \text{--} n \rightarrow (x, s')$
$G \vdash \text{Norm } s \text{ --In1r } (l \cdot \text{While}(e) \text{ } c)$	$\succ \text{--} n \rightarrow (x, s')$
$G \vdash \text{Norm } s \text{ --In1r } (c1 \text{ Finally } c2)$	$\succ \text{--} n \rightarrow (x, s')$
$G \vdash \text{Norm } s \text{ --In1r (Throw } e)$	$\succ \text{--} n \rightarrow (x, s')$
$G \vdash \text{Norm } s \text{ --In1l (NewC } C)$	$\succ \text{--} n \rightarrow (v, s')$
$G \vdash \text{Norm } s \text{ --In1l (New } T[e])$	$\succ \text{--} n \rightarrow (v, s')$
$G \vdash \text{Norm } s \text{ --In1l (Ass } va \text{ } e)$	$\succ \text{--} n \rightarrow (v, s')$
$G \vdash \text{Norm } s \text{ --In1r (Try } c1 \text{ Catch}(tn \text{ } vn) \text{ } c2)$	$\succ \text{--} n \rightarrow (x, s')$
$G \vdash \text{Norm } s \text{ --In2 } (\{accC, statDeclC, stat\}e..fn)$	$\succ \text{--} n \rightarrow (v, s')$
$G \vdash \text{Norm } s \text{ --In2 } (e1.[e2])$	$\succ \text{--} n \rightarrow (v, s')$

$$\begin{array}{l} G \vdash \text{Norm } s - \text{In1l } (\{accC, statT, mode\} e \cdot mn(\{pT\}p)) \succ -n \rightarrow (v, s') \\ G \vdash \text{Norm } s - \text{In1r } (\text{Init } C) \succ -n \rightarrow (x, s') \end{array}$$

declare *if-split* [split] *if-split-asm* [split]
option.split [split] *option.split-asm* [split]
not-None-eq [simp]
split-paired-All [simp] *split-paired-Ex* [simp]
declaration $\langle K (\text{Simplifier.map-ss } (fn \text{ ss } => \text{ ss } |> \text{Simplifier.add-loop } (\text{split-all-tac}, \text{split-all-tac}))) \rangle$

lemma *evaln-Inj-elim*: $G \vdash s - t \succ -n \rightarrow (w, s') \implies \text{case } t \text{ of } \text{In1 } ec \implies$
 $(\text{case } ec \text{ of } \text{Inl } e \implies (\exists v. w = \text{In1 } v) \mid \text{Inr } c \implies w = \Diamond)$
 $\mid \text{In2 } e \implies (\exists v. w = \text{In2 } v) \mid \text{In3 } e \implies (\exists v. w = \text{In3 } v)$

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*: $G \vdash s - \text{In1l } t \succ -n \rightarrow (w, s') = (\exists v. w = \text{In1 } v \wedge G \vdash s - t \succ v -n \rightarrow s')$
by (*auto*, *frule evaln-Inj-elim*, *auto*)

lemma *evaln-var-eq*: $G \vdash s - \text{In2 } t \succ -n \rightarrow (w, s') = (\exists vf. w = \text{In2 } vf \wedge G \vdash s - t = \succ vf -n \rightarrow s')$
by (*auto*, *frule evaln-Inj-elim*, *auto*)

lemma *evaln-exprs-eq*: $G \vdash s - \text{In3 } t \succ -n \rightarrow (w, s') = (\exists vs. w = \text{In3 } vs \wedge G \vdash s - t \dot{=} \succ vs -n \rightarrow s')$
by (*auto*, *frule evaln-Inj-elim*, *auto*)

lemma *evaln-stmt-eq*: $G \vdash s - \text{In1r } t \succ -n \rightarrow (w, s') = (w = \Diamond \wedge G \vdash s - t -n \rightarrow s')$
by (*auto*, *frule evaln-Inj-elim*, *auto*, *frule evaln-Inj-elim*, *auto*)

simplproc-setup *evaln-expr* ($G \vdash s - \text{In1l } t \succ -n \rightarrow (w, s')$) = \langle
 $K (K (fn \text{ ct } =>$
 $(\text{case } \text{Thm.term-of } \text{ct} \text{ of}$
 $(- \$ - \$ - \$ - \$ (Const - \$ -) \$ -) => \text{NONE}$
 $\mid - => \text{SOME } (mk\text{-meta-eq } @\{\text{thm evaln-expr-eq}\}))) \rangle$

simplproc-setup *evaln-var* ($G \vdash s - \text{In2 } t \succ -n \rightarrow (w, s')$) = \langle
 $K (K (fn \text{ ct } =>$
 $(\text{case } \text{Thm.term-of } \text{ct} \text{ of}$
 $(- \$ - \$ - \$ - \$ (Const - \$ -) \$ -) => \text{NONE}$
 $\mid - => \text{SOME } (mk\text{-meta-eq } @\{\text{thm evaln-var-eq}\}))) \rangle$

simplproc-setup *evaln-exprs* ($G \vdash s - \text{In3 } t \succ -n \rightarrow (w, s')$) = \langle
 $K (K (fn \text{ ct } =>$
 $(\text{case } \text{Thm.term-of } \text{ct} \text{ of}$
 $(- \$ - \$ - \$ - \$ (Const - \$ -) \$ -) => \text{NONE}$
 $\mid - => \text{SOME } (mk\text{-meta-eq } @\{\text{thm evaln-exprs-eq}\}))) \rangle$

simplproc-setup *evaln-stmt* ($G \vdash s - \text{In1r } t \succ -n \rightarrow (w, s')$) = \langle
 $K (K (fn \text{ ct } =>$
 $(\text{case } \text{Thm.term-of } \text{ct} \text{ of}$
 $(- \$ - \$ - \$ - \$ (Const - \$ -) \$ -) => \text{NONE}$
 $\mid - => \text{SOME } (mk\text{-meta-eq } @\{\text{thm evaln-stmt-eq}\}))) \rangle$

ML $\langle \text{ML-Thms.bind-thms } (\text{evaln-AbruptIs}, \text{sum3-instantiate } \text{context } @\{\text{thm evaln.Abrupt}\}) \rangle$

declare *evaln-AbruptIs* [intro!]

lemma *evaln-Callee*: $G \vdash \text{Norm } s - \text{In1l } (\text{Callee } l \ e) \succ -n \rightarrow (v, s') = \text{False}$

proof –

have *False*
if *eval*: $G \vdash s - t \succ -n \rightarrow (v, s')$
and *normal*: *normal* *s*
and *callee*: $t = \text{In1l } (\text{Callee } l \ e)$
for *s t v s'*
using *that* **by** *induct auto*
then show *?thesis*
by (*cases s'*) *fastforce*

qed

lemma *evaln-InsInitE*: $G \vdash \text{Norm } s - \text{In1l } (\text{InsInitE } c \ e) \succ -n \rightarrow (v, s') = \text{False}$

proof –

have *False*
if *eval*: $G \vdash s - t \succ -n \rightarrow (v, s')$
and *normal*: *normal* *s*
and *callee*: $t = \text{In1l } (\text{InsInitE } c \ e)$
for *s t v s'*
using *that* **by** *induct auto*
then show *?thesis*
by (*cases s'*) *fastforce*

qed

lemma *evaln-InsInitV*: $G \vdash \text{Norm } s - \text{In2 } (\text{InsInitV } c \ w) \succ -n \rightarrow (v, s') = \text{False}$

proof –

have *False*
if *eval*: $G \vdash s - t \succ -n \rightarrow (v, s')$
and *normal*: *normal* *s*
and *callee*: $t = \text{In2 } (\text{InsInitV } c \ w)$
for *s t v s'*
using *that* **by** *induct auto*
then show *?thesis*
by (*cases s'*) *fastforce*

qed

lemma *evaln-FinA*: $G \vdash \text{Norm } s - \text{In1r } (\text{FinA } a \ c) \succ -n \rightarrow (v, s') = \text{False}$

proof –

have *False*
if *eval*: $G \vdash s - t \succ -n \rightarrow (v, s')$
and *normal*: *normal* *s*
and *callee*: $t = \text{In1r } (\text{FinA } a \ c)$
for *s t v s'*
using *that* **by** *induct auto*
then show *?thesis*
by (*cases s'*) *fastforce*

qed

lemma *evaln-abrupt-lemma*: $G \vdash s - e \succ -n \rightarrow (v, s') \implies$

$\text{fst } s = \text{Some } xc \longrightarrow s' = s \wedge v = \text{undefined3 } e$

apply (*erule evaln-cases* , *auto*)

done

lemma *evaln-abrupt*:

$\bigwedge s'. G \vdash (\text{Some } xc, s) - e \succ -n \rightarrow (w, s') = (s' = (\text{Some } xc, s) \wedge$
 $w = \text{undefined3 } e \wedge G \vdash (\text{Some } xc, s) - e \succ -n \rightarrow (\text{undefined3 } e, (\text{Some } xc, s)))$

apply *auto*

apply (*frule evaln-abrupt-lemma*, *auto*)
done

simproc-setup *evaln-abrupt* ($G \vdash (Some\ xc, s) \multimap e \multimap n \rightarrow (w, s')$) = \langle
 $K\ (K\ (fn\ ct \Rightarrow$
 $(case\ Thm.term-of\ ct\ of$
 $(- \$ - \$ - \$ - \$ - \$ - \$ (Const\ (\mathbf{const-name}\ \langle Pair \rangle, -) \$ (Const\ (\mathbf{const-name}\ \langle Some \rangle, -) \$ -) \$ -))$
 $\Rightarrow\ NONE$
 $| - \Rightarrow\ SOME\ (mk-meta-eq\ @\{thm\ evaln-abrupt\})))$
 \rangle

lemma *evaln-LitI*: $G \vdash s \multimap Lit\ v \multimap (if\ normal\ s\ then\ v\ else\ undefined) \multimap n \rightarrow s$
apply (*case-tac s*, *case-tac a = None*)
by (*auto intro!*: *evaln.Lit*)

lemma *CondI*:
 $\bigwedge s1. \llbracket G \vdash s \multimap e \multimap b \multimap n \rightarrow s1; G \vdash s1 \multimap (if\ the-Bool\ b\ then\ e1\ else\ e2) \multimap v \multimap n \rightarrow s2 \rrbracket \implies$
 $G \vdash s \multimap e\ ?\ e1 : e2 \multimap (if\ normal\ s1\ then\ v\ else\ undefined) \multimap n \rightarrow s2$
apply (*case-tac s*, *case-tac a = None*)
by (*auto intro!*: *evaln.Cond*)

lemma *evaln-SkipI* [*intro!*]: $G \vdash s \multimap Skip \multimap n \rightarrow s$
apply (*case-tac s*, *case-tac a = None*)
by (*auto intro!*: *evaln.Skip*)

lemma *evaln-ExprI*: $G \vdash s \multimap e \multimap v \multimap n \rightarrow s' \implies G \vdash s \multimap Expr\ e \multimap n \rightarrow s'$
apply (*case-tac s*, *case-tac a = None*)
by (*auto intro!*: *evaln.Expr*)

lemma *evaln-CompI*: $\llbracket G \vdash s \multimap c1 \multimap n \rightarrow s1; G \vdash s1 \multimap c2 \multimap n \rightarrow s2 \rrbracket \implies G \vdash s \multimap c1;;\ c2 \multimap n \rightarrow s2$
apply (*case-tac s*, *case-tac a = None*)
by (*auto intro!*: *evaln.Comp*)

lemma *evaln-IfI*:
 $\llbracket G \vdash s \multimap e \multimap v \multimap n \rightarrow s1; G \vdash s1 \multimap (if\ the-Bool\ v\ then\ c1\ else\ c2) \multimap n \rightarrow s2 \rrbracket \implies$
 $G \vdash s \multimap If(e)\ c1\ Else\ c2 \multimap n \rightarrow s2$
apply (*case-tac s*, *case-tac a = None*)
by (*auto intro!*: *evaln.If*)

lemma *evaln-SkipD* [*dest!*]: $G \vdash s \multimap Skip \multimap n \rightarrow s' \implies s' = s$
by (*erule evaln-cases*, *auto*)

lemma *evaln-Skip-eq* [*simp*]: $G \vdash s \multimap Skip \multimap n \rightarrow s' = (s = s')$
apply *auto*
done

evaln implies eval

lemma *evaln-eval*:
assumes *evaln*: $G \vdash s0 \multimap t \multimap n \rightarrow (v, s1)$
shows $G \vdash s0 \multimap t \multimap \rightarrow (v, s1)$
using *evaln*
proof (*induct*)
case (*Loop s0 e b n s1 c s2 l s3*)
note $\langle G \vdash Norm\ s0 \multimap e \multimap b \rightarrow s1 \rangle$
moreover
have *if the-Bool b*
then ($G \vdash s1 \multimap c \rightarrow s2$) \wedge
 $G \multimap abupd\ (absorb\ (Cont\ l))\ s2 \multimap l \cdot While(e)\ c \rightarrow s3$

```

      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 -xalloc→ 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 initd 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*: $\llbracket \text{Suc } n \leq m'; (\bigwedge m. n \leq m \implies P (\text{Suc } m)) \rrbracket \implies P m'$
apply (frule *Suc-le-D*)
apply *fast*
done

lemma *evaln-nonstrict* [rule-format (no-asm), elim]:
 $G \vdash s -t \succ -n \rightarrow (w, s') \implies \forall m. n \leq m \longrightarrow G \vdash s -t \succ -m \rightarrow (w, s')$
apply (erule *evaln.induct*)
apply (tactic ⟨ALLGOALS (EVERY' [strip-tac **context**,
 TRY o eresolve-tac **context** @ {thms *Suc-le-D-lemma*},
 REPEAT o smt-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*: $\llbracket G \vdash s1 -t1 \succ -n1 \rightarrow (w1, s1'); G \vdash s2 -t2 \succ -n2 \rightarrow (w2, s2') \rrbracket \implies$
 $G \vdash s1 -t1 \succ -\max n1 n2 \rightarrow (w1, s1') \wedge G \vdash s2 -t2 \succ -\max n1 n2 \rightarrow (w2, s2')$
by (*fast intro: max.cobounded1 max.cobounded2*)

corollary *evaln-max2E* [consumes 2]:
 $\llbracket G \vdash s1 -t1 \succ -n1 \rightarrow (w1, s1'); G \vdash s2 -t2 \succ -n2 \rightarrow (w2, s2') \rrbracket$
 $\llbracket G \vdash s1 -t1 \succ -\max n1 n2 \rightarrow (w1, s1'); G \vdash s2 -t2 \succ -\max n1 n2 \rightarrow (w2, s2') \rrbracket \implies P \rrbracket \implies P$
by (*drule* (1) *evaln-max2*) *simp*

lemma *evaln-max3*:
 $\llbracket G \vdash s1 -t1 \succ -n1 \rightarrow (w1, s1'); G \vdash s2 -t2 \succ -n2 \rightarrow (w2, s2'); G \vdash s3 -t3 \succ -n3 \rightarrow (w3, s3') \rrbracket \implies$
 $G \vdash s1 -t1 \succ -\max (\max n1 n2) n3 \rightarrow (w1, s1') \wedge$
 $G \vdash s2 -t2 \succ -\max (\max n1 n2) n3 \rightarrow (w2, s2') \wedge$
 $G \vdash s3 -t3 \succ -\max (\max n1 n2) n3 \rightarrow (w3, s3')$

```

apply (drule (1) evaln-max2, erule thin-rl)
apply (fast intro!: max.cobounded1 max.cobounded2)
done

```

corollary *evaln-max3E*:

```


$$\llbracket G \vdash s1 - t1 \succ - n1 \rightarrow (w1, s1'); G \vdash s2 - t2 \succ - n2 \rightarrow (w2, s2'); G \vdash s3 - t3 \succ - n3 \rightarrow (w3, s3') \rrbracket$$


$$\llbracket G \vdash s1 - t1 \succ - \max (\max n1 n2) n3 \rightarrow (w1, s1') \rrbracket$$


$$G \vdash s2 - t2 \succ - \max (\max n1 n2) n3 \rightarrow (w2, s2');$$


$$G \vdash s3 - t3 \succ - \max (\max n1 n2) n3 \rightarrow (w3, s3')$$


$$\rrbracket \Rightarrow P$$


$$\rrbracket \Rightarrow P$$

by (drule (2) evaln-max3) simp

```

lemma *le-max3I1*: $(n2::nat) \leq \max n1 (\max n2 n3)$

```

proof -
  have  $n2 \leq \max n2 n3$ 
    by (rule max.cobounded1)
  also
  have  $\max n2 n3 \leq \max n1 (\max n2 n3)$ 
    by (rule max.cobounded2)
  finally
  show ?thesis .
qed

```

lemma *le-max3I2*: $(n3::nat) \leq \max n1 (\max n2 n3)$

```

proof -
  have  $n3 \leq \max n2 n3$ 
    by (rule max.cobounded2)
  also
  have  $\max n2 n3 \leq \max n1 (\max n2 n3)$ 
    by (rule max.cobounded2)
  finally
  show ?thesis .
qed

```

declare $[[simproc\ del:\ wt\text{-}expr\ wt\text{-}var\ wt\text{-}exprs\ wt\text{-}stmt]]$

eval implies evaln

lemma *eval-evaln*:

```

  assumes eval:  $G \vdash s0 - t \succ \rightarrow (v, s1)$ 
  shows  $\exists n. G \vdash s0 - t \succ - n \rightarrow (v, s1)$ 
using eval
proof (induct)
  case (Abrupt xc s t)
  obtain n where
     $G \vdash (Some\ xc, s) - t \succ - n \rightarrow (undefined3\ t, (Some\ xc, s))$ 
    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
     $G \vdash Norm\ s0 - e - \succ v - n \rightarrow s1$ 
    by (iprover)
  then have  $G \vdash Norm\ s0 - Expr\ e - n \rightarrow s1$ 

```

```

    by (rule evaln.Expr)
  then show ?case ..
next
  case (Lab s0 c s1 l)
  then obtain n where
     $G \vdash \text{Norm } s0 \text{ } -c-n \rightarrow s1$ 
    by (iprover)
  then have  $G \vdash \text{Norm } s0 \text{ } -l \cdot c-n \rightarrow \text{abupd } (\text{absorb } l) \text{ } s1$ 
    by (rule evaln.Lab)
  then show ?case ..
next
  case (Comp s0 c1 s1 c2 s2)
  then obtain n1 n2 where
     $G \vdash \text{Norm } s0 \text{ } -c1-n1 \rightarrow s1$ 
     $G \vdash s1 \text{ } -c2-n2 \rightarrow s2$ 
    by (iprover)
  then have  $G \vdash \text{Norm } s0 \text{ } -c1;; c2-\text{max } n1 \text{ } n2 \rightarrow s2$ 
    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
     $G \vdash \text{Norm } s0 \text{ } -e-\succ b-n1 \rightarrow s1$ 
     $G \vdash s1 \text{ } -(if \text{ the-Bool } b \text{ then } c1 \text{ else } c2)-n2 \rightarrow s2$ 
    by (iprover)
  then have  $G \vdash \text{Norm } s0 \text{ } -If(e) \text{ } c1 \text{ Else } c2-\text{max } n1 \text{ } n2 \rightarrow s2$ 
    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
     $G \vdash \text{Norm } s0 \text{ } -e-\succ b-n1 \rightarrow s1$ 
    by (iprover)
  moreover from Loop.hyps obtain n2 where
    if the-Bool b
      then ( $G \vdash s1 \text{ } -c-n2 \rightarrow s2 \wedge$ 
         $G \vdash (\text{abupd } (\text{absorb } (\text{Cont } l)) \text{ } s2)-l \cdot \text{While}(e) \text{ } c-n2 \rightarrow s3$ )
      else  $s3 = s1$ 
    by simp (iprover intro: evaln-nonstrict max.cobounded1 max.cobounded2)
  ultimately
  have  $G \vdash \text{Norm } s0 \text{ } -l \cdot \text{While}(e) \text{ } c-\text{max } n1 \text{ } n2 \rightarrow s3$ 
    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  $G \vdash \text{Norm } s \text{ } -\text{Jmp } j-n \rightarrow (\text{Some } (\text{Jump } j), s)$ 
    by (rule evaln.Jmp)
  then show ?case ..
next
  case (Throw s0 e a s1)
  then obtain n where
     $G \vdash \text{Norm } s0 \text{ } -e-\succ a-n \rightarrow s1$ 
    by (iprover)
  then have  $G \vdash \text{Norm } s0 \text{ } -\text{Throw } e-n \rightarrow \text{abupd } (\text{throw } a) \text{ } s1$ 
    by (rule evaln.Throw)

```



```

then show ?case ..
next
  case (Try s0 c1 s1 s2 catchC vn c2 s3)
  from Try.hyps obtain n1 where
     $G \vdash \text{Norm } s0 \text{ } -c1-n1 \rightarrow s1$ 
    by (iprover)
  moreover
  note salloc =  $\langle G \vdash s1 \text{ } -salloc \rightarrow s2 \rangle$ 
  moreover
  from Try.hyps obtain n2 where
    if G,s2  $\vdash$  catch catchC then  $G \vdash \text{new-xcpt-var } vn \text{ } s2 \text{ } -c2-n2 \rightarrow s3$  else  $s3 = s2$ 
    by fastforce
  ultimately
  have  $G \vdash \text{Norm } s0 \text{ } -\text{Try } c1 \text{ } \text{Catch}(\text{catchC } vn) \text{ } c2\text{-max } n1 \text{ } n2 \rightarrow s3$ 
    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
     $G \vdash \text{Norm } s0 \text{ } -c1-n1 \rightarrow (x1, s1)$ 
     $G \vdash \text{Norm } s1 \text{ } -c2-n2 \rightarrow s2$ 
    by iprover
  moreover
  note s3 =  $\langle s3 = (\text{if } \exists \text{err. } x1 = \text{Some } (\text{Error } \text{err})$ 
    then  $(x1, s1)$ 
    else abruptd (abrupt-if  $(x1 \neq \text{None}) \text{ } x1) \text{ } s2) \rangle$ 
  ultimately
  have
     $G \vdash \text{Norm } s0 \text{ } -c1 \text{ } \text{Finally } c2\text{-max } n1 \text{ } n2 \rightarrow s3$ 
    by (blast intro: evaln.Fin dest: evaln-max2)
  then show ?case ..
next
  case (Init C c s0 s3 s1 s2)
  note cls =  $\langle \text{the } (\text{class } G \text{ } C) = c \rangle$ 
  moreover from Init.hyps obtain n where
    if initd C (globs s0) then  $s3 = \text{Norm } s0$ 
    else  $(G \vdash \text{Norm } (\text{init-class-obj } G \text{ } C \text{ } s0)$ 
       $-(\text{if } C = \text{Object then Skip else Init } (\text{super } c))\text{-}n \rightarrow s1 \wedge$ 
       $G \vdash \text{set-lvars Map.empty } s1 \text{ } -\text{init } c\text{-}n \rightarrow s2 \wedge$ 
       $s3 = \text{restore-lvars } s1 \text{ } s2)$ 
    by (auto intro: evaln-nonstrict max.cobounded1 max.cobounded2)
  ultimately have  $G \vdash \text{Norm } s0 \text{ } -\text{Init } C\text{-}n \rightarrow s3$ 
    by (rule evaln.Init)
  then show ?case ..
next
  case (NewC s0 C s1 a s2)
  then obtain n where
     $G \vdash \text{Norm } s0 \text{ } -\text{Init } C\text{-}n \rightarrow s1$ 
    by (iprover)
  with NewC
  have  $G \vdash \text{Norm } s0 \text{ } -\text{NewC } C\text{-}\succ \text{Addr } a\text{-}n \rightarrow s2$ 
    by (iprover intro: evaln.NewC)
  then show ?case ..
next
  case (NewA s0 T s1 e i s2 a s3)
  then obtain n1 n2 where
     $G \vdash \text{Norm } s0 \text{ } -\text{init-comp-ty } T\text{-}n1 \rightarrow s1$ 
     $G \vdash s1 \text{ } -e\text{-}\succ i\text{-}n2 \rightarrow s2$ 
    by (iprover)

```

```

moreover
note  $\langle G \vdash \text{abupd } (\text{check-neg } i) \ s2 \text{ --halloc Arr } T \ (\text{the-Intg } i) \succ a \rightarrow s3 \rangle$ 
ultimately
have  $G \vdash \text{Norm } s0 \text{ --New } T[e] \text{ --} \succ \text{Addr } a \text{ --max } n1 \ n2 \rightarrow s3$ 
  by (blast intro: evaln.NewA dest: evaln-max2)
then show ?case ..
next
case (Cast  $s0 \ e \ v \ s1 \ s2 \ \text{cast}T$ )
then obtain  $n$  where
   $G \vdash \text{Norm } s0 \text{ --} e \text{ --} \succ v \text{ --} n \rightarrow s1$ 
  by (iprover)
moreover
note  $\langle s2 = \text{abupd } (\text{raise-if } (\neg G, \text{snd } s1 \vdash v \text{ fits } \text{cast}T) \ \text{ClassCast}) \ s1 \rangle$ 
ultimately
have  $G \vdash \text{Norm } s0 \text{ --Cast } \text{cast}T \ e \text{ --} \succ v \text{ --} n \rightarrow s2$ 
  by (rule evaln.Cast)
then show ?case ..
next
case (Inst  $s0 \ e \ v \ s1 \ b \ T$ )
then obtain  $n$  where
   $G \vdash \text{Norm } s0 \text{ --} e \text{ --} \succ v \text{ --} n \rightarrow s1$ 
  by (iprover)
moreover
note  $\langle b = (v \neq \text{Null} \wedge G, \text{snd } s1 \vdash v \text{ fits } \text{Ref}T \ T) \rangle$ 
ultimately
have  $G \vdash \text{Norm } s0 \text{ --} e \ \text{InstOf } T \text{ --} \succ \text{Bool } b \text{ --} n \rightarrow s1$ 
  by (rule evaln.Inst)
then show ?case ..
next
case (Lit  $s \ v$ )
fix  $n$  have  $G \vdash \text{Norm } s \text{ --Lit } v \text{ --} \succ v \text{ --} n \rightarrow \text{Norm } s$ 
  by (rule evaln.Lit)
then show ?case ..
next
case (UnOp  $s0 \ e \ v \ s1 \ \text{unop}$ )
then obtain  $n$  where
   $G \vdash \text{Norm } s0 \text{ --} e \text{ --} \succ v \text{ --} n \rightarrow s1$ 
  by (iprover)
hence  $G \vdash \text{Norm } s0 \text{ --UnOp } \text{unop} \ e \text{ --} \succ \text{eval-unop } \text{unop} \ v \text{ --} n \rightarrow s1$ 
  by (rule evaln.UnOp)
then show ?case ..
next
case (BinOp  $s0 \ e1 \ v1 \ s1 \ \text{binop} \ e2 \ v2 \ s2$ )
then obtain  $n1 \ n2$  where
   $G \vdash \text{Norm } s0 \text{ --} e1 \text{ --} \succ v1 \text{ --} n1 \rightarrow s1$ 
   $G \vdash s1 \text{ --(if need-second-arg binop } v1 \text{ then In1l } e2$ 
     $\text{else In1r Skip)} \text{ --} n2 \rightarrow (\text{In1 } v2, \ s2)$ 
  by (iprover)
hence  $G \vdash \text{Norm } s0 \text{ --BinOp } \text{binop} \ e1 \ e2 \text{ --} \succ (\text{eval-binop } \text{binop} \ v1 \ v2) \text{ --max } n1 \ n2$ 
   $\rightarrow s2$ 
  by (blast intro!: evaln.BinOp dest: evaln-max2)
then show ?case ..
next
case (Super  $s$ )
fix  $n$  have  $G \vdash \text{Norm } s \text{ --Super --} \succ \text{val-this } s \text{ --} n \rightarrow \text{Norm } s$ 
  by (rule evaln.Super)
then show ?case ..
next
case (Acc  $s0 \ va \ v \ f \ s1$ )

```

```

then obtain  $n$  where
   $G \vdash \text{Norm } s0 \text{ } -va = \succ(v, f) - n \rightarrow s1$ 
  by (iprover)
then
have  $G \vdash \text{Norm } s0 \text{ } -\text{Acc } va - \succ v - n \rightarrow s1$ 
  by (rule evaln.Acc)
then show ?case ..
next
case (Ass  $s0$   $var$   $w$   $f$   $s1$   $e$   $v$   $s2$ )
then obtain  $n1$   $n2$  where
   $G \vdash \text{Norm } s0 \text{ } -var = \succ(w, f) - n1 \rightarrow s1$ 
   $G \vdash s1 \text{ } -e - \succ v - n2 \rightarrow s2$ 
  by (iprover)
then
have  $G \vdash \text{Norm } s0 \text{ } -var := e - \succ v - \max n1 \ n2 \rightarrow \text{assign } f \ v \ s2$ 
  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
   $G \vdash \text{Norm } s0 \text{ } -e0 - \succ b - n1 \rightarrow s1$ 
   $G \vdash s1 \text{ } -(if \ the\ Bool \ b \ then \ e1 \ else \ e2) - \succ v - n2 \rightarrow s2$ 
  by (iprover)
then
have  $G \vdash \text{Norm } s0 \text{ } -e0 \ ? \ e1 : e2 - \succ v - \max n1 \ n2 \rightarrow s2$ 
  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
   $G \vdash \text{Norm } s0 \text{ } -e - \succ a' - n1 \rightarrow s1$ 
   $G \vdash s1 \text{ } -args = \succ vs - n2 \rightarrow s2$ 
  by iprover
moreover
note  $\langle invDeclC = \text{invocation-declclass } G \ mode \ (store \ s2) \ a' \ statT$ 
   $\ \ \ \ \ \langle name = mn, parTs = pTs' \rangle \rangle$ 
moreover
note  $\langle s3 = \text{init-lvars } G \ invDeclC \ \langle name = mn, parTs = pTs' \rangle \ mode \ a' \ vs \ s2 \rangle$ 
moreover
note  $\langle s3' = \text{check-method-access } G \ accC' \ statT \ mode \ \langle name = mn, parTs = pTs' \rangle \ a' \ s3 \rangle$ 
moreover
from Call.hyps
obtain  $m$  where
   $G \vdash s3' \text{ } -\text{Methd } invDeclC \ \langle name = mn, parTs = pTs' \rangle - \succ v - m \rightarrow s4$ 
  by iprover
ultimately
have  $G \vdash \text{Norm } s0 \text{ } -\{accC', statT, mode\}e.mn(\{pTs'\}args) - \succ v - \max n1 \ (\max n2 \ m) \rightarrow$ 
   $\ \ \ \ \ \ (set-lvars \ (locals \ (store \ s2))) \ s4$ 
  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
   $G \vdash \text{Norm } s0 \text{ } -body \ G \ D \ sig - \succ v - n \rightarrow s1$ 
  by iprover
then have  $G \vdash \text{Norm } s0 \text{ } -\text{Methd } D \ sig - \succ v - Suc \ n \rightarrow s1$ 
  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:  $G \vdash \text{Norm } s0 \text{ --Init } D \text{--} n1 \rightarrow s1$  and
  evaln-c:  $G \vdash s1 \text{ --c--} n2 \rightarrow s2$ 
  by (iprover)
moreover
note  $\langle s3 = (\text{if } \exists l. \text{fst } s2 = \text{Some } (\text{Jump } (\text{Break } l)) \vee$ 
       $\text{fst } s2 = \text{Some } (\text{Jump } (\text{Cont } l))$ 
       $\text{then abupd } (\lambda x. \text{Some } (\text{Error CrossMethodJump})) s2$ 
       $\text{else } s2) \rangle$ 
ultimately
have
   $G \vdash \text{Norm } s0 \text{ --Body } D \text{ c--} \rightarrow \text{the } (\text{locals } (\text{store } s2) \text{ Result}) \text{--max } n1 \text{ } n2$ 
   $\rightarrow \text{abupd } (\text{absorb Ret}) s3$ 
  by (iprover intro: evaln.Body dest: evaln-max2)
then show ?case ..
next
case (LVar s vn )
obtain n where
   $G \vdash \text{Norm } s \text{ --LVar } vn \text{--} \rightarrow \text{lvar } vn \text{ s--} n \rightarrow \text{Norm } s$ 
  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
   $G \vdash \text{Norm } s0 \text{ --Init } statDeclC \text{--} n1 \rightarrow s1$ 
   $G \vdash s1 \text{ --e--} \rightarrow a \text{--} n2 \rightarrow s2$ 
  by iprover
moreover
note  $\langle s3 = \text{check-field-access } G \text{ accC } statDeclC \text{ fn stat } a \text{ } s2' \rangle$ 
  and  $\langle (v, s2') = \text{fvar } statDeclC \text{ stat fn } a \text{ } s2 \rangle$ 
ultimately
have  $G \vdash \text{Norm } s0 \text{ --}\{accC, statDeclC, stat\}e..fn \text{--} \rightarrow v \text{--max } n1 \text{ } n2 \rightarrow s3$ 
  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
   $G \vdash \text{Norm } s0 \text{ --e1--} \rightarrow a \text{--} n1 \rightarrow s1$ 
   $G \vdash s1 \text{ --e2--} \rightarrow i \text{--} n2 \rightarrow s2$ 
  by iprover
moreover
note  $\langle (v, s2') = \text{avar } G \text{ } i \text{ } a \text{ } s2 \rangle$ 
ultimately
have  $G \vdash \text{Norm } s0 \text{ --e1.[e2]--} \rightarrow v \text{--max } n1 \text{ } n2 \rightarrow s2'$ 
  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
   $G \vdash \text{Norm } s0 \text{ --e--} \rightarrow v \text{--} n1 \rightarrow s1$ 
   $G \vdash s1 \text{ --es--} \rightarrow vs \text{--} n2 \rightarrow s2$ 
  by iprover
then
have  $G \vdash \text{Norm } s0 \text{ --e \# es--} \rightarrow v \text{ \# vs--max } n1 \text{ } n2 \rightarrow s2$ 
  by (blast intro!: evaln.Cons dest: evaln-max2)

```

```
  then show ?case ..
qed
end
```


Chapter 21

Trans

theory *Trans* **imports** *Evaln* **begin**

definition

groundVar :: *var* \Rightarrow *bool* **where**
groundVar *v* \longleftrightarrow (case *v* of
 LVar *ln* \Rightarrow *True*
 | {*accC*,*statDeclC*,*stat*}*e*..*fn* $\Rightarrow \exists$ *a*. *e*=*Lit* *a*
 | *e1*..*e2* $\Rightarrow \exists$ *a* *i*. *e1* = *Lit* *a* \wedge *e2* = *Lit* *i*
 | *InsInitV* *c* *v* \Rightarrow *False*)

lemma *groundVar-cases*:

assumes *ground*: *groundVar* *v*
obtains (*LVar*) *ln* **where** *v*=*LVar* *ln*
 | (*FVar*) *accC* *statDeclC* *stat* *a* *fn* **where** *v*={*accC*,*statDeclC*,*stat*}(*Lit* *a*)..*fn*
 | (*AVar*) *a* *i* **where** *v*=(*Lit* *a*).[*Lit* *i*]
using *ground* *LVar* *FVar* *AVar*
by (cases *v*) (auto simp add: *groundVar-def*)

definition

groundExprs :: *expr* *list* \Rightarrow *bool*
where *groundExprs* *es* $\longleftrightarrow (\forall e \in \text{set } es. \exists v. e = \text{Lit } v)$

primrec *the-val*:: *expr* \Rightarrow *val*

where *the-val* (*Lit* *v*) = *v*

primrec *the-var*:: *prog* \Rightarrow *state* \Rightarrow *var* \Rightarrow (*vvar* \times *state*) **where**

the-var *G* *s* (*LVar* *ln*) = (*lvar* *ln* (*store* *s*),*s*)
| *the-var-FVar-def*: *the-var* *G* *s* ({*accC*,*statDeclC*,*stat*}*a*..*fn*) = *fvar* *statDeclC* *stat* *fn* (*the-val* *a*) *s*
| *the-var-AVar-def*: *the-var* *G* *s* (*a*..*i*) = *avar* *G* (*the-val* *i*) (*the-val* *a*) *s*

lemma *the-var-FVar-simp*[*simp*]:

the-var *G* *s* ({*accC*,*statDeclC*,*stat*}(*Lit* *a*)..*fn*) = *fvar* *statDeclC* *stat* *fn* *a* *s*

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* \Rightarrow *expr*

where $Ref\ a == Lit\ (Addr\ a)$

abbreviation

$SKIP :: expr$

where $SKIP == Lit\ Unit$

inductive

$step :: [prog, term \times state, term \times state] \Rightarrow bool\ (\vdash \mapsto 1 \rightarrow [61, 82, 82]\ 81)$

for $G :: prog$

where

Abrupt:

$$\begin{aligned} & \llbracket \forall v. t \neq \langle Lit\ v \rangle; \\ & \quad \forall t. t \neq \langle l \cdot Skip \rangle; \\ & \quad \forall C\ vn\ c. t \neq \langle Try\ Skip\ Catch(C\ vn)\ c \rangle; \\ & \quad \forall x\ c. t \neq \langle Skip\ Finally\ c \rangle \wedge xc \neq Xcpt\ x; \\ & \quad \forall a\ c. t \neq \langle FinA\ a\ c \rangle \rrbracket \\ & \implies \\ & \quad G \vdash (t, Some\ xc, s) \mapsto 1\ (\langle Lit\ undefined \rangle, Some\ xc, s) \end{aligned}$$

| *InsInitE*: $\llbracket G \vdash (\langle c \rangle, Norm\ s) \mapsto 1\ (\langle c' \rangle, s') \rrbracket$

$$\implies$$

$G \vdash (\langle InsInitE\ c\ e \rangle, Norm\ s) \mapsto 1\ (\langle InsInitE\ c'\ e' \rangle, s')$

| *NewC*: $G \vdash (\langle NewC\ C' \rangle, Norm\ s) \mapsto 1\ (\langle InsInitE\ (Init\ C)\ (NewC\ C') \rangle, Norm\ s)$

| *NewCInitd*: $\llbracket G \vdash Norm\ s \text{ --halloc } (CInst\ C) \text{ --} a \rightarrow s' \rrbracket$

$$\implies$$

$G \vdash (\langle InsInitE\ Skip\ (NewC\ C') \rangle, Norm\ s) \mapsto 1\ (\langle Ref\ a \rangle, s')$

| *NewA*:

$$G \vdash (\langle New\ T[e] \rangle, Norm\ s) \mapsto 1\ (\langle InsInitE\ (init\ comp\ ty\ T)\ (New\ T[e]) \rangle, Norm\ s)$$

| *InsInitNewAIdx*:

$$\llbracket G \vdash (\langle e \rangle, Norm\ s) \mapsto 1\ (\langle e' \rangle, s') \rrbracket$$

$$\implies$$

$$G \vdash (\langle InsInitE\ Skip\ (New\ T[e]) \rangle, Norm\ s) \mapsto 1\ (\langle InsInitE\ Skip\ (New\ T[e']) \rangle, s')$$

| *InsInitNewA*:

$$\llbracket G \vdash abupd\ (check\ neg\ i)\ (Norm\ s) \text{ --halloc } (Arr\ T\ (the\ Intg\ i)) \text{ --} a \rightarrow s' \rrbracket$$

$$\implies$$

$G \vdash (\langle InsInitE\ Skip\ (New\ T[Lit\ i]) \rangle, Norm\ s) \mapsto 1\ (\langle Ref\ a \rangle, s')$

| *CastE*:

$$\llbracket G \vdash (\langle e \rangle, Norm\ s) \mapsto 1\ (\langle e' \rangle, s') \rrbracket$$

$$\implies$$

$$G \vdash (\langle Cast\ T\ e \rangle, None, s) \mapsto 1\ (\langle Cast\ T\ e' \rangle, s')$$

| *Cast*:

$$\llbracket s' = abupd\ (raise\ if\ (\neg G, s \vdash v\ fits\ T)\ ClassCast)\ (Norm\ s) \rrbracket$$

$$\implies$$

$G \vdash (\langle Cast\ T\ (Lit\ v) \rangle, Norm\ s) \mapsto 1\ (\langle Lit\ v \rangle, s')$

$$\begin{array}{l}
| \text{InstE: } \llbracket G \vdash (\langle e \rangle, \text{Norm } s) \mapsto 1 \ (\langle e'::\text{expr} \rangle, s') \rrbracket \\
\quad \Longrightarrow \\
\quad G \vdash (\langle e \text{ InstOf } T \rangle, \text{Norm } s) \mapsto 1 \ (\langle e' \rangle, s') \\
| \text{Inst: } \llbracket b = (v \neq \text{Null} \wedge G, s \vdash v \text{ fits RefT } T) \rrbracket \\
\quad \Longrightarrow \\
\quad G \vdash (\langle (\text{Lit } v) \text{ InstOf } T \rangle, \text{Norm } s) \mapsto 1 \ (\langle \text{Lit } (\text{Bool } b) \rangle, s') \\
\\
| \text{UnOpE: } \llbracket G \vdash (\langle e \rangle, \text{Norm } s) \mapsto 1 \ (\langle e' \rangle, s') \rrbracket \\
\quad \Longrightarrow \\
\quad G \vdash (\langle \text{UnOp unop } e \rangle, \text{Norm } s) \mapsto 1 \ (\langle \text{UnOp unop } e' \rangle, s') \\
| \text{UnOp: } G \vdash (\langle \text{UnOp unop } (\text{Lit } v) \rangle, \text{Norm } s) \mapsto 1 \ (\langle \text{Lit } (\text{eval-unop unop } v) \rangle, \text{Norm } s) \\
\\
| \text{BinOpE1: } \llbracket G \vdash (\langle e1 \rangle, \text{Norm } s) \mapsto 1 \ (\langle e1' \rangle, s') \rrbracket \\
\quad \Longrightarrow \\
\quad G \vdash (\langle \text{BinOp binop } e1 \ e2 \rangle, \text{Norm } s) \mapsto 1 \ (\langle \text{BinOp binop } e1' \ e2 \rangle, s') \\
| \text{BinOpE2: } \llbracket \text{need-second-arg binop } v1; G \vdash (\langle e2 \rangle, \text{Norm } s) \mapsto 1 \ (\langle e2' \rangle, s') \rrbracket \\
\quad \Longrightarrow \\
\quad G \vdash (\langle \text{BinOp binop } (\text{Lit } v1) \ e2 \rangle, \text{Norm } s) \\
\quad \mapsto 1 \ (\langle \text{BinOp binop } (\text{Lit } v1) \ e2' \rangle, s') \\
| \text{BinOpTerm: } \llbracket \neg \text{need-second-arg binop } v1 \rrbracket \\
\quad \Longrightarrow \\
\quad G \vdash (\langle \text{BinOp binop } (\text{Lit } v1) \ e2 \rangle, \text{Norm } s) \\
\quad \mapsto 1 \ (\langle \text{Lit } v1 \rangle, \text{Norm } s) \\
| \text{BinOp: } G \vdash (\langle \text{BinOp binop } (\text{Lit } v1) \ (\text{Lit } v2) \rangle, \text{Norm } s) \\
\quad \mapsto 1 \ (\langle \text{Lit } (\text{eval-binop binop } v1 \ v2) \rangle, \text{Norm } s) \\
\\
| \text{Super: } G \vdash (\langle \text{Super} \rangle, \text{Norm } s) \mapsto 1 \ (\langle \text{Lit } (\text{val-this } s) \rangle, \text{Norm } s) \\
\\
| \text{AccVA: } \llbracket G \vdash (\langle va \rangle, \text{Norm } s) \mapsto 1 \ (\langle va' \rangle, s') \rrbracket \\
\quad \Longrightarrow \\
\quad G \vdash (\langle \text{Acc } va \rangle, \text{Norm } s) \mapsto 1 \ (\langle \text{Acc } va' \rangle, s') \\
| \text{Acc: } \llbracket \text{groundVar } va; ((v, vf), s') = \text{the-var } G \ (\text{Norm } s) \ va \rrbracket \\
\quad \Longrightarrow \\
\quad G \vdash (\langle \text{Acc } va \rangle, \text{Norm } s) \mapsto 1 \ (\langle \text{Lit } v \rangle, s') \\
\\
| \text{AssVA: } \llbracket G \vdash (\langle va \rangle, \text{Norm } s) \mapsto 1 \ (\langle va' \rangle, s') \rrbracket \\
\quad \Longrightarrow \\
\quad G \vdash (\langle va := e \rangle, \text{Norm } s) \mapsto 1 \ (\langle va' := e \rangle, s') \\
| \text{AssE: } \llbracket \text{groundVar } va; G \vdash (\langle e \rangle, \text{Norm } s) \mapsto 1 \ (\langle e' \rangle, s') \rrbracket \\
\quad \Longrightarrow \\
\quad G \vdash (\langle va := e \rangle, \text{Norm } s) \mapsto 1 \ (\langle va := e' \rangle, s') \\
| \text{Ass: } \llbracket \text{groundVar } va; ((w, f), s') = \text{the-var } G \ (\text{Norm } s) \ va \rrbracket \\
\quad \Longrightarrow \\
\quad G \vdash (\langle va := (\text{Lit } v) \rangle, \text{Norm } s) \mapsto 1 \ (\langle \text{Lit } v \rangle, \text{assign } f \ v \ s') \\
\\
| \text{CondC: } \llbracket G \vdash (\langle e0 \rangle, \text{Norm } s) \mapsto 1 \ (\langle e0' \rangle, s') \rrbracket \\
\quad \Longrightarrow \\
\quad G \vdash (\langle e0? \ e1:e2 \rangle, \text{Norm } s) \mapsto 1 \ (\langle e0'? \ e1:e2 \rangle, s') \\
| \text{Cond: } G \vdash (\langle \text{Lit } b? \ e1:e2 \rangle, \text{Norm } s) \mapsto 1 \ (\langle \text{if the-Bool } b \text{ then } e1 \text{ else } e2 \rangle, \text{Norm } s) \\
\\
| \text{CallTarget: } \llbracket G \vdash (\langle e \rangle, \text{Norm } s) \mapsto 1 \ (\langle e' \rangle, s') \rrbracket
\end{array}$$

$$\begin{array}{l}
\begin{array}{l}
\Rightarrow \\
G \vdash (\langle \{accC, statT, mode\} e \cdot mn(\{pTs\} args) \rangle, Norm\ s) \\
\mapsto 1 (\langle \{accC, statT, mode\} e' \cdot mn(\{pTs\} args) \rangle, s')
\end{array} \\
| \text{ CallArgs: } \llbracket G \vdash (\langle args \rangle, Norm\ s) \mapsto 1 (\langle args' \rangle, s') \rrbracket \\
\begin{array}{l}
\Rightarrow \\
G \vdash (\langle \{accC, statT, mode\} Lit\ a \cdot mn(\{pTs\} args) \rangle, Norm\ s) \\
\mapsto 1 (\langle \{accC, statT, mode\} Lit\ a \cdot mn(\{pTs\} args') \rangle, s')
\end{array} \\
| \text{ Call: } \llbracket \text{groundExprs } args; vs = \text{map the-val } args; \\
D = \text{invocation-declclass } G\ mode\ s\ a\ statT\ (\langle name=mn, parTs=pTs \rangle); \\
s' = \text{init-lvars } G\ D\ (\langle name=mn, parTs=pTs \rangle\ mode\ a'\ vs\ (Norm\ s)) \rrbracket \\
\begin{array}{l}
\Rightarrow \\
G \vdash (\langle \{accC, statT, mode\} Lit\ a \cdot mn(\{pTs\} args) \rangle, Norm\ s) \\
\mapsto 1 (\langle \text{Callee } (locals\ s)\ (\text{Methd } D\ (\langle name=mn, parTs=pTs \rangle)) \rangle, s')
\end{array} \\
| \text{ Callee: } \llbracket G \vdash (\langle e \rangle, Norm\ s) \mapsto 1 (\langle e'::expr \rangle, s') \rrbracket \\
\begin{array}{l}
\Rightarrow \\
G \vdash (\langle \text{Callee lcls-caller } e \rangle, Norm\ s) \mapsto 1 (\langle e' \rangle, s')
\end{array} \\
| \text{ CalleeRet: } \begin{array}{l} G \vdash (\langle \text{Callee lcls-caller } (Lit\ v) \rangle, Norm\ s) \\ \mapsto 1 (\langle Lit\ v \rangle, (\text{set-lvars lcls-caller } (Norm\ s))) \end{array} \\
| \text{ Methd: } G \vdash (\langle \text{Methd } D\ sig \rangle, Norm\ s) \mapsto 1 (\langle \text{body } G\ D\ sig \rangle, Norm\ s) \\
| \text{ Body: } G \vdash (\langle \text{Body } D\ c \rangle, Norm\ s) \mapsto 1 (\langle \text{InsInitE } (Init\ D)\ (\text{Body } D\ c) \rangle, Norm\ s) \\
| \text{ InsInitBody: } \llbracket G \vdash (\langle c \rangle, Norm\ s) \mapsto 1 (\langle c' \rangle, s') \rrbracket \\
\begin{array}{l}
\Rightarrow \\
G \vdash (\langle \text{InsInitE Skip } (\text{Body } D\ c) \rangle, Norm\ s) \mapsto 1 (\langle \text{InsInitE Skip } (\text{Body } D\ c') \rangle, s')
\end{array} \\
| \text{ InsInitBodyRet: } \begin{array}{l} G \vdash (\langle \text{InsInitE Skip } (\text{Body } D\ Skip) \rangle, Norm\ s) \\ \mapsto 1 (\langle Lit\ (\text{the } ((locals\ s)\ \text{Result})) \rangle, \text{abupd } (\text{absorb } Ret)\ (Norm\ s)) \end{array} \\
| \text{ FVar: } \llbracket \neg\ \text{inited statDeclC } (globs\ s) \rrbracket \\
\begin{array}{l}
\Rightarrow \\
G \vdash (\langle \{accC, statDeclC, stat\} e..fn \rangle, Norm\ s) \\
\mapsto 1 (\langle \text{InsInitV } (Init\ statDeclC)\ (\{accC, statDeclC, stat\} e..fn) \rangle, Norm\ s)
\end{array} \\
| \text{ InsInitFVarE: } \llbracket G \vdash (\langle e \rangle, Norm\ s) \mapsto 1 (\langle e' \rangle, s') \rrbracket \\
\begin{array}{l}
\Rightarrow \\
G \vdash (\langle \text{InsInitV Skip } (\{accC, statDeclC, stat\} e..fn) \rangle, Norm\ s) \\
\mapsto 1 (\langle \text{InsInitV Skip } (\{accC, statDeclC, stat\} e'..fn) \rangle, s')
\end{array} \\
| \text{ InsInitFVar: } \begin{array}{l} G \vdash (\langle \text{InsInitV Skip } (\{accC, statDeclC, stat\} Lit\ a..fn) \rangle, Norm\ s) \\ \mapsto 1 (\langle \{accC, statDeclC, stat\} Lit\ a..fn \rangle, Norm\ s) \end{array} \\
\text{— Notice, that we do not have literal values for } vars. \text{ The rules for accessing variables (} Acc \text{) and assigning to} \\
\text{variables (} Ass \text{), test this with the predicate } groundVar. \text{ After initialisation is done and the } FVar \text{ is evaluated,} \\
\text{we can't just throw away the } InsInitFVar \text{ term and return a literal value, as in the cases of } New \text{ or } NewC. \\
\text{Instead we just return the evaluated } FVar \text{ and test for initialisation in the rule } FVar.
\end{array}$$

$$\begin{array}{l}
| \text{ AVarE1: } \llbracket G \vdash (\langle e1 \rangle, Norm\ s) \mapsto 1 (\langle e1' \rangle, s') \rrbracket \\
\begin{array}{l}
\Rightarrow \\
G \vdash (\langle e1.[e2] \rangle, Norm\ s) \mapsto 1 (\langle e1'.[e2] \rangle, s')
\end{array} \\
| \text{ AVarE2: } G \vdash (\langle e2 \rangle, Norm\ s) \mapsto 1 (\langle e2' \rangle, s') \\
\Rightarrow
\end{array}$$

$$G \vdash (\langle \text{Lit } a.[e2] \rangle, \text{Norm } s) \mapsto 1 (\langle \text{Lit } a.[e2'] \rangle, s')$$

— *Nil* is fully evaluated

$$\begin{aligned} | \text{ConsHd}: & \llbracket G \vdash (\langle e::\text{expr} \rangle, \text{Norm } s) \mapsto 1 (\langle e'::\text{expr} \rangle, s') \rrbracket \\ & \implies \\ & G \vdash (\langle e\#es \rangle, \text{Norm } s) \mapsto 1 (\langle e'\#es \rangle, s') \end{aligned}$$

$$\begin{aligned} | \text{ConsTl}: & \llbracket G \vdash (\langle es \rangle, \text{Norm } s) \mapsto 1 (\langle es' \rangle, s') \rrbracket \\ & \implies \\ & G \vdash (\langle (\text{Lit } v)\#es \rangle, \text{Norm } s) \mapsto 1 (\langle (\text{Lit } v)\#es' \rangle, s') \end{aligned}$$

$$| \text{Skip}: G \vdash (\langle \text{Skip} \rangle, \text{Norm } s) \mapsto 1 (\langle \text{SKIP} \rangle, \text{Norm } s)$$

$$\begin{aligned} | \text{ExprE}: & \llbracket G \vdash (\langle e \rangle, \text{Norm } s) \mapsto 1 (\langle e' \rangle, s') \rrbracket \\ & \implies \\ & G \vdash (\langle \text{Expr } e \rangle, \text{Norm } s) \mapsto 1 (\langle \text{Expr } e' \rangle, s') \\ | \text{Expr}: & G \vdash (\langle \text{Expr } (\text{Lit } v) \rangle, \text{Norm } s) \mapsto 1 (\langle \text{Skip} \rangle, \text{Norm } s) \end{aligned}$$

$$\begin{aligned} | \text{LabC}: & \llbracket G \vdash (\langle c \rangle, \text{Norm } s) \mapsto 1 (\langle c' \rangle, s') \rrbracket \\ & \implies \\ & G \vdash (\langle l \cdot c \rangle, \text{Norm } s) \mapsto 1 (\langle l \cdot c' \rangle, s') \\ | \text{Lab}: & G \vdash (\langle l \cdot \text{Skip} \rangle, s) \mapsto 1 (\langle \text{Skip} \rangle, \text{abupd } (\text{absorb } l) s) \end{aligned}$$

$$\begin{aligned} | \text{CompC1}: & \llbracket G \vdash (\langle c1 \rangle, \text{Norm } s) \mapsto 1 (\langle c1' \rangle, s') \rrbracket \\ & \implies \\ & G \vdash (\langle c1;; c2 \rangle, \text{Norm } s) \mapsto 1 (\langle c1';; c2 \rangle, s') \end{aligned}$$

$$| \text{Comp}: G \vdash (\langle \text{Skip};; c2 \rangle, \text{Norm } s) \mapsto 1 (\langle c2 \rangle, \text{Norm } s)$$

$$\begin{aligned} | \text{IfE}: & \llbracket G \vdash (\langle e \rangle, \text{Norm } s) \mapsto 1 (\langle e' \rangle, s') \rrbracket \\ & \implies \\ & G \vdash (\langle \text{If}(e) s1 \text{ Else } s2 \rangle, \text{Norm } s) \mapsto 1 (\langle \text{If}(e') s1 \text{ Else } s2 \rangle, s') \\ | \text{If}: & G \vdash (\langle \text{If}(\text{Lit } v) s1 \text{ Else } s2 \rangle, \text{Norm } s) \\ & \mapsto 1 (\langle \text{if the-Bool } v \text{ then } s1 \text{ else } s2 \rangle, \text{Norm } s) \end{aligned}$$

$$\begin{aligned} | \text{Loop}: & G \vdash (\langle l \cdot \text{While}(e) c \rangle, \text{Norm } s) \\ & \mapsto 1 (\langle \text{If}(e) (\text{Cont } l \cdot c;; l \cdot \text{While}(e) c) \text{ Else } \text{Skip} \rangle, \text{Norm } s) \end{aligned}$$

$$| \text{Jmp}: G \vdash (\langle \text{Jmp } j \rangle, \text{Norm } s) \mapsto 1 (\langle \text{Skip} \rangle, (\text{Some } (\text{Jump } j), s))$$

$$\begin{aligned} | \text{ThrowE}: & \llbracket G \vdash (\langle e \rangle, \text{Norm } s) \mapsto 1 (\langle e' \rangle, s') \rrbracket \\ & \implies \\ & G \vdash (\langle \text{Throw } e \rangle, \text{Norm } s) \mapsto 1 (\langle \text{Throw } e' \rangle, s') \\ | \text{Throw}: & G \vdash (\langle \text{Throw } (\text{Lit } a) \rangle, \text{Norm } s) \mapsto 1 (\langle \text{Skip} \rangle, \text{abupd } (\text{throw } a) (\text{Norm } s)) \end{aligned}$$

$$\begin{aligned} | \text{TryC1}: & \llbracket G \vdash (\langle c1 \rangle, \text{Norm } s) \mapsto 1 (\langle c1' \rangle, s') \rrbracket \\ & \implies \end{aligned}$$

$G \vdash (\langle \text{Try } c1 \text{ Catch}(C \text{ vn}) \ c2 \rangle, \text{Norm } s) \mapsto 1 \ (\langle \text{Try } c1' \text{ Catch}(C \text{ vn}) \ c2 \rangle, s')$
| *Try*: $\llbracket G \vdash s \text{ --} s\text{alloc} \rightarrow s \rrbracket$
 \implies
 $G \vdash (\langle \text{Try Skip Catch}(C \text{ vn}) \ c2 \rangle, s)$
 $\mapsto 1 \ (\text{if } G, s \vdash \text{catch } C \text{ then } (\langle c2 \rangle, \text{new-}x\text{cpt-var } \text{vn } s') \text{ else } (\langle \text{Skip} \rangle, s'))$

| *FinC1*: $\llbracket G \vdash (\langle c1 \rangle, \text{Norm } s) \mapsto 1 \ (\langle c1' \rangle, s') \rrbracket$
 \implies
 $G \vdash (\langle c1 \text{ Finally } c2 \rangle, \text{Norm } s) \mapsto 1 \ (\langle c1' \text{ Finally } c2 \rangle, s')$

| *Fin*: $G \vdash (\langle \text{Skip Finally } c2 \rangle, (a, s)) \mapsto 1 \ (\langle \text{FinA } a \ c2 \rangle, \text{Norm } s)$

| *FinAC*: $\llbracket G \vdash (\langle c \rangle, s) \mapsto 1 \ (\langle c' \rangle, s') \rrbracket$
 \implies
 $G \vdash (\langle \text{FinA } a \ c \rangle, s) \mapsto 1 \ (\langle \text{FinA } a \ c' \rangle, s')$

| *FinA*: $G \vdash (\langle \text{FinA } a \text{ Skip} \rangle, s) \mapsto 1 \ (\langle \text{Skip} \rangle, \text{abupd } (\text{abrupt-if } (a \neq \text{None}) \ a) \ s)$

| *Init1*: $\llbracket \text{inited } C \text{ (globs } s) \rrbracket$
 \implies
 $G \vdash (\langle \text{Init } C \rangle, \text{Norm } s) \mapsto 1 \ (\langle \text{Skip} \rangle, \text{Norm } s)$

| *Init*: $\llbracket \text{the (class } G \ C) = c; \neg \text{inited } C \text{ (globs } s) \rrbracket$
 \implies
 $G \vdash (\langle \text{Init } C \rangle, \text{Norm } s)$
 $\mapsto 1 \ ((\text{if } C = \text{Object then Skip else (Init (super } c))) ;$
 $\text{Expr (Callee (locals } s) (\text{InsInitE (init } c) \text{ SKIP}))}$
 $, \text{Norm (init-class-obj } G \ C \ s))$

— *InsInitE* is just used as trick to embed the statement *init c* into an expression
| *InsInitESKIP*:
 $G \vdash (\langle \text{InsInitE Skip SKIP} \rangle, \text{Norm } s) \mapsto 1 \ (\langle \text{SKIP} \rangle, \text{Norm } s)$

abbreviation

$\text{stepn} :: [\text{prog}, \text{term} \times \text{state}, \text{nat}, \text{term} \times \text{state}] \Rightarrow \text{bool} \ (\langle \vdash - \mapsto - \rangle [61, 82, 82] \ 81)$
where $G \vdash p \mapsto n \ p' \equiv (p, p') \in \{(x, y). \text{step } G \ x \ y\}^{\sim n}$

abbreviation

$\text{steptr} :: [\text{prog}, \text{term} \times \text{state}, \text{term} \times \text{state}] \Rightarrow \text{bool} \ (\langle \vdash - \mapsto * \rangle [61, 82, 82] \ 81)$
where $G \vdash p \mapsto * \ p' \equiv (p, p') \in \{(x, y). \text{step } G \ x \ y\}^*$

end

Chapter 22

AxSem

1 Axiomatic semantics of Java expressions and statements (see also Eval.thy)

theory *AxSem* **imports** *Evaln TypeSafe* **begin**

design issues:

- a strong version of validity for triples with premises, namely one that takes the recursive depth needed to complete execution, enables correctness proof
- auxiliary variables are handled first-class (-> Thomas Kleymann)
- expressions not flattened to elementary assignments (as usual for axiomatic semantics) but treated first-class => explicit result value handling
- intermediate values not on triple, but on assertion level (with result entry)
- multiple results with semantical substitution mechanism not requiring a stack
- because of dynamic method binding, terms need to be dependent on state. this is also useful for conditional expressions and statements
- result values in triples exactly as in eval relation (also for xcpt states)
- validity: additional assumption of state conformance and well-typedness, which is required for soundness and thus rule hazard required of completeness

restrictions:

- all triples in a derivation are of the same type (due to weak polymorphism)

type-synonym *res = vals* — result entry

abbreviation (*input*)

Val **where** *Val* *x* == *In1* *x*

abbreviation (*input*)

Var **where** *Var* *x* == *In2* *x*

abbreviation (*input*)

Vals **where** *Vals* *x* == *In3* *x*

syntax

- *Val* :: [*pttrn*] => *pttrn* (‹ *Val* :-> [951] 950)

- *Var* :: [*pttrn*] => *pttrn* (‹ *Var* :-> [951] 950)

-Vals :: [pttrn] => pttrn (⋄ Vals:-> [951] 950)

translations

$\lambda Val:v . b == (\lambda v. b) \circ CONST\ the-In1$
 $\lambda Var:v . b == (\lambda v. b) \circ CONST\ the-In2$
 $\lambda Vals:v. b == (\lambda v. b) \circ CONST\ the-In3$

— 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** $\langle \Rightarrow \rangle$ 25)
where $(P \Rightarrow Q) = (\forall Y\ s\ Z. P\ Y\ s\ Z \longrightarrow Q\ Y\ s\ Z)$

lemma assn-imp-def2 [iff]: $(P \Rightarrow Q) = (\forall Y\ s\ Z. P\ Y\ s\ Z \longrightarrow Q\ Y\ s\ Z)$

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** $\langle \wedge. \rangle$ 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 $\wedge.$ 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

$assn-supd :: 'a\ assn \Rightarrow (state \Rightarrow state) \Rightarrow 'a\ assn$ (**infixl** $\langle ; \rangle$ 13)
where $(P ;. f) = (\lambda Y\ s'\ Z.\ \exists s.\ P\ Y\ s\ Z \wedge s' = f\ s)$

lemma *assn-supd-def2* [*simp*]: $assn-supd\ P\ f\ Y\ s'\ Z = (\exists s.\ P\ Y\ s\ Z \wedge s' = f\ s)$
apply (*unfold assn-supd-def*)
apply (*simp (no-asm)*)
done

4 supd-assn

definition

$supd-assn :: (state \Rightarrow state) \Rightarrow 'a\ assn \Rightarrow 'a\ assn$ (**infixr** $\langle ; \rangle$ 13)
where $(f ;. P) = (\lambda Y\ s.\ P\ Y\ (f\ s))$

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*]: $((f ;. Q) ;. f)\ Y\ s\ Z \Longrightarrow Q\ Y\ s\ Z$
apply *auto*
done

lemma *supd-assn-supdI* [*elim*]: $Q\ Y\ s\ Z \Longrightarrow (f ;. (Q ;. f))\ Y\ s\ Z$
apply (*auto simp del: split-paired-Ex*)
done

5 subst-res

definition

$subst-res :: 'a\ assn \Rightarrow res \Rightarrow 'a\ assn$ ($\langle \leftarrow \leftarrow \rangle$ [60,61] 60)
where $P \leftarrow w = (\lambda Y.\ P\ w)$

lemma *subst-res-def2* [*simp*]: $(P \leftarrow w)\ Y = P\ w$
apply (*unfold subst-res-def*)
apply (*simp (no-asm)*)
done

lemma *subst-subst-res* [*simp*]: $P \leftarrow w \leftarrow v = P \leftarrow w$
apply (*rule ext*)
apply (*simp (no-asm)*)
done

lemma *peek-and-subst-res* [*simp*]: $(P \wedge. p) \leftarrow w = (P \leftarrow w \wedge. p)$
apply (*rule ext*)
apply (*rule ext*)
apply (*simp (no-asm)*)
done

6 subst-Bool

definition

subst-Bool :: 'a assn \Rightarrow bool \Rightarrow 'a assn (\hookleftarrow [60,61] 60)
where $P \hookleftarrow b = (\lambda Y s Z. \exists v. P (Val v) s Z \wedge (normal\ s \longrightarrow the-Bool\ v=b))$

lemma *subst-Bool-def2* [simp]:
 $(P \hookleftarrow b)\ Y\ s\ Z = (\exists v. P (Val v) s Z \wedge (normal\ s \longrightarrow the-Bool\ v=b))$
apply (*unfold subst-Bool-def*)
apply (*simp (no-asm)*)
done

lemma *subst-Bool-the-BoolI*: $P (Val\ b)\ s\ Z \Longrightarrow (P \hookleftarrow the-Bool\ b)\ Y\ s\ Z$
apply *auto*
done

7 peek-res

definition

peek-res :: (res \Rightarrow 'a assn) \Rightarrow 'a assn
where *peek-res Pf* = $(\lambda Y. Pf\ Y\ Y)$

syntax

-peek-res :: *pttrn* \Rightarrow 'a assn \Rightarrow 'a assn (\hookleftarrow [0,3] 3)

syntax-consts

-peek-res == *peek-res*

translations

$\lambda w. P == CONST\ peek-res\ (\lambda w. P)$

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* \hookleftarrow *w* = *P w* \hookleftarrow *w*
apply (*rule ext*)
apply (*simp (no-asm)*)
done

lemma peek-subst-res-allI:

$(\bigwedge a. T\ a\ (P\ (f\ a) \hookleftarrow f\ a)) \Longrightarrow \forall a. T\ a\ (peek-res\ P \hookleftarrow f\ a)$
apply (*rule allI*)
apply (*simp (no-asm)*)
apply *fast*
done

8 ign-res

definition

ign-res :: 'a assn \Rightarrow 'a assn (\hookdownarrow [1000] 1000)
where $P \hookdownarrow = (\lambda Y s Z. \exists Y. P\ Y\ s\ Z)$

lemma *ign-res-def2* [simp]: $P \hookdownarrow\ Y\ s\ Z = (\exists Y. P\ Y\ s\ Z)$
apply (*unfold ign-res-def*)
apply (*simp (no-asm)*)
done

lemma *ign-ign-res* [simp]: $P \hookdownarrow \hookdownarrow = P \hookdownarrow$
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 :: $(st \Rightarrow 'a \text{ assn}) \Rightarrow 'a \text{ assn}$
where *peek-st* $P = (\lambda Y s. P (\text{store } s) Y s)$

syntax

-peek-st :: $pttrn \Rightarrow 'a \text{ assn} \Rightarrow 'a \text{ assn}$ $(\langle \lambda \dots \rightarrow [0,3] \ 3)$

syntax-consts

-peek-st == *peek-st*

translations

$\lambda s.. P == \text{CONST } \text{peek-st } (\lambda s. P)$

lemma *peek-st-def2* [*simp*]: $(\lambda s.. Pf s) Y s = Pf (\text{store } s) Y s$
apply (*unfold peek-st-def*)
apply (*simp* (*no-asm*))
done

lemma *peek-st-triv* [*simp*]: $(\lambda s.. P) = P$
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*]: $(\lambda s.. \lambda Y s'. P s Y s') = (\lambda Y s. P (\text{store } s) Y s)$
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*]: $(\lambda s.. (\text{Normal } (P s))) = \text{Normal } (\lambda s.. P s)$

```

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 ( $\hookrightarrow$   $\downarrow$   $\Rightarrow$ ) [60,61] 60)
  where  $P \downarrow = w \equiv (\lambda Y s. P \downarrow \wedge. (\lambda s. Y = w))$ 

```

```

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  $\Rightarrow$  vvar  $\times$  state)  $\Rightarrow$  'a assn  $\Rightarrow$  'a assn (infixr  $\hookrightarrow$   $\downarrow$  13)
  where  $(vf \ ..; P) = (\lambda Y \ s. \text{let } (v, s') = vf \ s \text{ in } P \ (Var \ v) \ s')$ 

```

```

lemma RefVar-def2 [simp]:  $(vf \ ..; P) \ Y \ s =$ 
   $P \ (Var \ (fst \ (vf \ s))) \ (snd \ (vf \ s))$ 
apply (unfold RefVar-def Let-def)
apply (simp (no-asm) add: split-beta)
done

```

12 allocation

definition

```

  Alloc :: prog  $\Rightarrow$  obj-tag  $\Rightarrow$  'a assn  $\Rightarrow$  'a assn
  where  $Alloc \ G \ otag \ P = (\lambda Y \ s \ Z. \forall s' \ a. G \vdash s - \text{halloc } otag \succ a \rightarrow s' \longrightarrow P \ (Val \ (Addr \ a)) \ s' \ Z)$ 

```

definition

```

  SXAlloc :: prog  $\Rightarrow$  'a assn  $\Rightarrow$  'a assn
  where  $SXAlloc \ G \ P = (\lambda Y \ s \ Z. \forall s'. G \vdash s - \text{salloc} \rightarrow s' \longrightarrow P \ Y \ s' \ Z)$ 

```

lemma *Alloc-def2* [simp]: $\text{Alloc } G \text{ otag } P \ Y \ s \ Z =$
 $(\forall s' \ a. \ G \vdash s \text{ --halloc otag--> } a \rightarrow s' \longrightarrow P \ (\text{Val } (\text{Addr } a)) \ s' \ Z)$
apply (*unfold Alloc-def*)
apply (*simp (no-asm)*)
done

lemma *SXAlloc-def2* [simp]:
 $\text{SXAlloc } G \ P \ Y \ s \ Z = (\forall s'. \ G \vdash s \text{ --salloc--> } s' \longrightarrow P \ Y \ s' \ Z)$
apply (*unfold SXAlloc-def*)
apply (*simp (no-asm)*)
done

validity

definition

type-ok :: *prog* \Rightarrow *term* \Rightarrow *state* \Rightarrow *bool* **where**
type-ok *G t s* =
 $(\exists L \ T \ C \ A. \ (\text{normal } s \longrightarrow (\text{prg}=G, \text{cls}=C, \text{lcl}=L) \vdash t :: T \wedge$
 $(\text{prg}=G, \text{cls}=C, \text{lcl}=L) \vdash \text{dom } (\text{locals } (\text{store } s)) \gg t \gg A) \wedge s :: \preceq (G, L))$

datatype 'a *triple* = *triple* ('a *assn*) *term* ('a *assn*)
 $(\langle \{(1-)\} / \text{--}\rangle / \{(1-)\})$ [3,65,3] 75)

type-synonym 'a *triples* = 'a *triple* *set*

abbreviation

var-triple :: ['a *assn*, *var* , 'a *assn*] \Rightarrow 'a *triple*
 $(\langle \{(1-)\} / \text{--}\rangle / \{(1-)\})$ [3,80,3] 75)
where $\{P\} \ e \text{--}\rangle \{Q\} == \{P\} \ \text{In2} \ e \text{--}\rangle \{Q\}$

abbreviation

expr-triple :: ['a *assn*, *expr* , 'a *assn*] \Rightarrow 'a *triple*
 $(\langle \{(1-)\} / \text{--}\rangle / \{(1-)\})$ [3,80,3] 75)
where $\{P\} \ e \text{--}\rangle \{Q\} == \{P\} \ \text{In1l} \ e \text{--}\rangle \{Q\}$

abbreviation

exprs-triple :: ['a *assn*, *expr list* , 'a *assn*] \Rightarrow 'a *triple*
 $(\langle \{(1-)\} / \text{--}\rangle / \{(1-)\})$ [3,65,3] 75)
where $\{P\} \ e \text{--}\rangle \{Q\} == \{P\} \ \text{In3} \ e \text{--}\rangle \{Q\}$

abbreviation

stmt-triple :: ['a *assn*, *stmt*, 'a *assn*] \Rightarrow 'a *triple*
 $(\langle \{(1-)\} / \text{--}\rangle / \{(1-)\})$ [3,65,3] 75)
where $\{P\} \ .c. \{Q\} == \{P\} \ \text{In1r} \ c \text{--}\rangle \{Q\}$

notation (ASCII)

triple $(\langle \{(1-)\} / \text{--}\rangle / \{(1-)\})$ [3,65,3] 75) **and**
var-triple $(\langle \{(1-)\} / \text{--}\rangle / \{(1-)\})$ [3,80,3] 75) **and**
expr-triple $(\langle \{(1-)\} / \text{--}\rangle / \{(1-)\})$ [3,80,3] 75) **and**
exprs-triple $(\langle \{(1-)\} / \text{--}\rangle / \{(1-)\})$ [3,65,3] 75)

lemma *inj-triple*: *inj* $(\lambda(P, t, Q). \{P\} \ t \text{--}\rangle \{Q\})$
apply (*rule inj-onI*)
apply *auto*
done

lemma *triple-inj-eq*: $(\{P\} \ t \text{--}\rangle \{Q\} = \{P'\} \ t' \text{--}\rangle \{Q'\}) = (P=P' \wedge t=t' \wedge Q=Q')$
apply *auto*
done

definition $mtriples :: ('c \Rightarrow 'sig \Rightarrow 'a \text{ assn}) \Rightarrow ('c \Rightarrow 'sig \Rightarrow \text{expr}) \Rightarrow$
 $('c \Rightarrow 'sig \Rightarrow 'a \text{ assn}) \Rightarrow ('c \times 'sig) \text{ set} \Rightarrow 'a \text{ triples } (\langle \{(1-\)} / \text{ --} \succ / \{(1-\)} \rangle \mid - \rangle \rangle [3,65,3,65] 75)$

where

$\{\{P\} \text{ tf} - \succ \{Q\} \mid ms\} = (\lambda(C, sig). \{Normal(P \ C \ sig)\} \text{ tf } C \ sig - \succ \{Q \ C \ sig\}) 'ms$

definition

$\text{triple-valid} :: \text{prog} \Rightarrow \text{nat} \Rightarrow 'a \text{ triple} \Rightarrow \text{bool} \ (\langle - \mid \vdash - \rangle \rightarrow [61,0, 58] 57)$

where

$G \models n:t =$
 $(\text{case } t \text{ of } \{P\} \ t \succ \{Q\} \Rightarrow$
 $\forall Y \ s \ Z. P \ Y \ s \ Z \longrightarrow \text{type-ok } G \ t \ s \longrightarrow$
 $(\forall Y' \ s'. G \vdash s - t \succ - n \rightarrow (Y', s') \longrightarrow Q \ Y' \ s' \ Z))$

abbreviation

$\text{triples-valid} :: \text{prog} \Rightarrow \text{nat} \Rightarrow 'a \text{ triples} \Rightarrow \text{bool} \ (\langle - \mid \vdash - \rangle \rightarrow [61,0, 58] 57)$

where $G \models n:ts == \text{Ball } ts \ (\text{triple-valid } G \ n)$

notation (ASCII)

$\text{triples-valid} \ (\ \langle - \mid \vdash - \rangle \rightarrow [61,0, 58] 57)$

definition

$\text{ax-valids} :: \text{prog} \Rightarrow 'b \text{ triples} \Rightarrow 'a \text{ triples} \Rightarrow \text{bool} \ (\langle -, - \mid \vdash - \rangle [61,58,58] 57)$

where $(G, A \mid \vdash ts) = (\forall n. G \models n:A \longrightarrow G \models n:ts)$

abbreviation

$\text{ax-valid} :: \text{prog} \Rightarrow 'b \text{ triples} \Rightarrow 'a \text{ triple} \Rightarrow \text{bool} \ (\langle -, - \mid \vdash - \rangle [61,58,58] 57)$

where $G, A \mid \vdash t == G, A \mid \vdash \{t\}$

notation (ASCII)

$\text{ax-valid} \ (\ \langle -, - \mid \vdash - \rangle [61,58,58] 57)$

lemma $\text{triple-valid-def2}: G \models n:\{P\} \ t \succ \{Q\} =$

$(\forall Y \ s \ Z. P \ Y \ s \ Z$
 $\longrightarrow (\exists L. (\text{normal } s \longrightarrow (\exists \ C \ T \ A. (\text{prg}=G, \text{cls}=C, \text{lcl}=L) \vdash t :: T \wedge$
 $(\text{prg}=G, \text{cls}=C, \text{lcl}=L) \vdash \text{dom } (\text{locals } (\text{store } s)) \gg t \gg A)) \wedge$
 $s :: \preceq (G, L))$
 $\longrightarrow (\forall Y' \ s'. G \vdash s - t \succ - n \rightarrow (Y', s') \longrightarrow Q \ Y' \ s' \ Z))$

apply $(\text{unfold } \text{triple-valid-def } \text{type-ok-def})$

apply $(\text{simp } (\text{no-asm}))$

done

declare $\text{split-paired-All} \ [\text{simp del}] \ \text{split-paired-Ex} \ [\text{simp del}]$

declare $\text{if-split} \ [\text{split del}] \ \text{if-split-asm} \ [\text{split del}]$

$\text{option.split} \ [\text{split del}] \ \text{option.split-asm} \ [\text{split del}]$

setup $\langle \text{map-theory-simpset } (\text{fn } \text{ctxt} \Rightarrow \text{ctxt} \mid \succ \text{Simplifier.del-loop } \text{split-all-tac}) \rangle$

setup $\langle \text{map-theory-claset } (\text{fn } \text{ctxt} \Rightarrow \text{ctxt delSWrapper } \text{split-all-tac}) \rangle$

inductive

$\text{ax-derivs} :: \text{prog} \Rightarrow 'a \text{ triples} \Rightarrow 'a \text{ triples} \Rightarrow \text{bool} \ (\langle -, - \mid \vdash - \rangle [61,58,58] 57)$

and $\text{ax-deriv} :: \text{prog} \Rightarrow 'a \text{ triples} \Rightarrow 'a \text{ triple} \Rightarrow \text{bool} \ (\langle -, - \mid \vdash - \rangle [61,58,58] 57)$

for $G :: \text{prog}$

where

$G, A \vdash t \equiv G, A \mid \vdash \{t\}$

- | *empty*: $G, A \vdash \{\}$
- | *insert*: $\llbracket G, A \vdash t; G, A \vdash ts \rrbracket \implies G, A \vdash \text{insert } t \text{ } ts$

- | *asm*: $ts \subseteq A \implies G, A \vdash ts$

- | *weaken*: $\llbracket G, A \vdash ts'; ts \subseteq ts' \rrbracket \implies G, A \vdash ts$

- | *conseq*: $\forall Y \ s \ Z \ . \ P \ Y \ s \ Z \longrightarrow (\exists P' \ Q'. \ G, A \vdash \{P'\} \ t \succ \{Q'\} \wedge (\forall Y' \ s' \ Z'. \ P' \ Y' \ s' \ Z' \longrightarrow Q' \ Y' \ s' \ Z')) \implies G, A \vdash \{P\} \ t \succ \{Q\}$

- | *hazard*: $G, A \vdash \{P \wedge. \text{Not} \circ \text{type-ok } G \ t\} \ t \succ \{Q\}$

- | *Abrupt*: $G, A \vdash \{P \leftarrow (\text{undefined3 } t) \wedge. \text{Not} \circ \text{normal}\} \ t \succ \{P\}$

- variables
- | *LVar*: $G, A \vdash \{\text{Normal } (\lambda s.. P \leftarrow \text{Var } (lvar \ vn \ s))\} \ LVar \ vn = \succ \{P\}$

- | *FVar*: $\llbracket G, A \vdash \{\text{Normal } P\} \ . \text{Init } C. \{Q\}; G, A \vdash \{Q\} \ e \succ \{\lambda Val: a.. fvar \ C \ stat \ fn \ a \ ..; R\} \rrbracket \implies G, A \vdash \{\text{Normal } P\} \{accC, C, stat\} e..fn = \succ \{R\}$

- | *AVar*: $\llbracket G, A \vdash \{\text{Normal } P\} \ e1 \succ \{Q\}; \forall a. \ G, A \vdash \{Q \leftarrow Val \ a\} \ e2 \succ \{\lambda Val: i.. avar \ G \ i \ a \ ..; R\} \rrbracket \implies G, A \vdash \{\text{Normal } P\} \ e1.[e2] = \succ \{R\}$
- expressions

- | *NewC*: $\llbracket G, A \vdash \{\text{Normal } P\} \ . \text{Init } C. \{\text{Alloc } G \ (CInst \ C) \ Q\} \rrbracket \implies G, A \vdash \{\text{Normal } P\} \ NewC \ C \succ \{Q\}$

- | *NewA*: $\llbracket G, A \vdash \{\text{Normal } P\} \ . \text{init-comp-ty } T. \{Q\}; G, A \vdash \{Q\} \ e \succ \{\lambda Val: i.. abupd \ (\text{check-neg } i) \ .; \text{Alloc } G \ (\text{Arr } T \ (\text{the-Intg } i)) \ R\} \rrbracket \implies G, A \vdash \{\text{Normal } P\} \ New \ T[e] \succ \{R\}$

- | *Cast*: $\llbracket G, A \vdash \{\text{Normal } P\} \ e \succ \{\lambda Val: v.. \lambda s.. abupd \ (\text{raise-if } (\neg G, s \vdash v \text{ fits } T) \ \text{ClassCast}) \ .; Q \leftarrow Val \ v\} \rrbracket \implies G, A \vdash \{\text{Normal } P\} \ Cast \ T \ e \succ \{Q\}$

- | *Inst*: $\llbracket G, A \vdash \{\text{Normal } P\} \ e \succ \{\lambda Val: v.. \lambda s.. Q \leftarrow Val \ (\text{Bool } (v \neq \text{Null} \wedge G, s \vdash v \text{ fits } RefT \ T))\} \rrbracket \implies G, A \vdash \{\text{Normal } P\} \ e \ InstOf \ T \succ \{Q\}$

- | *Lit*: $G, A \vdash \{\text{Normal } (P \leftarrow Val \ v)\} \ Lit \ v \succ \{P\}$

- | *UnOp*: $\llbracket G, A \vdash \{\text{Normal } P\} \ e \succ \{\lambda Val: v.. Q \leftarrow Val \ (\text{eval-unop } unop \ v)\} \rrbracket \implies G, A \vdash \{\text{Normal } P\} \ UnOp \ unop \ e \succ \{Q\}$

- | *BinOp*: $\llbracket G, A \vdash \{\text{Normal } P\} \ e1 \succ \{Q\}; \forall v1. \ G, A \vdash \{Q \leftarrow Val \ v1\} \ (\text{if need-second-arg binop } v1 \text{ then } (In1l \ e2) \text{ else } (In1r \ Skip)) \succ \{\lambda Val: v2.. R \leftarrow Val \ (\text{eval-binop } binop \ v1 \ v2)\} \rrbracket \implies G, A \vdash \{\text{Normal } P\} \ BinOp \ binop \ e1 \ e2 \succ \{R\}$

- | *Super*: $G, A \vdash \{ \text{Normal } (\lambda s.. P \leftarrow \text{Val } (\text{val-this } s)) \} \text{ Super} \multimap \{P\}$
- | *Acc*: $\llbracket G, A \vdash \{ \text{Normal } P \} \text{ va} \multimap \{ \lambda \text{Var}:(v,f):. Q \leftarrow \text{Val } v \} \rrbracket \implies$
 $G, A \vdash \{ \text{Normal } P \} \text{ Acc va} \multimap \{Q\}$
- | *Ass*: $\llbracket G, A \vdash \{ \text{Normal } P \} \text{ va} \multimap \{Q\};$
 $\forall \text{vf}. G, A \vdash \{ Q \leftarrow \text{Var } \text{vf} \} e \multimap \{ \lambda \text{Val}:v:.. \text{assign } (\text{snd } \text{vf}) \text{ v } .; R \} \rrbracket \implies$
 $G, A \vdash \{ \text{Normal } P \} \text{ va} := e \multimap \{R\}$
- | *Cond*: $\llbracket G, A \vdash \{ \text{Normal } P \} e0 \multimap \{P'\};$
 $\forall b. G, A \vdash \{ P' \leftarrow b \} (\text{if } b \text{ then } e1 \text{ else } e2) \multimap \{Q\} \rrbracket \implies$
 $G, A \vdash \{ \text{Normal } P \} e0 \text{ ? } e1 : e2 \multimap \{Q\}$
- | *Call*:
 $\llbracket G, A \vdash \{ \text{Normal } P \} e \multimap \{Q\}; \forall a. G, A \vdash \{ Q \leftarrow \text{Val } a \} \text{ args} \multimap \{R \text{ a}\};$
 $\forall a \text{ vs invC declC l}. G, A \vdash \{ (R \text{ a} \leftarrow \text{Vals vs } \wedge.$
 $(\lambda s. \text{declC} = \text{invocation-declclass } G \text{ mode } (\text{store } s) \text{ a statT } (\text{name} = \text{mn}, \text{parTs} = \text{pTs})) \wedge$
 $\text{invC} = \text{invocation-class mode } (\text{store } s) \text{ a statT } \wedge$
 $\text{l} = \text{locals } (\text{store } s)) .;$
 $\text{init-lvars } G \text{ declC } (\text{name} = \text{mn}, \text{parTs} = \text{pTs}) \text{ mode } a \text{ vs} \} \wedge.$
 $(\lambda s. \text{normal } s \longrightarrow G \vdash \text{mode} \rightarrow \text{invC} \preceq \text{statT}) \rrbracket$
 $\text{Methd declC } (\text{name} = \text{mn}, \text{parTs} = \text{pTs}) \multimap \{ \text{set-lvars l } .; S \} \rrbracket \implies$
 $G, A \vdash \{ \text{Normal } P \} \{ \text{accC}, \text{statT}, \text{mode} \} e \cdot \text{mn}(\{ \text{pTs} \} \text{args}) \multimap \{S\}$
- | *Methd*: $\llbracket G, A \cup \{ \{P\} \text{ Methd} \multimap \{Q\} \mid \text{ms} \} \vdash \{ \{P\} \text{ body } G \multimap \{Q\} \mid \text{ms} \} \rrbracket \implies$
 $G, A \vdash \{ \{P\} \text{ Methd} \multimap \{Q\} \mid \text{ms} \}$
- | *Body*: $\llbracket G, A \vdash \{ \text{Normal } P \} . \text{Init } D. \{Q\};$
 $G, A \vdash \{Q\} .c. \{ \lambda s.. \text{abupd } (\text{absorb Ret}) .; R \leftarrow (\text{In1 } (\text{the } (\text{locals } s \text{ Result}))) \} \rrbracket$
 \implies
 $G, A \vdash \{ \text{Normal } P \} \text{ Body } D \text{ c} \multimap \{R\}$
- expression lists
- | *Nil*: $G, A \vdash \{ \text{Normal } (P \leftarrow \text{Vals } []) \} [] \multimap \{P\}$
- | *Cons*: $\llbracket G, A \vdash \{ \text{Normal } P \} e \multimap \{Q\};$
 $\forall v. G, A \vdash \{ Q \leftarrow \text{Val } v \} \text{ es} \multimap \{ \lambda \text{Vals}:vs:.. R \leftarrow \text{Vals } (v \# \text{vs}) \} \rrbracket \implies$
 $G, A \vdash \{ \text{Normal } P \} e \# \text{es} \multimap \{R\}$
- statements
- | *Skip*: $G, A \vdash \{ \text{Normal } (P \leftarrow \diamond) \} . \text{Skip}. \{P\}$
- | *Expr*: $\llbracket G, A \vdash \{ \text{Normal } P \} e \multimap \{ Q \leftarrow \diamond \} \rrbracket \implies$
 $G, A \vdash \{ \text{Normal } P \} . \text{Expr } e. \{Q\}$
- | *Lab*: $\llbracket G, A \vdash \{ \text{Normal } P \} .c. \{ \text{abupd } (\text{absorb l}) .; Q \} \rrbracket \implies$
 $G, A \vdash \{ \text{Normal } P \} .l. c. \{Q\}$
- | *Comp*: $\llbracket G, A \vdash \{ \text{Normal } P \} .c1. \{Q\};$
 $G, A \vdash \{Q\} .c2. \{R\} \rrbracket \implies$
 $G, A \vdash \{ \text{Normal } P \} .c1;;c2. \{R\}$
- | *If*: $\llbracket G, A \vdash \{ \text{Normal } P \} e \multimap \{P'\};$
 $\forall b. G, A \vdash \{ P' \leftarrow b \} .(\text{if } b \text{ then } c1 \text{ else } c2). \{Q\} \rrbracket \implies$
 $G, A \vdash \{ \text{Normal } P \} . \text{If}(e) \text{ c1 Else } c2. \{Q\}$
- | *Loop*: $\llbracket G, A \vdash \{P\} e \multimap \{P'\};$

$$\begin{array}{l}
G, A \vdash \{ \text{Normal } (P' \leftarrow \text{True}) \} .c. \{ \text{abupd } (\text{absorb } (\text{Cont } l)) .; P \} \implies \\
G, A \vdash \{ P \} .l. \text{While}(e) \text{ } c. \{ (P' \leftarrow \text{False}) \downarrow = \Diamond \} \\
| \text{Jmp: } G, A \vdash \{ \text{Normal } (\text{abupd } (\lambda a. (\text{Some } (\text{Jump } j)))) .; P \leftarrow \Diamond \} .\text{Jmp } j. \{ P \} \\
| \text{Throw: } \llbracket G, A \vdash \{ \text{Normal } P \} \text{ } e \multimap \{ \lambda \text{Val}:a:. \text{abupd } (\text{throw } a) .; Q \leftarrow \Diamond \} \rrbracket \implies \\
G, A \vdash \{ \text{Normal } P \} .\text{Throw } e. \{ Q \} \\
| \text{Try: } \llbracket G, A \vdash \{ \text{Normal } P \} .c1. \{ \text{SXAlloc } G \text{ } Q \}; \\
G, A \vdash \{ Q \wedge. (\lambda s. G, s \vdash \text{catch } C) ;. \text{new-xcpt-var } vn \} .c2. \{ R \}; \\
(Q \wedge. (\lambda s. \neg G, s \vdash \text{catch } C)) \Rightarrow R \rrbracket \implies \\
G, A \vdash \{ \text{Normal } P \} .\text{Try } c1 \text{ Catch}(C \text{ } vn) \text{ } c2. \{ R \} \\
| \text{Fin: } \llbracket G, A \vdash \{ \text{Normal } P \} .c1. \{ Q \}; \\
\forall x. G, A \vdash \{ Q \wedge. (\lambda s. x = \text{fst } s) ;. \text{abupd } (\lambda x. \text{None}) \} \\
.c2. \{ \text{abupd } (\text{abrupt-if } (x \neq \text{None}) \text{ } x) .; R \} \rrbracket \implies \\
G, A \vdash \{ \text{Normal } P \} .c1 \text{ Finally } c2. \{ R \} \\
| \text{Done: } G, A \vdash \{ \text{Normal } (P \leftarrow \Diamond \wedge. \text{initd } C) \} .\text{Init } C. \{ P \} \\
| \text{Init: } \llbracket \text{the } (\text{class } G \text{ } C) = c; \\
G, A \vdash \{ \text{Normal } ((P \wedge. \text{Not } \circ \text{initd } C) ;. \text{supd } (\text{init-class-obj } G \text{ } C)) \} \\
.\text{if } C = \text{Object then Skip else Init } (\text{super } c)). \{ Q \}; \\
\forall l. G, A \vdash \{ Q \wedge. (\lambda s. l = \text{locals } (\text{store } s)) ;. \text{set-lvars Map.empty} \} \\
.\text{init } c. \{ \text{set-lvars } l .; R \} \rrbracket \implies \\
G, A \vdash \{ \text{Normal } (P \wedge. \text{Not } \circ \text{initd } C) \} .\text{Init } C. \{ R \}
\end{array}$$

— Some dummy rules for the intermediate terms *Callee*, *InsInitE*, *InsInitV*, *FinA* only used by the smallstep semantics.

$$\begin{array}{l}
| \text{InsInitV: } G, A \vdash \{ \text{Normal } P \} \text{ } \text{InsInitV } c \text{ } v \multimap \{ Q \} \\
| \text{InsInitE: } G, A \vdash \{ \text{Normal } P \} \text{ } \text{InsInitE } c \text{ } e \multimap \{ Q \} \\
| \text{Callee: } G, A \vdash \{ \text{Normal } P \} \text{ } \text{Callee } l \text{ } e \multimap \{ Q \} \\
| \text{FinA: } G, A \vdash \{ \text{Normal } P \} .\text{FinA } a \text{ } c. \{ Q \}
\end{array}$$

definition

adapt-pre :: 'a assn \Rightarrow 'a assn \Rightarrow 'a assn \Rightarrow 'a assn

where *adapt-pre* $P \ Q \ Q' = (\lambda Y \ s \ Z. \forall Y' \ s'. \exists Z'. P \ Y \ s \ Z' \wedge (Q \ Y' \ s' \ Z' \longrightarrow Q' \ Y' \ s' \ Z))$

rules derived by induction

lemma *cut-valid*: $\llbracket G, A' \models ts; G, A \models A \rrbracket \implies G, A \models ts$

apply (*unfold ax-valids-def*)

apply *fast*

done

lemma *ax-thin* [*rule-format* (*no-asm*)]:

$G, (A'::'a \text{ triple set}) \vdash (ts::'a \text{ triple set}) \implies \forall A. A' \subseteq A \longrightarrow G, A \vdash ts$

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 $\langle \text{TRYALL (resolve-tac **context** ((funpow 5 tl) @\{thms ax-derivs.intros\}))} \rangle$)

apply *auto*

done

lemma *ax-thin-insert*: $G, (A :: 'a \text{ triple set}) \vdash (t :: 'a \text{ triple}) \implies G, \text{insert } x \text{ A} \vdash t$

apply (erule *ax-thin*)

apply *fast*

done

lemma *subset-mtriples-iff*:

$ts \subseteq \{\{P\} \text{ mb} \multimap \{Q\} \mid ms\} = (\exists ms'. ms' \subseteq ms \wedge ts = \{\{P\} \text{ mb} \multimap \{Q\} \mid ms'\})$

apply (unfold *mtriples-def*)

apply (rule *subset-image-iff*)

done

lemma *weaken*:

$G, (A :: 'a \text{ triple set}) \vdash (ts' :: 'a \text{ triple set}) \implies \forall ts. ts \subseteq ts' \longrightarrow G, A \vdash ts$

apply (erule *ax-derivs.induct*)

apply (tactic *ALLGOALS (strip-tac **context**)*)

apply (tactic $\langle \text{ALLGOALS (REPEAT o (EVERY [dresolve-tac **context** @\{thms subset-singletonD\},$
 $\text{eresolve-tac **context** [disjE],$
 $\text{fast-tac (**context** addSIs @\{thms ax-derivs.empty\})} \rangle \rangle \rangle$)

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 *simp-tac **context** 3 1*, *blast*)

apply (tactic $\langle \text{TRYALL (resolve-tac **context** ((funpow 5 tl) @\{thms ax-derivs.intros\})$
 $\text{THEN-ALL-NEW fast-tac **context**)} \rangle$)

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: $G, A \vdash \{P\} t \multimap \{Q\}$. Given only the term above without annotations, Isabelle would infer a more general type were we could have different types of auxiliary variables in the assumption set (A) and in the triple itself (P and Q). But *ax-derivs.Methd* enforces the same type in the inductive definition of the derivation. So we have to restrict the types to be able to apply the rules.

lemma *conseq12*: $\llbracket G, (A :: 'a \text{ triple set}) \vdash \{P' :: 'a \text{ assn}\} t \multimap \{Q'\};$

$\forall Y s Z. P Y s Z \longrightarrow (\forall Y' s'. (\forall Y Z'. P' Y s Z' \longrightarrow Q' Y' s' Z') \longrightarrow$
 $Q Y' s' Z) \rrbracket$

$\implies G, A \vdash \{P :: 'a \text{ assn}\} t \multimap \{Q\}$

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'*: $\llbracket G, (A::'a \text{ triple set}) \vdash \{P'::'a \text{ assn}\} \text{ } t> \{Q'\}; \forall s \ Y' \ s'. \$
 $(\forall Y \ Z. \ P' \ Y \ s \ Z \longrightarrow Q' \ Y' \ s' \ Z) \longrightarrow$
 $(\forall Y \ Z. \ P \ Y \ s \ Z \longrightarrow Q \ Y' \ s' \ Z) \rrbracket$
 $\implies G, A \vdash \{P::'a \text{ assn}\} \text{ } t> \{Q\}$

apply (*erule conseq12*)
apply *fast*
done

lemma *conseq12-from-conseq12'*: $\llbracket G, (A::'a \text{ triple set}) \vdash \{P'::'a \text{ assn}\} \text{ } t> \{Q'\}; \$
 $\forall Y \ s \ Z. \ P \ Y \ s \ Z \longrightarrow (\forall Y' \ s'. \ (\forall Y \ Z'. \ P' \ Y \ s \ Z' \longrightarrow Q' \ Y' \ s' \ Z') \longrightarrow$
 $Q \ Y' \ s' \ Z) \rrbracket$
 $\implies G, A \vdash \{P::'a \text{ assn}\} \text{ } t> \{Q\}$

apply (*erule conseq12'*)
apply *blast*
done

lemma *conseq1*: $\llbracket G, (A::'a \text{ triple set}) \vdash \{P'::'a \text{ assn}\} \text{ } t> \{Q\}; P \Rightarrow P' \rrbracket$
 $\implies G, A \vdash \{P::'a \text{ assn}\} \text{ } t> \{Q\}$
apply (*erule conseq12*)
apply *blast*
done

lemma *conseq2*: $\llbracket G, (A::'a \text{ triple set}) \vdash \{P::'a \text{ assn}\} \text{ } t> \{Q'\}; Q' \Rightarrow Q \rrbracket$
 $\implies G, A \vdash \{P::'a \text{ assn}\} \text{ } t> \{Q\}$
apply (*erule conseq12*)
apply *blast*
done

lemma *ax-escape*:
 $\llbracket \forall Y \ s \ Z. \ P \ Y \ s \ Z \longrightarrow G, (A::'a \text{ triple set}) \vdash \{\lambda Y' \ s'. \ (Z'::'a). \ (Y', s') = (Y, s)\} \text{ } t>$
 $\{\lambda Y \ s \ Z'. \ Q \ Y \ s \ Z\} \rrbracket$
 $\rrbracket \implies G, A \vdash \{P::'a \text{ assn}\} \text{ } t> \{Q::'a \text{ assn}\}$
apply (*rule ax-derivs.conseq*)
apply *force*
done

lemma *ax-constant*: $\llbracket C \implies G, (A::'a \text{ triple set}) \vdash \{P::'a \text{ assn}\} \text{ } t> \{Q\} \rrbracket$
 $\implies G, A \vdash \{\lambda Y \ s \ Z. \ C \wedge P \ Y \ s \ Z\} \text{ } t> \{Q\}$
apply (*rule ax-escape*)
apply *clarify*
apply (*rule conseq12*)
apply *fast*
apply *auto*
done

lemma *ax-impossible* [*intro*]:
 $G, (A::'a \text{ triple set}) \vdash \{\lambda Y \ s \ Z. \ \text{False}\} \text{ } t> \{Q::'a \text{ assn}\}$
apply (*rule ax-escape*)
apply *clarify*

done

lemma *ax-nochange-lemma*: $\llbracket P \ Y \ s; \text{All } ((=) \ w) \rrbracket \implies P \ w \ s$
apply *auto*
done

lemma *ax-nochange*:
 $G, (A::(\text{res} \times \text{state}) \ \text{triple} \ \text{set}) \vdash \{\lambda Y \ s \ Z. (Y, s) = Z\} \ t \succ \{\lambda Y \ s \ Z. (Y, s) = Z\}$
 $\implies G, A \vdash \{P::(\text{res} \times \text{state}) \ \text{assn}\} \ t \succ \{P\}$
apply (*erule* *conseq12*)
apply *auto*
apply (*erule* (1) *ax-nochange-lemma*)
done

lemma *ax-trivial*: $G, (A::'a \ \text{triple} \ \text{set}) \vdash \{P::'a \ \text{assn}\} \ t \succ \{\lambda Y \ s \ Z. \text{True}\}$
apply (*rule* *ax-derivs.conseq*)
apply *auto*
done

lemma *ax-disj*:
 $\llbracket G, (A::'a \ \text{triple} \ \text{set}) \vdash \{P1::'a \ \text{assn}\} \ t \succ \{Q1\}; G, A \vdash \{P2::'a \ \text{assn}\} \ t \succ \{Q2\} \rrbracket$
 $\implies G, A \vdash \{\lambda Y \ s \ Z. P1 \ Y \ s \ Z \vee P2 \ Y \ s \ Z\} \ t \succ \{\lambda Y \ s \ Z. Q1 \ Y \ s \ Z \vee Q2 \ Y \ s \ Z\}$
apply (*rule* *ax-escape*)
apply *safe*
apply (*erule* *conseq12*, *fast*) +
done

lemma *ax-supd-shuffle*:
 $(\exists Q. G, (A::'a \ \text{triple} \ \text{set}) \vdash \{P::'a \ \text{assn}\} \ .c1. \{Q\} \wedge G, A \vdash \{Q ; . f\} \ .c2. \{R\}) =$
 $(\exists Q'. G, A \vdash \{P\} \ .c1. \{f ; Q'\} \wedge G, A \vdash \{Q'\} \ .c2. \{R\})$
apply (*best elim!*: *conseq1* *conseq2*)
done

lemma *ax-cases*:
 $\llbracket G, (A::'a \ \text{triple} \ \text{set}) \vdash \{P \wedge. C\} \ t \succ \{Q::'a \ \text{assn}\};$
 $G, A \vdash \{P \wedge. \text{Not} \circ C\} \ t \succ \{Q\} \rrbracket \implies G, A \vdash \{P\} \ t \succ \{Q\}$
apply (*unfold* *peek-and-def*)
apply (*rule* *ax-escape*)
apply *clarify*
apply (*case-tac* *C s*)
apply (*erule* *conseq12*, *force*) +
done

lemma *ax-adapt*: $G, (A::'a \ \text{triple} \ \text{set}) \vdash \{P::'a \ \text{assn}\} \ t \succ \{Q\}$
 $\implies G, A \vdash \{\text{adapt-pre } P \ Q \ Q'\} \ t \succ \{Q'\}$
apply (*unfold* *adapt-pre-def*)
apply (*erule* *conseq12*)
apply *fast*
done

lemma *adapt-pre-adapts*: $G, (A::'a \ \text{triple} \ \text{set}) \models \{P::'a \ \text{assn}\} \ t \succ \{Q\}$
 $\longrightarrow G, A \models \{\text{adapt-pre } P \ Q \ Q'\} \ t \succ \{Q'\}$
apply (*unfold* *adapt-pre-def*)
apply (*simp* *add*: *ax-valids-def* *triple-valid-def2*)

apply *fast*
done

lemma *adapt-pre-weakest*:

$\forall G, (A::'a \text{ triple set}) \ t. \ G, A \models \{P\} \ t \succ \{Q\} \longrightarrow G, A \models \{P'\} \ t \succ \{Q'\} \implies$
 $P' \Rightarrow \text{adapt-pre } P \ Q \ (Q'::'a \text{ assn})$

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*:

$G, (A::'a \text{ triple set}) \vdash \{\text{Normal } P\} \ t \succ \{Q::'a \text{ assn}\}$

$\implies G, A \vdash \{\text{Normal } (P \wedge. p)\} \ t \succ \{Q\}$

apply (*erule conseq1*)

apply (*simp (no-asm)*)

done

lemma *peek-and-forget1*:

$G, (A::'a \text{ triple set}) \vdash \{P::'a \text{ assn}\} \ t \succ \{Q\}$

$\implies G, A \vdash \{P \wedge. p\} \ t \succ \{Q\}$

apply (*erule conseq1*)

apply (*simp (no-asm)*)

done

lemmas *ax-NormalD = peek-and-forget1 [of - - - - normal]*

lemma *peek-and-forget2*:

$G, (A::'a \text{ triple set}) \vdash \{P::'a \text{ assn}\} \ t \succ \{Q \wedge. p\}$

$\implies G, A \vdash \{P\} \ t \succ \{Q\}$

apply (*erule conseq2*)

apply (*simp (no-asm)*)

done

lemma *ax-subst-Val-allI*:

$\forall v. \ G, (A::'a \text{ triple set}) \vdash \{(P' \quad v) \leftarrow \text{Val } v\} \ t \succ \{(Q \ v)::'a \text{ assn}\}$

$\implies \forall v. \ G, A \vdash \{(\lambda w. \ P' \ (the-In1 \ w)) \leftarrow \text{Val } v\} \ t \succ \{Q \ v\}$

apply (*force elim!: conseq1*)

done

lemma *ax-subst-Var-allI*:

$\forall v. \ G, (A::'a \text{ triple set}) \vdash \{(P' \quad v) \leftarrow \text{Var } v\} \ t \succ \{(Q \ v)::'a \text{ assn}\}$

$\implies \forall v. \ G, A \vdash \{(\lambda w. \ P' \ (the-In2 \ w)) \leftarrow \text{Var } v\} \ t \succ \{Q \ v\}$

apply (*force elim!: conseq1*)

done

lemma *ax-subst-Vals-allI*:

$(\forall v. \ G, (A::'a \text{ triple set}) \vdash \{(P' \quad v) \leftarrow \text{Vals } v\} \ t \succ \{(Q \ v)::'a \text{ assn}\})$

$\implies \forall v. \ G, A \vdash \{(\lambda w. \ P' \ (the-In3 \ w)) \leftarrow \text{Vals } v\} \ t \succ \{Q \ v\}$

apply (*force elim!: conseq1*)

done

alternative axioms

lemma *ax-Lit2*:

$G, (A::'a \text{ triple set}) \vdash \{\text{Normal } P::'a \text{ assn}\} \ Lit \ v \prec \{\text{Normal } (P \downarrow = \text{Val } v)\}$

apply (*rule* *ax-derivs.Lit* [*THEN* *conseq1*])

apply *force*

done

lemma *ax-Lit2-test-complete*:

$G, (A :: 'a \text{ triple set}) \vdash \{ \text{Normal } (P \leftarrow \text{Val } v) :: 'a \text{ assn} \} \text{ Lit } v \multimap \{ P \}$

apply (*rule* *ax-Lit2* [*THEN* *conseq2*])

apply *force*

done

lemma *ax-LVar2*: $G, (A :: 'a \text{ triple set}) \vdash \{ \text{Normal } P :: 'a \text{ assn} \} \text{ LVar } vn \multimap \{ \text{Normal } (\lambda s.. P \downarrow = \text{Var } (lvar \text{ } vn \text{ } s)) \}$

apply (*rule* *ax-derivs.LVar* [*THEN* *conseq1*])

apply *force*

done

lemma *ax-Super2*: $G, (A :: 'a \text{ triple set}) \vdash$

$\{ \text{Normal } P :: 'a \text{ assn} \} \text{ Super} \multimap \{ \text{Normal } (\lambda s.. P \downarrow = \text{Val } (val\text{-this } s)) \}$

apply (*rule* *ax-derivs.Super* [*THEN* *conseq1*])

apply *force*

done

lemma *ax-Nil2*:

$G, (A :: 'a \text{ triple set}) \vdash \{ \text{Normal } P :: 'a \text{ assn} \} [] \multimap \{ \text{Normal } (P \downarrow = \text{Vals } []) \}$

apply (*rule* *ax-derivs.Nil* [*THEN* *conseq1*])

apply *force*

done

misc derived structural rules

lemma *ax-finite-mtriples-lemma*: $\llbracket F \subseteq ms; \text{finite } ms; \forall (C, sig) \in ms.$

$G, (A :: 'a \text{ triple set}) \vdash \{ \text{Normal } (P \text{ } C \text{ } sig) :: 'a \text{ assn} \} \text{ mb } C \text{ } sig \multimap \{ Q \text{ } C \text{ } sig \} \rrbracket \implies$

$G, A \vdash \{ \{ P \} \text{ mb } \multimap \{ Q \} \mid F \}$

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*:

$G, (A :: 'a \text{ triple set}) \vdash \text{insert } (t :: 'a \text{ triple}) \text{ } ts \implies G, A \vdash t \wedge G, A \vdash ts$

apply (*fast* *intro*: *ax-derivs.weaken*)

done

lemma *ax-methods-spec*:

$\llbracket G, (A :: 'a \text{ triple set}) \vdash \text{case-prod } f \text{ } 'ms; (C, sig) \in ms \rrbracket \implies G, A \vdash ((f \text{ } C \text{ } sig) :: 'a \text{ triple})$

apply (*erule* *ax-derivs.weaken*)

apply (*force* *del*: *image-eqI* *intro*: *rev-image-eqI*)

done

lemma *ax-finite-pointwise-lemma* [*rule-format*]: $\llbracket F \subseteq ms; \text{finite } ms \rrbracket \implies$

$((\forall (C, sig) \in F. G, (A :: 'a \text{ triple set}) \vdash (f \text{ } C \text{ } sig) :: 'a \text{ triple})) \longrightarrow (\forall (C, sig) \in ms. G, A \vdash (g \text{ } C \text{ } sig) :: 'a \text{ triple})) \longrightarrow$

$G, A \vdash \text{case-prod } f \text{ } F \longrightarrow G, A \vdash \text{case-prod } g \text{ } F$

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:
   $G, (A::'a \text{ triple set}) \vdash \{P \wedge. \text{type-ok } G \ t\} \ t \succ \{Q::'a \text{ assn}\} \implies G, A \vdash \{P\} \ t \succ \{Q\}$ 
apply (erule ax-cases)
apply (rule ax-derivs.hazard [THEN conseq1])
apply force
done

```

```

lemma ax-free-wt:
   $(\exists T \ L \ C. (\text{prg}=G, \text{cls}=C, \text{lcl}=L) \vdash t::T) \implies G, (A::'a \text{ triple set}) \vdash \{\text{Normal } P\} \ t \succ \{Q::'a \text{ assn}\} \implies G, A \vdash \{\text{Normal } P\} \ t \succ \{Q\}$ 
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!]:  $G, (A::'a \text{ triple set}) \vdash \{P \leftarrow \Diamond\} . \text{Skip}. \{P::'a \text{ assn}\}$ 
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:
   $\llbracket G \vdash \text{IntVir} \rightarrow C \preceq \text{statT};$ 
   $\forall a \text{ vs } l. G, A \vdash \{(R \ a \leftarrow \text{Vals } \text{vs} \wedge. (\lambda s. l = \text{locals } (\text{store } s)) \text{ ;}$ 
   $\text{init-lvars } G \ C \ (\text{name}=\text{mn}, \text{parTs}=\text{pTs}) \ \text{IntVir } a \ \text{vs})\}$ 
   $\text{Methd } C \ (\text{name}=\text{mn}, \text{parTs}=\text{pTs}) \multimap \{\text{set-lvars } l \text{ ; } S\};$ 
   $\forall a. G, A \vdash \{Q \leftarrow \text{Val } a\} \ \text{args} \multimap$ 
   $\{R \ a \wedge. (\lambda s. C = \text{obj-class } (\text{the } (\text{heap } (\text{store } s) \ (\text{the-Addr } a))) \wedge$ 
   $C = \text{invocation-declclass}$ 
   $G \ \text{IntVir } (\text{store } s) \ a \ \text{statT } (\text{name}=\text{mn}, \text{parTs}=\text{pTs}) \ \}\};$ 
   $G, (A::'a \text{ triple set}) \vdash \{\text{Normal } P\} \ e \multimap \{Q::'a \text{ assn}\}$ 
   $\implies G, A \vdash \{\text{Normal } P\} \ \{\text{accC}, \text{statT}, \text{IntVir}\} e \cdot \text{mn}(\{\text{pTs}\} \text{args}) \multimap \{S\}$ 
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*:

```


$$\llbracket \forall a \text{ vs } l. G, A \vdash \{R \leftarrow \text{Vals } vs \wedge. (\lambda s. l = \text{locals } (\text{store } s)) \};$$


$$\text{init-lvars } G \ C \ (\llbracket \text{name}=\text{mn}, \text{parTs}=\text{pTs} \rrbracket \text{ Static any-Addr } vs)$$


$$\text{Methd } C \ (\llbracket \text{name}=\text{mn}, \text{parTs}=\text{pTs} \rrbracket) \multimap \{\text{set-lvars } l \ ; \ S\};$$


$$G, A \vdash \{\text{Normal } P\} \text{ e} \multimap \{Q\};$$


$$\forall a. G, (A::'a \text{ triple set}) \vdash \{Q \leftarrow \text{Val } a\} \text{ args} \multimap \{(R::\text{val} \Rightarrow 'a \text{ assn}) \ a$$


$$\wedge. (\lambda s. C = \text{invocation-declclass}$$


$$G \text{ Static } (\text{store } s) \ a \ \text{statT } (\llbracket \text{name}=\text{mn}, \text{parTs}=\text{pTs} \rrbracket))\}$$


$$\rrbracket \implies G, A \vdash \{\text{Normal } P\} \{\text{accC}, \text{statT}, \text{Static}\} \text{e} \cdot \text{mn}(\{\text{pTs}\} \text{args}) \multimap \{S\}$$

apply (erule ax-derivs.Call)
apply safe
apply (erule spec)
apply (rule ax-escape, clarsimp)
apply (erule-tac  $V = P \longrightarrow Q$  for  $P \ Q$  in thin-rt)
apply (drule spec, drule spec, drule spec, erule conseq12)
apply (force simp add: init-lvars-def Let-def)
done

```

lemma *ax-Methd1*:

```


$$\llbracket G, A \cup \{\{P\} \text{ Methd} \multimap \{Q\} \mid ms\} \vdash \{\{P\} \text{ body } G \multimap \{Q\} \mid ms\}; (C, sig) \in ms \rrbracket \implies$$


$$G, A \vdash \{\text{Normal } (P \ C \ sig)\} \text{ Methd } C \ sig \multimap \{Q \ C \ sig\}$$

apply (drule ax-derivs.Methd)
apply (unfold mtriples-def)
apply (erule (1) ax-methods-spec)
done

```

lemma *ax-MethdN*:

```


$$G, \text{insert}(\{\text{Normal } P\} \text{ Methd } C \ sig \multimap \{Q\}) \ A \vdash$$


$$\{\text{Normal } P\} \text{ body } G \ C \ sig \multimap \{Q\} \implies$$


$$G, A \vdash \{\text{Normal } P\} \text{ Methd } C \ sig \multimap \{Q\}$$

apply (rule ax-Methd1)
apply (rule-tac [2] singletonI)
apply (unfold mtriples-def)
apply clarsimp
done

```

lemma *ax-StatRef*:

```


$$G, (A::'a \text{ triple set}) \vdash \{\text{Normal } (P \leftarrow \text{Val } \text{Null})\} \text{ StatRef } rt \multimap \{P::'a \text{ assn}\}$$

apply (rule ax-derivs.Cast)
apply (rule ax-Lit2 [THEN conseq2])
apply clarsimp
done

```

rules derived from Init and Done

```

lemma ax-InitS:  $\llbracket \text{the } (\text{class } G \ C) = c; C \neq \text{Object};$ 

$$\forall l. G, A \vdash \{Q \wedge. (\lambda s. l = \text{locals } (\text{store } s)) \}; \text{set-lvars } \text{Map.empty}\}$$


$$\text{.init } c. \{\text{set-lvars } l \ ; \ R\};$$


$$G, A \vdash \{\text{Normal } ((P \wedge. \text{Not } \circ \text{initd } C) \ ; \ \text{supd } (\text{init-class-obj } G \ C))\}$$


$$\text{.Init } (\text{super } c). \{Q\} \rrbracket \implies$$


$$G, (A::'a \text{ triple set}) \vdash \{\text{Normal } (P \wedge. \text{Not } \circ \text{initd } C)\} \text{.Init } C. \{R::'a \text{ assn}\}$$

apply (erule ax-derivs.Init)
apply (simp (no-asm-simp))
apply assumption
done

```

lemma *ax-Init-Skip-lemma*:

$\forall l. G, (A::'a \text{ triple set}) \vdash \{P \leftarrow \Diamond \wedge. (\lambda s. l = \text{locals } (store\ s)) \ ;. \text{set-lvars } l'\}$
 $.Skip. \{(set-lvars\ l \ ;. P)::'a \text{ assn}\}$
apply (rule allI)
apply (rule ax-SkipI)
apply clarsimp
done

lemma ax-triv-InitS: $\llbracket the\ (class\ G\ C) = c; init\ c = Skip; C \neq Object;$
 $P \leftarrow \Diamond \Rightarrow (supd\ (init-class-obj\ G\ C) \ ;. P);$
 $G, A \vdash \{Normal\ (P \wedge. initd\ C)\} .Init\ (super\ c). \{(P \wedge. initd\ C) \leftarrow \Diamond\} \rrbracket \Longrightarrow$
 $G, (A::'a \text{ triple set}) \vdash \{Normal\ P \leftarrow \Diamond\} .Init\ C. \{(P \wedge. initd\ C)::'a \text{ assn}\}$
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: $wf-prog\ G \Longrightarrow G, (A::'a \text{ triple set}) \vdash$
 $\{Normal\ ((supd\ (init-class-obj\ G\ Object) \ ;. P \leftarrow \Diamond) \wedge. Not \circ initd\ Object)\}$
 $.Init\ Object. \{(P \wedge. initd\ Object)::'a \text{ assn}\}$
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: $\llbracket wf-prog\ G;$
 $(P::'a \text{ assn}) \Rightarrow (supd\ (init-class-obj\ G\ Object) \ ;. P) \rrbracket \Longrightarrow$
 $G, (A::'a \text{ triple set}) \vdash \{Normal\ P \leftarrow \Diamond\} .Init\ Object. \{P \wedge. initd\ Object\}$
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:
 $G, (A::'a \text{ triple set}) \vdash \{P::'a \text{ assn}\} .c. \{Normal\ Q\}$
 $\Longrightarrow G, A \vdash \{P\} .c. \{SXAlloc\ G\ Q\}$
apply (erule conseq2)
apply (clarsimp elim!: sxalloc-elim-cases simp add: split-tupled-all)
done

lemma ax-Alloc:
 $G, (A::'a \text{ triple set}) \vdash \{P::'a \text{ assn}\} t \succ$
 $\{Normal\ (\lambda Y\ (x, s)\ Z. (\forall a. new-Addr\ (heap\ s) = Some\ a \longrightarrow$
 $Q\ (Val\ (Addr\ a))\ (Norm(init-obj\ G\ (CInst\ C)\ (Heap\ a)\ s))\ Z)) \wedge.$
 $heap-free\ (Suc\ (Suc\ 0))\}$
 $\Longrightarrow G, A \vdash \{P\} t \succ \{Alloc\ G\ (CInst\ C)\ Q\}$
apply (erule conseq2)
apply (auto elim!: halloc-elim-cases)
done

lemma *ax-Alloc-Arr*:

$G, (A :: 'a \text{ triple set}) \vdash \{P :: 'a \text{ assn}\} \text{ } t \succ$
 $\{\lambda \text{Val} : i. \text{Normal } (\lambda Y \ (x, s) \ Z. \neg \text{the-Intg } i < 0 \wedge$
 $(\forall a. \text{new-Addr } (\text{heap } s) = \text{Some } a \longrightarrow$
 $Q \ (Val \ (\text{Addr } a)) \ (\text{Norm } (\text{init-obj } G \ (\text{Arr } T \ (\text{the-Intg } i)) \ (\text{Heap } a) \ s)) \ Z)) \wedge.$
 $\text{heap-free } (\text{Suc } (\text{Suc } 0))\}$
 \implies
 $G, A \vdash \{P\} \text{ } t \succ \{\lambda \text{Val} : i. \text{abupd } (\text{check-neg } i) \ .; \text{Alloc } G \ (\text{Arr } T \ (\text{the-Intg } i)) \ Q\}$
apply (*erule consec2*)
apply (*auto elim! : halloc-elim-cases*)
done

lemma *ax-SXAlloc-catch-SXcpt*:

$\llbracket G, (A :: 'a \text{ triple set}) \vdash \{P :: 'a \text{ assn}\} \text{ } t \succ$
 $\{(\lambda Y \ (x, s) \ Z. x = \text{Some } (Xcpt \ (\text{Std } xn)) \wedge$
 $(\forall a. \text{new-Addr } (\text{heap } s) = \text{Some } a \longrightarrow$
 $Q \ Y \ (\text{Some } (Xcpt \ (\text{Loc } a)), \text{init-obj } G \ (CInst \ (SXcpt \ xn)) \ (\text{Heap } a) \ s) \ Z))$
 $\wedge. \text{heap-free } (\text{Suc } (\text{Suc } 0))\}\rrbracket$
 \implies
 $G, A \vdash \{P\} \text{ } t \succ \{SXAlloc \ G \ (\lambda Y \ s \ Z. Q \ Y \ s \ Z \wedge G, s \vdash \text{catch } SXcpt \ xn)\}$
apply (*erule consec2*)
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  $\implies$  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 $\multimap$  {Q}
by (auto elim!: evaln-elim-cases simp add: triple-valid2-def2)

```

```

lemma Methd-triple-valid2-SucI:
   $\llbracket G \models n::\{Normal P\} \text{ body } G \text{ C sig} \multimap \{Q\} \rrbracket$ 
 $\implies G \models$ Suc n::{Normal P} Methd C sig $\multimap$  {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))  $\implies$  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 $\cup$  {{P} Methd $\multimap$  {Q} | ms} $\models::$ {{P} body G $\multimap$  {Q} | ms}
  shows G,A $\models::$ {{P} Methd $\multimap$  {Q} | ms}
proof -
  have  $\forall t \in A. G \models n::t \implies \forall t \in \{\{P\} \text{ Methd} \multimap \{Q\} \mid ms\}. G \models n::t$ 
    if rec:  $\bigwedge n. \forall t \in (A \cup \{\{P\} \text{ Methd} \multimap \{Q\} \mid ms\}). G \models n::t$ 
       $\implies \forall t \in \{\{P\} \text{ body } G \multimap \{Q\} \mid ms\}. G \models n::t$ 
    for n
  proof (induct n)

```

```

case 0
show  $\forall t \in \{\{P\} \text{ Methd} \multimap \{Q\} \mid ms\}. G \models 0 :: t$ 
proof -
  have  $G \models 0 :: \{Normal (P \ C \ sig)\} \text{ Methd } C \ sig \multimap \{Q \ C \ sig\}$ 
  if  $(C, sig) \in ms$ 
  for  $C \ sig$ 
  by (rule Methd-triple-valid2-0)
  thus ?thesis
  by (simp add: mtriples-def split-def)
qed
next
case (Suc m)
note hyp =  $\langle \forall t \in A. G \models m :: t \implies \forall t \in \{\{P\} \text{ Methd} \multimap \{Q\} \mid ms\}. G \models m :: t \rangle$ 
note prem =  $\langle \forall t \in A. G \models Suc \ m :: t \rangle$ 
show  $\forall t \in \{\{P\} \text{ Methd} \multimap \{Q\} \mid ms\}. G \models Suc \ m :: t$ 
proof -
  have  $G \models Suc \ m :: \{Normal (P \ C \ sig)\} \text{ Methd } C \ sig \multimap \{Q \ C \ sig\}$ 
  if  $m: (C, sig) \in ms$ 
  for  $C \ sig$ 
  proof -
    from prem have prem-m:  $\forall t \in A. G \models m :: t$ 
    by (rule triples-valid2-Suc)
    hence  $\forall t \in \{\{P\} \text{ Methd} \multimap \{Q\} \mid ms\}. G \models m :: t$ 
    by (rule hyp)
    with prem-m
    have  $\forall t \in (A \cup \{\{P\} \text{ Methd} \multimap \{Q\} \mid ms\}). G \models m :: t$ 
    by (simp add: ball-Un)
    hence  $\forall t \in \{\{P\} \text{ body } G \multimap \{Q\} \mid ms\}. G \models m :: t$ 
    by (rule rec)
    with m have  $G \models m :: \{Normal (P \ C \ sig)\} \text{ body } G \ C \ sig \multimap \{Q \ C \ sig\}$ 
    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:  $\forall s \ t \ n \ Y' \ s'. G \vdash s - t \multimap - n \rightarrow (Y', s') \rightarrow t = c \rightarrow$ 
  Ball A (triple-valid2 G n)  $\rightarrow (\forall Y \ Z. P \ Y \ s \ Z \rightarrow$ 
   $(\forall L. s :: \preceq (G, L) \rightarrow$ 
   $(\forall T \ C \ A. (normal \ s \rightarrow (\llbracket prg=G, cls=C, lcl=L \rrbracket \vdash t :: T) \wedge$ 
   $(\llbracket prg=G, cls=C, lcl=L \rrbracket \vdash_{dom} (locals \ (store \ s)) \gg t \gg A) \rightarrow$ 
   $Q \ Y' \ s' \ Z \wedge s' :: \preceq (G, L))) \implies$ 
   $G, A \models :: \{\{P\} \ c \multimap \{Q\}\}$ 
apply (simp (no-asm) add: ax-valids2-def triple-valid2-def2)
apply clarsimp
done

```

```

lemma da-good-approx-evalnE [consumes 4]:
assumes evaln:  $G \vdash s0 - t \multimap - n \rightarrow (v, s1)$ 
and wt:  $(\llbracket prg=G, cls=C, lcl=L \rrbracket \vdash t :: T$ 
and da:  $(\llbracket prg=G, cls=C, lcl=L \rrbracket \vdash_{dom} (locals \ (store \ s0)) \gg t \gg A$ 
and wf: wf-prog G

```

```

and   elim:  $\llbracket \text{normal } s1 \implies \text{nrm } A \subseteq \text{dom } (\text{locals } (\text{store } s1));$ 
         $\wedge l. \llbracket \text{abrupt } s1 = \text{Some } (\text{Jump } (\text{Break } l)); \text{normal } s0 \rrbracket$ 
         $\implies \text{brk } A \ l \subseteq \text{dom } (\text{locals } (\text{store } s1));$ 
         $\llbracket \text{abrupt } s1 = \text{Some } (\text{Jump } \text{Ret}); \text{normal } s0 \rrbracket$ 
         $\implies \text{Result} \in \text{dom } (\text{locals } (\text{store } s1))$ 
         $\rrbracket \implies P$ 

shows  $P$ 
proof -
  from evaln have  $G \vdash s0 \multimap \rightarrow (v, s1)$ 
  by (rule evaln-eval)
  from this wt da wf elim show  $P$ 
  by (rule da-good-approxE') iprover +
qed

lemma validI:
  assumes  $I: \bigwedge n \ s0 \ L \ \text{acc}C \ T \ C \ v \ s1 \ Y \ Z.$ 
     $\llbracket \forall t \in A. \ G \models n::t; \ s0::\preceq(G, L);$ 
     $\text{normal } s0 \implies \llbracket \text{prg} = G, \text{cls} = \text{acc}C, \text{lcl} = L \rrbracket \vdash t::T;$ 
     $\text{normal } s0 \implies \llbracket \text{prg} = G, \text{cls} = \text{acc}C, \text{lcl} = L \rrbracket \vdash \text{dom } (\text{locals } (\text{store } s0)) \gg t \gg C;$ 
     $G \vdash s0 \multimap \rightarrow \neg n \rightarrow (v, s1); \ P \ Y \ s0 \ Z \rrbracket \implies Q \ v \ s1 \ Z \wedge s1::\preceq(G, L)$ 

  shows  $G, A \models::\{ \{P\} \multimap \{Q\} \}$ 
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  $\llbracket \text{simproc add: wt-expr wt-var wt-exprs wt-stmt} \rrbracket$ 

lemma valid-stmtI:
  assumes  $I: \bigwedge n \ s0 \ L \ \text{acc}C \ C \ s1 \ Y \ Z.$ 
     $\llbracket \forall t \in A. \ G \models n::t; \ s0::\preceq(G, L);$ 
     $\text{normal } s0 \implies \llbracket \text{prg} = G, \text{cls} = \text{acc}C, \text{lcl} = L \rrbracket \vdash c::\checkmark;$ 
     $\text{normal } s0 \implies \llbracket \text{prg} = G, \text{cls} = \text{acc}C, \text{lcl} = L \rrbracket \vdash \text{dom } (\text{locals } (\text{store } s0)) \gg \langle c \rangle_s \gg C;$ 
     $G \vdash s0 \multimap \rightarrow c \multimap \rightarrow s1; \ P \ Y \ s0 \ Z \rrbracket \implies Q \ \Diamond \ s1 \ Z \wedge s1::\preceq(G, L)$ 

  shows  $G, A \models::\{ \{P\} \langle c \rangle_s \multimap \{Q\} \}$ 
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  $I: \bigwedge n \ s0 \ L \ \text{acc}C \ C \ s1 \ Y \ Z.$ 
     $\llbracket \forall t \in A. \ G \models n::t; \ s0::\preceq(G, L); \text{normal } s0; \llbracket \text{prg} = G, \text{cls} = \text{acc}C, \text{lcl} = L \rrbracket \vdash c::\checkmark;$ 
     $\llbracket \text{prg} = G, \text{cls} = \text{acc}C, \text{lcl} = L \rrbracket \vdash \text{dom } (\text{locals } (\text{store } s0)) \gg \langle c \rangle_s \gg C;$ 
     $G \vdash s0 \multimap \rightarrow c \multimap \rightarrow s1; \ (\text{Normal } P) \ Y \ s0 \ Z \rrbracket \implies Q \ \Diamond \ s1 \ Z \wedge s1::\preceq(G, L)$ 

  shows  $G, A \models::\{ \{ \text{Normal } P \} \langle c \rangle_s \multimap \{Q\} \}$ 
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 $I: \bigwedge n\ s0\ L\ accC\ T\ C\ vf\ s1\ Y\ Z.$
 $\llbracket \forall t \in A. G \models n::t; s0::\preceq(G, L); \text{normal } s0;$
 $\langle prg=G, cls=accC, lcl=L \rangle \vdash t::=T;$
 $\langle prg=G, cls=accC, lcl=L \rangle \vdash_{dom} (locals\ (store\ s0)) \gg \langle t \rangle_v \gg C;$
 $G \vdash s0 -t = \succ vf -n \rightarrow s1; (Normal\ P)\ Y\ s0\ Z \rrbracket$
 $\implies Q\ (In2\ vf)\ s1\ Z \wedge s1::\preceq(G, L)$
shows $G, A \models::\{ \{Normal\ P\} \langle t \rangle_v \succ \{Q\} \}$
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 *valid-expr-NormalI*:

assumes $I: \bigwedge n\ s0\ L\ accC\ T\ C\ v\ s1\ Y\ Z.$
 $\llbracket \forall t \in A. G \models n::t; s0::\preceq(G, L); \text{normal } s0;$
 $\langle prg=G, cls=accC, lcl=L \rangle \vdash t::= -T;$
 $\langle prg=G, cls=accC, lcl=L \rangle \vdash_{dom} (locals\ (store\ s0)) \gg \langle t \rangle_e \gg C;$
 $G \vdash s0 -t - \succ v -n \rightarrow s1; (Normal\ P)\ Y\ s0\ Z \rrbracket$
 $\implies Q\ (In1\ v)\ s1\ Z \wedge s1::\preceq(G, L)$
shows $G, A \models::\{ \{Normal\ P\} \langle t \rangle_e \succ \{Q\} \}$
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 *valid-expr-list-NormalI*:

assumes $I: \bigwedge n\ s0\ L\ accC\ T\ C\ vs\ s1\ Y\ Z.$
 $\llbracket \forall t \in A. G \models n::t; s0::\preceq(G, L); \text{normal } s0;$
 $\langle prg=G, cls=accC, lcl=L \rangle \vdash t::=T;$
 $\langle prg=G, cls=accC, lcl=L \rangle \vdash_{dom} (locals\ (store\ s0)) \gg \langle t \rangle_l \gg C;$
 $G \vdash s0 -t \doteq \succ vs -n \rightarrow s1; (Normal\ P)\ Y\ s0\ Z \rrbracket$
 $\implies Q\ (In3\ vs)\ s1\ Z \wedge s1::\preceq(G, L)$
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*: $\text{normal } s0 \implies \langle prg=G, cls=accC, lcl=L \rangle \vdash t::=T$
and *da*: $\text{normal } s0 \implies \langle prg=G, cls=accC, lcl=L \rangle \vdash_{dom} (locals\ (store\ s0)) \gg t \gg C$
and *elim*: $\llbracket Q\ v\ s1\ Z; s1::\preceq(G, L) \rrbracket \implies \text{concl}$
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*: $G \vdash s0 \rightarrow t \rightarrow n \rightarrow (v, s1)$ **and**
 $wt: (\text{prg}=G, \text{cls}=\text{acc } C, \text{lcl}=L) \vdash t::T$ **and**
 $da: (\text{prg}=G, \text{cls}=\text{acc } C, \text{lcl}=L) \vdash \text{dom}(\text{locals}(\text{store } s0)) \gg t \gg A$ **and**
 $\text{conf-}s0: s0::\leq(G, L)$ **and**
 $wf: wf\text{-prog } G$
shows $s1::\leq(G, L) \wedge (\text{normal } s1 \longrightarrow G, L, \text{store } s1 \vdash t \rightarrow v::\leq T) \wedge$
 $(\text{error-free } s0 = \text{error-free } s1)$

proof –

from *evaln* **have** $G \vdash s0 \rightarrow t \rightarrow (v, s1)$
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
 $\text{evaln}: G \vdash s0 \rightarrow t \rightarrow n \rightarrow (v, s1)$ **and**
 $\text{hyps}: \llbracket \text{dom}(\text{locals}(\text{store } s0)) \subseteq \text{dom}(\text{locals}(\text{store } s1)) \rrbracket$
 $\wedge \text{vv } s \text{ val. } \llbracket v = \text{In2 } vv; \text{normal } s1 \rrbracket$
 $\implies \text{dom}(\text{locals}(\text{store } s))$
 $\subseteq \text{dom}(\text{locals}(\text{store } ((\text{snd } vv) \text{ val } s))) \rrbracket \implies P$

shows P

proof –

from *evaln* **have** $G \vdash s0 \rightarrow t \rightarrow (v, s1)$ **by** (rule *evaln-eval*)
from *this hyps* **show** ?thesis
by (rule *dom-locals-eval-mono-elim*) *iprover+*

qed

lemma *evaln-no-abrupt*:

$\wedge s \ s'. \llbracket G \vdash s \rightarrow t \rightarrow n \rightarrow (w, s'); \text{normal } s' \rrbracket \implies \text{normal } s$
by (rule *evaln-cases, auto*)

declare *inj-term-simps* [simp]

lemma *ax-sound2*:

assumes $wf: wf\text{-prog } G$
and $\text{deriv}: G, A \vdash ts$
shows $G, A \models ts$
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 $\text{valid-}t = \langle G, A \models \{t\} \rangle$
moreover
note $\text{valid-}ts = \langle G, A \models ts \rangle$
have $\forall t' \in \text{insert } t \text{ ts. } G \models n::t'$
if $\text{valid-}A: \forall t \in A. G \models n::t$
for n
proof –

```

have  $G \models_{n::t}$ 
  using valid-A valid-t by (simp add: ax-valids2-def)
moreover
have  $\forall t \in ts. G \models_{n::t}$ 
  using valid-A valid-ts by (unfold ax-valids2-def) blast
ultimately show  $\forall t' \in \text{insert } t \text{ } ts. G \models_{n::t'}$ 
  by simp
qed
thus ?case
  by (unfold ax-valids2-def) blast
next
case (asm ts A)
from  $\langle ts \subseteq A \rangle$ 
show  $G, A \models_{::ts}$ 
  by (auto simp add: ax-valids2-def triple-valid2-def)
next
case (weaken A ts' ts)
note  $\langle G, A \models_{::ts'} \rangle$ 
moreover note  $\langle ts \subseteq ts' \rangle$ 
ultimately show  $G, A \models_{::ts}$ 
  by (unfold ax-valids2-def triple-valid2-def) blast
next
case (conseq P A t Q)
note con =  $\langle \forall Y s Z. P Y s Z \longrightarrow$ 
   $(\exists P' Q'.$ 
     $(G, A \vdash \{P'\} t \succ \{Q'\} \wedge G, A \models_{::\{ \{P'\} t \succ \{Q'\} \}}) \wedge$ 
     $(\forall Y' s'. (\forall Y Z'. P' Y s Z' \longrightarrow Q' Y' s' Z') \longrightarrow Q Y' s' Z)) \rangle$ 
show  $G, A \models_{::\{ \{P\} t \succ \{Q\} \}}$ 
proof (rule validI)
  fix n s0 L accC T C v s1 Y Z
  assume valid-A:  $\forall t \in A. G \models_{n::t}$ 
  assume conf:  $s0 :: \preceq (G, L)$ 
  assume wt:  $\text{normal } s0 \implies \langle \text{prg} = G, \text{cls} = \text{accC}, \text{lcl} = L \rangle \vdash_{t::T}$ 
  assume da:  $\text{normal } s0$ 
     $\implies \langle \text{prg} = G, \text{cls} = \text{accC}, \text{lcl} = L \rangle \vdash_{\text{dom } (\text{locals } (\text{store } s0))} \gg t \gg C$ 
  assume eval:  $G \vdash s0 - t \succ - n \rightarrow (v, s1)$ 
  assume P:  $P Y s0 Z$ 
  show  $Q v s1 Z \wedge s1 :: \preceq (G, L)$ 
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  $s1 :: \preceq (G, L)$ 
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 (λs.. 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-> {λVal: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: ( $\langle \text{prg} = G, \text{cls} = \text{acc} C', \text{lcl} = L \rangle \vdash \{ \text{acc} C, \text{statDecl} C, \text{stat} \} e..fn ::= T$ )
assume da: ( $\langle \text{prg} = G, \text{cls} = \text{acc} C', \text{lcl} = L \rangle \vdash \text{dom} (\text{locals} (\text{store } s0)) \gg \langle \{ \text{acc} C, \text{statDecl} C, \text{stat} \} e..fn \rangle_v \gg V$ )
assume eval:  $G \vdash s0 \dashv \{ \text{acc} C, \text{statDecl} C, \text{stat} \} e..fn \dashv \text{vf} - n \rightarrow s3$ 
assume P: (Normal P) Y s0 Z
show  $R \lfloor \text{vf} \rfloor_v s3 Z \wedge s3 :: \preceq (G, L)$ 
proof –
  from wt obtain statC f where
    wt-e: ( $\langle \text{prg} = G, \text{cls} = \text{acc} C, \text{lcl} = L \rangle \vdash e :: \text{Class } \text{stat} C$  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: ( $\langle \text{prg} = G, \text{cls} = \text{acc} C, \text{lcl} = L \rangle \vdash \text{dom} (\text{locals} (\text{store } s0)) \gg \langle e \rangle_e \gg V$ )
    by cases simp
  from eval obtain a s1 s2 s2' where
    eval-init:  $G \vdash s0 \dashv \text{Init } \text{statDecl} C - n \rightarrow s1$  and
    eval-e:  $G \vdash s1 \dashv e - \text{a} - n \rightarrow s2$  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: ( $\langle \text{prg} = G, \text{cls} = \text{acc} C, \text{lcl} = L \rangle \vdash (\text{Init } \text{statDecl} C) :: \checkmark$ )
  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: ( $\langle \text{prg} = G, \text{cls} = \text{acc} C, \text{lcl} = L \rangle \vdash \text{dom} (\text{locals} (\text{store } s0)) \gg \langle \text{Init } \text{statDecl} C \rangle_s \gg I$ )
    by (auto intro: da-Init [simplified] assigned.select-conv)
  from valid-init P valid-A conf-s0 eval-init wt-init da-init
  obtain Q:  $Q \diamond s1 Z$  and conf-s1:  $s1 :: \preceq (G, L)$ 
    by (rule validE)
  obtain
    R:  $R \lfloor \text{vf} \rfloor_v s2' Z$  and
    conf-s2:  $s2 :: \preceq (G, L)$  and
    conf-a: normal s2  $\longrightarrow G, \text{store } s2 \vdash a :: \preceq \text{Class } \text{stat} C$ 
  proof (cases normal s1)
    case True
    obtain V' where
      da-e': ( $\langle \text{prg} = G, \text{cls} = \text{acc} C, \text{lcl} = L \rangle \vdash \text{dom} (\text{locals} (\text{store } s1)) \gg \langle e \rangle_e \gg V'$ )
    proof –
      from eval-init
      have ( $\text{dom} (\text{locals} (\text{store } s0)) \subseteq \text{dom} (\text{locals} (\text{store } s1))$ )
        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  $R \lfloor \text{vf} \rfloor_v s2' Z$  and  $s2 :: \preceq (G, L)$ 

```

```

    by (rule validE) (simp add: fvar [symmetric])
  moreover
  from eval-e wt-e da-e' conf-s1 wf
  have normal s2  $\longrightarrow$  G,store s2 $\vdash$ a:: $\preceq$ 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:: $\preceq$ (G, L)
    by (cases rule: validE) (simp add: fvar [symmetric])+
  moreover from False eval-e have  $\neg$  normal s2
    by auto
  hence normal s2  $\longrightarrow$  G,store s2 $\vdash$ a:: $\preceq$ 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:: $\preceq$ (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 =  $\langle G, A \mid \vdash :: \{ \{ \text{Normal } P \} \ e1 - \succ \{ Q \} \} \rangle$ 
have valid-e2:  $\bigwedge a. G, A \mid \vdash :: \{ \{ Q \leftarrow \text{In1 } a \} \ e2 - \succ \{ \lambda \text{Val}:i.. \text{avar } G \ i \ a \ ..; R \} \}$ 
  using AVar.hyps by simp
show  $G, A \mid \vdash :: \{ \{ \text{Normal } P \} \ e1.[e2] = \succ \{ R \} \}$ 
proof (rule valid-var-NormalI)
  fix n s0 L accC T V vf s2' 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:  $(\langle \text{prg} = G, \text{cls} = \text{accC}, \text{lcl} = L \rangle) \vdash e1.[e2]::=T$ 
  assume da:  $(\langle \text{prg} = G, \text{cls} = \text{accC}, \text{lcl} = L \rangle) \vdash \text{dom}(\text{locals}(\text{store } s0)) \gg \langle e1.[e2] \rangle_v \gg V$ 
  assume eval:  $G \vdash s0 - e1.[e2] = \succ \text{vf} - n \rightarrow s2'$ 
  assume P: (Normal P) Y s0 Z
  show R [vf]v s2' Z  $\wedge$  s2':: $\preceq$ (G, L)
proof -
  from wt obtain
    wt-e1:  $(\langle \text{prg} = G, \text{cls} = \text{accC}, \text{lcl} = L \rangle) \vdash e1::-T.[]$  and
    wt-e2:  $(\langle \text{prg} = G, \text{cls} = \text{accC}, \text{lcl} = L \rangle) \vdash e2::-\text{PrimT Integer}$ 
  by (rule wt-elim-cases) simp
  from da obtain E1 where
    da-e1:  $(\langle \text{prg} = G, \text{cls} = \text{accC}, \text{lcl} = L \rangle) \vdash \text{dom}(\text{locals}(\text{store } s0)) \gg \langle e1 \rangle_e \gg E1$  and
    da-e2:  $(\langle \text{prg} = G, \text{cls} = \text{accC}, \text{lcl} = L \rangle) \vdash \text{nrm } E1 \gg \langle e2 \rangle_e \gg V$ 
  by (rule da-elim-cases) simp
  from eval obtain s1 a i s2 where
    eval-e1:  $G \vdash s0 - e1 - \succ a - n \rightarrow s1$  and
    eval-e2:  $G \vdash s1 - e2 - \succ i - n \rightarrow s2$  and
    avar:  $\text{avar } G \ i \ a \ s2 = (\text{vf}, s2')$ 
  using normal-s0 by (fastforce elim: evaln-elim-cases)
  from valid-e1 P valid-A conf-s0 eval-e1 wt-e1 da-e1

```

```

obtain  $Q: Q \downarrow a \downarrow_e s1 \ Z$  and  $conf\text{-}s1: s1::\preceq(G, L)$ 
  by (rule validE)
from  $Q$  have  $Q': \bigwedge v. (Q \leftarrow In1 \ a) \ v \ s1 \ Z$ 
  by simp
have  $R \downarrow vf \downarrow_v s2' \ Z$ 
proof (cases normal s1)
  case True
    obtain  $V'$  where
       $(\langle prg=G, cls=accC, lcl=L \rangle \vdash dom (locals (store \ s1)) \rangle \langle e2 \rangle_e \rangle V'$ 
    proof –
      from eval-e1 wt-e1 da-e1 wf True
      have  $nrm \ E1 \subseteq dom (locals (store \ s1))$ 
        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  $s2'::\preceq(G, L)$ 
  by (rule evaln-type-sound [elim-format]) simp
ultimately show ?thesis ..
qed
qed
next
case (NewC A P C Q)
note  $valid\text{-}init = \langle G, A \mid \models :: \{ \{ Normal \ P \} . Init \ C. \{ Alloc \ G \ (CInst \ C) \ Q \} \} \rangle$ 
show  $G, A \mid \models :: \{ \{ Normal \ P \} \ NewC \ C \multimap \{ Q \} \}$ 
proof (rule valid-expr-NormalI)
  fix  $n \ s0 \ L \ accC \ T \ E \ v \ s2 \ Y \ Z$ 
  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:  $(\langle prg=G, cls=accC, lcl=L \rangle \vdash NewC \ C :: - \ T)$ 
  assume da:  $(\langle prg=G, cls=accC, lcl=L \rangle \vdash dom (locals (store \ s0)) \rangle \langle NewC \ C \rangle_e \rangle E$ 
  assume eval:  $G \vdash s0 \multimap NewC \ C \multimap v \multimap n \rightarrow s2$ 
  assume P: (Normal P)  $Y \ s0 \ Z$ 
  show  $Q \downarrow v \downarrow_e s2 \ Z \wedge s2::\preceq(G, L)$ 
proof –
  from wt obtain is-cls-C: is-class G C
    by (rule wt-elim-cases) (auto dest: is-acc-classD)
  hence wt-init:  $(\langle prg=G, cls=accC, lcl=L \rangle \vdash Init \ C :: \checkmark)$ 
    by auto
  obtain  $I$  where
    da-init:  $(\langle prg=G, cls=accC, lcl=L \rangle \vdash dom (locals (store \ s0)) \rangle \langle Init \ C \rangle_s \rangle I$ 
    by (auto intro: da-Init [simplified] assigned.select-conv)
  from eval obtain  $s1 \ a$  where
    eval-init:  $G \vdash s0 \multimap Init \ C \multimap n \rightarrow s1$  and
    alloc:  $G \vdash s1 \multimap halloc \ CInst \ C \multimap a \rightarrow s2$  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)  $\Diamond$  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:: $\preceq$ (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 =  $\langle G, A \rangle \models \{ \{ \text{Normal } P \} . \text{init-comp-ty } T. \{ Q \} \}$ 
note valid-e =  $\langle G, A \rangle \models \{ \{ Q \} e \multimap \{ \lambda \text{Val} : i. . \text{abupd } (\text{check-neg } i) . ;$ 
 $\text{Alloc } G (\text{Arr } T (\text{the-Intg } i)) R \}$ 
show  $G, A \models \{ \{ \text{Normal } P \} \text{New } T[e] \multimap \{ R \} \}$ 
proof (rule valid-expr-NormalI)
  fix n s0 L accC arrT E v 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:  $(\text{prg} = G, \text{cls} = \text{accC}, \text{lcl} = L) \vdash \text{New } T[e] :: - \text{arrT}$ 
  assume da:  $(\text{prg} = G, \text{cls} = \text{accC}, \text{lcl} = L) \vdash \text{dom } (\text{locals } (\text{store } s0)) \gg \langle \text{New } T[e] \rangle_e \gg E$ 
  assume eval:  $G \vdash s0 \multimap \text{New } T[e] \multimap v \multimap n \rightarrow s3$ 
  assume P:  $(\text{Normal } P) Y s0 Z$ 
  show R [v]e s3 Z  $\wedge$  s3:: $\preceq$ (G, L)
  proof -
    from wt obtain
      wt-init:  $(\text{prg} = G, \text{cls} = \text{accC}, \text{lcl} = L) \vdash \text{init-comp-ty } T :: \checkmark$  and
      wt-e:  $(\text{prg} = G, \text{cls} = \text{accC}, \text{lcl} = L) \vdash e :: - \text{PrimT Integer}$ 
    by (rule wt-elim-cases) (auto intro: wt-init-comp-ty )
    from da obtain
      da-e:  $(\text{prg} = G, \text{cls} = \text{accC}, \text{lcl} = L) \vdash \text{dom } (\text{locals } (\text{store } s0)) \gg \langle e \rangle_e \gg E$ 
    by cases simp
    from eval obtain s1 i s2 a where
      eval-init:  $G \vdash s0 \multimap \text{init-comp-ty } T \multimap n \rightarrow s1$  and
      eval-e:  $G \vdash s1 \multimap e \multimap i \multimap n \rightarrow s2$  and
      alloc:  $G \vdash \text{abupd } (\text{check-neg } i) s2 \multimap \text{halloc Arr } T (\text{the-Intg } i) \multimap a \rightarrow s3$  and
      v:  $v = \text{Addr } a$ 
    using normal-s0 by (fastforce elim: evaln-elim-cases)
    obtain I where
      da-init:
 $(\text{prg} = G, \text{cls} = \text{accC}, \text{lcl} = L) \vdash \text{dom } (\text{locals } (\text{store } s0)) \gg \langle \text{init-comp-ty } T \rangle_s \gg I$ 
    proof (cases  $\exists C. T = \text{Class } C$ )
      case True
      show ?thesis
      by (rule that)
      (use True in
         $\langle \text{auto intro: da-Init [simplified] assigned.select-convs}$ 
 $\text{simp add: init-comp-ty-def} \rangle$ )
    next
    case False
    show ?thesis
    by (rule that)
    (use False in

```

⟨*auto intro: da-Skip [simplified] assigned.select-conv*
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 (λ*Val: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.. λs.. 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.. λs.. 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-⋈a-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 (λ Val: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
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:  $G, A \models \{ \{ \text{Normal } P \} e \multimap \{ \lambda \text{Val}.v. Q \leftarrow \text{In1 } (\text{eval-unop unop } v) \} \}$ 
show  $G, A \models \{ \{ \text{Normal } P \} \text{UnOp unop } e \multimap \{ Q \} \}$ 
proof (rule valid-expr-NormalI)
  fix  $n \ s0 \ L \ \text{acc}C \ T \ E \ v \ s1 \ Y \ Z$ 
  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:  $(\text{prg}=G, \text{cls}=\text{acc}C, \text{lcl}=L) \vdash \text{UnOp unop } e::-T$ 
  assume da:  $(\text{prg}=G, \text{cls}=\text{acc}C, \text{lcl}=L) \vdash \text{dom } (\text{locals } (\text{store } s0)) \gg \langle \text{UnOp unop } e \rangle_e \gg E$ 
  assume eval:  $G \vdash s0 \multimap \text{UnOp unop } e \multimap v \multimap n \rightarrow s1$ 
  assume  $P$ : (Normal P)  $Y \ s0 \ Z$ 
  show  $Q \ [v]_e \ s1 \ Z \wedge s1::\preceq(G, L)$ 
proof –
  from wt obtain  $eT$  where
     $wt\text{-}e$ :  $(\text{prg}=G, \text{cls}=\text{acc}C, \text{lcl}=L) \vdash e::-eT$ 
  by cases simp
  from da obtain
     $da\text{-}e$ :  $(\text{prg}=G, \text{cls}=\text{acc}C, \text{lcl}=L) \vdash \text{dom } (\text{locals } (\text{store } s0)) \gg \langle e \rangle_e \gg E$ 
  by cases simp
  from eval obtain  $ve$  where
     $eval\text{-}e$ :  $G \vdash s0 \multimap e \multimap ve \multimap n \rightarrow s1$  and
     $v$ :  $v = \text{eval-unop unop } ve$ 
  using normal-s0 by (fastforce elim: evaln-elim-cases)
  from valid-e P valid-A conf-s0 eval-e wt-e da-e
  have  $(\lambda \text{Val}.v. Q \leftarrow \text{In1 } (\text{eval-unop unop } v)) \ [ve]_e \ s1 \ Z$ 
  by (rule validE)
  with  $v$  have  $Q \ [v]_e \ s1 \ Z$ 
  by simp
  moreover
  from eval wt da conf-s0 wf
  have  $s1::\preceq(G, L)$ 
  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:  $G, A \models \{ \{ \text{Normal } P \} e1 \multimap \{ Q \} \}$ 
have valid-e2:  $\bigwedge v1. G, A \models \{ \{ Q \leftarrow \text{In1 } v1 \}$ 
  (if need-second-arg binop v1 then In1l e2 else In1r Skip)  $\multimap$ 
   $\{ \lambda \text{Val}.v2. R \leftarrow \text{In1 } (\text{eval-binop binop } v1 \ v2) \} \}$ 
using BinOp.hyps by simp
show  $G, A \models \{ \{ \text{Normal } P \} \text{BinOp binop } e1 \ e2 \multimap \{ R \} \}$ 
proof (rule valid-expr-NormalI)
  fix  $n \ s0 \ L \ \text{acc}C \ T \ E \ v \ s2 \ Y \ Z$ 
  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:  $(\text{prg}=G, \text{cls}=\text{acc}C, \text{lcl}=L) \vdash \text{BinOp binop } e1 \ e2::-T$ 
  assume da:  $(\text{prg}=G, \text{cls}=\text{acc}C, \text{lcl}=L) \vdash \text{dom } (\text{locals } (\text{store } s0)) \gg \langle \text{BinOp binop } e1 \ e2 \rangle_e \gg E$ 

```

```

assume eval:  $G \vdash s0 \text{ --BinOp binop } e1 \ e2 \text{ --} \succ v \text{ --} n \rightarrow s2$ 
assume P: (Normal P)  $Y \ s0 \ Z$ 
show  $R \ [v]_e \ s2 \ Z \wedge s2::\preceq(G, L)$ 
proof –
  from wt obtain  $e1T \ e2T$  where
    wt-e1:  $(\text{prg}=G, \text{cls}=\text{acc}C, \text{lcl}=L) \vdash e1::\text{--}e1T$  and
    wt-e2:  $(\text{prg}=G, \text{cls}=\text{acc}C, \text{lcl}=L) \vdash e2::\text{--}e2T$  and
    wt-binop: wt-binop G binop e1T e2T
    by cases simp
  have wt-Skip:  $(\text{prg} = G, \text{cls} = \text{acc}C, \text{lcl} = L) \vdash \text{Skip}::\checkmark$ 
    by simp

  from da obtain E1 where
    da-e1:  $(\text{prg}=G, \text{cls}=\text{acc}C, \text{lcl}=L) \vdash \text{dom}(\text{locals}(\text{store } s0)) \gg \langle e1 \rangle_e \gg E1$ 
    by cases simp+
  from eval obtain  $v1 \ s1 \ v2$  where
    eval-e1:  $G \vdash s0 \text{ --} e1 \text{ --} \succ v1 \text{ --} n \rightarrow s1$  and
    eval-e2:  $G \vdash s1 \text{ --} (\text{if need-second-arg binop } v1 \text{ then } \langle e2 \rangle_e \text{ else } \langle \text{Skip} \rangle_s) \text{ --} \succ \text{ --} n \rightarrow ([v2]_e, s2)$  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:  $s1::\preceq(G, L)$ 
    by (rule validE)
  from Q have  $Q': \bigwedge v. (Q \leftarrow \text{In1 } v1) \ v \ s1 \ Z$ 
    by simp
  have  $(\lambda \text{Val}.v2:. R \leftarrow \text{In1 } (\text{eval-binop binop } v1 \ v2)) \ [v2]_e \ s2 \ Z$ 
proof (cases normal s1)
    case True
    from eval-e1 wt-e1 da-e1 conf-s0 wf
    have conf-v1:  $G, \text{store } s1 \vdash v1::\preceq e1T$ 
      by (rule evaln-type-sound [elim-format]) (use True in simp)
    from eval-e1
    have  $G \vdash s0 \text{ --} e1 \text{ --} \succ v1 \rightarrow s1$ 
      by (rule evaln-eval)
    from da wt-e1 wt-e2 wt-binop conf-s0 True this conf-v1 wf
obtain E2 where
      da-e2:  $(\text{prg}=G, \text{cls}=\text{acc}C, \text{lcl}=L) \vdash \text{dom}(\text{locals}(\text{store } s1)) \gg (\text{if need-second-arg binop } v1 \text{ then } \langle e2 \rangle_e \text{ else } \langle \text{Skip} \rangle_s) \gg E2$ 
      by (rule da-e2-BinOp [elim-format]) iprover
    from wt-e2 wt-Skip obtain T2
      where  $(\text{prg}=G, \text{cls}=\text{acc}C, \text{lcl}=L) \vdash (\text{if need-second-arg binop } v1 \text{ then } \langle e2 \rangle_e \text{ else } \langle \text{Skip} \rangle_s)::T2$ 
      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  $s2::\preceq(G, L)$ 

```



```

    by (rule evaln-type-sound [elim-format]) simp
  ultimately show ?thesis ..
qed
qed
next
case (Super A P)
show  $G, A \models :: \{ \{ \text{Normal } (\lambda s.. P \leftarrow \text{In1 } (\text{val-this } s)) \} \text{ Super} \multimap \{ P \} \}$ 
proof (rule valid-expr-NormalI)
  fix  $n L s0 s1 v Y Z$ 
  assume  $\text{conf-s0}: s0 :: \preceq (G, L)$ 
  assume  $\text{normal-s0}: \text{normal } s0$ 
  assume  $\text{eval}: G \vdash s0 \multimap \text{Super} \multimap v \multimap n \rightarrow s1$ 
  assume  $P: (\text{Normal } (\lambda s.. P \leftarrow \text{In1 } (\text{val-this } s))) Y s0 Z$ 
  show  $P \llbracket v \rrbracket_e s1 Z \wedge s1 :: \preceq (G, L)$ 
  proof -
    from  $\text{eval}$  have  $s1 = s0$  and  $v = \text{val-this } (\text{store } s0)$ 
    using  $\text{normal-s0}$  by (auto elim: evaln-elim-cases)
    with  $P$   $\text{conf-s0}$  show ?thesis by simp
  qed
qed
next
case (Acc A P var Q)
note  $\text{valid-var} = \langle G, A \models :: \{ \{ \text{Normal } P \} \text{ var} \multimap \{ \lambda \text{Var}:(v, f)::. Q \leftarrow \text{In1 } v \} \} \rangle$ 
show  $G, A \models :: \{ \{ \text{Normal } P \} \text{ Acc var} \multimap \{ Q \} \}$ 
proof (rule valid-expr-NormalI)
  fix  $n s0 L \text{accC } T E v s1 Y Z$ 
  assume  $\text{valid-A}: \forall t \in A. G \models n :: t$ 
  assume  $\text{conf-s0}: s0 :: \preceq (G, L)$ 
  assume  $\text{normal-s0}: \text{normal } s0$ 
  assume  $\text{wt}: (\text{prg} = G, \text{cls} = \text{accC}, \text{lcl} = L) \vdash \text{Acc var} :: - T$ 
  assume  $\text{da}: (\text{prg} = G, \text{cls} = \text{accC}, \text{lcl} = L) \vdash \text{dom } (\text{locals } (\text{store } s0)) \gg \langle \text{Acc var} \rangle_e \gg E$ 
  assume  $\text{eval}: G \vdash s0 \multimap \text{Acc var} \multimap v \multimap n \rightarrow s1$ 
  assume  $P: (\text{Normal } P) Y s0 Z$ 
  show  $Q \llbracket v \rrbracket_e s1 Z \wedge s1 :: \preceq (G, L)$ 
  proof -
    from  $\text{wt}$  obtain
       $\text{wt-var}: (\text{prg} = G, \text{cls} = \text{accC}, \text{lcl} = L) \vdash \text{var} :: = T$ 
    by cases simp
    from  $\text{da}$  obtain  $V$  where
       $\text{da-var}: (\text{prg} = G, \text{cls} = \text{accC}, \text{lcl} = L) \vdash \text{dom } (\text{locals } (\text{store } s0)) \gg \langle \text{var} \rangle_v \gg V$ 
    by (cases  $\exists n. \text{var} = L\text{Var } n$ ) (use  $\text{da.LVar}$  in  $\langle \text{auto elim!}: \text{da-elim-cases} \rangle$ )
    from  $\text{eval}$  obtain  $\text{upd}$  where
       $\text{eval-var}: G \vdash s0 \multimap \text{var} \multimap (v, \text{upd}) \multimap n \rightarrow s1$ 
    using  $\text{normal-s0}$  by (fastforce elim: evaln-elim-cases)
    from  $\text{valid-var } P \text{ valid-A conf-s0 eval-var wt-var da-var}$ 
    have  $(\lambda \text{Var}:(v, f)::. Q \leftarrow \text{In1 } v) \llbracket (v, \text{upd}) \rrbracket_v s1 Z$ 
    by (rule validE)
    then have  $Q \llbracket v \rrbracket_e s1 Z$ 
    by simp
    moreover
    from  $\text{eval wt da conf-s0 wf}$ 
    have  $s1 :: \preceq (G, L)$ 
    by (rule evaln-type-sound [elim-format]) simp
    ultimately show ?thesis ..
  qed
qed
qed
next
case (Ass A P var Q e R)
note  $\text{valid-var} = \langle G, A \models :: \{ \{ \text{Normal } P \} \text{ var} \multimap \{ Q \} \} \rangle$ 

```

```

have valid-e:  $\bigwedge$  vf.
   $G, A \models :: \{ \{ Q \leftarrow \text{In2 } vf \} \ e \multimap \{ \lambda \text{Val}:v.. \text{assign } (\text{snd } vf) \ v \ ; \ R \} \}$ 
  using Ass.hyps by simp
show  $G, A \models :: \{ \{ \text{Normal } P \} \ \text{var}::e \multimap \{ R \} \}$ 
proof (rule valid-expr-NormalI)
  fix n s0 L accC T E v 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:  $(\text{prg}=G, \text{cls}=\text{accC}, \text{lcl}=L) \vdash \text{var}::e::-T$ 
  assume da:  $(\text{prg}=G, \text{cls}=\text{accC}, \text{lcl}=L) \vdash \text{dom } (\text{locals } (\text{store } s0)) \gg \langle \text{var}::e \rangle_e \gg E$ 
  assume eval:  $G \vdash s0 \multimap \text{var}::e \multimap v \multimap n \rightarrow s3$ 
  assume P:  $(\text{Normal } P) \ Y \ s0 \ Z$ 
  show  $R \ [v]_e \ s3 \ Z \wedge s3::\preceq(G, L)$ 
proof -
  from wt obtain varT where
    wt-var:  $(\text{prg}=G, \text{cls}=\text{accC}, \text{lcl}=L) \vdash \text{var}::\text{varT}$  and
    wt-e:  $(\text{prg}=G, \text{cls}=\text{accC}, \text{lcl}=L) \vdash e::-T$ 
  by cases simp
from eval obtain w upd s1 s2 where
    eval-var:  $G \vdash s0 \multimap \text{var}::e \multimap (w, \text{upd}) \multimap n \rightarrow s1$  and
    eval-e:  $G \vdash s1 \multimap e \multimap v \multimap n \rightarrow s2$  and
    s3:  $s3 = \text{assign } \text{upd } v \ s2$ 
  using normal-s0 by (auto elim: evaln-elim-cases)
have  $R \ [v]_e \ s3 \ Z$ 
proof (cases  $\exists \text{vn}. \text{var} = \text{LVar vn}$ )
  case False
  with da obtain V where
    da-var:  $(\text{prg}=G, \text{cls}=\text{accC}, \text{lcl}=L) \vdash \text{dom } (\text{locals } (\text{store } s0)) \gg \langle \text{var} \rangle_v \gg V$  and
    da-e:  $(\text{prg}=G, \text{cls}=\text{accC}, \text{lcl}=L) \vdash \text{nrm } V \gg \langle e \rangle_e \gg E$ 
  by cases simp+
from valid-var P valid-A conf-s0 eval-var wt-var da-var
obtain Q:  $Q \ [ (w, \text{upd}) ]_v \ s1 \ Z$  and conf-s1:  $s1::\preceq(G, L)$ 
  by (rule validE)
hence Q':  $\bigwedge v. (Q \leftarrow \text{In2 } (w, \text{upd})) \ v \ s1 \ Z$ 
  by simp
have  $(\lambda \text{Val}:v.. \text{assign } (\text{snd } (w, \text{upd})) \ v \ ; \ R) \ [v]_e \ s2 \ Z$ 
proof (cases normal s1)
  case True
  obtain E' where
    da-e':  $(\text{prg}=G, \text{cls}=\text{accC}, \text{lcl}=L) \vdash \text{dom } (\text{locals } (\text{store } s1)) \gg \langle e \rangle_e \gg E'$ 
  proof -
    from eval-var wt-var da-var wf True
    have  $\text{nrm } V \subseteq \text{dom } (\text{locals } (\text{store } s1))$ 
    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  $R \ [v]_e \ s3 \ Z$ 

```

```

    by simp
next
case True
then obtain vn where
  vn: var = LVar vn
  by auto
with da obtain E where
  da-e:  $\langle \text{prg}=G, \text{cls}=\text{acc}C, \text{lcl}=L \rangle \vdash \text{dom} (\text{locals} (\text{store } s0)) \gg \langle e \rangle_e \gg E$ 
  by cases simp+
from da.LVar vn obtain V where
  da-var:  $\langle \text{prg}=G, \text{cls}=\text{acc}C, \text{lcl}=L \rangle \vdash \text{dom} (\text{locals} (\text{store } s0)) \gg \langle \text{var} \rangle_v \gg V$ 
  by auto
from valid-var P valid-A conf-s0 eval-var wt-var da-var
obtain Q:  $Q \llbracket (w, \text{upd}) \rrbracket_v s1 Z$  and conf-s1:  $s1 :: \preceq (G, L)$ 
  by (rule validE)
hence Q':  $\bigwedge v. (Q \leftarrow \text{In2 } (w, \text{upd})) v s1 Z$ 
  by simp
have  $(\lambda \text{Val}:v. \text{assign } (\text{snd } (w, \text{upd})) v \text{ ; } R) \llbracket v \rrbracket_e s2 Z$ 
proof (cases normal s1)
case True
obtain E' where
  da-e':  $\langle \text{prg}=G, \text{cls}=\text{acc}C, \text{lcl}=L \rangle \vdash \text{dom} (\text{locals} (\text{store } s1)) \gg \langle e \rangle_e \gg E'$ 
proof -
  from eval-var
  have  $\text{dom} (\text{locals} (\text{store } s0)) \subseteq \text{dom} (\text{locals} (\text{store } (s1)))$ 
  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  $R \llbracket v \rrbracket_e s3 Z$ 
  by simp
qed
moreover
from eval wt da conf-s0 wf
have  $s3 :: \preceq (G, L)$ 
  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 =  $\langle G, A \rangle \models \{ \{ \text{Normal } P \} e0 \rightarrow \{ P' \} \}$ 
have valid-then-else:  $\bigwedge b. \langle G, A \rangle \models \{ \{ P' \leftarrow b \} (\text{if } b \text{ then } e1 \text{ else } e2) \rightarrow \{ Q \} \}$ 
  using Cond.hyps by simp
show  $\langle G, A \rangle \models \{ \{ \text{Normal } P \} e0 \text{ ? } e1 : e2 \rightarrow \{ Q \} \}$ 
proof (rule valid-expr-NormalI)
  fix n s0 L accC T E v s2 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:  $(\text{prg} = G, \text{cls} = \text{accC}, \text{lcl} = L) \vdash e0 \text{ ? } e1 : e2 :: - T$ 
assume da:  $(\text{prg} = G, \text{cls} = \text{accC}, \text{lcl} = L) \vdash \text{dom} (\text{locals} (\text{store } s0)) \gg \langle e0 \text{ ? } e1 : e2 \rangle_e \gg E$ 
assume eval:  $G \vdash s0 - e0 \text{ ? } e1 : e2 \multimap v - n \rightarrow s2$ 
assume P:  $(\text{Normal } P) \text{ Y } s0 \text{ Z}$ 
show  $Q \llbracket v \rrbracket_e s2 \text{ Z} \wedge s2 :: \preceq (G, L)$ 
proof –
  from wt obtain T1 T2 where
    wt-e0:  $(\text{prg} = G, \text{cls} = \text{accC}, \text{lcl} = L) \vdash e0 :: - \text{PrimT Boolean}$  and
    wt-e1:  $(\text{prg} = G, \text{cls} = \text{accC}, \text{lcl} = L) \vdash e1 :: - T1$  and
    wt-e2:  $(\text{prg} = G, \text{cls} = \text{accC}, \text{lcl} = L) \vdash e2 :: - T2$ 
  by cases simp
  from da obtain E0 E1 E2 where
    da-e0:  $(\text{prg} = G, \text{cls} = \text{accC}, \text{lcl} = L) \vdash \text{dom} (\text{locals} (\text{store } s0)) \gg \langle e0 \rangle_e \gg E0$  and
    da-e1:  $(\text{prg} = G, \text{cls} = \text{accC}, \text{lcl} = L) \vdash (\text{dom} (\text{locals} (\text{store } s0)) \cup \text{assigns-if True } e0) \gg \langle e1 \rangle_e \gg E1$  and
    da-e2:  $(\text{prg} = G, \text{cls} = \text{accC}, \text{lcl} = L) \vdash (\text{dom} (\text{locals} (\text{store } s0)) \cup \text{assigns-if False } e0) \gg \langle e2 \rangle_e \gg E2$ 
  by cases simp+
  from eval obtain b s1 where
    eval-e0:  $G \vdash s0 - e0 \multimap b - n \rightarrow s1$  and
    eval-then-else:  $G \vdash s1 - (\text{if the-Bool } b \text{ then } e1 \text{ else } e2) \multimap v - n \rightarrow s2$ 
  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:  $s1 :: \preceq (G, L)$ 
  by (rule validE)
  hence P':  $\bigwedge v. (P' \leftarrow (\text{the-Bool } b)) \text{ v } s1 \text{ Z}$ 
  by (cases normal s1) auto
  have  $Q \llbracket v \rrbracket_e s2 \text{ Z}$ 
  proof (cases normal s1)
    case True
    note normal-s1=this
    from wt-e1 wt-e2 obtain T' where
      wt-then-else:
         $(\text{prg} = G, \text{cls} = \text{accC}, \text{lcl} = L) \vdash (\text{if the-Bool } b \text{ then } e1 \text{ else } e2) :: - T'$ 
      by (cases the-Bool b) simp+
    have s0-s1:  $\text{dom} (\text{locals} (\text{store } s0)) \cup \text{assigns-if } (\text{the-Bool } b) \text{ } e0 \subseteq \text{dom} (\text{locals} (\text{store } s1))$ 
    proof –
      from eval-e0
      have eval-e0':  $G \vdash s0 - e0 \multimap b \rightarrow s1$ 
      by (rule evaln-eval)
      hence
         $\text{dom} (\text{locals} (\text{store } s0)) \subseteq \text{dom} (\text{locals} (\text{store } s1))$ 
      by (rule dom-locals-eval-mono-elim)
      moreover
      from eval-e0' True wt-e0
      have assigns-if (the-Bool b) e0  $\subseteq \text{dom} (\text{locals} (\text{store } s1))$ 
      by (rule assigns-if-good-approx')
      ultimately show ?thesis by (rule Un-least)
    qed
  obtain E' where
    da-then-else:
       $(\text{prg} = G, \text{cls} = \text{accC}, \text{lcl} = L) \vdash \text{dom} (\text{locals} (\text{store } s1)) \gg (\text{if the-Bool } b \text{ then } e1 \text{ else } e2)_e \gg E'$ 
  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:: $\preceq$ (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 =  $\langle G, A \mid \models :: \{ \{ \text{Normal } P \} \} e \multimap \{ Q \} \rangle$ 
have valid-args:  $\bigwedge a. G, A \mid \models :: \{ \{ Q \leftarrow \text{In1 } a \} \} \text{args} \multimap \{ R \ a \} \}$ 
  using Call.hyps by simp
have valid-methd:  $\bigwedge a \text{ vs invC declC } l.$ 
   $G, A \mid \models :: \{ \{ R \ a \leftarrow \text{In3 } \text{vs} \wedge.$ 
     $(\lambda s. \text{declC} =$ 
       $\text{invocation-declclass } G \text{ mode } (\text{store } s) \ a \ \text{statT}$ 
       $\langle \text{name} = \text{mn}, \text{parTs} = \text{pTs}' \rangle \wedge$ 
       $\text{invC} = \text{invocation-class mode } (\text{store } s) \ a \ \text{statT} \wedge$ 
       $l = \text{locals } (\text{store } s) \rangle ;.$ 
       $\text{init-lvars } G \ \text{declC} \langle \text{name} = \text{mn}, \text{parTs} = \text{pTs}' \rangle \text{ mode } a \ \text{vs} \wedge.$ 
       $(\lambda s. \text{normal } s \longrightarrow G \vdash \text{mode} \rightarrow \text{invC} \preceq \text{statT}) \}$ 
       $\text{Methd declC } \langle \text{name} = \text{mn}, \text{parTs} = \text{pTs}' \rangle \multimap \{ \text{set-lvars } l \ ; \ S \} \}$ 
  using Call.hyps by simp
show  $G, A \mid \models :: \{ \{ \text{Normal } P \} \} \{ \text{accC}', \text{statT}, \text{mode} \} e \cdot \text{mn} ( \{ \text{pTs}' \} \text{args} ) \multimap \{ S \} \}$ 
proof (rule valid-expr-NormalI)
  fix n s0 L accC T E v s5 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:  $\langle \text{prg} = G, \text{cls} = \text{accC}, \text{lcl} = L \rangle \vdash \{ \text{accC}', \text{statT}, \text{mode} \} e \cdot \text{mn} ( \{ \text{pTs}' \} \text{args} ) :: - T$ 
  assume da:  $\langle \text{prg} = G, \text{cls} = \text{accC}, \text{lcl} = L \rangle \vdash \text{dom } (\text{locals } (\text{store } s0))$ 
     $\gg \{ \{ \text{accC}', \text{statT}, \text{mode} \} e \cdot \text{mn} ( \{ \text{pTs}' \} \text{args} ) \}_e \gg E$ 
  assume eval:  $G \vdash s0 \multimap \{ \text{accC}', \text{statT}, \text{mode} \} e \cdot \text{mn} ( \{ \text{pTs}' \} \text{args} ) \multimap v \multimap n \rightarrow s5$ 
  assume P:  $(\text{Normal } P) \ Y \ s0 \ Z$ 
  show  $S \ [v]_e \ s5 \ Z \wedge s5 :: \preceq (G, L)$ 
proof -
  from wt obtain pTs statDeclT statM where
    wt-e:  $\langle \text{prg} = G, \text{cls} = \text{accC}, \text{lcl} = L \rangle \vdash e :: - \text{RefT statT}$  and
    wt-args:  $\langle \text{prg} = G, \text{cls} = \text{accC}, \text{lcl} = L \rangle \vdash \text{args} :: \dot{=} pTs$  and
    statM:  $\text{max-spec } G \ \text{accC} \ \text{statT} \langle \text{name} = \text{mn}, \text{parTs} = \text{pTs} \rangle$ 
       $= \{ ((\text{statDeclT}, \text{statM}), \text{pTs}') \}$  and
    mode:  $\text{mode} = \text{invmode statM } e$  and
    T:  $T = (\text{resTy statM})$  and
    eq-accC-accC':  $\text{accC} = \text{accC}'$ 

```

by cases fastforce+
from *da* **obtain** *C* **where**
da-e: $(\llbracket \text{prg} = G, \text{cls} = \text{acc}C, \text{lcl} = L \rrbracket) \vdash (\text{dom} (\text{locals} (\text{store } s0))) \gg \langle e \rangle_e \gg C$ **and**
da-args: $(\llbracket \text{prg} = G, \text{cls} = \text{acc}C, \text{lcl} = L \rrbracket) \vdash \text{nrm } C \gg \langle \text{args} \rangle_l \gg E$
 by cases simp
from *eval eq-accC-accC'* **obtain** *a s1 vs s2 s3 s3' s4 invDeclC* **where**
evaln-e: $G \vdash s0 \dashv e \dashv a \dashv n \rightarrow s1$ **and**
evaln-args: $G \vdash s1 \dashv \text{args} \dashv vs \dashv n \rightarrow s2$ **and**
invDeclC: *invDeclC* = *invocation-declclass*
 $G \text{ mode } (\text{store } s2) \text{ a statT } (\llbracket \text{name} = \text{mn}, \text{parTs} = \text{pTs} \rrbracket)$ **and**
s3: *s3* = *init-lvars* *G invDeclC* $(\llbracket \text{name} = \text{mn}, \text{parTs} = \text{pTs} \rrbracket)$ *mode a vs s2* **and**
check: *s3'* = *check-method-access* *G*
 $\text{acc}C' \text{ statT mode } (\llbracket \text{name} = \text{mn}, \text{parTs} = \text{pTs} \rrbracket) \text{ a } s3$ **and**
evaln-methd:
 $G \vdash s3' \dashv \text{Methd } \text{invDeclC } (\llbracket \text{name} = \text{mn}, \text{parTs} = \text{pTs} \rrbracket) \dashv v \dashv n \rightarrow s4$ **and**
s5: *s5* = *(set-lvars (locals (store s2))) s4*
using *normal-s0* **by** (*auto elim: evaln-elim-cases*)

from *evaln-e*
have *eval-e*: $G \vdash s0 \dashv e \dashv a \rightarrow s1$
by (*rule evaln-eval*)

from *eval-e - wt-e wf*
have *s1-no-return*: *abrupt s1* $\neq \text{Some } (\text{Jump Ret})$
by (*rule eval-expression-no-jump*
 $[\text{where } ?\text{Env} = (\llbracket \text{prg} = G, \text{cls} = \text{acc}C, \text{lcl} = L \rrbracket), \text{simplified}]$)
(use normal-s0 in auto)

from *valid-e P valid-A conf-s0 evaln-e wt-e da-e*
obtain *Q* $\lfloor a \rfloor_e s1 Z$ **and** *conf-s1*: $s1 :: \preceq (G, L)$
by (*rule validE*)
hence *Q*: $\bigwedge v. (Q \leftarrow \text{In1 } a) v s1 Z$
by *simp*
obtain
R: $(R \text{ a}) \lfloor vs \rfloor_l s2 Z$ **and**
conf-s2: $s2 :: \preceq (G, L)$ **and**
s2-no-return: *abrupt s2* $\neq \text{Some } (\text{Jump Ret})$
proof (*cases normal s1*)
case *True*
obtain *E'* **where**
da-args':
 $(\llbracket \text{prg} = G, \text{cls} = \text{acc}C, \text{lcl} = L \rrbracket) \vdash \text{dom} (\text{locals} (\text{store } s1)) \gg \langle \text{args} \rangle_l \gg E'$
proof –
from *evaln-e wt-e da-e wf True*
have *nrm C* $\subseteq \text{dom} (\text{locals} (\text{store } s1))$
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 $(R \text{ a}) \lfloor vs \rfloor_l s2 Z s2 :: \preceq (G, L)$
by (*rule validE*)
moreover
from *evaln-args*
have *e*: $G \vdash s1 \dashv \text{args} \dashv vs \rightarrow s2$
by (*rule evaln-eval*)
from *this s1-no-return wt-args wf*
have *abrupt s2* $\neq \text{Some } (\text{Jump Ret})$
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)) ;.
      (λs. normal s ⟶ G⊢mode→invC≼statT)
    ) v s3' Z
  )
have abrupt-s3-lemma: S [v]e s5 Z
  if abrupt-s3: ¬ normal s3
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

have S [v]e s5 Z
proof (cases normal s2)
  case False
  with s3 have abrupt-s3:  $\neg$  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':
      ( $\text{prg} = G, \text{cls} = \text{accC}, \text{lcl} = L$ )  $\vdash$  dom (locals (store s1))  $\gg \langle \text{args} \rangle_l \gg E'$ 
  proof -
    from evaln-e wt-e da-e wf normal-s1
    have nrm C  $\subseteq$  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:  $G \vdash s1 - \text{args} \dot{=} vs \rightarrow s2$ 
    by (rule evaln-eval)
  from evaln-e wt-e da-e conf-s0 wf
  have conf-a:  $G, \text{store } s1 \vdash a :: \preceq \text{RefT statT}$ 
    by (rule evaln-type-sound [elim-format]) (use normal-s1 in simp)
  with normal-s1 normal-s2 eval-args
  have conf-a-s2:  $G, \text{store } s2 \vdash a :: \preceq \text{RefT statT}$ 
    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]) (use normal-s2 in simp)
  from statM
  obtain
    statM': ( $\text{statDeclT}, \text{statM}$ )  $\in$  mheads G accC statT ( $\text{name} = \text{mn}, \text{parTs} = pTs'$ )
    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  $notNull: mode = IntVir \longrightarrow a \neq Null$ 
  by (cases  $s2$ ) (auto simp add: init-lvars-def2)
from  $conf\text{-}s2$   $conf\text{-}a\text{-}s2$   $wf\ notNull\ invC$ 
have  $dynT\text{-}prop: G \vdash mode \rightarrow invC \preceq statT$ 
  by (cases  $s2$ ) (auto intro: DynT-propI)

with  $wt\text{-}e\ statM' invC mode wf$ 
obtain  $dynM$  where
   $dynM: dynlookup\ G\ statT\ invC\ (\langle name=mn, parTs=pTs \rangle) = Some\ dynM$  and
   $acc\text{-}dynM: G \vdash Methd\ (\langle name=mn, parTs=pTs \rangle)\ dynM$ 
     $in\ invC\ dyn\text{-}accessible\text{-}from\ accC$ 
  by (force dest!: call-access-ok)
with  $invC'$   $check\ eq\text{-}accC\text{-}accC'$ 
have  $eq\text{-}s3'\text{-}s3: s3' = s3$ 
  by (auto simp add: check-method-access-def Let-def)

with  $dynT\text{-}prop\ R\ s3\ invDeclC\ invC\ l$ 
have  $R': PROP\ ?R$ 
  by auto

from  $dynT\text{-}prop\ wf\ wt\text{-}e\ statM'\ mode\ invC\ invDeclC\ dynM$ 
obtain
   $dynM: dynlookup\ G\ statT\ invC\ (\langle name=mn, parTs=pTs \rangle) = Some\ dynM$  and
   $wf\text{-}dynM: wf\text{-}mdecl\ G\ invDeclC\ (\langle name=mn, parTs=pTs \rangle, mthd\ dynM)$  and
   $dynM': methd\ G\ invDeclC\ (\langle name=mn, parTs=pTs \rangle) = Some\ dynM$  and
   $iscls\text{-}invDeclC: is\text{-}class\ G\ invDeclC$  and
   $invDeclC': invDeclC = declclass\ dynM$  and
   $invC\text{-}widen: G \vdash invC \preceq_C invDeclC$  and
   $resTy\text{-}widen: G \vdash resTy\ dynM \preceq resTy\ statM$  and
   $is\text{-}static\text{-}eq: is\text{-}static\ dynM = is\text{-}static\ statM$  and
   $involved\text{-}classes\text{-}prop:$ 
    (if  $invmode\ statM\ e = IntVir$ 
      then  $\forall statC. statT = ClassT\ statC \longrightarrow G \vdash invC \preceq_C statC$ 
      else  $((\exists statC. statT = ClassT\ statC \wedge G \vdash statC \preceq_C invDeclC) \vee$ 
         $(\forall statC. statT \neq ClassT\ statC \wedge invDeclC = Object)) \wedge$ 
         $statDeclT = ClassT\ invDeclC$ )
  by (cases rule: DynT-mheadsE) simp
obtain  $L'$  where
   $L': L' = (\lambda k.$ 
    (case  $k$  of
       $EName\ e$ 
       $\Rightarrow$  (case  $e$  of
         $VName\ v$ 
         $\Rightarrow ((table\text{-}of\ (lcls\ (mbody\ (mthd\ dynM))))$ 
           $(pars\ (mthd\ dynM) [\mapsto] pTs))\ v$ 
        |  $Res \Rightarrow Some\ (resTy\ dynM)$ 
        |  $This \Rightarrow$  if  $is\text{-}static\ statM$ 
          then  $None$  else  $Some\ (Class\ invDeclC)$ )))
    by simp
from  $wf\text{-}dynM\ [THEN\ wf\text{-}mdeclD1,\ THEN\ conjunct1]\ normal\text{-}s2\ conf\text{-}s2\ wt\text{-}e$ 
   $wf\ eval\text{-}args\ conf\text{-}a\ mode\ notNull\ wf\text{-}dynM\ involved\text{-}classes\text{-}prop$ 
have  $conf\text{-}s3: s3 :: \preceq (G, L')$ 
apply –

apply (drule conforms-init-lvars [of  $G\ invDeclC$ 
   $(\langle name=mn, parTs=pTs \rangle)\ dynM\ store\ s2\ vs\ pTs\ abrupt\ s2$ 
   $L\ statT\ invC\ a\ (statDeclT, statM)\ e]$ )
apply (rule  $wf$ )
apply (rule  $conf\text{-}args$ )

```

```

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-gert)
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':: $\preceq$ (G,L') by simp
from is-static-eq wf-dynM L'
obtain mthdT where
  ( $\langle \text{prg} = G, \text{cls} = \text{invDeclC}, \text{lcl} = L' \rangle$ )
   $\vdash \text{Body invDeclC (stmt (mbody (mthd dynM)))} :: \neg \text{mthdT}$  and
  mthdT-widen:  $G \vdash \text{mthdT} \preceq \text{resTy dynM}$ 
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:
  ( $\langle \text{prg} = G, \text{cls} = \text{invDeclC}, \text{lcl} = L' \rangle$ )
   $\vdash (\text{Methd invDeclC } \langle \text{name} = mn, \text{parTs} = pTs' \rangle) :: \neg \text{mthdT}$ 
by (auto intro: wt.Methd)
obtain M where
  da-methd:
  ( $\langle \text{prg} = G, \text{cls} = \text{invDeclC}, \text{lcl} = L' \rangle$ )
   $\vdash \text{dom (locals (store s3'))}$ 
   $\gg \langle \text{Methd invDeclC } \langle \text{name} = mn, \text{parTs} = pTs' \rangle \rangle_e \gg M$ 
proof -
from wf-dynM
obtain M' where
  da-body:
  ( $\langle \text{prg} = G, \text{cls} = \text{invDeclC}$ 
  ,  $\text{lcl} = \text{callee-lcl invDeclC } \langle \text{name} = mn, \text{parTs} = pTs' \rangle \rangle$  (mthd dynM)
  )  $\vdash \text{parameters (mthd dynM)} \gg \langle \text{stmt (mbody (mthd dynM))} \rangle \gg M'$  and
  res: Result  $\in \text{nrm } M'$ 
by (rule wf-mdeclE) iprover
from da-body is-static-eq L' have
  ( $\langle \text{prg} = G, \text{cls} = \text{invDeclC}, \text{lcl} = L' \rangle$ )
   $\vdash \text{parameters (mthd dynM)} \gg \langle \text{stmt (mbody (mthd dynM))} \rangle \gg M'$ 
by (simp add: callee-lcl-def
      cong add: lname.case-cong ename.case-cong)
moreover have parameters (mthd dynM)  $\subseteq \text{dom (locals (store s3'))}$ 
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-conv)
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  $valid-A: \forall t \in A. G \models n :: t$ 
  assume  $conf-s0: s0 :: \preceq(G, L)$ 
  assume  $normal-s0: normal\ s0$ 
  assume  $wt: (\text{prg}=G, \text{cls}=accC, \text{lcl}=L) \vdash Body\ D\ c :: -T$ 
  assume  $da: (\text{prg}=G, \text{cls}=accC, \text{lcl}=L) \vdash dom\ (locals\ (store\ s0)) \gg \langle Body\ D\ c \rangle_e \gg E$ 
  assume  $eval: G \vdash s0 \rightarrow Body\ D\ c \rightarrow v \rightarrow n \rightarrow s4$ 
  assume  $P: (Normal\ P)\ Y\ s0\ Z$ 
  show  $R\ [v]_e\ s4\ Z \wedge s4 :: \preceq(G, L)$ 
proof -
  from  $wt$  obtain
     $iscls-D: is-class\ G\ D$  and
     $wt-init: (\text{prg}=G, \text{cls}=accC, \text{lcl}=L) \vdash Init\ D :: \checkmark$  and
     $wt-c: (\text{prg}=G, \text{cls}=accC, \text{lcl}=L) \vdash c :: \checkmark$ 
  by cases auto
obtain  $I$  where
   $da-init: (\text{prg}=G, \text{cls}=accC, \text{lcl}=L) \vdash dom\ (locals\ (store\ s0)) \gg \langle Init\ D \rangle_s \gg I$ 
  by (auto intro: da-Init [simplified] assigned.select-convs)
from  $da$  obtain  $C$  where
   $da-c: (\text{prg}=G, \text{cls}=accC, \text{lcl}=L) \vdash (dom\ (locals\ (store\ s0))) \gg \langle c \rangle_s \gg C$  and
   $jmpOk: jumpNestingOkS\ \{Ret\}\ c$ 
  by cases simp
from  $eval$  obtain  $s1\ s2\ s3$  where
   $eval-init: G \vdash s0 \rightarrow Init\ D \rightarrow n \rightarrow s1$  and
   $eval-c: G \vdash s1 \rightarrow c \rightarrow n \rightarrow s2$  and
   $v: v = the\ (locals\ (store\ s2)\ Result)$  and
   $s3: s3 = (if\ \exists l. abrupt\ s2 = Some\ (Jump\ (Break\ l)) \vee$ 
     $abrupt\ s2 = Some\ (Jump\ (Cont\ l))$ 
     $then\ abupd\ (\lambda x. Some\ (Error\ CrossMethodJump))\ s2\ else\ s2)$  and
   $s4: s4 = abupd\ (absorb\ Ret)\ s3$ 
  using  $normal-s0$  by (fastforce elim: evaln-elim-cases)
obtain  $C'$  where
   $da-c': (\text{prg}=G, \text{cls}=accC, \text{lcl}=L) \vdash (dom\ (locals\ (store\ s1))) \gg \langle c \rangle_s \gg C'$ 
proof -
  from  $eval-init$ 
  have  $(dom\ (locals\ (store\ s0))) \subseteq (dom\ (locals\ (store\ s1)))$ 
  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  $Q: Q \diamond s1\ Z$  and  $conf-s1: s1 :: \preceq(G, L)$ 
  by (rule validE)
from  $valid-c\ Q\ valid-A\ conf-s1\ eval-c\ wt-c\ da-c'$ 
have  $R: (\lambda s.. abupd\ (absorb\ Ret)\ s) \cdot R \leftarrow In1\ (the\ (locals\ s\ Result))$ 
   $\diamond s2\ Z$ 
  by (rule validE)
have  $s3=s2$ 
proof -
  have  $s1-no-jmp: \bigwedge j. abrupt\ s1 \neq Some\ (Jump\ j)$ 
  by (rule eval-statement-no-jump [OF - - wt-init])
    (use eval-init [THEN evaln-eval] wf normal-s0 in auto)
from  $eval-c$  [THEN evaln-eval] -  $wt-c$  wf
have  $\bigwedge j. abrupt\ s2 = Some\ (Jump\ j) \implies j=Ret$ 
  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  $s4::\preceq(G, L)$ 
by (rule evaln-type-sound [elim-format]) simp
ultimately show ?thesis ..
qed
qed
next
case (Nil A P)
show  $G, A \models::\{ \{ \text{Normal } (P \leftarrow [\ ]_i) \} \models \{P\} \}$ 
proof (rule valid-expr-list-NormalI)
  fix  $s0 \ s1 \ vs \ n \ L \ Y \ Z$ 
  assume conf-s0:  $s0::\preceq(G, L)$ 
  assume normal-s0: normal s0
  assume eval:  $G \vdash s0 \multimap [ ] \multimap vs \multimap n \rightarrow s1$ 
  assume  $P: (\text{Normal } (P \leftarrow [\ ]_i)) \ Y \ s0 \ Z$ 
  show  $P \ [vs]_i \ s1 \ Z \wedge s1::\preceq(G, L)$ 
  proof –
    from eval obtain  $vs = [ ] \ s1 = s0$ 
    using normal-s0 by (auto elim: evaln-elim-cases)
    with  $P \ \text{conf-s0}$  show ?thesis
    by simp
  qed
qed
next
case (Cons A P e Q es R)
note valid-e =  $\langle G, A \models::\{ \{ \text{Normal } P \} \ e \multimap \{Q\} \} \rangle$ 
have valid-es:  $\bigwedge v. G, A \models::\{ \{ Q \leftarrow [v]_e \} \ es \multimap \{ \lambda Vals:vs. R \leftarrow [(v \# vs)]_i \} \}$ 
using Cons.hyps by simp
show  $G, A \models::\{ \{ \text{Normal } P \} \ e \# es \multimap \{R\} \}$ 
proof (rule valid-expr-list-NormalI)
  fix  $n \ s0 \ L \ accC \ T \ E \ v \ s2 \ Y \ Z$ 
  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:  $(\text{prg} = G, \text{cls} = accC, \text{lcl} = L) \vdash e \# es::\vdash T$ 
  assume da:  $(\text{prg} = G, \text{cls} = accC, \text{lcl} = L) \vdash \text{dom } (\text{locals } (\text{store } s0)) \gg \langle e \# es \rangle_i \gg E$ 
  assume eval:  $G \vdash s0 \multimap e \# es \multimap v \multimap n \rightarrow s2$ 
  assume  $P: (\text{Normal } P) \ Y \ s0 \ Z$ 
  show  $R \ [v]_i \ s2 \ Z \wedge s2::\preceq(G, L)$ 
  proof –
    from wt obtain  $eT \ esT$  where
       $wt\text{-}e: (\text{prg} = G, \text{cls} = accC, \text{lcl} = L) \vdash e::\vdash eT$  and
       $wt\text{-}es: (\text{prg} = G, \text{cls} = accC, \text{lcl} = L) \vdash es::\vdash esT$ 
    by cases simp
    from da obtain  $E1$  where
       $da\text{-}e: (\text{prg} = G, \text{cls} = accC, \text{lcl} = L) \vdash (\text{dom } (\text{locals } (\text{store } s0))) \gg \langle e \rangle_e \gg E1$  and
       $da\text{-}es: (\text{prg} = G, \text{cls} = accC, \text{lcl} = L) \vdash \text{nrm } E1 \gg \langle es \rangle_i \gg E$ 
    by cases simp
    from eval obtain  $s1 \ ve \ vs$  where
       $eval\text{-}e: G \vdash s0 \multimap e \multimap ve \multimap n \rightarrow s1$  and
       $eval\text{-}es: G \vdash s1 \multimap es \multimap vs \multimap n \rightarrow s2$  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:  $s1::\preceq(G, L)$ 

```

```

    by (rule validE)
  from Q have Q':  $\bigwedge v. (Q \leftarrow [ve]_e) v s1 Z$ 
  by simp
  have  $(\lambda Vals:vs.. R \leftarrow [(ve \# vs)]_l) [vs]_l s2 Z$ 
  proof (cases normal s1)
    case True
    obtain E' where
      da-es':  $(\text{prg}=G, \text{cls}=\text{acc}C, \text{lcl}=L) \vdash \text{dom}(\text{locals}(\text{store } s1)) \gg \langle es \rangle_l \gg E'$ 
    proof -
      from eval-e wt-e da-e wf True
      have  $\text{nrm } E1 \subseteq \text{dom}(\text{locals}(\text{store } s1))$ 
      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  $s2::\preceq(G, L)$ 
  by (rule evaln-type-sound [elim-format]) simp
  ultimately show ?thesis ..
  qed
  qed
  next
  case (Skip A P)
  show  $G, A \models::\{ \{ \text{Normal } (P \leftarrow \Diamond) \} . \text{Skip. } \{ P \} \}$ 
  proof (rule valid-stmt-NormalI)
    fix s0 s1 n L Y Z
    assume conf-s0:  $s0::\preceq(G, L)$ 
    assume normal-s0: normal s0
    assume eval:  $G \vdash s0 \text{ --Skip--} n \rightarrow s1$ 
    assume P:  $(\text{Normal } (P \leftarrow \Diamond)) Y s0 Z$ 
    show  $P \Diamond s1 Z \wedge s1::\preceq(G, L)$ 
    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 =  $\langle G, A \models::\{ \{ \text{Normal } P \} e \text{ --} \succ \{ Q \leftarrow \Diamond \} \} \rangle$ 
  show  $G, A \models::\{ \{ \text{Normal } P \} . \text{Expr } e. \{ Q \} \}$ 
  proof (rule valid-stmt-NormalI)
    fix n s0 L accC C s1 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: ( $\text{prg} = G, \text{cls} = \text{acc}C, \text{lcl} = L$ )  $\vdash \text{Expr } e :: \sqrt{\phantom{x}}$ 
assume da: ( $\text{prg} = G, \text{cls} = \text{acc}C, \text{lcl} = L$ )  $\vdash \text{dom } (\text{locals } (\text{store } s0)) \gg \langle \text{Expr } e \rangle_s \gg C$ 
assume eval:  $G \vdash s0 \multimap \text{Expr } e \multimap n \rightarrow s1$ 
assume P: (Normal P)  $Y s0 Z$ 
show  $Q \diamond s1 Z \wedge s1 :: \preceq (G, L)$ 
proof –
  from wt obtain eT where
    wt-e: ( $\text{prg} = G, \text{cls} = \text{acc}C, \text{lcl} = L$ )  $\vdash e :: -eT$ 
    by cases simp
  from da obtain E where
    da-e: ( $\text{prg} = G, \text{cls} = \text{acc}C, \text{lcl} = L$ )  $\vdash \text{dom } (\text{locals } (\text{store } s0)) \gg \langle e \rangle_e \gg E$ 
    by cases simp
  from eval obtain v where
    eval-e:  $G \vdash s0 \multimap e \multimap v \multimap n \rightarrow s1$ 
    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: ( $Q \leftarrow \diamond$ )  $[v]_e s1 Z$  and  $s1 :: \preceq (G, L)$ 
    by (rule validE)
  thus ?thesis by simp
qed
qed
next
case (Lab A P c l Q)
note valid-c =  $\langle G, A \mid \vdash :: \{ \{ \text{Normal } P \} .c. \{ \text{abupd } (\text{absorb } l) .; Q \} \} \rangle$ 
show  $G, A \mid \vdash :: \{ \{ \text{Normal } P \} .l. c. \{ Q \} \}$ 
proof (rule valid-stmt-NormalI)
  fix n s0 L accC C s2 Y Z
assume valid-A:  $\forall t \in A. G \mid \vdash n :: t$ 
assume conf-s0:  $s0 :: \preceq (G, L)$ 
assume normal-s0: normal s0
assume wt: ( $\text{prg} = G, \text{cls} = \text{acc}C, \text{lcl} = L$ )  $\vdash l. c :: \sqrt{\phantom{x}}$ 
assume da: ( $\text{prg} = G, \text{cls} = \text{acc}C, \text{lcl} = L$ )  $\vdash \text{dom } (\text{locals } (\text{store } s0)) \gg \langle l. c \rangle_s \gg C$ 
assume eval:  $G \vdash s0 \multimap l. c \multimap n \rightarrow s2$ 
assume P: (Normal P)  $Y s0 Z$ 
show  $Q \diamond s2 Z \wedge s2 :: \preceq (G, L)$ 
proof –
  from wt obtain
    wt-c: ( $\text{prg} = G, \text{cls} = \text{acc}C, \text{lcl} = L$ )  $\vdash c :: \sqrt{\phantom{x}}$ 
    by cases simp
  from da obtain E where
    da-c: ( $\text{prg} = G, \text{cls} = \text{acc}C, \text{lcl} = L$ )  $\vdash \text{dom } (\text{locals } (\text{store } s0)) \gg \langle c \rangle_s \gg E$ 
    by cases simp
  from eval obtain s1 where
    eval-c:  $G \vdash s0 \multimap c \multimap n \rightarrow s1$  and
    s2:  $s2 = \text{abupd } (\text{absorb } l) s1$ 
    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: ( $\text{abupd } (\text{absorb } l) .; Q$ )  $\diamond s1 Z$ 
    by (rule validE)
  with s2 have  $Q \diamond s2 Z$ 
    by simp
  moreover
  from eval wt da conf-s0 wf
have  $s2 :: \preceq (G, L)$ 
    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 =  $\langle G, A \models :: \{ \{ Normal P \} \} e \multimap \{ P' \} \rangle$ 
have valid-then-else:  $\bigwedge b. G, A \models :: \{ \{ P' \leftarrow b \} \} . (if\ b\ then\ c1\ else\ c2) . \{ Q \} \}$ 
  using If.hyps by simp
show  $G, A \models :: \{ \{ Normal P \} \} . If(e)\ c1\ Else\ c2 . \{ Q \} \}$ 
proof (rule valid-stmt-NormalI)
  fix n s0 L accC C s2 Y Z
  assume valid-A:  $\forall t \in A. G \models n :: t$ 
  assume conf-s0:  $s0 :: \preceq (G, L)$ 
  assume normal-s0: normal s0
  assume wt:  $(\langle prg = G, cls = accC, lcl = L \rangle \vdash If(e)\ c1\ Else\ c2 :: \checkmark)$ 
  assume da:  $(\langle prg = G, cls = accC, lcl = L \rangle \vdash \text{dom}(\text{locals}(\text{store } s0)) \gg \langle If(e)\ c1\ Else\ c2 \rangle_s \gg C)$ 
  assume eval:  $G \vdash s0 \multimap If(e)\ c1\ Else\ c2 \multimap n \rightarrow s2$ 
  assume P: (Normal P) Y s0 Z
  show  $Q \diamond s2\ Z \wedge s2 :: \preceq (G, L)$ 
proof –
  from eval obtain b s1 where
    eval-e:  $G \vdash s0 \multimap e \multimap b \multimap n \rightarrow s1$  and
    eval-then-else:  $G \vdash s1 \multimap (if\ the\ Bool\ b\ then\ c1\ else\ c2) \multimap n \rightarrow s2$ 
  using normal-s0 by (auto elim: evaln-elim-cases)
  from wt obtain
    wt-e:  $(\langle prg = G, cls = accC, lcl = L \rangle \vdash e :: \neg PrimT\ Boolean)$  and
    wt-then-else:  $(\langle prg = G, cls = accC, lcl = L \rangle \vdash (if\ the\ Bool\ b\ then\ c1\ else\ c2) :: \checkmark)$ 
  by cases (simp split: if-split)
  from da obtain E S where
    da-e:  $(\langle prg = G, cls = accC, lcl = L \rangle \vdash \text{dom}(\text{locals}(\text{store } s0)) \gg \langle e \rangle_e \gg E)$  and
    da-then-else:
       $(\langle prg = G, cls = accC, lcl = L \rangle \vdash$ 
         $(\text{dom}(\text{locals}(\text{store } s0)) \cup \text{assigns-if}(the\ Bool\ b)\ e)$ 
         $\gg \langle if\ the\ Bool\ b\ then\ c1\ else\ c2 \rangle_s \gg S)$ 
  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:  $s1 :: \preceq (G, L)$ 
  by (rule validE)
  hence P':  $\bigwedge v. (P' \leftarrow the\ Bool\ b)\ v\ s1\ Z$ 
  by (cases normal s1) auto
  have  $Q \diamond s2\ Z$ 
proof (cases normal s1)
  case True
  have s0-s1:  $\text{dom}(\text{locals}(\text{store } s0)) \cup \text{assigns-if}(the\ Bool\ b)\ e \subseteq \text{dom}(\text{locals}(\text{store } s1))$ 
proof –
  from eval-e
  have eval-e':  $G \vdash s0 \multimap e \multimap b \rightarrow s1$ 
  by (rule evaln-eval)
  hence
     $\text{dom}(\text{locals}(\text{store } s0)) \subseteq \text{dom}(\text{locals}(\text{store } s1))$ 
  by (rule dom-locals-eval-mono-elim)
  moreover
  from eval-e' True wt-e
  have assigns-if (the-Bool b) e  $\subseteq \text{dom}(\text{locals}(\text{store } s1))$ 
  by (rule assigns-if-good-approx')
  ultimately show ?thesis by (rule Un-least)
qed
with da-then-else
obtain S' where
   $(\langle prg = G, cls = accC, lcl = L \rangle \vdash \text{dom}(\text{locals}(\text{store } s1)) \gg \langle if\ the\ Bool\ b\ then\ c1\ else\ c2 \rangle_s \gg S')$ 

```

```

    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 hypotheses valid-e and valid-c we can only reach the state after unfolding the
    loop once, i.e.  $P \Diamond s2 Z$ , where  $s2$  is the state after executing  $c$ . To gain validity of the further execution of
    while, to finally get  $(P'←=False↓=◇) \Diamond s3 Z$  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.
    have generalized:
      ∧ 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')
    if G⊢s -t⋗-n→ (v, s')
    for t s s' v
    using that
  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.premis eqs by simp

```

```

have da: ( $\langle \text{prg} = G, \text{cls} = \text{acc}C, \text{lcl} = L \rangle \vdash$ 
   $\text{dom}(\text{locals}(\text{store}((\text{Norm } s0')::\text{state}))) \gg \langle l \cdot \text{While}(e) \ c \rangle_s \gg E$ 
  using Loop.premis eqs by simp
have evaln-e:  $G \vdash \text{Norm } s0' - e \rightarrow b - n' \rightarrow s1'$ 
  using Loop.hyps eqs by simp
show ( $P' \leftarrow \text{False} \downarrow = \Diamond$ )  $\Diamond s3' \ Z$ 
proof –
  from wt obtain
    wt-e: ( $\langle \text{prg} = G, \text{cls} = \text{acc}C, \text{lcl} = L \rangle \vdash e :: -\text{Prim}T \ \text{Boolean}$  and
    wt-c: ( $\langle \text{prg} = G, \text{cls} = \text{acc}C, \text{lcl} = L \rangle \vdash c :: \checkmark$ 
    by cases (simp add: eqs)
  from da obtain E S where
    da-e: ( $\langle \text{prg} = G, \text{cls} = \text{acc}C, \text{lcl} = L \rangle$ 
       $\vdash \text{dom}(\text{locals}(\text{store}((\text{Norm } s0')::\text{state}))) \gg \langle e \rangle_e \gg E$  and
    da-c: ( $\langle \text{prg} = G, \text{cls} = \text{acc}C, \text{lcl} = L \rangle$ 
       $\vdash (\text{dom}(\text{locals}(\text{store}((\text{Norm } s0')::\text{state})))$ 
         $\cup \text{assigns-if } \text{True } e) \gg \langle c \rangle_s \gg S$ 
      by cases (simp add: eqs)
  from evaln-e
have eval-e:  $G \vdash \text{Norm } s0' - e \rightarrow b \rightarrow s1'$ 
  by (rule evaln-eval)
from valid-e P valid-A conf-s0' evaln-e wt-e da-e
obtain  $P'$ :  $P' \vdash [b]_e \ s1' \ Z$  and  $\text{conf-s1}': s1'::\preceq(G, L)$ 
  by (rule validE)
show ( $P' \leftarrow \text{False} \downarrow = \Diamond$ )  $\Diamond s3' \ Z$ 
proof (cases normal s1')
  case True
  note normal-s1' = this
  show ?thesis
  proof (cases the-Bool b)
  case True
  with  $P'$  normal-s1' have  $P''$ : ( $\text{Normal}(P' \leftarrow \text{True})) \ [b]_e \ s1' \ Z$ 
  by auto
  from True Loop.hyps obtain
    eval-c:  $G \vdash s1' - c - n' \rightarrow s2'$  and
    eval-while:
       $G \vdash \text{abupd}(\text{absorb}(\text{Cont } l)) \ s2' - l \cdot \text{While}(e) \ c - n' \rightarrow s3'$ 
    by (simp add: eqs)
  from True Loop.hyps have
    hyp:  $\text{PROP } ?\text{Hyp } n' \langle l \cdot \text{While}(e) \ c \rangle_s$ 
       $(\text{abupd}(\text{absorb}(\text{Cont } l')) \ s2') \Diamond s3'$ 
    apply (simp only: True if-True eqs)
    apply (elim conjE)
    apply (tactic smp-tac context 3 1)
    apply fast
    done
  from eval-e
have  $s0' - s1'$ :  $\text{dom}(\text{locals}(\text{store}((\text{Norm } s0')::\text{state})))$ 
   $\subseteq \text{dom}(\text{locals}(\text{store } s1'))$ 
  by (rule dom-locals-eval-mono-elim)
obtain  $S'$  where
  da-c':
    ( $\langle \text{prg} = G, \text{cls} = \text{acc}C, \text{lcl} = L \rangle \vdash (\text{dom}(\text{locals}(\text{store } s1')) \gg \langle c \rangle_s \gg S'$ 
  proof –
  note  $s0' - s1'$ 
  moreover
  from eval-e normal-s1' wt-e
  have  $\text{assigns-if } \text{True } e \subseteq \text{dom}(\text{locals}(\text{store } s1'))$ 
  by (rule assigns-if-good-approx' [elim-format])

```

```

      (simp add: True)
    ultimately
    have dom (locals (store ((Norm s0')::state)))
       $\cup$  assigns-if True  $e \subseteq$  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)  $\Diamond$  s2' Z and
    conf-s2': s2':: $\preceq$ (G,L)
  by (rule validE)
  hence P-s2': P  $\Diamond$  (abupd (absorb (Cont l)) s2') Z
  by simp
  from conf-s2'
  have conf-absorb: abupd (absorb (Cont l)) s2' :: $\preceq$ (G, L)
  by (cases s2') (auto intro: conforms-absorb)
  moreover
  obtain E' where
    da-while':
      ( $\text{prg} = G, \text{cls} = \text{accC}, \text{lcl} = L$ ) $\vdash$ 
        dom (locals (store (abupd (absorb (Cont l)) s2')))
        »  $\langle l \cdot \text{While}(e) \ c \rangle_s$  » E'
  proof -
    note s0'-s1'
    also
    from eval-c
    have  $G \vdash s1' - c \rightarrow s2'$ 
    by (rule evaln-eval)
    hence dom (locals (store s1'))  $\subseteq$  dom (locals (store s2'))
    by (rule dom-locals-eval-mono-elim)
    also
    have ...  $\subseteq$  dom (locals (store (abupd (absorb (Cont l)) s2')))
    by simp
    finally
    have dom (locals (store ((Norm s0')::state)))  $\subseteq$  ...
    with da show thesis
    by (rule da-weakenE) (rule that)
  qed
  from valid-A P-s2' conf-absorb wt da-while'
  show (P'  $\leftarrow$  False $\downarrow$  =  $\Diamond$ )  $\Diamond$  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:  $\bigwedge j. \text{abrupt } s1' \neq \text{Some } (\text{Jump } j)$ 
  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:  $G \vdash s1' - c - n' \rightarrow s2'$  and
    eval-while:
       $G \vdash \text{abupd} (\text{absorb} (\text{Cont } l)) s2' - l \cdot \text{While}(e) c - n' \rightarrow s3'$ 
    by (simp add: eqs)
  from eval-c abr have  $s2' = s1'$  by auto
  moreover from calculation no-jmp
  have  $\text{abupd} (\text{absorb} (\text{Cont } l)) s2' = s2'$ 
    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' =  $\langle t' = \langle l \cdot \text{While}(e) c \rangle_s \rangle$ 
note conf =  $\langle (\text{Some } \text{abr}, s) :: \preceq (G, L) \rangle$ 
note P =  $\langle P \ Y' (\text{Some } \text{abr}, s) \ Z \rangle$ 
note valid-A =  $\langle \forall t \in A. G \models n' :: t \rangle$ 
show  $(P' \leftarrow \text{False} \downarrow = \Diamond) (\text{undefined3 } t') (\text{Some } \text{abr}, s) \ Z$ 
proof -
  have eval-e:
     $G \vdash (\text{Some } \text{abr}, s) - \langle e \rangle_e \rightarrow - n' \rightarrow (\text{undefined3 } \langle e \rangle_e, (\text{Some } \text{abr}, s))$ 
  by auto
  from valid-e P valid-A conf eval-e
  have P'  $(\text{undefined3 } \langle e \rangle_e) (\text{Some } \text{abr}, s) \ Z$ 
    by (cases rule: validE [where ?P=P]) simp+
  with t' show ?thesis
    by auto
  qed
qed simp-all
from eval - valid-A P conf-s0 wt da
have  $(P' \leftarrow \text{False} \downarrow = \Diamond) \Diamond s3 \ Z$ 
  by (rule generalized) simp-all
moreover
have  $s3 :: \preceq (G, L)$ 
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  $G, A \models :: \{ \{ \text{Normal } (abupd (\lambda a. \text{Some } (\text{Jump } j)) \text{ } .; P \leftarrow \Diamond) \} \} . \text{Jump } j. \{ P \} \}$ 
  proof (rule valid-stmt-NormalI)
    fix n s0 L accC C s1 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:  $(\text{prg} = G, \text{cls} = \text{accC}, \text{lcl} = L) \vdash \text{Jump } j :: \checkmark$ 
    assume da:  $(\text{prg} = G, \text{cls} = \text{accC}, \text{lcl} = L) \vdash \text{dom } (\text{locals } (\text{store } s0)) \gg \langle \text{Jump } j \rangle_s \gg C$ 
    assume eval:  $G \vdash s0 \text{ -- Jump } j \text{ -- } n \rightarrow s1$ 
    assume P:  $(\text{Normal } (abupd (\lambda a. \text{Some } (\text{Jump } j)) \text{ } .; P \leftarrow \Diamond)) \text{ } Y \text{ } s0 \text{ } Z$ 
    show  $P \Diamond s1 \text{ } Z \wedge s1 :: \preceq (G, L)$ 
    proof -
      from eval obtain s where
        s:  $s0 = \text{Norm } s \text{ } s1 = (\text{Some } (\text{Jump } j), s)$ 
      using normal-s0 by (auto elim: evaln-elim-cases)
      with P have  $P \Diamond s1 \text{ } Z$ 
      by simp
    moreover
      from eval wt da conf-s0 wf
      have  $s1 :: \preceq (G, L)$ 
      by (rule evaln-type-sound [elim-format]) simp
    ultimately show ?thesis ..
  qed
  qed
next
  case (Throw A P e Q)
  note valid-e =  $\langle G, A \models :: \{ \{ \text{Normal } P \} \text{ } e \text{ -- } \succ \{ \lambda \text{Val} : a. . abupd (\text{throw } a) \text{ } .; Q \leftarrow \Diamond \} \} \rangle$ 
  show  $G, A \models :: \{ \{ \text{Normal } P \} \text{ } . \text{Throw } e. \{ Q \} \}$ 
  proof (rule valid-stmt-NormalI)
    fix n s0 L accC C s2 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:  $(\text{prg} = G, \text{cls} = \text{accC}, \text{lcl} = L) \vdash \text{Throw } e :: \checkmark$ 
    assume da:  $(\text{prg} = G, \text{cls} = \text{accC}, \text{lcl} = L) \vdash \text{dom } (\text{locals } (\text{store } s0)) \gg \langle \text{Throw } e \rangle_s \gg C$ 
    assume eval:  $G \vdash s0 \text{ -- Throw } e \text{ -- } n \rightarrow s2$ 
    assume P:  $(\text{Normal } P) \text{ } Y \text{ } s0 \text{ } Z$ 
    show  $Q \Diamond s2 \text{ } Z \wedge s2 :: \preceq (G, L)$ 
    proof -
      from eval obtain s1 a where
        eval-e:  $G \vdash s0 \text{ -- } e \text{ -- } \succ a \text{ -- } n \rightarrow s1$  and
        s2:  $s2 = abupd (\text{throw } a) \text{ } s1$ 
      using normal-s0 by (auto elim: evaln-elim-cases)
      from wt obtain T where
        wt-e:  $(\text{prg} = G, \text{cls} = \text{accC}, \text{lcl} = L) \vdash e :: -T$ 
      by cases simp
      from da obtain E where
        da-e:  $(\text{prg} = G, \text{cls} = \text{accC}, \text{lcl} = L) \vdash \text{dom } (\text{locals } (\text{store } s0)) \gg \langle e \rangle_e \gg E$ 
      by cases simp
      from valid-e P valid-A conf-s0 eval-e wt-e da-e

```

```

obtain ( $\lambda Val:a.. abupd (throw a) .; Q \leftarrow \Diamond$ )  $[a]_e s1 Z$ 
  by (rule validE)
with  $s2$  have  $Q \Diamond s2 Z$ 
  by simp
moreover
from eval wt da conf-s0 wf
have  $s2::\preceq(G,L)$ 
  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 =  $\langle G, A \models::\{ \{Normal P\} .c1. \{SXAlloc G Q\} \} \rangle$ 
note valid-c2 =  $\langle G, A \models::\{ \{Q \wedge. (\lambda s. G, s \vdash catch C) ;. new-xcpt-var vn\} .c2. \{R\} \} \rangle$ 
note  $Q-R = \langle (Q \wedge. (\lambda s. \neg G, s \vdash catch C)) \Rightarrow R \rangle$ 
show  $G, A \models::\{ \{Normal P\} .Try c1 Catch(C vn) c2. \{R\} \}$ 
proof (rule valid-stmt-NormalI)
  fix  $n s0 L accC E s3 Y Z$ 
  assume valid-A:  $\forall t \in A. G \models n::t$ 
  assume conf-s0:  $s0::\preceq(G,L)$ 
  assume normal-s0: normal s0
  assume wt:  $(\text{prg}=G, \text{cls}=accC, \text{lcl}=L) \vdash Try c1 Catch(C vn) c2::\checkmark$ 
  assume da:  $(\text{prg}=G, \text{cls}=accC, \text{lcl}=L) \vdash \text{dom} (locals (store s0)) \gg \langle Try c1 Catch(C vn) c2 \rangle_s \gg E$ 
  assume eval:  $G \vdash s0 - Try c1 Catch(C vn) c2 - n \rightarrow s3$ 
  assume P:  $(Normal P) Y s0 Z$ 
  show  $R \Diamond s3 Z \wedge s3::\preceq(G,L)$ 
proof –
  from eval obtain  $s1 s2$  where
    eval-c1:  $G \vdash s0 - c1 - n \rightarrow s1$  and
    sxalloc:  $G \vdash s1 - sxalloc \rightarrow s2$  and
     $s3$ : if  $G, s2 \vdash catch C$ 
      then  $G \vdash new-xcpt-var vn s2 - c2 - n \rightarrow s3$ 
      else  $s3 = s2$ 
  using normal-s0 by (fastforce elim: evaln-elim-cases)
from wt obtain
    wt-c1:  $(\text{prg}=G, \text{cls}=accC, \text{lcl}=L) \vdash c1::\checkmark$  and
    wt-c2:  $(\text{prg}=G, \text{cls}=accC, \text{lcl}=L (VName vn \mapsto Class C)) \vdash c2::\checkmark$ 
  by cases simp
from da obtain  $C1 C2$  where
    da-c1:  $(\text{prg}=G, \text{cls}=accC, \text{lcl}=L) \vdash \text{dom} (locals (store s0)) \gg \langle c1 \rangle_s \gg C1$  and
    da-c2:  $(\text{prg}=G, \text{cls}=accC, \text{lcl}=L (VName vn \mapsto Class C)) \vdash (\text{dom} (locals (store s0)) \cup \{VName vn\}) \gg \langle c2 \rangle_s \gg C2$ 
  by cases simp
from valid-c1 P valid-A conf-s0 eval-c1 wt-c1 da-c1
obtain sxQ:  $(SXAlloc G Q) \Diamond s1 Z$  and conf-s1:  $s1::\preceq(G,L)$ 
  by (rule validE)
from sxalloc sxQ
have  $Q \Diamond s2 Z$ 
  by auto
have  $R \Diamond s3 Z$ 
proof (cases  $\exists x. abrupt s1 = Some (Xcpt x)$ )
  case False
  from sxalloc wf
  have  $s2=s1$ 
  by (rule sxalloc-type-sound [elim-format])

```

```

      (use False in ⟨auto split: option.splits abrupt.splits⟩)
with False
have no-catch:  $\neg G, s2 \vdash \text{catch } C$ 
  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  $G, s2 \vdash \text{catch } C$ )
  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:  $G \vdash \text{new-xcpt-var } vn \ s2 \ -c2-n \rightarrow s3$ 
  by simp
from conf-s1 xalloc wf
have conf-s2:  $s2 :: \preceq (G, L)$ 
  by (auto dest: xalloc-type-sound
      split: option.splits abrupt.splits)
from exception-s1 xalloc wf
obtain a
  where xcpt-s2:  $\text{abrupt } s2 = \text{Some } (Xcpt (Loc a))$ 
  by (auto dest!: xalloc-type-sound
      split: option.splits abrupt.splits)
with True
have  $G \vdash \text{obj-ty } (the (globs (store s2) (Heap a))) \preceq \text{Class } C$ 
  by (cases s2) simp
with xcpt-s2 conf-s2 wf
have conf-new-xcpt:  $\text{new-xcpt-var } vn \ s2 :: \preceq (G, L (VName \text{vn} \mapsto \text{Class } C))$ 
  by (auto dest: Try-lemma)
obtain C2' where
  da-c2':
    ( $\text{prg} = G, \text{cls} = \text{acc } C, \text{lcl} = L (VName \text{vn} \mapsto \text{Class } C)$ )
     $\vdash (\text{dom } (locals (store (\text{new-xcpt-var } vn \ s2)))) \gg \langle c2 \rangle_s \gg C2'$ 
proof -
  have  $(\text{dom } (locals (store s0))) \cup \{VName \text{vn}\} \subseteq \text{dom } (locals (store (\text{new-xcpt-var } vn \ s2)))$ 
  proof -
    from eval-c1
    have  $\text{dom } (locals (store s0)) \subseteq \text{dom } (locals (store s1))$ 
      by (rule dom-locals-evaln-mono-elim)
    also
    from xalloc
    have  $\dots \subseteq \text{dom } (locals (store s2))$ 
      by (rule dom-locals-xalloc-mono)
    also
    have  $\dots \subseteq \text{dom } (locals (store (\text{new-xcpt-var } vn \ s2)))$ 
      by (cases s2) (simp add: new-xcpt-var-def, blast)

```



```

    also
    have { VName vn }  $\subseteq$  ...
    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  $\wedge$ . ( $\lambda s$ . G, s  $\vdash$  catch C) ;. new-xcpt-var vn)
     $\Diamond$  (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  $\Diamond$  s3 Z
  by (rule validE)
  qed
  qed
  moreover
  from eval wt da conf-s0 wf
  have s3 ::  $\preceq$ (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 =  $\langle G, A \models :: \{ \{ \text{Normal } P \} . c1 . \{ Q \} \} \rangle$ 
have valid-c2:  $\bigwedge$  abr. G, A  $\models :: \{ \{ Q \wedge . (\lambda s. \text{abr} = \text{fst } s) ;. \text{abupd } (\lambda x. \text{None}) \}$ 
   $.c2.$ 
   $\{ \text{abupd } (\text{abrupt-if } (\text{abr} \neq \text{None}) \text{abr}) .; R \} \}$ 
  using Fin.hyps by simp
show G, A  $\models :: \{ \{ \text{Normal } P \} . c1 \text{ Finally } c2 . \{ R \} \}$ 
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:  $(\langle \text{prg} = G, \text{cls} = \text{accC}, \text{lcl} = L \rangle \vdash c1 \text{ Finally } c2 :: \checkmark)$ 
  assume da:  $(\langle \text{prg} = G, \text{cls} = \text{accC}, \text{lcl} = L \rangle \vdash \text{dom } (\text{locals } (\text{store } s0)) \gg \langle c1 \text{ Finally } c2 \rangle_s \gg E)$ 
  assume eval:  $G \vdash s0 - c1 \text{ Finally } c2 - n \rightarrow s3$ 
  assume P: (Normal P) Y s0 Z
  show R  $\Diamond$  s3 Z  $\wedge$  s3 ::  $\preceq$ (G, L)
  proof -
    from eval obtain s1 abr1 s2 where
      eval-c1:  $G \vdash s0 - c1 - n \rightarrow (\text{abr1}, s1)$  and
      eval-c2:  $G \vdash \text{Norm } s1 - c2 - n \rightarrow s2$  and
      s3:  $s3 = (\text{if } \exists \text{err}. \text{abr1} = \text{Some } (\text{Error } \text{err})$ 
        then (abr1, s1)
        else  $\text{abupd } (\text{abrupt-if } (\text{abr1} \neq \text{None}) \text{abr1}) s2)$ 
    using normal-s0 by (fastforce elim: evaln-elim-cases)
    from wt obtain
      wt-c1:  $(\langle \text{prg} = G, \text{cls} = \text{accC}, \text{lcl} = L \rangle \vdash c1 :: \checkmark)$  and
      wt-c2:  $(\langle \text{prg} = G, \text{cls} = \text{accC}, \text{lcl} = L \rangle \vdash c2 :: \checkmark)$ 
    by cases simp
    from da obtain C1 C2 where
      da-c1:  $(\langle \text{prg} = G, \text{cls} = \text{accC}, \text{lcl} = L \rangle \vdash \text{dom } (\text{locals } (\text{store } s0)) \gg \langle c1 \rangle_s \gg C1)$  and
      da-c2:  $(\langle \text{prg} = G, \text{cls} = \text{accC}, \text{lcl} = L \rangle \vdash \text{dom } (\text{locals } (\text{store } s0)) \gg \langle c2 \rangle_s \gg C2)$ 

```

```

    by cases simp
  from valid-c1 P valid-A conf-s0 eval-c1 wt-c1 da-c1
  obtain Q:  $Q \Diamond (abr1, s1) Z$  and conf-s1:  $(abr1, s1) :: \preceq(G, L)$ 
    by (rule validE)
  from Q
  have Q':  $(Q \wedge. (\lambda s. abr1 = fst\ s) \therefore abupd\ (\lambda x. None)) \Diamond (Norm\ s1) Z$ 
    by auto
  from eval-c1 wt-c1 da-c1 conf-s0 wf
  have error-free (abr1, s1)
    by (rule evaln-type-sound [elim-format]) (use normal-s0 in 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':  $(\langle prg = G, cls = accC, lcl = L \rangle \vdash dom\ (locals\ (store\ ((Norm\ s1) :: state))) \gg \langle c2 \rangle_s \gg C2')$ 
  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  $(abupd\ (abrupt\text{-}if\ (abr1 \neq None)\ abr1) \therefore R) \Diamond s2 Z$ 
    by (rule validE)
  with s3' have  $R \Diamond s3 Z$ 
    by simp
  moreover
  from eval wt da conf-s0 wf
  have  $s3 :: \preceq(G, L)$ 
    by (rule evaln-type-sound [elim-format]) simp
  ultimately show ?thesis ..
qed
qed
next
case (Done A P C)
show  $G, A \models :: \{ \{ Normal\ (P \leftarrow \Diamond \wedge. initd\ C) \} . Init\ C. \{ P \} \}$ 
  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:  $(\langle prg = G, cls = accC, lcl = L \rangle \vdash Init\ C :: \checkmark)$ 
    assume da:  $(\langle prg = G, cls = accC, lcl = L \rangle \vdash dom\ (locals\ (store\ s0)) \gg \langle Init\ C \rangle_s \gg E)$ 
    assume eval:  $G \vdash s0 - Init\ C - n \rightarrow s3$ 
    assume P:  $(Normal\ (P \leftarrow \Diamond \wedge. initd\ C))\ Y\ s0\ Z$ 
    show  $P \Diamond s3 Z \wedge s3 :: \preceq(G, L)$ 
  proof -
    from P have initd:  $initd\ C\ (globs\ (store\ s0))$ 
      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 ( $\text{Init } C \ c \ A \ P \ Q \ R$ )
note  $c = \langle \text{the } (\text{class } G \ C) = c \rangle$ 
note valid-super =
   $\langle G, A \rangle \models \{ \{ \text{Normal } (P \wedge. \text{Not} \circ \text{initd } C \ ;. \text{supd } (\text{init-class-obj } G \ C)) \}$ 
     $\cdot (\text{if } C = \text{Object} \text{ then } \text{Skip} \text{ else } \text{Init } (\text{super } c)).$ 
     $\{ Q \} \}$ 
have valid-init:
   $\bigwedge l. \ G, A \models \{ \{ Q \wedge. (\lambda s. l = \text{locals } (\text{snd } s)) \ ;. \text{set-lvars } \text{Map.empty} \}$ 
     $\cdot \text{init } c.$ 
     $\{ \text{set-lvars } l \ ;. R \} \}$ 
using Init.hyps by simp
show  $G, A \models \{ \{ \text{Normal } (P \wedge. \text{Not} \circ \text{initd } C) \} \cdot \text{Init } C. \{ R \} \}$ 
proof (rule valid-stmt-NormalI)
  fix  $n \ s0 \ L \ \text{acc} \ C \ E \ s3 \ Y \ Z$ 
  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:  $\langle \text{prg} = G, \text{cls} = \text{acc } C, \text{lcl} = L \rangle \vdash \text{Init } C :: \checkmark$ 
  assume da:  $\langle \text{prg} = G, \text{cls} = \text{acc } C, \text{lcl} = L \rangle$ 
     $\vdash \text{dom } (\text{locals } (\text{store } s0)) \gg \langle \text{Init } C \rangle_s \gg E$ 
  assume eval:  $G \vdash s0 \rightarrow \text{Init } C \rightarrow n \rightarrow s3$ 
  assume  $P$ :  $(\text{Normal } (P \wedge. \text{Not} \circ \text{initd } C)) \ Y \ s0 \ Z$ 
  show  $R \ \Diamond \ s3 \ Z \wedge s3 :: \preceq (G, L)$ 
proof –
  from  $P$  have not-initd:  $\neg \text{initd } C \ (\text{globs } (\text{store } s0))$  by simp
  with eval c obtain  $s1 \ s2$  where
    eval-super:
       $G \vdash \text{Norm } ((\text{init-class-obj } G \ C) \ (\text{store } s0))$ 
       $\rightarrow (\text{if } C = \text{Object} \text{ then } \text{Skip} \text{ else } \text{Init } (\text{super } c)) \rightarrow n \rightarrow s1$  and
    eval-init:  $G \vdash (\text{set-lvars } \text{Map.empty}) \ s1 \rightarrow \text{init } c \rightarrow n \rightarrow s2$  and
     $s3: s3 = (\text{set-lvars } (\text{locals } (\text{store } s1))) \ s2$ 
    using normal-s0 by (auto elim!: evaln-elim-cases)
  from wt c have
    cls-C: class  $G \ C = \text{Some } c$ 
    by cases auto
  from wf cls-C have
    wt-super:  $\langle \text{prg} = G, \text{cls} = \text{acc } C, \text{lcl} = L \rangle$ 
       $\vdash (\text{if } C = \text{Object} \text{ then } \text{Skip} \text{ else } \text{Init } (\text{super } c)) :: \checkmark$ 
    by (cases C=Object)
    (auto dest: wf-prog-cdecl wf-cdecl-supD is-acc-classD)
  obtain  $S$  where
    da-super:
       $\langle \text{prg} = G, \text{cls} = \text{acc } C, \text{lcl} = L \rangle$ 
       $\vdash \text{dom } (\text{locals } (\text{store } ((\text{Norm } ((\text{init-class-obj } G \ C) \ (\text{store } s0))) :: \text{state})))$ 
       $\gg \langle \text{if } C = \text{Object} \text{ then } \text{Skip} \text{ else } \text{Init } (\text{super } c) \rangle_s \gg S$ 
  proof (cases C=Object)
    case True
    with da-Skip show ?thesis
      using that by (auto intro: assigned.select-convs)
  next
    case False
    show ?thesis
      by (rule that) (use da-Init False in  $\langle \text{auto intro$ : assigned.select-convs  $\rangle$ )

```

```

qed
from normal-s0 conf-s0 wf cls-C not-initd
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=acc C, 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  $G, A \models :: \{ \{ \text{Normal } P \} \text{ InsInitV } c \ v = \succ \{ Q \} \}$ 
proof (rule valid-var-NormalI)
  fix  $s0 \ v \ n \ s1 \ L \ Z$ 
  assume  $\text{normal } s0$ 
  moreover
  assume  $G \vdash s0 \ \neg \text{InsInitV } c \ v = \succ \ v \neg n \rightarrow s1$ 
  ultimately have  $\text{False}$ 
  by (cases  $s0$ ) (simp add: evaln-InsInitV)
  thus  $Q \ [v]_v \ s1 \ Z \wedge s1 :: \preceq (G, L) ..$ 
qed
next
case ( $\text{InsInitE } A \ P \ c \ e \ Q$ )
show  $G, A \models :: \{ \{ \text{Normal } P \} \text{ InsInitE } c \ e = \succ \{ Q \} \}$ 
proof (rule valid-expr-NormalI)
  fix  $s0 \ v \ n \ s1 \ L \ Z$ 
  assume  $\text{normal } s0$ 
  moreover
  assume  $G \vdash s0 \ \neg \text{InsInitE } c \ e = \succ \ v \neg n \rightarrow s1$ 
  ultimately have  $\text{False}$ 
  by (cases  $s0$ ) (simp add: evaln-InsInitE)
  thus  $Q \ [v]_e \ s1 \ Z \wedge s1 :: \preceq (G, L) ..$ 
qed
next
case ( $\text{Callee } A \ P \ l \ e \ Q$ )
show  $G, A \models :: \{ \{ \text{Normal } P \} \text{ Callee } l \ e = \succ \{ Q \} \}$ 
proof (rule valid-expr-NormalI)
  fix  $s0 \ v \ n \ s1 \ L \ Z$ 
  assume  $\text{normal } s0$ 
  moreover
  assume  $G \vdash s0 \ \neg \text{Callee } l \ e = \succ \ v \neg n \rightarrow s1$ 
  ultimately have  $\text{False}$ 
  by (cases  $s0$ ) (simp add: evaln-Callee)
  thus  $Q \ [v]_e \ s1 \ Z \wedge s1 :: \preceq (G, L) ..$ 
qed
next
case ( $\text{FinA } A \ P \ a \ c \ Q$ )
show  $G, A \models :: \{ \{ \text{Normal } P \} \text{ .FinA } a \ c. \{ Q \} \}$ 
proof (rule valid-stmt-NormalI)
  fix  $s0 \ v \ n \ s1 \ L \ Z$ 
  assume  $\text{normal } s0$ 
  moreover
  assume  $G \vdash s0 \ \neg \text{FinA } a \ c \neg n \rightarrow s1$ 
  ultimately have  $\text{False}$ 
  by (cases  $s0$ ) (simp add: evaln-FinA)
  thus  $Q \ \Diamond \ s1 \ Z \wedge s1 :: \preceq (G, L) ..$ 
qed
qed
declare inj-term-simps [simp del]

theorem ax-sound:
   $\text{wf-prog } G \implies G, (A :: 'a \text{ triple set}) \vdash (ts :: 'a \text{ triple set}) \implies G, A \models ts$ 
apply (subst ax-valids2-eq [symmetric])
apply assumption
apply (erule (1) ax-sound2)
done

lemma sound-valid2-lemma:
   $\llbracket \forall v \ n. \text{Ball } A \ (\text{triple-valid2 } G \ n) \longrightarrow P \ v \ n; \text{Ball } A \ (\text{triple-valid2 } G \ n) \rrbracket$ 
   $\implies P \ v \ n$ 

```

494

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 initialized classes

definition

nyinitcls :: *prog* \Rightarrow *state* \Rightarrow *qname* *set*
where *nyinitcls* *G* *s* = { *C*. *is-class* *G* *C* \wedge \neg *initd* *C* *s* }

lemma *nyinitcls-subset-class*: *nyinitcls* *G* *s* \subseteq { *C*. *is-class* *G* *C* }

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*) \leq *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*)) \leq *n* \Longrightarrow *card* (*nyinitcls* *G* (*y*, *s*)) \leq *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*: $\llbracket nyinitcls\ G\ s = \{\};\ is-class\ G\ C \rrbracket \implies initd\ C\ s$

unfolding *nyinitcls-def* **by** *fast*

lemma *card-Suc-lemma*:

$\llbracket card\ (insert\ a\ A) \leq Suc\ n;\ a \notin A;\ finite\ A \rrbracket \implies card\ A \leq n$

by *auto*

lemma *nyinitcls-le-SucD*:

$\llbracket card\ (nyinitcls\ G\ (x,s)) \leq Suc\ n;\ \neg initd\ C\ (globs\ s);\ class\ G\ C = Some\ y \rrbracket \implies$

$card\ (nyinitcls\ G\ (x,init-class-obj\ G\ C\ s)) \leq n$

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-initdD* *elim!:*

simp *add: nyinitcls-def initd-def split: if-split-asm*)

done

lemma *initd-gext'*: $\llbracket s \leq |s'|; initd\ C\ (globs\ s) \rrbracket \implies initd\ C\ (globs\ s')$

by (*rule* *initd-gext*)

lemma *nyinitcls-gext*: $snd\ s \leq |snd\ s' \implies nyinitcls\ G\ s' \subseteq nyinitcls\ G\ s$

unfolding *nyinitcls-def* **by** (*force* *dest!:* *initd-gext'*)

lemma *card-nyinitcls-gext*:

$\llbracket snd\ s \leq |snd\ s'; card\ (nyinitcls\ G\ s) \leq n \rrbracket \implies card\ (nyinitcls\ G\ s') \leq n$

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* \Rightarrow *nat* \Rightarrow *state* \Rightarrow *bool* ($\lhd \vdash init \leq \rhd$ [51,51] 50)

where $G \vdash init \leq n = (\lambda s. card\ (nyinitcls\ G\ s) \leq n)$

lemma *init-le-def2* [*simp*]: $(G \vdash init \leq n)\ s = (card\ (nyinitcls\ G\ s) \leq n)$

apply (*unfold* *init-le-def*)

apply *auto*

done

lemma *All-init-leD*:

$\forall n::nat. G, (A::'a\ triple\ set) \vdash \{P \wedge. G \vdash init \leq n\} \ t \succ \{Q::'a\ assn\}$

$\implies G, A \vdash \{P\} \ t \succ \{Q\}$

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* ($\langle \dot{=} \rangle$)
where $\dot{=} \equiv \lambda Y s Z. s = Z$

lemma *remember-init-state-def2* [*simp*]: $\dot{=} Y = (=)$

apply (*unfold remember-init-state-def*)

apply (*simp* (*no-asm*))

done

definition

MGF :: [*state assn*, *term*, *prog*] \Rightarrow *state triple* ($\langle \{-\} \rightarrow \{-\rightarrow\} \rangle [3,65,3] 62$)
where $\{P\} t \succ \{G \rightarrow\} = \{P\} t \succ \{\lambda Y s' s. G \vdash s - t \rightarrow (Y, s')\}$

definition

MGFn :: [*nat*, *term*, *prog*] \Rightarrow *state triple* ($\langle \{=: \} \rightarrow \{-\rightarrow\} \rangle [3,65,3] 62$)
where $\{=:n\} t \succ \{G \rightarrow\} = \{\dot{=} \wedge. G \vdash \text{init} \leq n\} t \succ \{G \rightarrow\}$

lemma *MGF-valid*: *wf-prog* $G \Longrightarrow G, \{\dot{=}\} \models \{\dot{=}\} t \succ \{G \rightarrow\}$

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*:

$G, A \vdash \{=:n\} t \succ \{G \rightarrow\} = G, A \vdash \{\dot{=} \wedge. G \vdash \text{init} \leq n\}$
 $t \succ \{\lambda Y s' s. G \vdash s - t \rightarrow (Y, s')\}$

unfolding *MGFn-def MGF-def* **by** *fast*

lemma *MGF-MGFn-iff*:

$G, (A :: \text{state triple set}) \vdash \{\dot{=}\} t \succ \{G \rightarrow\} = (\forall n. G, A \vdash \{=:n\} t \succ \{G \rightarrow\})$

apply (*simp add: MGFn-def2 MGF-def*)

apply *safe*

apply (*erule-tac* [2] *All-init-leD*)

apply (*erule consequ1*)

apply *clarsimp*

done

lemma *MGFnD*:

$G, (A :: \text{state triple set}) \vdash \{=:n\} t \succ \{G \rightarrow\} \Longrightarrow$
 $G, A \vdash \{(\lambda Y' s' s. s' = s \wedge P s) \wedge. G \vdash \text{init} \leq n\}$
 $t \succ \{(\lambda Y' s' s. G \vdash s - t \rightarrow (Y', s') \wedge P s) \wedge. G \vdash \text{init} \leq n\}$

apply (*unfold init-le-def*)

apply (*simp* (*no-asm-use*) *add: MGFn-def2*)

apply (*erule consequ12*)

apply *clarsimp*

apply (*erule* (1) *eval-geat* [*THEN card-nyinitcls-geat*])

done

lemmas *MGFnD' = MGFnD* [*of - - -* $\lambda x. \text{True}$]

To derive the most general formula, we can always assume a normal state in the precondition, since abrupt cases can be handled uniformly by the abrupt rule.

lemma *MGFNormalI*: $G, A \vdash \{Normal \dot{=}\} t \succ \{G \rightarrow\} \implies$
 $G, (A :: state\ triple\ set) \vdash \{\dot{=} :: state\ assn\} t \succ \{G \rightarrow\}$
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*:
 $G, (A :: state\ triple\ set) \vdash \{\dot{=}\} t \succ \{G \rightarrow\} \implies G, A \vdash \{Normal \dot{=}\} t \succ \{G \rightarrow\}$
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*:
 $G, (A :: state\ triple\ set) \vdash \{Normal((\lambda Y' s' s. s' = s \wedge normal\ s) \wedge. G \vdash init \leq n)\} t \succ$
 $\{\lambda Y s' s. G \vdash s - t \rightarrow (Y, s')\} \implies G, A \vdash \{=:n\} t \succ \{G \rightarrow\}$
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 uniformly handled by the hazard rule.

lemma *MGFn-free-wt*:
 $(\exists T L C. (\text{prg} = G, \text{cls} = C, \text{lcl} = L) \vdash t :: T)$
 $\longrightarrow G, (A :: state\ triple\ set) \vdash \{=:n\} t \succ \{G \rightarrow\}$
 $\implies G, A \vdash \{=:n\} t \succ \{G \rightarrow\}$
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 uniformly handled by the hazard rule.

lemma *MGFn-free-wt-NormalConformI*:
 $(\forall T L C. (\text{prg} = G, \text{cls} = C, \text{lcl} = L) \vdash t :: T)$
 $\longrightarrow G, (A :: state\ triple\ set)$
 $\vdash \{Normal((\lambda Y' s' s. s' = s \wedge normal\ s) \wedge. G \vdash init \leq n) \wedge. (\lambda s. s :: \preceq (G, L))\}$
 $t \succ$
 $\{\lambda Y s' s. G \vdash s - t \rightarrow (Y, s')\}$
 $\implies G, A \vdash \{=:n\} t \succ \{G \rightarrow\}$
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 definitely assigned with respect to this state. All type violations can be uniformly handled by the hazard rule.

lemma *MGFn-free-wt-da-NormalConformI*:

```

(∀ T L C B. (prg=G, cls=C, lcl=L) ⊢ t :: T
  → G, (A :: state triple set)
    ⊢ {Normal((λ Y' s' s. s' = s ∧ normal s) ∧. G ⊢ init ≤ n) ∧. (λ s. s :: ⊢ (G, L))
      ∧. (λ s. (prg=G, cls=C, lcl=L) ⊢ dom (locals (store s)) » t » B)}
    t >
    {λ Y s' s. G ⊢ s - t > → (Y, s')})
  ⇒ G, A ⊢ {=: n} t > {G →}
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: ∀ m. Suc m ≤ n → (∀ t. G, A ⊢ {=: m} t > {G →})
shows G, (A :: state triple set) ⊢ {=: n} ⟨Init C⟩s > {G →}
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
    show ?thesis
    by (rule ax-impossible [THEN conseq1]) (use is-cls 0 in ⟨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 ∧ ¬initd C (globs s) ∧. G ⊢ init ≤ m
  from c
  show ?thesis
  proof (rule ax-derivs.Init [where ?Q = ?Q])
    let ?P' = Normal ((λY 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])
      using 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-methods:
      ∀ 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 $G, A \vdash \{=:n\} \langle \{accC, statT, mode\} e \cdot mn(\{pTs'\}ps) \rangle_e \succ \{G \rightarrow\}$

proof (rule *MGFn-free-wt-da-NormalConformI* [rule-format], clarsimp)

note *inj-term-simps* [simp]

fix $T \ L \ accC' \ E$

assume $wt: (\text{prg}=G, \text{cls}=accC', \text{lcl}=L) \vdash \langle \{accC, statT, mode\} e \cdot mn(\{pTs'\}ps) \rangle_e :: T$

then obtain $pTs \ statDeclT \ statM$ **where**

$wt-e: (\text{prg}=G, \text{cls}=accC', \text{lcl}=L) \vdash e :: -RefT \ statT$ **and**

$wt-args: (\text{prg}=G, \text{cls}=accC', \text{lcl}=L) \vdash ps :: \dot{=} pTs$ **and**

$statM: \text{max-spec } G \ accC \ statT \ (\text{name}=mn, \text{parTs}=pTs)$

$= \{((statDeclT, statM), pTs')\}$ **and**

$mode: mode = \text{invmode } statM \ e$ **and**

$T: T = \text{Inl } (\text{resTy } statM)$ **and**

$eq\text{-}accC\text{-}accC': accC = accC'$

by cases *fastforce+*

let $?Q = (\lambda Y \ s1 \ (x, s) . x = \text{None} \wedge$

$(\exists a. G \vdash \text{Norm } s - e - \succ a \rightarrow s1 \wedge$

$(\text{normal } s1 \longrightarrow G, \text{store } s1 \vdash a :: \preceq RefT \ statT)$

$\wedge Y = \text{Inl } a) \wedge$

$(\exists P. \text{normal } s1$

$\longrightarrow (\text{prg}=G, \text{cls}=accC', \text{lcl}=L) \vdash \text{dom } (\text{locals } (\text{store } s1)) \gg \langle ps \rangle_l \gg P))$

$\wedge G \vdash \text{init} \leq n \wedge (\lambda s. s :: \preceq (G, L)) :: \text{state } \text{assn}$

let $?R = \lambda a. ((\lambda Y \ (x2, s2) \ (x, s) . x = \text{None} \wedge$

$(\exists s1 \ pvs. G \vdash \text{Norm } s - e - \succ a \rightarrow s1 \wedge$

$(\text{normal } s1 \longrightarrow G, \text{store } s1 \vdash a :: \preceq RefT \ statT) \wedge$

$Y = [pvs]_l \wedge G \vdash s1 - ps \dot{=} \succ pvs \rightarrow (x2, s2)))$

$\wedge G \vdash \text{init} \leq n \wedge (\lambda s. s :: \preceq (G, L)) :: \text{state } \text{assn}$

show $G, A \vdash \{Normal \ ((\lambda Y' \ s' \ s. s' = s \wedge \text{abrupt } s = \text{None}) \wedge G \vdash \text{init} \leq n \wedge$

$(\lambda s. s :: \preceq (G, L)) \wedge$

$(\lambda s. (\text{prg}=G, \text{cls}=accC', \text{lcl}=L) \vdash \text{dom } (\text{locals } (\text{store } s))$

$\gg \langle \{accC, statT, mode\} e \cdot mn(\{pTs'\}ps) \rangle_e \gg E)\}$

$\{accC, statT, mode\} e \cdot mn(\{pTs'\}ps) - \succ$

$\{\lambda Y \ s' \ s. \exists v. Y = [v]_e \wedge$

$G \vdash s - \{accC, statT, mode\} e \cdot mn(\{pTs'\}ps) - \succ v \rightarrow s'\}$

$\}$ **(is** $G, A \vdash \{Normal \ ?P\} \{accC, statT, mode\} e \cdot mn(\{pTs'\}ps) - \succ \{?S\}$

proof (rule *ax-derivs.Call* [where $?Q = ?Q$ and $?R = ?R$])

from *mgf-e*

show $G, A \vdash \{Normal \ ?P\} e - \succ \{?Q\}$

proof (rule *MGFnD'* [THEN *conseq12*], clarsimp)

fix $s0 \ s1 \ a$

assume $\text{conf-s0}: \text{Norm } s0 :: \preceq (G, L)$

assume $da: (\text{prg}=G, \text{cls}=accC', \text{lcl}=L) \vdash$

$\text{dom } (\text{locals } s0) \gg \langle \{accC, statT, mode\} e \cdot mn(\{pTs'\}ps) \rangle_e \gg E$

assume $\text{eval-e}: G \vdash \text{Norm } s0 - e - \succ a \rightarrow s1$

show $(\text{abrupt } s1 = \text{None} \longrightarrow G, \text{store } s1 \vdash a :: \preceq RefT \ statT) \wedge$

$(\text{abrupt } s1 = \text{None} \longrightarrow$

$(\exists P. (\text{prg}=G, \text{cls}=accC', \text{lcl}=L) \vdash \text{dom } (\text{locals } (\text{store } s1)) \gg \langle ps \rangle_l \gg P))$

$\wedge s1 :: \preceq (G, L)$

proof –

from *da* **obtain** C **where**

$da-e: (\text{prg}=G, \text{cls}=accC, \text{lcl}=L) \vdash$

$\text{dom } (\text{locals } (\text{store } ((\text{Norm } s0) :: \text{state}))) \gg \langle e \rangle_e \gg C$ **and**

$da-ps: (\text{prg}=G, \text{cls}=accC, \text{lcl}=L) \vdash \text{nrm } C \gg \langle ps \rangle_l \gg E$

by cases (*simp add: eq-accC-accC'*)

from *eval-e conf-s0 wt-e da-e wf*

obtain $(\text{abrupt } s1 = \text{None} \longrightarrow G, \text{store } s1 \vdash a :: \preceq RefT \ statT)$

and $s1 :: \preceq (G, L)$

by (rule *eval-type-soundE*) *simp*

moreover have $\exists P. (\text{prg}=G, \text{cls}=accC, \text{lcl}=L) \vdash \text{dom } (\text{locals } (\text{store } s1)) \gg \langle ps \rangle_l \gg P$

```

    if normal-s1: normal s1
  proof -
    from eval-e wt-e da-e wf normal-s1
    have nrm  $C \subseteq \text{dom} (\text{locals} (\text{store } s1))$ 
      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  $\forall a. G, A \vdash \{ ?Q \leftarrow \text{In1 } a \} ps \dot{=} \succ \{ ?R \ a \} \text{ (is } \forall a. ?PS \ a)$ 
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:  $s1 :: \preceq (G, L)$ 
  assume eval-e:  $G \vdash \text{Norm } s0 \text{ } -e \text{ } \succ a \rightarrow s1$ 
  assume conf-a:  $\text{abrupt } s1 = \text{None} \longrightarrow G, \text{store } s1 \vdash a :: \preceq \text{RefT } \text{statT}$ 
  assume eval-ps:  $G \vdash s1 \text{ } -ps \dot{=} \succ vs \rightarrow s2$ 
  assume da-ps:  $\text{abrupt } s1 = \text{None} \longrightarrow$ 
     $(\exists P. (\text{prg} = G, \text{cls} = \text{accC}, \text{lcl} = L) \vdash$ 
       $\text{dom} (\text{locals} (\text{store } s1)) \gg \langle ps \rangle_l \gg P)$ 
  show  $(\exists s1. G \vdash \text{Norm } s0 \text{ } -e \text{ } \succ a \rightarrow s1 \wedge$ 
     $(\text{abrupt } s1 = \text{None} \longrightarrow G, \text{store } s1 \vdash a :: \preceq \text{RefT } \text{statT}) \wedge$ 
     $G \vdash s1 \text{ } -ps \dot{=} \succ vs \rightarrow s2) \wedge$ 
     $s2 :: \preceq (G, L)$ 
  proof (cases normal s1)
    case True
    with da-ps obtain P where
       $(\text{prg} = G, \text{cls} = \text{accC}, \text{lcl} = L) \vdash \text{dom} (\text{locals} (\text{store } s1)) \gg \langle ps \rangle_l \gg P$ 
    by auto
    from eval-ps conf-s1 wt-args this wf
    have  $s2 :: \preceq (G, L)$ 
      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  $\forall a \text{ vs } \text{invC } \text{declC } l.$ 
 $G, A \vdash \{ ?R \ a \leftarrow \lfloor vs \rfloor_l \wedge.$ 
 $(\lambda s. \text{declC} =$ 
   $\text{invocation-declclass } G \text{ mode } (\text{store } s) \ a \ \text{statT}$ 
 $\ (\text{name} = \text{mn}, \text{parTs} = \text{pTs}') \wedge$ 
 $\text{invC} = \text{invocation-class } \text{mode } (\text{store } s) \ a \ \text{statT} \wedge$ 
 $\ l = \text{locals } (\text{store } s)) ;.$ 

```

```

    init-lvars G declC ( $\langle \text{name}=\text{mn}, \text{parTs}=\text{pTs}' \rangle$ ) mode a vs  $\wedge$ .
    ( $\lambda s. \text{normal } s \longrightarrow G \vdash \text{mode} \rightarrow \text{invC} \preceq \text{statT}$ )
  Methd declC ( $\langle \text{name}=\text{mn}, \text{parTs}=\text{pTs}' \rangle$ )  $\neg$ 
  {set-lvars l .; ?S}
(is  $\forall a \text{ vs invC declC l. ?METHOD } a \text{ vs invC declC l}$ )
proof (intro allI)
  fix a vs invC declC l
  from mgf-methds [rule-format]
  show ?METHOD 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
    ( $\langle \text{name}=\text{mn}, \text{parTs}=\text{pTs}' \rangle$ )
  let ?s3= init-lvars G ?D ( $\langle \text{name}=\text{mn}, \text{parTs}=\text{pTs}' \rangle$ ) mode a vs s2
  assume inv-prop: abrupt ?s3=None
     $\longrightarrow G \vdash \text{mode} \rightarrow \text{invocation-class mode (store s2) a statT} \preceq \text{statT}$ 
  assume conf-s2: s2:: $\preceq$ (G, L)
  assume conf-a: abrupt s1 = None  $\longrightarrow G, \text{store s1} \vdash a::\preceq \text{RefT statT}$ 
  assume eval-e:  $G \vdash \text{Norm } s0 -e \neg a \rightarrow s1$ 
  assume eval-ps:  $G \vdash s1 -ps \neg vs \rightarrow s2$ 
  assume eval-mthd:  $G \vdash ?s3 -\text{Methd } ?D (\langle \text{name}=\text{mn}, \text{parTs}=\text{pTs}' \rangle) \neg v \rightarrow s4$ 
  show  $G \vdash \text{Norm } s0 -\{\text{accC}, \text{statT}, \text{mode}\} e \cdot \text{mn}(\{pTs'\}ps) \neg v$ 
     $\rightarrow (\text{set-lvars (locals (store s2))}) s4$ 
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
      ( $\langle \text{name}=\text{mn}, \text{parTs}=\text{pTs}' \rangle$ ) a s3
    by simp
  have eq-s3'-s3: s3'=s3
proof -
  from inv-prop s3 mode
  have normal s3  $\implies$ 
     $G \vdash \text{invmode statM } e \rightarrow \text{invocation-class mode (store s2) a statT} \preceq \text{statT}$ 
  by auto
  with eval-ps wt-e statM conf-s2 conf-a [rule-format]
  have check-method-access G accC statT (invmode statM e)
    ( $\langle \text{name}=\text{mn}, \text{parTs}=\text{pTs}' \rangle$ ) 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  $G \vdash s3' -\text{Methd } D (\langle \text{name}=\text{mn}, \text{parTs}=\text{pTs}' \rangle) \neg v \rightarrow s4$ 
  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:  $G \vdash s0 -e \neg v \rightarrow s1$ 
  and no-jmp: abrupt s0  $\neq \text{Some (Jump j)}$ 

```

```

and      wt: ( $\text{prg}=G, \text{cls}=C, \text{lcl}=L$ ) $\vdash e::-T$ 
and      wf: wf-prog  $G$ 
shows abrupt  $s1 \neq \text{Some } (\text{Jump } j)$ 
by (rule eval-expression-no-jump [where ?Env=( $\text{prg}=G, \text{cls}=C, \text{lcl}=L$ ),simplified])
    (use eval no-jmp wt wf in 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  $\Rightarrow$  label  $\Rightarrow$  expr  $\Rightarrow$  stmt  $\Rightarrow$  (state  $\times$  state) set where
unroll  $G \ l \ e \ c = \{(s, t). \exists \ v \ s1 \ s2. \quad$ 
     $G \vdash s - e \rightarrow v \rightarrow s1 \wedge \text{the-Bool } v \wedge \text{normal } s1 \wedge$ 
     $G \vdash s1 - c \rightarrow s2 \wedge t = (\text{abupd } (\text{absorb } (\text{Cont } l)) \ s2)\}$ 

```

lemma unroll-while:

```

assumes unroll:  $(s, t) \in (\text{unroll } G \ l \ e \ c)^*$ 
and      eval-e:  $G \vdash t - e \rightarrow v \rightarrow s'$ 
and      normal-termination:  $\text{normal } s' \longrightarrow \neg \text{the-Bool } v$ 
and      wt: ( $\text{prg}=G, \text{cls}=C, \text{lcl}=L$ ) $\vdash e::-T$ 
and      wf: wf-prog  $G$ 
shows  $G \vdash s - l \cdot \text{While}(e) \ c \rightarrow s'$ 

```

using unroll

proof (induct rule: converse-rtrancl-induct)

```

show  $G \vdash t - l \cdot \text{While}(e) \ c \rightarrow s'$ 

```

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  $s' \neq \text{Some } (\text{Jump } (\text{Cont } l))$ 

```

```

      by (rule eval-expression-no-jump') (use normal-t in simp)

```

```

    have

```

```

      if the-Bool  $v$ 

```

```

        then ( $G \vdash s' - c \rightarrow s' \wedge$ 

```

```

           $G \vdash (\text{abupd } (\text{absorb } (\text{Cont } l)) \ s') - l \cdot \text{While}(e) \ c \rightarrow s'$ )

```

```

        else  $s' = s'$ 

```

proof (cases the-Bool v)

```

  case False thus ?thesis by simp

```

next

```

  case True

```

```

    with abrupt-s' have  $G \vdash s' - c \rightarrow s'$  by auto

```

```

    moreover from abrupt-s' no-cont

```

```

    have no-absorb:  $(\text{abupd } (\text{absorb } (\text{Cont } l)) \ s') = s'$ 

```



```

    by (cases s') (simp add: absorb-def split: if-split)
  moreover
  from no-absorb abrupt-s'
  have  $G \vdash (\text{abupd } (\text{absorb } (\text{Cont } l)) \ s') \text{ --l. While}(e) \ c \rightarrow s'$ 
    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:  $(s, s3) \in \text{unroll } G \ l \ e \ c$ 
assume while:  $G \vdash s3 \text{ --l. While}(e) \ c \rightarrow s'$ 
show  $G \vdash s \text{ --l. While}(e) \ c \rightarrow s'$ 
proof -
  from unroll obtain v s1 s2 where
    normal-s1: normal s1 and
    eval-e:  $G \vdash s \text{ --e--> } v \rightarrow s1$  and
    continue: the-Bool v and
    eval-c:  $G \vdash s1 \text{ --c--> } s2$  and
    s3:  $s3 = (\text{abupd } (\text{absorb } (\text{Cont } l)) \ s2)$ 
  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:  $G, (A :: \text{state triple set}) \vdash \{=n\} \langle e \rangle_e \succ \{G \rightarrow\}$ 
  and mfg-c:  $G, A \vdash \{=n\} \langle c \rangle_s \succ \{G \rightarrow\}$ 
  and wf: wf-prog G
shows  $G, A \vdash \{=n\} \langle l. \text{While}(e) \ c \rangle_s \succ \{G \rightarrow\}$ 
proof (rule MGFn-free-wt [rule-format], elim exE)
  fix T L C
  assume wt:  $\langle \text{prg} = G, \text{cls} = C, \text{lcl} = L \rangle \vdash \langle l. \text{While}(e) \ c \rangle_{s::T}$ 
  then obtain eT where
    wt-e:  $\langle \text{prg} = G, \text{cls} = C, \text{lcl} = L \rangle \vdash e :: \text{--eT}$ 
  by cases simp
  show ?thesis
  proof (rule MGFn-NormalI)
    show  $G, A \vdash \{ \text{Normal } ((\lambda Y' s' s. s' = s \wedge \text{normal } s) \wedge. G \vdash \text{init} \leq n) \}$ 
      .l.  $\text{While}(e) \ c.$ 
       $\{ \lambda Y s' s. G \vdash s \text{ --In1r } (l. \text{While}(e) \ c) \succ \rightarrow (Y, s') \}$ 
    proof (rule conseq12)
      [where ?P' =  $(\lambda Y s' s. (s, s') \in (\text{unroll } G \ l \ e \ c)^*) \wedge. G \vdash \text{init} \leq n$ 
      and ?Q' =  $((\lambda Y s' s. (\exists t b. (s, t) \in (\text{unroll } G \ l \ e \ c)^* \wedge$ 
         $Y = \lfloor b \rfloor_e \wedge G \vdash t \text{ --e--> } b \rightarrow s'))$ 
         $\wedge. G \vdash \text{init} \leq n) \leftarrow \text{False} \downarrow = \Diamond$ ]]
      show  $G, A \vdash \{ (\lambda Y s' s. (s, s') \in (\text{unroll } G \ l \ e \ c)^*) \wedge. G \vdash \text{init} \leq n \}$ 
        .l.  $\text{While}(e) \ c.$ 
         $\{ ((\lambda Y s' s. (\exists t b. (s, t) \in (\text{unroll } G \ l \ e \ c)^* \wedge$ 
           $Y = \text{In1 } b \wedge G \vdash t \text{ --e--> } b \rightarrow s'))$ 

```

```

       $\wedge. G \vdash \text{init} \leq n) \leftarrow \text{False} \downarrow = \Diamond \}$ 
proof (rule ax-derivs.Loop)
  from mfg-e
  show  $G, A \vdash \{(\lambda Y s' s. (s, s') \in (\text{unroll } G \text{ l e c})^*) \wedge. G \vdash \text{init} \leq n\}$ 
     $e \multimap$ 
     $\{(\lambda Y s' s. (\exists t b. (s, t) \in (\text{unroll } G \text{ l e c})^* \wedge$ 
       $Y = \text{In1 } b \wedge G \vdash t - e \multimap b \rightarrow s'))$ 
     $\wedge. G \vdash \text{init} \leq n\}$ 
proof (rule MGFnD' [THEN conseq12], clarsimp)
  fix  $s Z s' v$ 
  assume  $(Z, s) \in (\text{unroll } G \text{ l e c})^*$ 
  moreover
  assume  $G \vdash s - e \multimap v \rightarrow s'$ 
  ultimately
  show  $\exists t. (Z, t) \in (\text{unroll } G \text{ l e c})^* \wedge G \vdash t - e \multimap v \rightarrow s'$ 
    by blast
qed
next
from mfg-c
show  $G, A \vdash \{\text{Normal } (((\lambda Y s' s. \exists t b. (s, t) \in (\text{unroll } G \text{ l e c})^* \wedge$ 
     $Y = \lfloor b \rfloor_e \wedge G \vdash t - e \multimap b \rightarrow s') \wedge. G \vdash \text{init} \leq n) \leftarrow \text{True})\}$ 
    .c.
     $\{abupd (\text{absorb } (\text{Cont } l)) \} ;$ 
     $\{(\lambda Y s' s. (s, s') \in (\text{unroll } G \text{ l e c})^*) \wedge. G \vdash \text{init} \leq n\}$ 
proof (rule MGFnD' [THEN conseq12], clarsimp)
  fix  $Z s' s v t$ 
  assume unroll:  $(Z, t) \in (\text{unroll } G \text{ l e c})^*$ 
  assume eval-e:  $G \vdash t - e \multimap v \rightarrow \text{Norm } s$ 
  assume true: the-Bool v
  assume eval-c:  $G \vdash \text{Norm } s - c \rightarrow s'$ 
  show  $(Z, abupd (\text{absorb } (\text{Cont } l)) s') \in (\text{unroll } G \text{ l e c})^*$ 
  proof -
    note unroll
    also
    from eval-e true eval-c
    have  $(t, abupd (\text{absorb } (\text{Cont } l)) s') \in \text{unroll } G \text{ l e c}$ 
    by (unfold unroll-def) force
    ultimately show ?thesis ..
  qed
qed
qed
next
show
   $\forall Y s Z.$ 
   $(\text{Normal } ((\lambda Y' s' s. s' = s \wedge \text{normal } s) \wedge. G \vdash \text{init} \leq n)) Y s Z$ 
   $\longrightarrow (\forall Y' s'.$ 
     $(\forall Y Z'.$ 
       $((\lambda Y s' s. (s, s') \in (\text{unroll } G \text{ l e c})^*) \wedge. G \vdash \text{init} \leq n) Y s Z'$ 
       $\longrightarrow (((\lambda Y s' s. \exists t b. (s, t) \in (\text{unroll } G \text{ l e c})^*$ 
         $\wedge Y = \lfloor b \rfloor_e \wedge G \vdash t - e \multimap b \rightarrow s')$ 
         $\wedge. G \vdash \text{init} \leq n) \leftarrow \text{False} \downarrow = \Diamond) Y' s' Z')$ 
       $\longrightarrow G \vdash Z - \langle l \cdot \text{While}(e) \ c \rangle_s \multimap (Y', s'))$ 
    )
  )
proof (clarsimp)
  fix  $Y' s' s$ 
  assume asm:
     $\forall Z'. (Z', \text{Norm } s) \in (\text{unroll } G \text{ l e c})^*$ 
     $\longrightarrow \text{card } (\text{nyinitcls } G s') \leq n \wedge$ 
     $(\exists v. (\exists t. (Z', t) \in (\text{unroll } G \text{ l e c})^* \wedge G \vdash t - e \multimap v \rightarrow s')) \wedge$ 

```

```

      (fst s' = None  $\longrightarrow$   $\neg$  the-Bool v))  $\wedge$  Y' =  $\Diamond$ 
show Y' =  $\Diamond$   $\wedge$  G $\vdash$ Norm s -l. While(e) c $\rightarrow$  s'
proof -
  from asm obtain v t where
    — Z' gets instantiated with Norm s
    unroll: (Norm s, t)  $\in$  (unroll G l e c)* and
    eval-e: G $\vdash$ t -e- $\succ$ v $\rightarrow$  s' and
    normal-termination: normal s'  $\longrightarrow$   $\neg$  the-Bool v and
    Y': Y' =  $\Diamond$ 
  by auto
  from unroll eval-e normal-termination wt-e wf
  have G $\vdash$ Norm s -l. While(e) c $\rightarrow$  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 $\vdash$ {=:n} (Init statDeclC)s $\succ$  {G $\rightarrow$ }
  and mgf-e: G, A $\vdash$ {=:n} (e)e $\succ$  {G $\rightarrow$ }
  and wf: wf-prog G
  shows G, A $\vdash$ {=:n} ({accC, statDeclC, stat}e..fn)v $\succ$  {G $\rightarrow$ }
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) $\vdash$ 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 = ( $\lambda$  Y s1 (x, s) . x = None  $\wedge$ 
    (G $\vdash$ Norm s -Init statDeclC $\rightarrow$  s1)  $\wedge$ 
    ( $\exists$  E. (prg = G, cls = accC', lcl = L) $\vdash$ dom (locals (store s1))  $\gg$  (e)e E))
     $\wedge$ . G $\vdash$ init $\leq$ n  $\wedge$ . ( $\lambda$  s. s:: $\preceq$ (G, L))
  show G, A $\vdash$ {Normal
    (( $\lambda$  Y' s' s. s' = s  $\wedge$  abrupt s = None)  $\wedge$ . G $\vdash$ init $\leq$ n  $\wedge$ .
    ( $\lambda$  s. s:: $\preceq$ (G, L))  $\wedge$ .
    ( $\lambda$  s. (prg = G, cls = accC', lcl = L)
       $\vdash$  dom (locals (store s))  $\gg$  ({accC, statDeclC, stat}e..fn)v V))
    } {accC, statDeclC, stat}e..fn $\succ$ 
    { $\lambda$  Y s' s.  $\exists$  vf. Y =  $\lfloor$ vf $\rfloor_v$   $\wedge$ 
      G $\vdash$ s -{accC, statDeclC, stat}e..fn $\succ$ vf $\rightarrow$  s'}
    (is G, A $\vdash$ {Normal ?P} {accC, statDeclC, stat}e..fn $\succ$  {?R})
  proof (rule ax-derivs.FVar [where ?Q=?Q])
    from mgf-init
    show G, A $\vdash$ {Normal ?P} .Init statDeclC. {?Q}
  proof (rule MGFnD' [THEN conseq12], clarsimp)
    fix s s'
    assume conf-s: Norm s:: $\preceq$ (G, L)
    assume da: (prg = G, cls = accC', lcl = L)
       $\vdash$  dom (locals s)  $\gg$  ({accC, statDeclC, stat}e..fn)v V

```

```

assume eval-init:  $G \vdash \text{Norm } s \rightarrow \text{Init statDeclC} \rightarrow s'$ 
show  $(\exists E. (\text{prg}=G, \text{cls}=\text{accC}', \text{lcl}=L) \vdash \text{dom } (\text{locals } (\text{store } s')) \gg \langle e \rangle_e \gg E) \wedge$ 
 $s'::\preceq(G, L)$ 
proof –
  from da
  obtain E where
     $(\text{prg}=G, \text{cls}=\text{accC}', \text{lcl}=L) \vdash \text{dom } (\text{locals } s) \gg \langle e \rangle_e \gg E$ 
    by cases simp
  moreover
  from eval-init
  have  $\text{dom } (\text{locals } s) \subseteq \text{dom } (\text{locals } (\text{store } s'))$ 
    by (rule dom-locals-eval-mono [elim-format]) simp
  ultimately obtain E' where
     $(\text{prg}=G, \text{cls}=\text{accC}', \text{lcl}=L) \vdash \text{dom } (\text{locals } (\text{store } s')) \gg \langle e \rangle_e \gg E'$ 
    by (rule da-weakenE)
  moreover
  have  $s'::\preceq(G, L)$ 
  proof –
    have wt-init:  $(\text{prg}=G, \text{cls}=\text{accC}, \text{lcl}=L) \vdash (\text{Init statDeclC})::\checkmark$ 
    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:  $(\text{prg}=G, \text{cls}=\text{accC}, \text{lcl}=L) \vdash$ 
 $\text{dom } (\text{locals } (\text{store } ((\text{Norm } s)::\text{state}))) \gg \langle \text{Init statDeclC} \rangle_s \gg I$ 
      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  $G, A \vdash \{?Q\} \ e \rightarrow \{ \lambda \text{Val}:a. \text{fvar statDeclC stat fn } a \ ..; ?R \}$ 
  proof (rule MGFnD' [THEN conseq12], clarsimp)
    fix s0 s1 s2 E a
    let ?fvar = fvar statDeclC stat fn a s2
    assume eval-init:  $G \vdash \text{Norm } s0 \rightarrow \text{Init statDeclC} \rightarrow s1$ 
    assume eval-e:  $G \vdash s1 \rightarrow e \rightarrow a \rightarrow s2$ 
    assume conf-s1:  $s1::\preceq(G, L)$ 
    assume da-e:  $(\text{prg}=G, \text{cls}=\text{accC}', \text{lcl}=L) \vdash \text{dom } (\text{locals } (\text{store } s1)) \gg \langle e \rangle_e \gg E$ 
    show  $G \vdash \text{Norm } s0 \rightarrow \{ \text{accC}, \text{statDeclC}, \text{stat} \} e.. \text{fn} \Rightarrow \text{fst } ?\text{fvar} \rightarrow \text{snd } ?\text{fvar}$ 
    proof –
      obtain v s2' where
         $v = \text{fst } ?\text{fvar}$  and  $s2' = \text{snd } ?\text{fvar}$ 
        by simp
      obtain s3 where
         $s3 = \text{check-field-access } G \text{ accC}' \text{ statDeclC fn stat } a \ s2'$ 
        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 consequ12],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: ( $\langle \text{prg} = G, \text{cls} = \text{acc}C, \text{lcl} = L \rangle \vdash \text{dom} (\text{locals } s0) \gg \langle c1 \rangle_s \gg C1$ )
  assume conf-s0: Norm s0 ::  $\preceq (G, L)$ 
  assume eval-c1:  $G \vdash \text{Norm } s0 - c1 \rightarrow s1$ 
  assume eval-c2:  $G \vdash \text{abupd } (\lambda \text{abr. None}) s1 - c2 \rightarrow s2$ 
  show  $G \vdash \text{Norm } s0 - c1 \text{ Finally } c2$ 
     $\rightarrow \text{abupd } (\text{abrupt-if } (\exists y. \text{abrupt } s1 = \text{Some } y) (\text{abrupt } s1)) s2$ 
proof -
  obtain abr1 str1 where s1: s1 = (abr1, str1)
  by (cases s1)
  with eval-c1 eval-c2 obtain
    eval-c1':  $G \vdash \text{Norm } s0 - c1 \rightarrow (\text{abr1}, \text{str1})$  and
    eval-c2':  $G \vdash \text{Norm } \text{str1} - c2 \rightarrow s2$ 
  by simp
  obtain s3 where
    s3: s3 = (if  $\exists \text{err. } \text{abr1} = \text{Some } (\text{Error } \text{err})$ 
      then (abr1, str1)
      else  $\text{abupd } (\text{abrupt-if } (\text{abr1} \neq \text{None}) \text{abr1}) s2$ )
  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 =  $\text{abupd } (\text{abrupt-if } (\text{abr1} \neq \text{None}) \text{abr1}) s2$ 
  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:  $G \vdash \text{Norm } s0 - \text{Init } D \rightarrow s1$ 
and eval-c:  $G \vdash s1 - c \rightarrow s2$ 
and jmpOk:  $\text{jumpNestingOkS } \{\text{Ret}\} c$ 
and wt-c: ( $\langle \text{prg} = G, \text{cls} = C, \text{lcl} = L \rangle \vdash c :: \checkmark$ )
and clsD:  $\text{class } G D = \text{Some } d$ 
and wf: wf-prog G
shows  $\forall l. \text{abrupt } s2 \neq \text{Some } (\text{Jump } (\text{Break } l)) \wedge$ 
   $\text{abrupt } s2 \neq \text{Some } (\text{Jump } (\text{Cont } l))$ 
proof
  fix l show  $\text{abrupt } s2 \neq \text{Some } (\text{Jump } (\text{Break } l)) \wedge$ 
     $\text{abrupt } s2 \neq \text{Some } (\text{Jump } (\text{Cont } l))$ 
  proof -
  fix accC
  from clsD have wt-init: ( $\langle \text{prg} = G, \text{cls} = \text{acc}C, \text{lcl} = L \rangle \vdash (\text{Init } D) :: \checkmark$ )
  by auto
  have s1-no-jmp:  $\bigwedge j. \text{abrupt } s1 \neq \text{Some } (\text{Jump } j)$ 
  by (rule eval-statement-no-jump [OF - - wt-init]) (use eval-init wf in auto)
  from eval-c - wt-c wf show ?thesis
  by (rule jumpNestingOk-eval [THEN conjE, elim-format]) (use jmpOk s1-no-jmp in auto)
qed
qed

lemma MGFn-Body:
assumes wf: wf-prog G
and mgf-init:  $G, A \vdash \{ =: n \} \langle \text{Init } D \rangle_s \succ \{ G \rightarrow \}$ 

```

and $\text{mgf-c: } G, A \vdash \{=:n\} \langle c \rangle_s \succ \{G \rightarrow\}$
shows $G, (A::\text{state triple set}) \vdash \{=:n\} \langle \text{Body } D \ c \rangle_e \succ \{G \rightarrow\}$
proof (rule *MGFn-free-wt-da-NormalConformI* [rule-format], clarsimp)
fix $T \ L \ \text{accC} \ E$
assume $\text{wt: } (\text{prg}=G, \text{cls}=\text{accC}, \text{lcl}=L) \vdash \langle \text{Body } D \ c \rangle_e :: T$
let $?Q = (\lambda Y' \ s' \ s. \text{normal } s \wedge G \vdash s - \text{Init } D \rightarrow s' \wedge \text{jumpNestingOkS } \{ \text{Ret} \} \ c)$
 $\wedge. G \vdash \text{init} \leq n$
show $G, A \vdash \{ \text{Normal} \}$
 $((\lambda Y' \ s' \ s. s' = s \wedge \text{fst } s = \text{None}) \wedge. G \vdash \text{init} \leq n \wedge.$
 $(\lambda s. s :: \preceq (G, L)) \wedge.$
 $(\lambda s. (\text{prg}=G, \text{cls}=\text{accC}, \text{lcl}=L))$
 $\vdash \text{dom } (\text{locals } (\text{store } s)) \gg \langle \text{Body } D \ c \rangle_e \gg E))\}$
 $\text{Body } D \ c \rightarrow$
 $\{ \lambda Y \ s' \ s. \exists v. Y = \text{In1 } v \wedge G \vdash s - \text{Body } D \ c \rightarrow v \rightarrow s' \}$
(is $G, A \vdash \{ \text{Normal } ?P \} \text{Body } D \ c \rightarrow \{ ?R \}$)
proof (rule *ax-derivs.Body* [where $?Q = ?Q$])
from *mgf-init*
show $G, A \vdash \{ \text{Normal } ?P \} . \text{Init } D. \{ ?Q \}$
proof (rule *MGFnD'* [THEN *conseq12*], clarsimp)
fix $s0$
assume $\text{da: } (\text{prg}=G, \text{cls}=\text{accC}, \text{lcl}=L) \vdash \text{dom } (\text{locals } s0) \gg \langle \text{Body } D \ c \rangle_e \gg E$
thus $\text{jumpNestingOkS } \{ \text{Ret} \} \ c$
by *cases simp*
qed
next
from *mgf-c*
show $G, A \vdash \{ ?Q \}. c. \{ \lambda s.. \text{abupd } (\text{absorb } \text{Ret}) .; ?R \leftarrow [\text{the } (\text{locals } s \ \text{Result})]_e \}$
proof (rule *MGFnD'* [THEN *conseq12*], clarsimp)
fix $s0 \ s1 \ s2$
assume $\text{eval-init: } G \vdash \text{Norm } s0 - \text{Init } D \rightarrow s1$
assume $\text{eval-c: } G \vdash s1 - c \rightarrow s2$
assume $\text{nestingOk: } \text{jumpNestingOkS } \{ \text{Ret} \} \ c$
show $G \vdash \text{Norm } s0 - \text{Body } D \ c \rightarrow \text{the } (\text{locals } (\text{store } s2) \ \text{Result})$
 $\rightarrow \text{abupd } (\text{absorb } \text{Ret}) \ s2$
proof –
from *wt obtain d where*
 $d: \text{class } G \ D = \text{Some } d$ **and**
 $\text{wt-c: } (\text{prg} = G, \text{cls} = \text{accC}, \text{lcl} = L) \vdash c :: \checkmark$
by *cases auto*
obtain $s3$ **where**
 $s3: s3 = (\text{if } \exists l. \text{fst } s2 = \text{Some } (\text{Jump } (\text{Break } l)) \vee$
 $\text{fst } s2 = \text{Some } (\text{Jump } (\text{Cont } l))$
 $\text{then } \text{abupd } (\lambda x. \text{Some } (\text{Error } \text{CrossMethodJump})) \ s2$
 $\text{else } s2)$
by *simp*
from *eval-init eval-c nestingOk wt-c d wf*
have $\text{eq-s3-s2: } s3 = s2$
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-methods*:
 $\wedge n. \forall C \ \text{sig}. G, (A::\text{state triple set}) \vdash \{=:n\} \langle \text{Methd } C \ \text{sig} \rangle_e \succ \{G \rightarrow\}$

```

and wf: wf-prog G
shows  $\bigwedge t. G, A \vdash \{=:n\} \ t \succ \{G \rightarrow\}$ 
proof (induct rule: full-nat-induct)
  fix n t
  assume hyp:  $\forall m. \text{Suc } m \leq n \longrightarrow (\forall t. G, A \vdash \{=:m\} \ t \succ \{G \rightarrow\})$ 
  show  $G, A \vdash \{=:n\} \ t \succ \{G \rightarrow\}$ 
  proof –
    fix v e c es
    have  $G, A \vdash \{=:n\} \ \langle v \rangle_v \succ \{G \rightarrow\}$  and
       $G, A \vdash \{=:n\} \ \langle e \rangle_e \succ \{G \rightarrow\}$  and
       $G, A \vdash \{=:n\} \ \langle c \rangle_s \succ \{G \rightarrow\}$  and
       $G, A \vdash \{=:n\} \ \langle es \rangle_l \succ \{G \rightarrow\}$ 
    for v e c es
  proof (induct rule: compat-var.induct compat-expr.induct compat-stmt.induct compat-expr-list.induct)
    case (LVar v)
      show  $G, A \vdash \{=:n\} \ \langle LVar \ v \rangle_v \succ \{G \rightarrow\}$ 
      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  $\langle G, A \vdash \{=:n\} \ \langle e \rangle_e \succ \{G \rightarrow\} \rangle$  and wf
      show ?case
      by (rule MGFn-FVar)
    next
      case (AVar e1 e2)
      note mgf-e1 =  $\langle G, A \vdash \{=:n\} \ \langle e1 \rangle_e \succ \{G \rightarrow\} \rangle$ 
      note mgf-e2 =  $\langle G, A \vdash \{=:n\} \ \langle e2 \rangle_e \succ \{G \rightarrow\} \rangle$ 
      show  $G, A \vdash \{=:n\} \ \langle e1.[e2] \rangle_v \succ \{G \rightarrow\}$ 
      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  $?Q = (\lambda Y' s' s. \text{normal } s \wedge G \vdash s - \text{In1r } (\text{init-comp-ty } T) \succ \rightarrow (Y', s')) \wedge. G \vdash \text{init} \leq n]$ 
      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  $G, A \vdash \{=:n\} \langle v := e \rangle_e \succ \{G \rightarrow\}$ 
  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  $G, A \vdash \{=:n\} \langle e1 \ ? \ e2 : e3 \rangle_e \succ \{G \rightarrow\}$ 
  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 =  $\langle G, A \vdash \{=:n\} \langle e \rangle_e \succ \{G \rightarrow\} \rangle$ 
note mgf-ps =  $\langle G, A \vdash \{=:n\} \langle ps \rangle_l \succ \{G \rightarrow\} \rangle$ 
from mgf-methds mgf-e mgf-ps wf
show  $G, A \vdash \{=:n\} \langle \{accC, statT, mode\} e \cdot mn (\{pTs'\} ps) \rangle_e \succ \{G \rightarrow\}$ 
  by (rule MGFn-Call)

```

```

next
  case (Methd D mn)
  from mgf-methds
  show  $G, A \vdash \{=:n\} \langle \text{Methd } D \text{ } mn \rangle_e \succ \{G \rightarrow\}$ 
  by simp
next
  case (Body D c)
  note mgf-c =  $\langle G, A \vdash \{=:n\} \langle c \rangle_s \succ \{G \rightarrow\} \rangle$ 
  from wf MGFn-Init [OF hyp] mgf-c
  show  $G, A \vdash \{=:n\} \langle \text{Body } D \text{ } c \rangle_e \succ \{G \rightarrow\}$ 
  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  $G, A \vdash \{=:n\} \langle l \cdot c \rangle_s \succ \{G \rightarrow\}$ 
  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  $G, A \vdash \{=:n\} \langle c1 ;; c2 \rangle_s \succ \{G \rightarrow\}$ 
  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  $G, A \vdash \{=:n\} \langle \text{If}(e) \text{ } c1 \text{ Else } c2 \rangle_s \succ \{G \rightarrow\}$ 
  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 =  $\langle G, A \vdash \{=:n\} \langle e \rangle_e \succ \{G \rightarrow\} \rangle$ 
note mgf-c =  $\langle G, A \vdash \{=:n\} \langle c \rangle_s \succ \{G \rightarrow\} \rangle$ 
from mgf-e mgf-c wf
show  $G, A \vdash \{=:n\} \langle l \cdot \text{While}(e) \ c \rangle_s \succ \{G \rightarrow\}$ 
  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  $G, A \vdash \{=:n\} \langle \text{Try } c1 \text{ Catch}(C \text{ vn}) \ c2 \rangle_s \succ \{G \rightarrow\}$ 
  apply -
  apply (rule MGFn-NormalI)
  apply (rule ax-derivs.Try [where
    ?Q =  $(\lambda Y' s' s. \text{normal } s \wedge (\exists s''. G \vdash s - \langle c1 \rangle_s \rightarrow (Y', s'') \wedge$ 
     $G \vdash s'' - s \text{alloc} \rightarrow s') \wedge. G \vdash \text{init} \leq n]$ )
  apply (erule MGFnD [THEN ax-NormalD, THEN conseq2])
  apply (fastforce elim: salloc-geat [THEN card-nyinitcls-geat])
  apply (erule MGFnD' [THEN conseq12])
  apply (fastforce intro: eval.Try)
  apply (fastforce intro: eval.Try)
  done
next
case (Fin c1 c2)
note mgf-c1 =  $\langle G, A \vdash \{=:n\} \langle c1 \rangle_s \succ \{G \rightarrow\} \rangle$ 
note mgf-c2 =  $\langle G, A \vdash \{=:n\} \langle c2 \rangle_s \succ \{G \rightarrow\} \rangle$ 
from wf mgf-c1 mgf-c2
show  $G, A \vdash \{=:n\} \langle c1 \text{ Finally } c2 \rangle_s \succ \{G \rightarrow\}$ 
  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  $G, A \vdash \{=:n\} \langle \text{Init } C \rangle_s \succ \{G \rightarrow\}$ 
    by (rule MGFn-Init)
next
    case Nil-expr
    show  $G, A \vdash \{=:n\} \langle [] \rangle_l \succ \{G \rightarrow\}$ 
    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:*

```


$$\llbracket \forall C \text{ sig. is-methd } G \ C \text{ sig} \longrightarrow G, A \vdash \{ \doteq \} \text{ In1l } (\text{Methd } C \text{ sig}) \succ \{G \rightarrow\}; \text{ wf-prog } G \rrbracket$$


$$\implies G, (A :: \text{state triple set}) \vdash \{ \doteq \} t \succ \{G \rightarrow\}$$

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:  $\bigwedge A \text{ ts. } ts \subseteq A \implies P \ A \ ts$ 
and MGF-nested-Methd:  $\bigwedge A \text{ pn. } \forall b \in \text{bdy } pn. P \ (\text{insert } (\text{mgf-call } pn) \ A) \ \{\text{mgf } b\}$ 
 $\implies P \ A \ \{\text{mgf-call } pn\}$ 
and MGF-asm:  $\bigwedge A \ t. \forall pn \in U. P \ A \ \{\text{mgf-call } pn\} \implies P \ A \ \{\text{mgf } t\}$ 
and finU: finite U
and uA:  $uA = \text{mgf-call}' U$ 
shows  $\forall A. A \subseteq uA \longrightarrow n \leq \text{card } uA \longrightarrow \text{card } A = \text{card } uA - n$ 
 $\longrightarrow (\forall t. P \ A \ \{\text{mgf } t\})$ 
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:  $\bigwedge A \ ts. \ ts \subseteq A \implies P \ A \ ts$ 
  and MGF-nested-Methd:  $\bigwedge A \ pn. \ \forall b \in \text{bdy } pn. \ P \ (\text{insert } (\text{mgf } (f \ pn)) \ A) \ \{\text{mgf } b\}$ 
     $\implies P \ A \ \{\text{mgf } (f \ pn)\}$ 
  and MGF-asm:  $\bigwedge A \ t. \ \forall pn \in U. \ P \ A \ \{\text{mgf } (f \ pn)\} \implies P \ A \ \{\text{mgf } t\}$ 
  and finU: finite U
shows  $P \ \{\} \ \{\text{mgf } t\}$ 
using ax-derivs-asm MGF-nested-Methd MGF-asm finU
by (rule nesting-lemma') (auto intro!: le-refl)

```

```

lemma MGF-nested-Methd:  $\llbracket$ 
   $G, \text{insert } (\{ \text{Normal} \doteq \} \langle \text{Methd } C \ \text{sig} \rangle_e \succ \{ G \rightarrow \}) \ A$ 
   $\vdash \{ \text{Normal} \doteq \} \langle \text{body } G \ C \ \text{sig} \rangle_e \succ \{ G \rightarrow \}$ 
 $\rrbracket \implies G, A \vdash \{ \text{Normal} \doteq \} \langle \text{Methd } C \ \text{sig} \rangle_e \succ \{ G \rightarrow \}$ 
apply (unfold MGF-def)
apply (rule ax-MethdN)
apply (erule conseq2)
apply clarsimp
apply (erule MethdI)
done

```

```

lemma MGF-deriv:  $wf\text{-prog } G \implies G, (\{ \} :: \text{state triple set}) \vdash \{ \doteq \} \ t \succ \{ G \rightarrow \}$ 
apply (rule MGFNormalI)
apply (rule-tac  $\text{mgf} = \lambda t. \ \{ \text{Normal} \doteq \} \ t \succ \{ G \rightarrow \}$  and
   $\text{bdy} = \lambda (C, \text{sig}). \ \{ \langle \text{body } G \ C \ \text{sig} \rangle_e \}$  and
   $f = \lambda (C, \text{sig}). \ \langle \text{Methd } C \ \text{sig} \rangle_e$  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  $\implies$ 
   $G, A \cup (\lambda (C, \text{sig}). \ \{ \text{Normal} \doteq \} \langle \text{Methd } C \ \text{sig} \rangle_e \succ \{ G \rightarrow \}) \ ' \ ms$ 

```

```

  |-(λ(C,sig). {Normal ≐} ⟨body G C sig⟩e > {G→}) ‘ms ⇒
  G,A|-(λ(C,sig). {Normal ≐} ⟨Methd C sig⟩e > {G→}) ‘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 ⇒
  G,({::state triple set)|-(λ(C,sig). {Normal ≐} ⟨Methd C sig⟩e > {G→})
  ‘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⊢s -t>→ (Y', s'); type-ok G t s; wf-prog G|]
  ⇒ ∃ n. G⊢s -t>-n→ (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,{}|= {P} t> {Q}
  and mgf: G,({::state triple set)|- {≐} t> {G→}
  and wf: wf-prog G
  shows G,({::state triple set)|- {P::state assn} t> {Q}
proof (rule ax-no-hazard)
  from mgf
  have G,({::state triple set)|- {≐} t> {λ Y s' s. G⊢s -t>→ (Y, s')}
    by (unfold MGF-def)
  thus G,({::state triple set)|- {P ∧. type-ok G t} t> {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⊢s -t>→ (Y', s')
  show Q Y' s' Z
proof -
  from eval-t type-ok wf
  obtain n where evaln: G⊢s -t>-n→ (Y', s')
  by (rule eval-to-evaln [elim-format]) iprover

```

```

from valid have
  valid-expanded:
     $\forall n\ Y\ s\ Z. P\ Y\ s\ Z \longrightarrow \text{type-ok}\ G\ t\ s$ 
     $\longrightarrow (\forall Y'\ s'. G \vdash s - t \succ - n \rightarrow (Y', s') \longrightarrow Q\ Y'\ s'\ Z)$ 
  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:  $G, \{\} \models \{P::\text{state assn}\} t \succ \{Q\}$ 
  shows  $G, (\{\}::\text{state triple set}) \vdash \{P\} t \succ \{Q\}$ 
proof –
  from wf have  $G, (\{\}::\text{state triple set}) \vdash \{\dot{=}\} t \succ \{G \rightarrow\}$ 
    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 = (λs. ∃ obj a T el. globs s (Stat Base) = Some obj ∧
    values obj (Inl (arr, Base)) = Some (Addr a) ∧
    heap s a = Some (⟦tag=Arr T 2,values=el⟧))
```

lemma *arr-inv-new-obj*:

```
  ∧a. ⟦arr-inv s; new-Addr (heap s)=Some a⟧ ⇒ arr-inv (gupd(Inl a↦x) s)
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 initd-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  $P' = \text{Normal } P \text{ and } P = P \text{ for } P \text{ in } \text{conseq1}$ )
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  $P' \lambda a \text{ vs } l \text{ vf. } PP \text{ a vs } l \text{ vf} \leftarrow x \wedge. p [PP, x, p]$ ))
apply (rule allI)
apply (tactic (simp-tac (context |> Simplifier.del-loop split-all-tac |> Simplifier.del-simps [@{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  $R \lambda a'. \text{Normal } ((\lambda \text{Vals:vs } (x, s) Z. \text{arr-inv } s \wedge \text{initd Ext (globs } s) \wedge a' \neq \text{Null} \wedge \text{vs} = [\text{Null}]) \wedge. \text{heap-free two}) []$ ))
apply fastforce
prefer 4
apply (rule ax-derivs.Done [THEN conseq1],force)
apply (rule ax-subst-Val-allI)
apply (tactic (inst1-tac context  $P' \lambda a. \text{Normal } (PP \text{ a} \leftarrow x) [PP, x]$ ))
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  $P' \lambda aa \text{ v. Normal } (QQ \text{ aa } v \leftarrow y) [QQ, y]$ ))
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  $Q' = \text{Normal } ((\lambda Y (x, s) Z. \text{arr-inv } s \wedge (\exists a. \text{the (locals } s (VName \text{ e})) = \text{Addr } a \wedge \text{obj-class (the (globs } s (\text{Inl } a))) = \text{Ext } \wedge \text{invocation-declclass tprg IntVir } s (\text{the (locals } s (VName \text{ e}))) (\text{ClassT Base}) (\text{name} = \text{foo, parTs} = [\text{Class Base}]) = \text{Ext})) \wedge. \text{initd Ext } \wedge. \text{heap-free two})$ )
  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  $P' \lambda \text{vf. Normal } (PP \text{ vf } \wedge. p) [PP, p]$ ))
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  $Q' = \text{Normal } ((\lambda Y \ s \ Z. \text{arr-inv } (\text{store } s) \wedge \text{vf=lvar } (\text{VName } e) (\text{store } s)) \wedge. \text{heap-free three}$ 
 $\wedge. \text{initd Ext})$  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 |> Simplifier.del-loop split-all-tac) 1>)
apply (rule ax-Init-Skip-lemma)
apply (tactic <simp-tac (context |> Simplifier.del-loop 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  $P' = \text{Normal } P$  and  $P = P$  for  $P$  in conseq1)
apply (tactic <simp-tac (context |> Simplifier.del-loop 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  $P' \lambda \text{vf } l \text{ vfa. Normal } (P \text{ vf } l \text{ vfa}) [P]$ >)
apply (tactic <simp-tac (context |> Simplifier.del-loop split-all-tac |> Simplifier.del-simps [@{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  $P \lambda \text{vf } l \text{ vfa. Normal } (P \text{ vf } l \text{ vfa} \leftarrow \Diamond) [P]$ >)
apply (tactic <simp-tac (context |> Simplifier.del-loop 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  $Q \lambda \text{vf. Normal } ((\lambda Y \ s \ Z. \text{vf=lvar } (\text{VName } e) (\text{snd } s)) \wedge. \text{heap-free four}$ 
 $\wedge. \text{initd Base} \wedge. \text{initd Ext})$  []>)
apply (clarsimp split del: if-split)
apply (frule atleast-free-weaken [THEN atleast-free-weaken])
apply (drule initdD)
apply (clarsimp elim!: atleast-free-SucD simp add: arr-inv-def)
apply force
apply (tactic <simp-tac (context |> Simplifier.del-loop split-all-tac) 1>)
apply (rule ax-triv-Init-Object [THEN peek-and-forget2, THEN conseq1])
apply (rule wf-tpg)
apply clarsimp
apply (tactic <inst1-tac context  $P \lambda \text{vf. Normal } ((\lambda Y \ s \ Z. \text{vf} = \text{lvar } (\text{VName } e) (\text{snd } s)) \wedge. \text{heap-free four}$ 
 $\wedge. \text{initd Ext})$  []>)
apply clarsimp
apply (tactic <inst1-tac context  $PP \lambda \text{vf. Normal } ((\lambda Y \ s \ Z. \text{vf} = \text{lvar } (\text{VName } e) (\text{snd } s)) \wedge. \text{heap-free}$ 
 $\text{four} \wedge. \text{Not} \circ \text{initd Base})$  []>)
apply clarsimp

apply (rule conseq1)

```

```

apply (tactic ax-tac context 1)
apply clarsimp
done

```

lemma *Loop-Xcpt-benchmark:*

```

 $Q = (\lambda Y (x,s) Z. x \neq \text{None} \longrightarrow \text{the-Bool} (\text{the} (\text{locals } s \ i))) \implies$ 
 $G, (\{\} :: 'a \text{ triple set}) \vdash \{\text{Normal} (\lambda Y s Z :: 'a. \text{True})\}$ 
 $\text{.lab1} \cdot \text{While}(\text{Lit} (\text{Bool True})) (\text{If}(\text{Acc} (\text{LVar } i)) (\text{Throw} (\text{Acc} (\text{LVar } xcpt))) \text{Else}$ 
 $(\text{Expr} (\text{Ass} (\text{LVar } i) (\text{Acc} (\text{LVar } j))))). \{Q\}$ 
apply (rule-tac  $P' = Q$  and  $Q' = Q \leftarrow \text{False} \downarrow = \Diamond$  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
   $\langle \text{inst1-tac } \text{context } P' \text{ Normal } (\lambda s.. (\lambda Y s Z. \text{True}) \downarrow = \text{Val} (\text{the} (\text{locals } s \ i))) \rangle$ )
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  $Q' = \text{Normal} (\lambda Y s Z. \text{True})$  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  $\langle \text{inst1-tac } \text{context } P' \lambda b Y ba Z \text{vf}. \lambda Y (x,s) Z. x = \text{None} \wedge \text{snd } \text{vf} = \text{snd} (\text{lvar } i \ s) \rangle$ )
apply (rule allI)
apply (rule-tac  $P' = \text{Normal } P$  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