

# Java Source and Bytecode Formalizations in Isabelle: Bali

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

June 8, 2008



# Contents

<b>1</b>	<b>Overview</b>	<b>7</b>
<b>2</b>	<b>Basis</b>	<b>11</b>
1	Definitions extending HOL as logical basis of Bali . . . . .	12
<b>3</b>	<b>Table</b>	<b>19</b>
2	Abstract tables and their implementation as lists . . . . .	20
<b>4</b>	<b>Name</b>	<b>29</b>
3	Java names . . . . .	30
<b>5</b>	<b>Value</b>	<b>33</b>
4	Java values . . . . .	34
<b>6</b>	<b>Type</b>	<b>35</b>
5	Java types . . . . .	36
<b>7</b>	<b>Term</b>	<b>37</b>
6	Java expressions and statements . . . . .	38
<b>8</b>	<b>Decl</b>	<b>47</b>
7	Field, method, interface, and class declarations, whole Java programs . . . .	48
8	Modifier . . . . .	48
9	Declaration (base "class" for member,interface and class declarations . . . .	50
10	Member (field or method) . . . . .	50
11	Field . . . . .	50
12	Method . . . . .	51
13	Interface . . . . .	53
14	Class . . . . .	53
<b>9</b>	<b>TypeRel</b>	<b>63</b>
15	The relations between Java types . . . . .	64
<b>10</b>	<b>DeclConcepts</b>	<b>77</b>
16	Advanced concepts on Java declarations like overriding, inheritance, dynamic method lookup . . . . .	78
17	accessibility of types (cf. 6.6.1) . . . . .	78
18	accessibility of members . . . . .	79
19	imethds . . . . .	104
20	accimethd . . . . .	104
21	methd . . . . .	105
22	accmethd . . . . .	107
23	dynmethd . . . . .	108

24	dynlookup . . . . .	114
25	fields . . . . .	114
26	accfield . . . . .	116
27	is methd . . . . .	117
<b>11</b>	<b>WellType</b>	<b>121</b>
28	Well-typedness of Java programs . . . . .	122
<b>12</b>	<b>DefiniteAssignment</b>	<b>135</b>
29	Definite Assignment . . . . .	136
30	Very restricted calculation fallback calculation . . . . .	137
31	Analysis of constant expressions . . . . .	139
32	Main analysis for boolean expressions . . . . .	140
33	Lifting set operations to range of tables (map to a set) . . . . .	143
<b>13</b>	<b>WellForm</b>	<b>167</b>
34	Well-formedness of Java programs . . . . .	168
35	accessibility concerns . . . . .	208
<b>14</b>	<b>State</b>	<b>219</b>
36	State for evaluation of Java expressions and statements . . . . .	220
37	access . . . . .	224
38	memory allocation . . . . .	224
39	initialization . . . . .	225
40	update . . . . .	225
41	update . . . . .	231
<b>15</b>	<b>Eval</b>	<b>237</b>
42	Operational evaluation (big-step) semantics of Java expressions and statements	238
<b>16</b>	<b>Example</b>	<b>259</b>
43	Example Bali program . . . . .	260
<b>17</b>	<b>Conform</b>	<b>287</b>
44	Conformance notions for the type soundness proof for Java . . . . .	288
<b>18</b>	<b>DefiniteAssignmentCorrect</b>	<b>299</b>
45	Correctness of Definite Assignment . . . . .	300
<b>19</b>	<b>TypeSafe</b>	<b>377</b>
46	The type soundness proof for Java . . . . .	378
47	accessibility . . . . .	394
48	Ideas for the future . . . . .	443
<b>20</b>	<b>Evaln</b>	<b>449</b>
49	Operational evaluation (big-step) semantics of Java expressions and statements	450
<b>21</b>	<b>Trans</b>	<b>465</b>
<b>22</b>	<b>AxSem</b>	<b>473</b>
50	Axiomatic semantics of Java expressions and statements (see also Eval.thy) .	474
51	peek-and . . . . .	475
52	assn-supd . . . . .	475
53	supd-assn . . . . .	476

54	subst-res . . . . .	476
55	subst-Bool . . . . .	476
56	peek-res . . . . .	477
57	ign-res . . . . .	477
58	peek-st . . . . .	478
59	ign-res-eq . . . . .	479
60	RefVar . . . . .	479
61	allocation . . . . .	480

## 23 AxSound 495

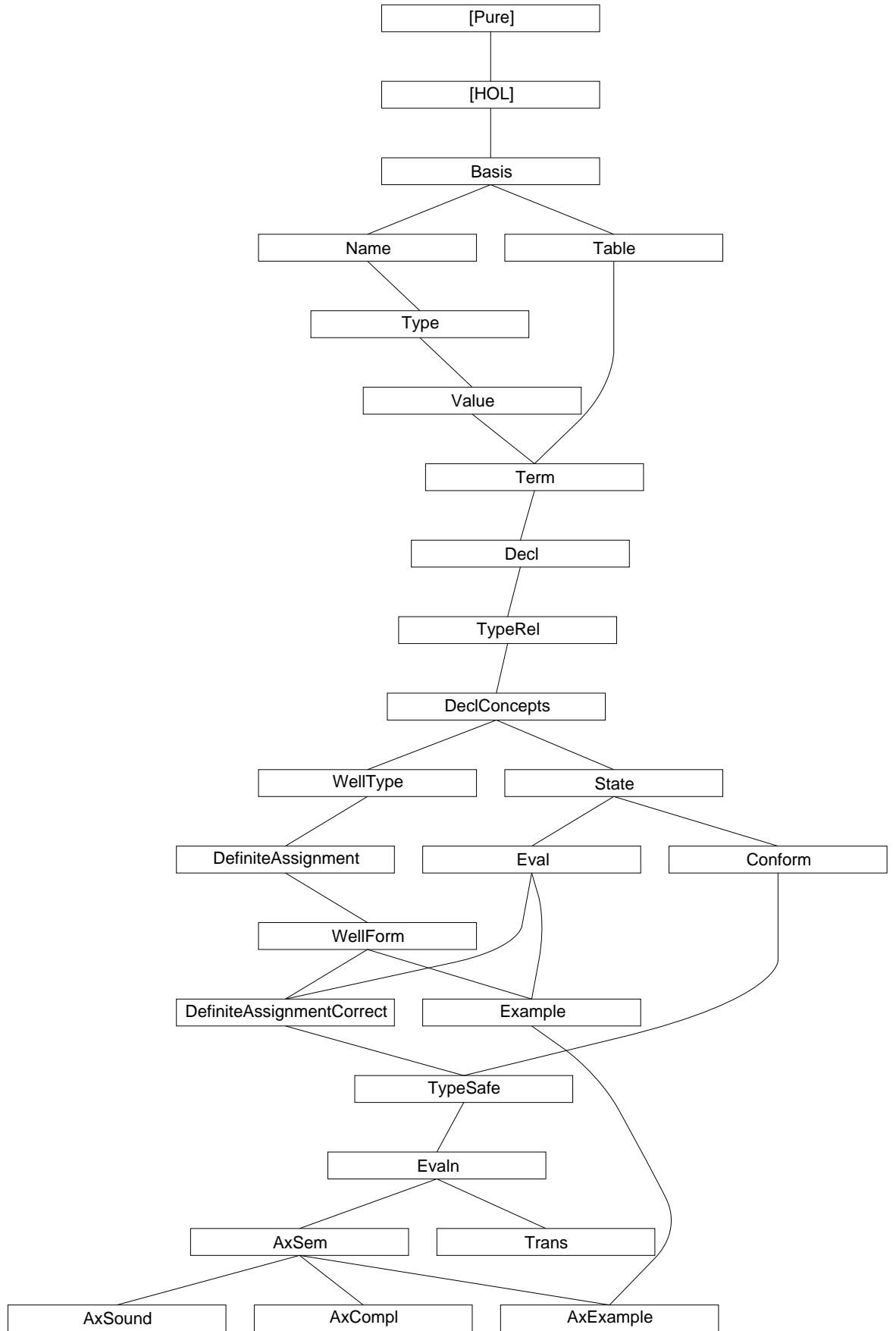
62	Soundness proof for Axiomatic semantics of Java expressions and statements	496
----	--	-----

## 24 AxCompl 543

63	Completeness proof for Axiomatic semantics of Java expressions and statements	544
----	---	-----

## 25 AxExample 571

64	Example of a proof based on the Bali axiomatic semantics . . . . .	572
----	--	-----



# 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**

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

**misc**

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

**declare** *split-if-asm*  $[split]$  *option.split*  $[split]$  *option.split-asm*  $[split]$

**declaration**  $\ll K (Simplifier.map-ss (fn ss => ss addloop (split-all-tac, split-all-tac))) \gg$

**declare** *if-weak-cong*  $[cong\ del]$  *option.weak-case-cong*  $[cong\ del]$

**declare** *length-Suc-conv*  $[iff]$

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

**apply** *auto*

**done**

**lemma** *subset-insertD*:

$A \leq insert\ x\ B \implies A \leq B \ \& \ x \sim: A \mid (EX\ B'. A = insert\ x\ B' \ \& \ B' \leq B)$

**apply**  $(case-tac\ x:A)$

**apply**  $(rule\ disjI2)$

**apply**  $(rule-tac\ x = A - \{x\}\ in\ exI)$

**apply** *fast+*

**done**

**syntax**

$3 :: nat \quad (3)$

$4 :: nat \quad (4)$

**translations**

$3 == Suc\ 2$

$4 == Suc\ 3$

**lemma** *range-bool-domain*:  $range\ f = \{f\ True, f\ False\}$

**apply** *auto*

**apply**  $(case-tac\ xa)$

**apply** *auto*

**done**

**lemma** *irrefl-tranclI'*:  $r^{\wedge}-1\ Int\ r^{\wedge}+ = \{\} \implies !x. (x, x) \sim: r^{\wedge}+$

**by**  $(blast\ elim: tranclE\ dest: trancl-into-rtrancl)$

**lemma** *trancl-rtrancl-trancl*:

$\ll (x,y) \in r^{\wedge}+; (y,z) \in r^{\wedge}* \rrbracket \implies (x,z) \in r^{\wedge}+$

**by**  $(auto\ dest: tranclD\ rtrancl-trans\ rtrancl-into-trancl2)$

**lemma** *rtrancl-into-trancl3*:

$\ll (a,b) \in r^{\wedge}*; a \neq b \rrbracket \implies (a,b) \in r^{\wedge}+$

**apply**  $(drule\ rtranclD)$

**apply** *auto*  
**done**

**lemma** *rtrancl-into-rtrancl2*:  
 $\llbracket (a, b) \in r; (b, c) \in r^* \rrbracket \implies (a, c) \in r^*$   
**by** (*auto intro: r-into-rtrancl rtrancl-trans*)

**lemma** *triangle-lemma*:  
 $\llbracket \bigwedge a b c. \llbracket (a, b) \in r; (a, c) \in r \rrbracket \implies b = c; (a, x) \in r^*; (a, y) \in r^* \rrbracket$   
 $\implies (x, y) \in r^* \vee (y, x) \in r^*$

**proof** –

**note** *converse-rtrancl-induct* = *converse-rtrancl-induct* [*consumes 1*]

**note** *converse-rtranclE* = *converse-rtranclE* [*consumes 1*]

**assume** *unique*:  $\bigwedge a b c. \llbracket (a, b) \in r; (a, c) \in r \rrbracket \implies b = c$

**assume**  $(a, x) \in r^*$

**then show**  $(a, y) \in r^* \implies (x, y) \in r^* \vee (y, x) \in r^*$

**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: r-into-rtrancl 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**

**qed**

**lemma** *rtrancl-cases* [*consumes 1, case-names Refl Trancl*]:

$\llbracket (a, b) \in r^*; a = b \implies P; (a, b) \in r^+ \implies P \rrbracket \implies P$

**apply** (*erule rtranclE*)

**apply** (*auto dest: rtrancl-into-trancl1*)

**done**

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

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

**lemma** *Ball-weaken*:  $\llbracket \text{Ball } s \ P; \bigwedge x. P \ x \longrightarrow Q \ x \rrbracket \Longrightarrow \text{Ball } s \ Q$   
**by** *auto*

**lemma** *finite-SetCompr2*:  $\llbracket \text{finite } (\text{Collect } P); !y. P \ y \longrightarrow \text{finite } (\text{range } (f \ y)) \rrbracket \Longrightarrow$   
 $\text{finite } \{f \ y \ x \mid x \ y. P \ y\}$   
**apply** (*subgoal-tac*  $\{f \ y \ x \mid x \ y. P \ y\} = \text{UNION } (\text{Collect } P) (\%y. \text{range } (f \ y))$ )  
**prefer** 2 **apply** *fast*  
**apply** (*erule ssubst*)  
**apply** (*erule finite-UN-I*)  
**apply** *fast*  
**done**

**lemma** *list-all2-trans*:  $\forall a \ b \ c. P1 \ a \ b \longrightarrow P2 \ b \ c \longrightarrow P3 \ a \ c \Longrightarrow$   
 $\forall xs2 \ xs3. \text{list-all2 } P1 \ xs1 \ xs2 \longrightarrow \text{list-all2 } P2 \ xs2 \ xs3 \longrightarrow \text{list-all2 } P3 \ xs1 \ xs3$   
**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*:  $p = (\text{fst } p, \text{fst } (\text{snd } p), \text{fst } (\text{snd } (\text{snd } p)), \text{fst } (\text{snd } (\text{snd } (\text{snd } p))),$   
 $\text{snd } (\text{snd } (\text{snd } (\text{snd } p))))$   
**apply** *auto*  
**done**

**lemma** *fst-splitE* [*elim!*]:  
 $\llbracket \text{fst } s' = x'; !x \ s. \llbracket s' = (x, s); x = x' \rrbracket \Longrightarrow Q \rrbracket \Longrightarrow Q$   
**by** (*cases s'*) *auto*

**lemma** *fst-in-set-lemma* [*rule-format (no-asm)*]:  $(x, y) : \text{set } l \longrightarrow x : \text{fst } ' \text{set } l$   
**apply** (*induct-tac l*)  
**apply** *auto*  
**done**

**quantifiers**

**lemma** *All-Ex-refl-eq2* [*simp*]:  
 $(!x. (? b. x = f \ b \ \& \ Q \ b) \longrightarrow P \ x) = (!b. Q \ b \longrightarrow P \ (f \ b))$

**apply** *auto*  
**done**

**lemma** *ex-ex-miniscope1* [*simp*]:  
 $(EX\ w\ v.\ P\ w\ v\ \&\ Q\ v) = (EX\ v.\ (EX\ w.\ P\ w\ v)\ \&\ Q\ v)$   
**apply** *auto*  
**done**

**lemma** *ex-miniscope2* [*simp*]:  
 $(EX\ v.\ P\ v\ \&\ Q\ \&\ R\ v) = (Q\ \&\ (EX\ v.\ P\ v\ \&\ R\ v))$   
**apply** *auto*  
**done**

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

**lemma** *All-Ex-refl-eq1* [*simp*]:  $(!x.\ (?\ b.\ x = f\ b) \longrightarrow P\ x) = (!b.\ P\ (f\ b))$   
**apply** *auto*  
**done**

**sums**

**hide** *const In0 In1*

**syntax**

*fun-sum* ::  $('a \Rightarrow 'c) \Rightarrow ('b \Rightarrow 'c) \Rightarrow (('a + 'b) \Rightarrow 'c)$  (**infixr**  $'(+)'80$ )

**translations**

*fun-sum* == *CONST sum-case*

**consts** *the-Inl* ::  $'a + 'b \Rightarrow 'a$

*the-Inr* ::  $'a + 'b \Rightarrow 'b$

**primrec** *the-Inl* (*Inl* *a*) = *a*

**primrec** *the-Inr* (*Inr* *b*) = *b*

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

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

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

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

**primrec** *the-In1* (*In1* *a*) = *a*

**primrec** *the-In2* (*In2* *b*) = *b*

**primrec** *the-In3* (*In3* *c*) = *c*

**syntax**

*In1l* ::  $'al \Rightarrow ('al + 'ar, 'b, 'c)\ sum3$

*In1r* ::  $'ar \Rightarrow ('al + 'ar, 'b, 'c)\ sum3$

**translations**

*In1l* *e* == *In1* (*Inl* *e*)

*In1r* *c* == *In1* (*Inr* *c*)

**syntax** *the-In1l* ::  $('al + 'ar, 'b, 'c)\ sum3 \Rightarrow 'al$

*the-In1r* ::  $('al + 'ar, 'b, 'c)\ sum3 \Rightarrow 'ar$

**translations**

*the-In1l* == *the-Inl*  $\circ$  *the-In1*

*the-In1r* == *the-Inr*  $\circ$  *the-In1*

**ML**  $\ll$   
*fun* *sum3-instantiate* *thm* = *map* (*fn* *s* => *simplify*(*simpset*()*delsimps*[@{*thm not-None-eq*}])  
 (*read-instantiate* [(*t*, *In* ^ *s* ^ ?*x*)] *thm*)) [*1l*, *2*, *3*, *1r*]  
 $\gg$

### translations

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

### quantifiers for option type

#### syntax

*Oall* :: [*pttrn*, '*a option*, *bool*] => *bool* (( $\exists!$  -::/ -) [*0*, *0*, *10*] *10*)  
*Oex* :: [*pttrn*, '*a option*, *bool*] => *bool* (( $\exists?$  -::/ -) [*0*, *0*, *10*] *10*)

#### syntax (*symbols*)

*Oall* :: [*pttrn*, '*a option*, *bool*] => *bool* (( $\exists \forall$  - $\in$ :/ -) [*0*, *0*, *10*] *10*)  
*Oex* :: [*pttrn*, '*a option*, *bool*] => *bool* (( $\exists \exists$  - $\in$ :/ -) [*0*, *0*, *10*] *10*)

### translations

! *x*:*A*: *P* == ! *x*:*o2s A*. *P*  
 ? *x*:*A*: *P* == ? *x*:*o2s A*. *P*

### Special map update

Deemed too special for theory Map.

#### constdefs

*chg-map* :: ('*b* => '*b*) => '*a* => ('*a*  $\leadsto$  => '*b*) => ('*a*  $\leadsto$  => '*b*)  
*chg-map* *f a m* == *case* *m a* *of* *None* => *m* | *Some b* => *m*(*a*|->*f b*)

**lemma** *chg-map-new*[*simp*]: *m a* = *None* ==> *chg-map f a m* = *m*  
**by** (*unfold chg-map-def*, *auto*)

**lemma** *chg-map-upd*[*simp*]: *m a* = *Some b* ==> *chg-map f a m* = *m*(*a*|->*f b*)  
**by** (*unfold chg-map-def*, *auto*)

**lemma** *chg-map-other* [*simp*]: *a*  $\neq$  *b*  $\implies$  *chg-map f a m* *b* = *m b*  
**by** (*auto simp: chg-map-def split add: option.split*)

### unique association lists

#### constdefs

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

**lemma** *uniqueD* [*rule-format* (*no-asm*)]:  
*unique l*  $\longrightarrow$  (!*x y*. (*x*,*y*):*set l*  $\longrightarrow$  (!*x' y'*. (*x'*,*y'*):*set l*  $\longrightarrow$  *x*=*x'*  $\longrightarrow$  *y*=*y'*))  
**apply** (*unfold unique-def o-def*)  
**apply** (*induct-tac l*)  
**apply** (*auto dest: fst-in-set-lemma*)  
**done**



```

lemma unique-Nil [simp]: unique []
apply (unfold unique-def)
apply (simp (no-asm))
done

```

```

lemma unique-Cons [simp]: unique ((x,y)#l) = (unique l & (!y. (x,y) ~: set l))
apply (unfold unique-def)
apply (auto dest: fst-in-set-lemma)
done

```

```

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

```

```

lemma unique-single [simp]: !!p. unique [p]
apply auto
done

```

```

lemma unique-ConsD: unique (x#xs) ==> unique xs
apply (simp add: unique-def)
done

```

```

lemma unique-append [rule-format (no-asm)]: unique l' ==> unique l -->
  (! (x,y):set l. ! (x',y'):set l'. x' ~ = x) --> unique (l @ l')
apply (induct-tac l)
apply (auto dest: fst-in-set-lemma)
done

```

```

lemma unique-map-inj [rule-format (no-asm)]: unique l --> inj f --> unique (map (%(k,x). (f k, g k
x)) l)
apply (induct-tac l)
apply (auto dest: fst-in-set-lemma simp add: inj-eq)
done

```

```

lemma map-of-SomeI [rule-format (no-asm)]: unique l --> (k, x) : set l --> map-of l k = Some x
apply (induct-tac l)
apply auto
done

```

## list patterns

```

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

```

```

syntax
  -lpttrn    :: [pttrn,pttrn] => pttrn    (-#/- [901,900] 900)

```

```

translations
  %y#x#xs. b == lsplit (%y x#xs. b)
  %x#xs . b == lsplit (%x xs . b)

```

```

lemma lsplit [simp]: lsplit c (x#xs) = c x xs
apply (unfold lsplit-def)
apply (simp (no-asm))
done

```

```

lemma lsplit2 [simp]: lsplit P (x#xs) y z = P x xs y z
apply (unfold lsplit-def)
apply simp
done

```

```

end

```

## Chapter 3

## Table

## 2 Abstract tables and their implementation as lists

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

design issues:

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

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

**map of / table of**

**syntax**

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

**translations**

*table-of* == *map-of*

$(\text{type})'a \multimap 'b \leq (\text{type})'a \Rightarrow 'b \text{ Datatype.option}$   
 $(\text{type})('a, 'b) \text{ table} \leq (\text{type})'a \multimap 'b$

**lemma** *map-add-find-left[simp]*:

$n \ k = \text{None} \implies (m \ ++ \ n) \ k = m \ k$

**by** (*simp add: map-add-def*)

### Conditional Override

**constdefs**

*cond-override*::

$('b \Rightarrow 'b \Rightarrow \text{bool}) \Rightarrow ('a, 'b) \text{ table} \Rightarrow ('a, 'b) \text{ table} \Rightarrow ('a, 'b) \text{ table}$

— when merging tables old and new, only override an entry of table old when the condition cond holds

*cond-override cond old new*  $\equiv$

$\lambda k.$

(*case new k of*  
   *None*  $\Rightarrow$  *old k*  
   | *Some new-val*  $\Rightarrow$  (*case old k of*  
     *None*  $\Rightarrow$  *Some new-val*  
     | *Some old-val*  $\Rightarrow$  (*if cond new-val old-val*  
       *then Some new-val*  
       else *Some old-val*)))

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

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

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

**lemma** *cond-override-override*:  
 $\llbracket \text{old } k = \text{Some } ov; \text{new } k = \text{Some } nv; C \text{ nv } ov \rrbracket$   
 $\implies (\text{cond-override } C \text{ old new}) k = \text{Some } nv$   
**by** (*auto simp add: cond-override-def*)

**lemma** *cond-override-noOverride*:  
 $\llbracket \text{old } k = \text{Some } ov; \text{new } k = \text{Some } nv; \neg (C \text{ nv } ov) \rrbracket$   
 $\implies (\text{cond-override } C \text{ old new}) k = \text{Some } ov$   
**by** (*auto simp add: cond-override-def*)

**lemma** *dom-cond-override*: *dom (cond-override C s t)  $\subseteq$  dom s  $\cup$  dom t*  
**by** (*auto simp add: cond-override-def dom-def*)

**lemma** *finite-dom-cond-override*:  
 $\llbracket \text{finite (dom } s); \text{finite (dom } t) \rrbracket \implies \text{finite (dom (cond-override } C \text{ s t))}$   
**apply** (*rule-tac B=dom s  $\cup$  dom t in finite-subset*)  
**apply** (*rule dom-cond-override*)  
**by** (*rule finite-UnI*)

## Filter on Tables

**constdefs**  
*filter-tab*:: ('a  $\Rightarrow$  'b  $\Rightarrow$  bool)  $\Rightarrow$  ('a, 'b) table  $\Rightarrow$  ('a, 'b) table  
*filter-tab c t*  $\equiv \lambda k. (\text{case } t \text{ k of}$   
 $\quad \text{None} \Rightarrow \text{None}$   
 $\quad | \text{Some } x \Rightarrow \text{if } c \text{ k } x \text{ then Some } x \text{ else None})$

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

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

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

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

**by** (*auto simp add: filter-tab-def ran-def*)

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

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

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

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

**lemma** *cond-override-filter*:

$\llbracket \bigwedge k \text{ old new}. \llbracket s k = \text{Some new}; t k = \text{Some old} \rrbracket \implies (\neg \text{overC new old} \longrightarrow \neg \text{filterC } k \text{ new}) \wedge (\text{overC new old} \longrightarrow \text{filterC } k \text{ old} \longrightarrow \text{filterC } k \text{ new}) \rrbracket \implies$   
 $\text{cond-override overC (filter-tab filterC } t) (\text{filter-tab filterC } s)$   
 $= \text{filter-tab filterC (cond-override overC } t \text{ } s)$   
**by** (*auto simp add: expand-fun-eq 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*:

$\llbracket (\forall (x,y) \in \text{set } l. P x y); x \in \text{o2s (table-of } l \ x) \rrbracket \implies P x x$   
**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*:

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

**apply** *auto*  
**done**

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

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

**lemma** *table-of-map-SomeI* [*rule-format* (*no-asm*)]:  $\text{table-of } t \text{ } k = \text{Some } x \longrightarrow$   
 $\text{table-of } (\text{map } (\lambda(k,x). (k, f \text{ } x)) \text{ } t) \text{ } k = \text{Some } (f \text{ } x)$   
**apply** (*induct-tac* *t*)  
**apply** *auto*  
**done**

**lemma** *table-of-remap-SomeD* [*rule-format* (*no-asm*)]:  
 $\text{table-of } (\text{map } (\lambda((k,k'),x). (k,(k',x))) \text{ } t) \text{ } k = \text{Some } (k',x) \longrightarrow$   
 $\text{table-of } t \text{ } (k, k') = \text{Some } x$   
**apply** (*induct-tac* *t*)  
**apply** *auto*  
**done**

**lemma** *table-of-mapf-Some* [*rule-format* (*no-asm*)]:  $\forall x \text{ } y. f \text{ } x = f \text{ } y \longrightarrow x = y \implies$   
 $\text{table-of } (\text{map } (\lambda(k,x). (k,f \text{ } x)) \text{ } t) \text{ } k = \text{Some } (f \text{ } x) \longrightarrow \text{table-of } t \text{ } k = \text{Some } x$   
**apply** (*induct-tac* *t*)  
**apply** *auto*  
**done**

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

**lemma** *table-of-mapf-NoneD* [*rule-format* (*no-asm*), *dest!*]:  
 $\text{table-of } (\text{map } (\lambda(k,x). (k, f \text{ } x)) \text{ } t) \text{ } k = \text{None} \longrightarrow (\text{table-of } t \text{ } k = \text{None})$   
**apply** (*induct-tac* *t*)  
**apply** *auto*  
**done**

**lemma** *table-of-mapkey-SomeD* [*rule-format* (*no-asm*), *dest!*]:  
 $\text{table-of } (\text{map } (\lambda(k,x). ((k,C),x)) \text{ } t) \text{ } (k,D) = \text{Some } x \longrightarrow C = D \wedge \text{table-of } t \text{ } k = \text{Some } x$   
**apply** (*induct-tac* *t*)  
**apply** *auto*  
**done**

**lemma** *table-of-mapkey-SomeD2* [*rule-format* (*no-asm*), *dest!*]:  
 $\text{table-of } (\text{map } (\lambda(k,x). ((k,C),x)) \text{ } t) \text{ } ek = \text{Some } x$   
 $\longrightarrow C = \text{snd } ek \wedge \text{table-of } t \text{ } (\text{fst } ek) = \text{Some } x$



```
apply (induct-tac t)
apply auto
done
```

```

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  $\implies$  unique xs  $\longrightarrow$  table-of xs k = Some z
apply (induct xs)
apply (auto del: map-of-SomeD intro!: map-of-SomeD)
done

```

**consts**
$$\begin{array}{ll}
\textit{Un-tables} & :: ('a, 'b) \textit{ tables set} \Rightarrow ('a, 'b) \textit{ tables} \\
\textit{overrides-t} & :: ('a, 'b) \textit{ tables} \Rightarrow ('a, 'b) \textit{ tables} \Rightarrow \\
& \quad ('a, 'b) \textit{ tables} \quad (\textbf{infixl } \oplus \oplus 100) \\
\textit{hidings-entails} & :: ('a, 'b) \textit{ tables} \Rightarrow ('a, 'c) \textit{ tables} \Rightarrow \\
& \quad ('b \Rightarrow 'c \Rightarrow \textit{bool}) \Rightarrow \textit{bool} \quad (- \textit{ hidings - entails - 20}) \\
\text{--- variant for unique table:} & \\
\textit{hiding-entails} & :: ('a, 'b) \textit{ table} \Rightarrow ('a, 'c) \textit{ table} \Rightarrow \\
& \quad ('b \Rightarrow 'c \Rightarrow \textit{bool}) \Rightarrow \textit{bool} \quad (- \textit{ hiding - entails - 20}) \\
\text{--- variant for a unique table and conditional overriding:} & \\
\textit{cond-hiding-entails} & :: ('a, 'b) \textit{ table} \Rightarrow ('a, 'c) \textit{ table} \\
& \quad \Rightarrow ('b \Rightarrow 'c \Rightarrow \textit{bool}) \Rightarrow ('b \Rightarrow 'c \Rightarrow \textit{bool}) \Rightarrow \textit{bool} \\
& \quad (- \textit{ hiding - under - entails - 20})
\end{array}$$

## overrides

**lemma** *empty-overrides-t* [simp]:  $(\lambda k. \{\}) \oplus \oplus m = m$   
**apply** (*unfold overrides-t-def*)  
**apply** (*simp (no-asm)*)  
**done**

**lemma** *overrides-empty-t* [simp]:  $m \oplus \oplus (\lambda k. \{\}) = m$   
**apply** (*unfold overrides-t-def*)  
**apply** (*simp (no-asm)*)  
**done**

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

**lemma** *empty-hiding-entails*: *empty hiding s entails R*  
**by** (*simp add: hiding-entails-def*)

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

## 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]: *empty hiding s under C entails R*  
**by** (*simp add: cond-hiding-entails-def*)

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

```

```

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

```

```

consts
  atleast-free ::  $('a \rightsquigarrow 'b) \Rightarrow \text{nat} \Rightarrow \text{bool}$ 
primrec
  atleast-free  $m$   $0$  = True
  atleast-free-Suc:
  atleast-free  $m$  (Suc  $n$ ) =  $(? a. m\ a = \text{None} \ \& \ (!b. \text{atleast-free } (m(a|-\>b))\ n))$ 

```

```

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

```

```

declare fun-upd-apply [simp del]

```

```

lemma atleast-free-SucD-lemma [rule-format (no-asm)]:
   $!m\ a. m\ a = \text{None} \longrightarrow (!c. \text{atleast-free } (m(a|-\>c))\ n) \longrightarrow$ 
   $(!b\ d. a \rightsquigarrow b \longrightarrow \text{atleast-free } (m(b|-\>d))\ n)$ 
apply (induct-tac n)
apply auto
apply (rule-tac x = a in exI)
apply (rule conjI)
apply (force simp add: fun-upd-apply)
apply (erule-tac V = m a = None in thin-rl)
apply clarify
apply (subst fun-upd-twist)
apply (erule not-sym)
apply (rename-tac ba)
apply (drule-tac x = ba in spec)
apply clarify
apply (tactic simp-tac 2 1)
apply (erule (1) notE impE)
apply (case-tac aa = b)
apply fast+

```

**done**

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

**lemma** *atleast-free-SucD* [*rule-format (no-asm)*]: *atleast-free* *h* (*Suc* *n*)  $\implies$  *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

### 3 Java names

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

**typeddecl** *tnam* — 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*

**syntax**

*VName* :: *vname*  $\Rightarrow$  *lname*

*Result* :: *lname*

**translations**

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

*Result* == *ENAME* *Res*

**datatype** *xname* — names of standard exceptions

= *Throwable*

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

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

**lemma** *xn-cases*:

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

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

*xn* = *NegArrSize*  $\vee$  *xn* = *IndOutBound*  $\vee$  *xn* = *ArrStore*

**apply** (*induct xn*)

**apply** *auto*

**done**

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

= *Object'*

| *SXcpt'* *xname*

| *TName* *tnam*

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

*pid* :: *pname*

*tid* :: *tname*

**axclass** *has-pname* < *type*

**consts** *pname*::*a*::*has-pname*  $\Rightarrow$  *pname*

**instance** *pname*::*has-pname* ..

**defs** (**overloaded**)

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

**axclass** *has-tname* < *type*

**consts**  $tname :: 'a :: has-tname \Rightarrow tname$

**instance**  $tname :: has-tname ..$

**defs** (overloaded)

$tname-tname-def: tname (t :: tname) \equiv t$

**axclass**  $has-qtname < type$

**consts**  $qtname :: 'a :: has-qtname \Rightarrow qtname$

**instance**  $qtname-ext-type :: (type) has-qtname ..$

**defs** (overloaded)

$qtname-qtname-def: qtname (q :: qtname) \equiv q$

**translations**

$mname <= Name.mname$

$xname <= Name.xname$

$tname <= Name.tname$

$ename <= Name.ename$

$qtname <= (type) (\lambda pid :: pname, tid :: tname) \langle \rangle$

$(type) 'a \ qtname-scheme <= (type) (\lambda pid :: pname, tid :: tname, \dots :: 'a) \langle \rangle$

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

**consts**

$Object :: qtname$

$SXcpt :: xname \Rightarrow qtname$

**defs**

$Object-def: Object \equiv (\lambda pid = java-lang, tid = Object) \langle \rangle$

$SXcpt-def: SXcpt \equiv \lambda x. (\lambda pid = java-lang, tid = SXcpt' x) \langle \rangle$

**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

## 4 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

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

**consts** *the-Bool* :: *val*  $\Rightarrow$  *bool*

**primrec** *the-Bool* (*Bool b*) = *b*

**consts** *the-Intg* :: *val*  $\Rightarrow$  *int*

**primrec** *the-Intg* (*Intg i*) = *i*

**consts** *the-Addr* :: *val*  $\Rightarrow$  *loc*

**primrec** *the-Addr* (*Addr a*) = *a*

**types** *dyn-ty* = *loc*  $\Rightarrow$  *ty option*

**consts**

*typeof* :: *dyn-ty*  $\Rightarrow$  *val*  $\Rightarrow$  *ty option*

*defpval* :: *prim-ty*  $\Rightarrow$  *val* — default value for primitive types

*default-val* :: *ty*  $\Rightarrow$  *val* — default value for all types

**primrec** *typeof dt Unit* = *Some (PrimT Void)*

*typeof dt (Bool b)* = *Some (PrimT Boolean)*

*typeof dt (Intg i)* = *Some (PrimT Integer)*

*typeof dt Null* = *Some NT*

*typeof dt (Addr a)* = *dt a*

**primrec** *defpval Void* = *Unit*

*defpval Boolean* = *Bool False*

*defpval Integer* = *Intg 0*

**primrec** *default-val (PrimT pt)* = *defpval pt*

*default-val (RefT r)* = *Null*

**end**

## Chapter 6

# Type

## 5 Java types

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

simplifications:

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

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

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

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

**translations**

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

**syntax**

*NT* :: *ty*  
*Iface* :: *qtname*  $\Rightarrow$  *ty*  
*Class* :: *qtname*  $\Rightarrow$  *ty*  
*Array* :: *ty*  $\Rightarrow$  *ty* ( $\cdot$ .[] [90] 90)

**translations**

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

**constdefs**

*the-Class* :: *ty*  $\Rightarrow$  *qtname*  
*the-Class T*  $\equiv$  *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

## 6 Java expressions and statements

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

design issues:

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

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

**datatype** *jump*  
= *Break label* — break

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

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

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

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

**types**  
*abopt* = *abrupt option*

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

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

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

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

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

— function codes for unary operations

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

— function codes for binary operations

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

```

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

```

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

#### **datatype** *var*

```

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

```

#### **and** *expr*

```

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

| Super — special Super keyword
| Acc var — variable access
| Ass var expr (-:= - [90,85 ]85)
    — variable assign
| Cond expr expr expr (- ? - : - [85,85,80]80) — conditional
| Call qname ref-ty inv-mode expr mname (ty list) (expr list)
    ({-, -, -}---'({-}'-')[10,10,10,85,99,10,10]85)
    — method call
    — {accC, statT, mode}e..mn( {pTs}args) "
    — accC: accessing class (static class were
    — the call code is declared. Annotation only needed for
    — evaluation to check accessibility)
    — statT: static declaration class/interface of
    — method
    — mode: invocation mode
    — e: reference to object

```



— *mn*: field name  
 — *pTs*: types of parameters  
 — *args*: the actual parameters/arguments  
 | *Method qname sig* — (folded) method (see below)  
 | *Body qname stmt* — (unfolded) method body  
 | *InsInitE stmt expr*  
   — insertion of initialization before  
   — evaluation of *expr* (technical term for smallstep sem.)  
 | *Callee locals expr* — save callers locals in callee-Frame  
   — (technical term for smallstep semantics)  
**and** *stmt*  
 = *Skip* — empty statement  
 | *Expr expr* — expression statement  
 | *Lab jump stmt* ( $\rightarrow$  - [ 99,66]66)  
   — labeled statement; handles break  
 | *Comp stmt stmt* ( $;$  - [ 66,65]65)  
 | *If' expr stmt stmt* (*If'*(-) - *Else* - [ 80,79,79]70)  
 | *Loop label expr stmt* ( $\rightarrow$  *While'*(-) - [ 99,80,79]70)  
 | *Jump jump* — break, continue, return  
 | *Throw expr*  
 | *TryC stmt qname vname stmt* (*Try* - *Catch'*(- -) - [79,99,80,79]70)  
   — *Try c1 Catch(C vn) c2*  
   — *c1*: block where exception may be thrown  
   — *C*: exception class to catch  
   — *vn*: local name for exception used in *c2*  
   — *c2*: block to execute when exception is caught  
 | *Fin stmt stmt* (*Finally* - [ 79,79]70)  
 | *FinA abrupt stmt* — Save abrupt of first statement  
   — technical term for smallstep sem.)  
 | *Init qname* — class initialization

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 semantics 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.

**types** *term* = (*expr+stmt,var,expr list*) *sum3*

**translations**

*sig* <= (*type*) *mname*  $\times$  *ty list*  
*var* <= (*type*) *Term.var*  
*expr* <= (*type*) *Term.expr*  
*stmt* <= (*type*) *Term.stmt*  
*term* <= (*type*) (*expr+stmt,var,expr list*) *sum3*

**syntax**

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

**translations**

```

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

```

### constdefs

```

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

```

**ML**  $\ll \text{bind-thms } (\text{is-stmt-rews}, \text{sum3-instantiate } @\{\text{thm is-stmt-def}\}) \gg$

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

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

### syntax

```

expr-inj-term:: expr ⇒ term (⟨-⟩e 1000)
stmt-inj-term:: stmt ⇒ term (⟨-⟩s 1000)
var-inj-term:: var ⇒ term (⟨-⟩v 1000)
lst-inj-term:: expr list ⇒ term (⟨-⟩l 1000)

```

### translations

```

⟨e⟩e ↦ In1l e
⟨c⟩s ↦ In1r c
⟨v⟩v ↦ In2 v
⟨es⟩l ↦ In3 es

```

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

```

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

```

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

**instance** *stmt::inj-term ..*

### defs (overloaded)

```

stmt-inj-term-def: ⟨c::stmt⟩ ≡ In1r c

```

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

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

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

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

**instance** *expr::inj-term ..*

### defs (overloaded)

```

expr-inj-term-def: ⟨e::expr⟩ ≡ In1l e

```

**lemma** *expr-inj-term-simp*:  $\langle e::\text{expr} \rangle = \text{In1 } e$   
**by** (*simp add: expr-inj-term-def*)

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

**instance** *var::inj-term ..*

**defs** (**overloaded**)  
*var-inj-term-def*:  $\langle v::\text{var} \rangle \equiv \text{In2 } v$

**lemma** *var-inj-term-simp*:  $\langle v::\text{var} \rangle = \text{In2 } v$   
**by** (*simp add: var-inj-term-def*)

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

**instance** *list::(type) inj-term ..*

**defs** (**overloaded**)  
*expr-list-inj-term-def*:  $\langle es::\text{expr list} \rangle \equiv \text{In3 } es$

**lemma** *expr-list-inj-term-simp*:  $\langle es::\text{expr list} \rangle = \text{In3 } es$   
**by** (*simp add: expr-list-inj-term-def*)

**lemma** *expr-list-inj-term [iff]*:  $\langle x::\text{expr list} \rangle = \langle y \rangle \equiv x = y$   
**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]*:  $\langle t::\text{stmt} \rangle \neq \langle w::\text{expr} \rangle$   
**by** (*simp add: inj-term-simps*)

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

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

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

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

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

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

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

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

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

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

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

**lemma** *elist-var-inj-term* [iff]:  $\langle t::\text{expr list} \rangle \neq \langle w::\text{var} \rangle$   
**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** (*case-tac a*)  
**apply** *auto*  
**done**

## Evaluation of unary operations

**consts** *eval-unop* :: *unop*  $\Rightarrow$  *val*  $\Rightarrow$  *val*  
**primrec**  
*eval-unop UPlus*  $v = \text{Intg } (\text{the-Intg } v)$   
*eval-unop UMinus*  $v = \text{Intg } (- (\text{the-Intg } v))$   
*eval-unop UBitNot*  $v = \text{Intg } 42$  — FIXME: Not yet implemented  
*eval-unop UNot*  $v = \text{Bool } (\neg \text{the-Bool } v)$

## Evaluation of binary operations

**consts** *eval-binop* :: *binop*  $\Rightarrow$  *val*  $\Rightarrow$  *val*  $\Rightarrow$  *val*  
**primrec**  
*eval-binop Mul*  $v1\ v2 = \text{Intg } ((\text{the-Intg } v1) * (\text{the-Intg } v2))$   
*eval-binop Div*  $v1\ v2 = \text{Intg } ((\text{the-Intg } v1) \text{ div } (\text{the-Intg } v2))$   
*eval-binop Mod*  $v1\ v2 = \text{Intg } ((\text{the-Intg } v1) \text{ mod } (\text{the-Intg } v2))$   
*eval-binop Plus*  $v1\ v2 = \text{Intg } ((\text{the-Intg } v1) + (\text{the-Intg } v2))$   
*eval-binop Minus*  $v1\ v2 = \text{Intg } ((\text{the-Intg } v1) - (\text{the-Intg } v2))$   
  
— Be aware of the explicit coercion of the shift distance to nat  
*eval-binop LShift*  $v1\ v2 = \text{Intg } ((\text{the-Intg } v1) * (2^{(\text{nat } (\text{the-Intg } v2))}))$   
*eval-binop RShift*  $v1\ v2 = \text{Intg } ((\text{the-Intg } v1) \text{ div } (2^{(\text{nat } (\text{the-Intg } v2))}))$   
*eval-binop RShiftU*  $v1\ v2 = \text{Intg } 42$  — FIXME: Not yet implemented  
  
*eval-binop Less*  $v1\ v2 = \text{Bool } ((\text{the-Intg } v1) < (\text{the-Intg } v2))$   
*eval-binop Le*  $v1\ v2 = \text{Bool } ((\text{the-Intg } v1) \leq (\text{the-Intg } v2))$   
*eval-binop Greater*  $v1\ v2 = \text{Bool } ((\text{the-Intg } v2) < (\text{the-Intg } v1))$   
*eval-binop Ge*  $v1\ v2 = \text{Bool } ((\text{the-Intg } v2) \leq (\text{the-Intg } v1))$   
  
*eval-binop Eq*  $v1\ v2 = \text{Bool } (v1=v2)$

```

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

```

```

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

```

*CondAnd* and *CondOr* only 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) = (¬ b)
by (simp add: need-second-arg-def)

```

```

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

```



## Chapter 8

### Decl

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

**theory** *Decl*  
**imports** *Term Table*  
**begin**

improvements:

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

simplifications:

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

## 8 Modifier

### Access modifier

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

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

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

#### definition

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

#### definition

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

#### instance proof

**fix**  $x\ y\ z::acc-modi$   
**{**  
**show**  $x \leq x$  — reflexivity



```

  by (auto simp add: le-acc-def)
next
assume  $x \leq y$   $y \leq z$  — transitivity
thus  $x \leq z$ 
  by (auto simp add: le-acc-def less-acc-def split add: acc-modi.split)
next
assume  $x \leq y$   $y \leq x$  — antisymmetry
thus  $x = y$ 
proof -
  have  $\forall x y. x < (y::acc-modi) \wedge y < x \longrightarrow False$ 
  by (auto simp add: less-acc-def split add: acc-modi.split)
  with prems show ?thesis
  by (unfold le-acc-def) iprover
qed
next
show  $(x < y) = (x \leq y \wedge x \neq y)$ 
  by (auto simp add: le-acc-def less-acc-def split add: acc-modi.split)
}
fix  $x y:: acc-modi$ 
show  $x \leq y \vee y \leq x$ 
  by (auto simp add: less-acc-def le-acc-def split add: acc-modi.split)
qed

end

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

lemma acc-modi-le-Package:
 $a \leq Package \implies a=Private \vee a = Package$ 
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*  
 [consumes 1, case-names *Package Protected Public*]:  
 $Package \leq m \implies (m = Package \implies P\ m) \implies (m = Protected \implies P\ m) \implies$   
 $(m = Public \implies P\ m) \implies P\ m$   
**by** (auto dest: *acc-modi-Package-le*)

## Static Modifier

**types** *stat-modi* = bool

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

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

**translations**  
 $decl \leq (type) \ (\downarrow access :: acc-modi)$   
 $decl \leq (type) \ (\downarrow access :: acc-modi, \dots :: 'a)$

## 10 Member (field or method)

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

**translations**  
 $member \leq (type) \ (\downarrow access :: acc-modi, static :: bool)$   
 $member \leq (type) \ (\downarrow access :: acc-modi, static :: bool, \dots :: 'a)$

## 11 Field

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

**translations**  
 $field \leq (type) \ (\downarrow access :: acc-modi, static :: bool, type :: ty)$   
 $field \leq (type) \ (\downarrow access :: acc-modi, static :: bool, type :: ty, \dots :: 'a)$

**types**  
 $fdecl$   
 $= vname \times field$

**translations**  
 $fdecl \leq (type) \ vname \times field$

## 12 Method

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

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

```
record methd = mhead +
  mbody :: mbody
```

```
types mdecl = sig × methd
```

### translations

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

### constdefs

```
mhead::methd ⇒ mhead
mhead m ≡ (|access=access m, static=static m, pars=pars m, resT=resT m|)
```

```
lemma access-mhead [simp]:access (mhead m) = access m
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
```

```
axclass has-memberid < type
```

```
consts
```

```
memberid :: 'a::has-memberid ⇒ memberid
```

**instance** *memberdecl::has-memberid ..*

**defs** (overloaded)

*memberdecl-memberid-def*:

*memberid m*  $\equiv$  (case *m* of  
     *fdecl (vn,f)*  $\Rightarrow$  *fid vn*  
   | *mdecl (sig,m)*  $\Rightarrow$  *mid sig*)

**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*)

**instance** \* :: (*type, has-memberid*) *has-memberid ..*

**defs** (overloaded)

*pair-memberid-def*:

*memberid p*  $\equiv$  *memberid (snd p)*

**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*)

**constdefs** *is-field* :: *qtname*  $\times$  *memberdecl*  $\Rightarrow$  *bool*  
*is-field m*  $\equiv \exists$  *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*)

**constdefs** *is-method* :: *qtname*  $\times$  *memberdecl*  $\Rightarrow$  *bool*  
*is-method membr*  $\equiv \exists$  *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)

### 13 Interface

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

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

**types**  
*idecl* — interface declaration, cf. 9.1  
= qname × *iface*

#### translations

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

#### constdefs

*ibody* :: *iface* ⇒ *ibody*  
*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)

### 14 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

**types**  
*cdecl* — class declaration, cf. 8.1  
= qname × *class*

#### translations

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

constdefs

$$cbody :: class \Rightarrow cbody$$
$$cbody\ c \equiv (\text{access} = \text{access}\ c, \text{cfields} = \text{cfields}\ c, \text{methods} = \text{methods}\ c, \text{init} = \text{init}\ c)$$

**lemma** *access-cbody*  $[simp]: \text{access } (\text{cbody } c) = \text{access } c$   
**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*

*ObjectC* :: *cdecl* — declaration of root class

$$SX_{cpt}C :: xname \Rightarrow cdecl \quad \text{— declarations of throwable classes}$$

defs

$$\begin{aligned} \text{ObjectC-def: ObjectC} &\equiv (\text{Object}, (\text{access}=\text{Public}, \text{cfields}=[], \text{methods}=\text{Object-mdecls}, \\ &\quad \text{init}=\text{Skip}, \text{super}=\text{arbitrary}, \text{superIfs}=[])) \\ \text{SXcptC-def: SXcptC } xn &\equiv (\text{SXcpt } xn, (\text{access}=\text{Public}, \text{cfields}=[], \text{methods}=\text{SXcpt-mdecls}, \\ &\quad \text{init}=\text{Skip}, \\ &\quad \text{super}=\text{if } xn = \text{Throwable then Object} \\ &\quad \quad \quad \text{else SXcpt Throwable}, \\ &\quad \text{superIfs}=[])) \end{aligned}$$

**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*)

```
constdefs standard-classes :: cdecl list
```

$$\textit{standard-classes} \equiv [\textit{ObjectC}, \textit{SXcptC Throwable}, \\ \textit{SXcptC NullPointer}, \textit{SXcptC OutOfMemory}, \textit{SXcptC ClassCast}, \\ \textit{SXcptC NegArrSize}, \textit{SXcptC IndOutBound}, \textit{SXcptC ArrStore}]$$

programs

```
record prog =
    ifaces :: idecl list
```

*classes::cdecl list*

### translations

*prog* ≤ (*type*) (|*ifaces::idecl list*, *classes::cdecl list*)  
*prog* ≤ (*type*) (|*ifaces::idecl list*, *classes::cdecl list*, ...::'a)

### syntax

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

### translations

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

### is type

#### consts

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

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

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

**lemma** *type-is-iface*: *is-type* *G* (*Iface* *I*) ⇒ *is-iface* *G* *I*  
**by** *auto*

**lemma** *type-is-class*: *is-type* *G* (*Class* *C*) ⇒ *is-class* *G* *C*  
**by** *auto*

### subinterface and subclass relation, in anticipation of TypeRel.thy

#### consts

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

#### defs

*subint1-def*: *subint1* *G* ≡ {(*I*, *J*). ∃ *i* ∈ *iface* *G* *I*: *J* ∈ *set* (*isuperIfs* *i*)}

*subcls1-def*: *subcls1* *G* ≡ {(*C*, *D*). *C* ≠ *Object* ∧ (∃ *c* ∈ *class* *G* *C*: *super* *c* = *D*)}

#### syntax

-*subcls1* :: *prog* ⇒ [*qname*, *qname*] ⇒ *bool* (|-<: *C1*- [71,71,71] 70)  
-*subclsseq*:: *prog* ⇒ [*qname*, *qname*] ⇒ *bool* (|-<=: *C* - [71,71,71] 70)  
-*subcls* :: *prog* ⇒ [*qname*, *qname*] ⇒ *bool* (|-<: *C* - [71,71,71] 70)

#### syntax (*xsymbols*)

-*subcls1* :: *prog* ⇒ [*qname*, *qname*] ⇒ *bool* (+-<<sub>*C1*</sub>- [71,71,71] 70)  
-*subclsseq*:: *prog* ⇒ [*qname*, *qname*] ⇒ *bool* (+-<=<sub>*C*</sub> - [71,71,71] 70)  
-*subcls* :: *prog* ⇒ [*qname*, *qname*] ⇒ *bool* (+-<<sub>*C*</sub> - [71,71,71] 70)

**translations**

$$\begin{aligned}
G \vdash C \prec_{C1} D &== (C, D) \in \text{subcls1 } G \\
G \vdash C \preceq_C D &== (C, D) \in (\text{subcls1 } G)^* \\
G \vdash C \prec_C D &== (C, D) \in (\text{subcls1 } G)^+
\end{aligned}$$

**lemma** *subint1I*:  $\llbracket \text{iface } G \ I = \text{Some } i; J \in \text{set } (\text{isuperIfs } i) \rrbracket$   
 $\implies (I, J) \in \text{subint1 } G$   
**apply** (*simp add: subint1-def*)  
**done**

**lemma** *subcls1I*:  $\llbracket \text{class } G \ C = \text{Some } c; C \neq \text{Object} \rrbracket \implies (C, (\text{super } c)) \in \text{subcls1 } G$   
**apply** (*simp add: subcls1-def*)  
**done**

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

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

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

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

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

**lemma** *no-subcls1-Object*:  $G \vdash \text{Object} \prec_{C1} D \implies P$   
**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****constdefs**

$ws\_idecl :: prog \Rightarrow qname \Rightarrow qname\ list \Rightarrow bool$   
 $ws\_idecl\ G\ I\ si \equiv \forall J \in set\ si. \ is\_iface\ G\ J \ \wedge \ (J, I) \notin (subint1\ G)^+ +$   
  
 $ws\_cdecl :: prog \Rightarrow qname \Rightarrow qname \Rightarrow bool$   
 $ws\_cdecl\ G\ C\ sc \equiv C \neq Object \longrightarrow is\_class\ G\ sc \wedge (sc, C) \notin (subcls1\ G)^+ +$   
  
 $ws\_prog :: prog \Rightarrow bool$   
 $ws\_prog\ G \equiv (\forall (I, i) \in set\ (ifaces\ G). \ ws\_idecl\ G\ I\ (isuperIfs\ i)) \wedge$   
 $(\forall (C, c) \in set\ (classes\ G). \ ws\_cdecl\ G\ C\ (super\ c))$

**lemma** *ws-progI*:

$\llbracket \forall (I, i) \in set\ (ifaces\ G). \ \forall J \in set\ (isuperIfs\ i). \ is\_iface\ G\ J \wedge$   
 $(J, I) \notin (subint1\ G)^+ +;$   
 $\forall (C, c) \in set\ (classes\ G). \ C \neq Object \longrightarrow is\_class\ G\ (super\ c) \wedge$   
 $((super\ c), C) \notin (subcls1\ G)^+ +$   
 $\rrbracket \implies ws\_prog\ G$   
**apply** (*unfold ws-prog-def ws-idecl-def ws-cdecl-def*)  
**apply** (*erule-tac conjI*)  
**apply** *blast*  
**done**

**lemma** *ws-prog-ideclD*:

$\llbracket iface\ G\ I = Some\ i; J \in set\ (isuperIfs\ i); ws\_prog\ G \rrbracket \implies$   
 $is\_iface\ G\ J \wedge (J, I) \notin (subint1\ G)^+ +$   
**apply** (*unfold ws-prog-def ws-idecl-def*)  
**apply** *clarify*  
**apply** (*drule-tac map-of-SomeD*)  
**apply** *auto*  
**done**

**lemma** *ws-prog-cdeclD*:

$\llbracket class\ G\ C = Some\ c; C \neq Object; ws\_prog\ G \rrbracket \implies$   
 $is\_class\ G\ (super\ c) \wedge (super\ c, C) \notin (subcls1\ G)^+ +$   
**apply** (*unfold ws-prog-def ws-cdecl-def*)  
**apply** *clarify*  
**apply** (*drule-tac map-of-SomeD*)  
**apply** *auto*  
**done**

**well-foundedness****lemma** *finite-is-iface*: *finite*  $\{I. \ is\_iface\ G\ I\}$ 

**apply** (*fold dom-def*)  
**apply** (*rule-tac finite-dom-map-of*)  
**done**

**lemma** *finite-is-class*: *finite*  $\{C. \ is\_class\ G\ C\}$ 

**apply** (*fold dom-def*)  
**apply** (*rule-tac finite-dom-map-of*)  
**done**

```

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

```

```

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

```

```

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

```

```

lemma subcls1-irrefl-lemma1:
  ws-prog G  $\implies$  (subcls1 G)-1  $\cap$  (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, standard]
lemmas subcls1-acyclic = subcls1-irrefl-lemma2 [THEN acyclicI, standard]

```

```

lemma wf-subint1: ws-prog G  $\implies$  wf ((subint1 G)-1)
by (auto intro: finite-acyclic-wf-converse finite-subint1 subint1-acyclic)

```

```

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

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

$\llbracket \text{class } G C = \text{Some } c; \text{ws-prog } G;$   
 $\bigwedge co. \text{class } G \text{Object} = \text{Some } co \implies P \text{Object};$   
 $\bigwedge C c. \llbracket \text{class } G C = \text{Some } c; C \neq \text{Object}; P (\text{super } c) \rrbracket \implies P C$   
 $\rrbracket \implies P C$   
**proof** –  
**assume** *clsC*:  $\text{class } G C = \text{Some } c$   
**and** *init*:  $\bigwedge co. \text{class } G \text{Object} = \text{Some } co \implies P \text{Object}$   
**and** *step*:  $\bigwedge C c. \llbracket \text{class } G C = \text{Some } c; C \neq \text{Object}; P (\text{super } c) \rrbracket \implies P C$   
**assume** *ws*:  $\text{ws-prog } G$   
**then have**  $\text{is-class } G C \implies P C$   
**proof** (*induct rule: subcls1-induct*)

```

fix C
assume hyp:  $\forall S. G \vdash C \prec_{C1} S \longrightarrow is\_class\ G\ S \longrightarrow P\ S$ 
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]:  
 $\llbracket is\_class\ G\ C; ws\_prog\ G; \bigwedge co. class\ G\ Object = Some\ co \implies P\ Object; \bigwedge C\ c. \llbracket class\ G\ C = Some\ c; C \neq Object; P\ (super\ c) \rrbracket \implies P\ C \rrbracket \implies P\ C$   
**by** (auto intro: ws-class-induct)

**lemma** ws-class-induct'' [consumes 2, case-names Object Subcls]:  
 $\llbracket class\ G\ C = Some\ c; ws\_prog\ G; \bigwedge co. class\ G\ Object = Some\ co \implies P\ Object\ co; \bigwedge C\ c\ sc. \llbracket class\ G\ C = Some\ c; class\ G\ (super\ c) = Some\ sc; C \neq Object; P\ (super\ c)\ sc \rrbracket \implies P\ C\ c \rrbracket \implies P\ C\ c$   
**proof** –  
**assume** clsC:  $class\ G\ C = Some\ c$   
**and** init:  $\bigwedge co. class\ G\ Object = Some\ co \implies P\ Object\ co$   
**and** step:  $\bigwedge C\ c\ sc. \llbracket class\ G\ C = Some\ c; class\ G\ (super\ c) = Some\ sc; C \neq Object; P\ (super\ c)\ sc \rrbracket \implies P\ C\ c$   
**assume** ws: ws-prog G  
**then have**  $\bigwedge c. class\ G\ C = Some\ c \implies P\ C\ c$   
**proof** (induct rule: subcls1-induct)  
**fix** C c  
**assume** hyp:  $\forall S. G \vdash C \prec_{C1} S \longrightarrow (\forall s. class\ G\ S = Some\ s \longrightarrow P\ S\ s)$   
**and** iscls:  $class\ G\ C = Some\ c$   
**show** P C c  
**proof** (cases C=Object)  
**case** True **with** iscls init **show** P C c **by** auto  
**next**  
**case** False  
**with** ws iscls **obtain** sc **where**  
   sc:  $class\ G\ (super\ c) = Some\ sc$   
   **by** (auto dest: ws-prog-cdeclD)  
**from** iscls False **have**  $G \vdash C \prec_{C1} (super\ c)$  **by** (rule subcls1I)  
**with** False ws step hyp iscls sc  
**show** P C c  
   **by** (auto)  
**qed**  
**qed**  
**with** clsC **show** P C c **by** auto  
**qed**

**lemma** ws-interface-induct [consumes 2, case-names Step]:

```

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

shows P I
proof –
from is-if-I ws
show P I
proof (rule ws-subint1-induct)
  fix I i
  assume hyp: iface G I = Some i  $\wedge$ 
             $(\forall J \in \text{set } (\text{isuperIfs } i). (I,J) \in \text{subint1 } G \wedge P J \wedge \text{is-iface } G J)$ 
  then have if-I: iface G I = Some i
    by blast
  show P I
  proof (cases isuperIfs i)
    case Nil
    with if-I hyp-sub
    show P I
    by auto
  next
    case (Cons hd tl)
    with hyp if-I hyp-sub
    show P I
    by auto
  qed
qed
qed

```

### general recursion operators for the interface and class hierarchies

```

consts
  iface-rec :: prog  $\times$  qtname  $\Rightarrow$  (qtname  $\Rightarrow$  iface  $\Rightarrow$  'a set  $\Rightarrow$  'a)  $\Rightarrow$  'a
  class-rec :: prog  $\times$  qtname  $\Rightarrow$  'a  $\Rightarrow$  (qtname  $\Rightarrow$  class  $\Rightarrow$  'a  $\Rightarrow$  'a)  $\Rightarrow$  'a

recdef iface-rec same-fst ws-prog ( $\lambda G. (\text{subint1 } G)^{-1}$ )
iface-rec (G,I) =
  ( $\lambda f. \text{case } \text{iface } G I \text{ of}$ 
    None  $\Rightarrow$  arbitrary
    | Some i  $\Rightarrow$  if ws-prog G
      then f I i
       $((\lambda J. \text{iface-rec } (G,J) f) ' \text{set } (\text{isuperIfs } i))$ 
      else arbitrary)
  (hints recdef-wf: wf-subint1 intro: subint1I)
declare iface-rec.simps [simp del]

```

```

lemma iface-rec:
 $\llbracket \text{iface } G I = \text{Some } i; \text{ws-prog } G \rrbracket \implies$ 
  iface-rec (G,I) f = f I i  $((\lambda J. \text{iface-rec } (G,J) f) ' \text{set } (\text{isuperIfs } i))$ 
apply (subst iface-rec.simps)
apply simp
done

```

```

recdef class-rec same-fst ws-prog ( $\lambda G. (\text{subcls1 } G)^{-1}$ )
class-rec(G,C) =
  ( $\lambda t f. \text{case } \text{class } G C \text{ of}$ 
    None  $\Rightarrow$  arbitrary

```

```

| Some c ⇒ if ws-prog G
            then f C c
              (if C = Object then t
               else class-rec (G,super c) t f)
            else arbitrary)
(hints recdef-wf: wf-subcls1 intro: subcls1I)
declare class-rec.simps [simp del]

lemma class-rec: [[class G C = Some c; ws-prog G]] ⇒
  class-rec (G,C) t f =
    f C c (if C = Object then t else class-rec (G,super c) t f)
apply (rule class-rec.simps [THEN trans [THEN fun-cong [THEN fun-cong]]])
apply simp
done

constdefs
imethds:: prog ⇒ qtname ⇒ (sig,qtname × mhead) tables
  — methods of an interface, with overriding and inheritance, cf. 9.2
imethds G I
  ≡ iface-rec (G,I)
    (λI i ts. (Un-tables ts) ⊕⊕
              (o2s ∘ table-of (map (λ(s,m). (s,I,m)) (imethds i))))

end

```

## Chapter 9

# TypeRel

## 15 The relations between Java types

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

simplifications:

- subinterface, subclass and widening relation includes identity

improvements over Java Specification 1.0:

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

design issues:

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

**consts**

*implmt1* :: *prog*  $\Rightarrow$  (*qname*  $\times$  *qname*) *set* — direct implementation

**syntax**

*-subint1* :: *prog*  $\Rightarrow$  [*qname*, *qname*]  $\Rightarrow$  *bool* ( $\neg$   $\neg$   $\neg$   $\neg$  *I1* - [*71*, *71*, *71*] *70*)

*-subint* :: *prog*  $\Rightarrow$  [*qname*, *qname*]  $\Rightarrow$  *bool* ( $\neg$   $\neg$   $\neg$   $\neg$   $\neg$  *I* - [*71*, *71*, *71*] *70*)

*@implmt1* :: *prog*  $\Rightarrow$  [*qname*, *qname*]  $\Rightarrow$  *bool* ( $\neg$   $\neg$   $\neg$   $\neg$   $\neg$   $\neg$  *I* - [*71*, *71*, *71*] *70*)

**syntax** (*xsymbols*)

*-subint1* :: *prog*  $\Rightarrow$  [*qname*, *qname*]  $\Rightarrow$  *bool* ( $\neg$   $\neg$   $\neg$   $\neg$  *I1* - [*71*, *71*, *71*] *70*)

*-subint* :: *prog*  $\Rightarrow$  [*qname*, *qname*]  $\Rightarrow$  *bool* ( $\neg$   $\neg$   $\neg$   $\neg$   $\neg$  *I* - [*71*, *71*, *71*] *70*)

*-implmt1* :: *prog*  $\Rightarrow$  [*qname*, *qname*]  $\Rightarrow$  *bool* ( $\neg$   $\neg$   $\neg$   $\neg$   $\neg$   $\neg$  *I* - [*71*, *71*, *71*] *70*)

**translations**

$G \vdash I \prec I1 J == (I, J) \in \text{subint1 } G$

$G \vdash I \preceq I J == (I, J) \in (\text{subint1 } G)^* \text{ — cf. 9.1.3}$

$G \vdash C \leadsto I I == (C, I) \in \text{implmt1 } G$

**subclass and subinterface relations**

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

**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*:  
 $\llbracket \text{class } G \ C = \text{Some } c; C \neq \text{Object}; D = \text{super } c \rrbracket \implies G \vdash C \prec_{C1} D$   
**apply** (*auto dest: subcls1I1*)  
**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*:  
 $\llbracket \text{class } G \ (\text{SXcpt } xn) = \text{Some } xc;$   
 $\text{super } xc = (\text{if } xn = \text{Throwable then Object else SXcpt Throwable}) \rrbracket$   
 $\implies G \vdash \text{SXcpt } xn \preceq_C \text{SXcpt Throwable}$   
**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*:  
 $\llbracket G \vdash A \prec_{C1} B; G \vdash A \prec_{C1} C \rrbracket \implies B = C$   
**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*:  $\llbracket G \vdash C \prec_{C1} D; \text{ws-prog } G \rrbracket \implies C \neq D$

**proof**

**assume** *ws*: *ws-prog* *G* **and**  
*subcls1*:  $G \vdash C \prec_{C1} D$  **and**  
*eq-C-D*:  $C = D$   
**from** *subcls1* **obtain** *c*  
**where**  
*neq-C-Object*:  $C \neq \text{Object}$  **and**  
*clsC*:  $\text{class } G \ C = \text{Some } c$  **and**  
*super-c*:  $\text{super } c = D$   
**by** (*auto simp add: subcls1-def*)  
**with** *super-c subcls1 eq-C-D*  
**have** *subcls-super-c-C*:  $G \vdash \text{super } c \prec_C C$   
**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:  $G \vdash C \prec_C D$ 
then show ?thesis
proof (induct rule: converse-trancl-induct)
  fix C
  assume subcls1-C-D:  $G \vdash C \prec_{C1} D$ 
  then obtain c where
     $C \neq \text{Object}$  and
     $\text{class } G \ C = \text{Some } c$  and
     $\text{super } c = D$ 
  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:  $G \vdash C \prec_{C1} Z$  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* [*consumes 1*, *case-names Eq Subcls*]:  
 $\llbracket G \vdash C \preceq_C D; C = D \implies P; G \vdash C \prec_C D \implies P \rrbracket \implies P$   
**by** (*blast intro: rtrancl-cases*)

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

**lemma** *subcls-irrefl*:  $\llbracket G \vdash C \prec_C D; \text{ws-prog } G \rrbracket$   
 $\implies C \neq D$   
**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**  $G \vdash C \prec_{C1} D$   
**with** *ws*  
**show**  $C \neq D$   
**by** (*blast dest: subcls1-irrefl*)  
**next**  
**fix** *C Z*

```

assume subcls1-C-Z:  $G \vdash C \prec_{C1} Z$  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 –
    from subcls1-C-Z eq-C-D
    have  $G \vdash D \prec_C Z$ 
      by (auto)
    also
    from subcls-Z-D ws
    have  $\neg G \vdash D \prec_C Z$ 
      by (rule subcls-acyclic)
    ultimately
    show ?thesis
      by – (rule notE)
  qed
qed
qed
qed

```

```

lemma invert-subclseq:
   $\llbracket G \vdash C \preceq_C D; ws\text{-}prog\ G \rrbracket$ 
   $\implies \neg G \vdash D \prec_C C$ 
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: rtrancl-into-trancl3)
    with ws
    show ?thesis
      by (blast dest: subcls-acyclic)
  qed
qed

```

```

lemma invert-subcls:
   $\llbracket G \vdash C \prec_C D; ws\text{-}prog\ G \rrbracket$ 
   $\implies \neg G \vdash D \preceq_C C$ 
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 subcls-C-C:  $G \vdash C \prec_C D$ 
  then obtain S where
     $G \vdash C \prec_{C1} S$  and
    subclseq-S-D:  $G \vdash S \preceq_C D$ 
    by (blast dest: tranclD)
  with clsC
  have  $S = \text{super } c$ 
    by (auto dest: subcls1D)
  with subclseq-S-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 neq-C-D:  $C \neq D$ 
  assume  $\text{cls}C: \text{class } G \ C = \text{Some } c$ 
  assume subclseq-C-D:  $G \vdash C \preceq_C D$ 
  then show ?thesis
    proof (cases rule: subclseq-cases)
      case Eq with neq-C-D show ?thesis by contradiction
    next
      case Subcls
      with clsC show ?thesis by (blast dest: subcls-superD)
    qed
  qed

```

## implementation relation

**defs**

— direct implementation, cf. 8.1.3

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

```

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

*implmt* :: *prog*  $\Rightarrow$  *qtname*  $\Rightarrow$  *qtname*  $\Rightarrow$  *bool* ( $\vdash \rightsquigarrow$  [71,71,71] 70)

for *G* :: *prog*

**where**

*direct*:  $G \vdash C \rightsquigarrow I J \implies G \vdash C \rightsquigarrow J$   
*subint*:  $\llbracket G \vdash C \rightsquigarrow I I; G \vdash I \preceq I J \rrbracket \implies G \vdash C \rightsquigarrow J$   
*subcls1*:  $\llbracket G \vdash C \prec_{C_1} D; G \vdash D \rightsquigarrow J \rrbracket \implies G \vdash C \rightsquigarrow J$

**lemma** *implmtD*:  $G \vdash C \rightsquigarrow J \implies (\exists I. G \vdash C \rightsquigarrow I I \wedge G \vdash I \preceq I J) \vee (\exists D. G \vdash C \prec_{C_1} D \wedge G \vdash D \rightsquigarrow J)$

**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 rtrancl-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

*widen* :: *prog*  $\Rightarrow$  *ty*  $\Rightarrow$  *ty*  $\Rightarrow$  *bool* ( $\vdash \preceq$  [71,71,71] 70)

for *G* :: *prog*

**where**

*refl*:  $G \vdash T \preceq T$  — identity conversion, cf. 5.1.1  
*subint*:  $G \vdash I \preceq I J \implies G \vdash \text{Iface } I \preceq \text{Iface } J$  — wid.ref.conv., cf. 5.1.4  
*int-obj*:  $G \vdash \text{Iface } I \preceq \text{Class } \text{Object}$   
*subcls*:  $G \vdash C \preceq_C D \implies G \vdash \text{Class } C \preceq \text{Class } D$   
*implmt*:  $G \vdash C \rightsquigarrow I \implies G \vdash \text{Class } C \preceq \text{Iface } I$   
*null*:  $G \vdash \text{NT} \preceq \text{RefT } R$   
*arr-obj*:  $G \vdash T.\boxed{\phantom{x}} \preceq \text{Class } \text{Object}$   
*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*:  $G \vdash \text{PrimT Boolean} \preceq T \implies T = \text{PrimT Boolean}$   
**by** (*ind-cases*  $G \vdash \text{PrimT Boolean} \preceq T$ ) *simp-all*

**lemma** *widen-Boolean2*:  $G \vdash S \preceq \text{PrimT Boolean} \implies S = \text{PrimT Boolean}$   
**by** (*ind-cases*  $G \vdash S \preceq \text{PrimT Boolean}$ ) *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*:  $G \vdash \text{Iface } I \preceq T \implies T = \text{Class Object} \vee (\exists J. T = \text{Iface } J)$   
**apply** (*ind-cases*  $G \vdash \text{Iface } I \preceq T$ )  
**by** *auto*

**lemma** *widen-Iface2*:  $G \vdash S \preceq \text{Iface } J \implies S = \text{NT} \vee (\exists I. S = \text{Iface } I) \vee (\exists D. S = \text{Class } D)$   
**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*:  $G \vdash S.\square \preceq T \implies T = \text{Class } \text{Object} \vee (\exists T'. T = T'.\square \wedge G \vdash S \preceq T')$   
**apply** (*ind-cases*  $G \vdash S.\square \preceq T$ )  
**by** *auto*

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

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

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



```

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

```

```

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

```

```

lemma widen-Array-Class:  $G \vdash S.\llbracket \preceq \text{Class } C \rrbracket \implies C = \text{Object}$ 
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:  $\llbracket \text{isrtype } G \text{ } T; \text{ws-prog } G \rrbracket \implies G \vdash \text{RefT } T \preceq \text{Class Object}$ 
apply (case-tac T)
apply (auto)
apply (subgoal-tac  $G \vdash \text{qtname-ext-type} \preceq_C \text{Object}$ )
apply (auto intro: subcls-ObjectI)
done

```

```

lemma widen-trans-lemma [rule-format (no-asm)]:
   $\llbracket G \vdash S \preceq U; \forall C. \text{is-class } G \text{ } C \longrightarrow G \vdash C \preceq_C \text{Object} \rrbracket \implies \forall T. G \vdash U \preceq T \longrightarrow G \vdash S \preceq T$ 
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 (tactic ALLGOALS (etac 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)]:  $\llbracket G \vdash S \preceq T;$   
 $\forall I J. G \vdash I \preceq I J \wedge G \vdash J \preceq I I \longrightarrow I = J;$   
 $\forall C D. G \vdash C \preceq_C D \wedge G \vdash D \preceq_C C \longrightarrow C = D;$   
 $\forall I. G \vdash \text{Object} \rightsquigarrow I \longrightarrow \text{False} \rrbracket \Longrightarrow G \vdash T \preceq S \longrightarrow S = T$   
**apply** (erule *widen.induct*)  
**apply** (auto dest: *widen-Iface widen-NT2 widen-Class*)  
**done**

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

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

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

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

**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* ( $\vdash$ - $\succ$ -[71,71,71] 70)

**for** *G* :: *prog*

**where**

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

**lemma** *narrow-RefT*:  $G \vdash \text{RefT } R \succ T \implies \exists t. T = \text{RefT } t$   
**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  
 $\text{cast} :: \text{prog} \Rightarrow \text{ty} \Rightarrow \text{bool} \quad (-\vdash-\preceq^? - [71, 71, 71] \ 70)$   
**for**  $G :: \text{prog}$   
**where**  
 $\text{widen} : G \vdash S \preceq T \implies G \vdash S \preceq^? T$   
 $\text{narrow} : G \vdash S \succ T \implies G \vdash S \preceq^? T$

**lemma** *cast-RefT*:  $G \vdash \text{RefT } R \preceq^? T \implies \exists t. T = \text{RefT } t$   
**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**  $T = \text{PrimT } \text{Boolean}$   
**using** *bool-cast*  
**proof** (*cases*)  
**case** *widen*  
**hence**  $G \vdash \text{PrimT } \text{Boolean} \preceq T$   
**by** *simp*  
**thus** *?thesis* **by** (*rule widen-Boolean*)  
**next**  
**case** *narrow*  
**hence**  $G \vdash \text{PrimT } \text{Boolean} \succ T$   
**by** *simp*  
**thus** *?thesis* **by** (*rule narrow-Boolean*)  
**qed**

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

## Chapter 10

# DeclConcepts

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

theory *DeclConcepts* imports *TypeRel* begin

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

**constdefs**

*is-public* :: *prog*  $\Rightarrow$  *qname*  $\Rightarrow$  *bool*  
*is-public* *G* *qn*  $\equiv$  (case class *G* *qn* of  
     *None*  $\Rightarrow$  (case iface *G* *qn* of  
         *None*  $\Rightarrow$  *False*  
         | *Some iface*  $\Rightarrow$  *access iface* = *Public*)  
     | *Some class*  $\Rightarrow$  *access class* = *Public*)

## 17 accessibility of types (cf. 6.6.1)

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

**consts** *accessible-in* :: *prog*  $\Rightarrow$  *ty*  $\Rightarrow$  *pname*  $\Rightarrow$  *bool*  
     (-  $\vdash$  - *accessible'-in* - [61,61,61] 60)  
     *rt-accessible-in*:: *prog*  $\Rightarrow$  *ref-ty*  $\Rightarrow$  *pname*  $\Rightarrow$  *bool*  
     (-  $\vdash$  - *accessible'-in'* - [61,61,61] 60)

**primrec**

$G \vdash (\text{PrimT } p) \text{ accessible-in pack} = \text{True}$   
*accessible-in-RefT-simp*:  
 $G \vdash (\text{RefT } r) \text{ accessible-in pack} = G \vdash r \text{ accessible-in' pack}$   
  
 $G \vdash (\text{NullT}) \text{ accessible-in' pack} = \text{True}$   
 $G \vdash (\text{IfaceT } I) \text{ accessible-in' pack} = ((\text{pid } I = \text{pack}) \vee \text{is-public } G \ I)$   
 $G \vdash (\text{ClassT } C) \text{ accessible-in' pack} = ((\text{pid } C = \text{pack}) \vee \text{is-public } G \ C)$   
 $G \vdash (\text{ArrayT } ty) \text{ accessible-in' pack} = G \vdash ty \text{ accessible-in pack}$

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

**constdefs**

*is-acc-class* :: *prog*  $\Rightarrow$  *pname*  $\Rightarrow$  *qname*  $\Rightarrow$  *bool*  
*is-acc-class* *G* *P* *C*  $\equiv$  *is-class* *G* *C*  $\wedge$   $G \vdash (\text{Class } C) \text{ accessible-in } P$   
*is-acc-iface* :: *prog*  $\Rightarrow$  *pname*  $\Rightarrow$  *qname*  $\Rightarrow$  *bool*  
*is-acc-iface* *G* *P* *I*  $\equiv$  *is-iface* *G* *I*  $\wedge$   $G \vdash (\text{Iface } I) \text{ accessible-in } P$   
*is-acc-type* :: *prog*  $\Rightarrow$  *pname*  $\Rightarrow$  *ty*  $\Rightarrow$  *bool*  
*is-acc-type* *G* *P* *T*  $\equiv$  *is-type* *G* *T*  $\wedge$   $G \vdash T \text{ accessible-in } P$   
*is-acc-reftype* :: *prog*  $\Rightarrow$  *pname*  $\Rightarrow$  *ref-ty*  $\Rightarrow$  *bool*  
*is-acc-reftype* *G* *P* *T*  $\equiv$  *isrtype* *G* *T*  $\wedge$   $G \vdash T \text{ accessible-in' } P$

**lemma** *is-acc-classD*:

*is-acc-class* *G* *P* *C*  $\implies$  *is-class* *G* *C*  $\wedge$   $G \vdash (\text{Class } C) \text{ accessible-in } P$   
**by** (*simp add: is-acc-class-def*)

**lemma** *is-acc-class-is-class*: *is-acc-class* *G* *P* *C*  $\implies$  *is-class* *G* *C*

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

**lemma** *is-acc-ifaceD*:

*is-acc-iface* *G* *P* *I*  $\implies$  *is-iface* *G* *I*  $\wedge$   $G \vdash (\text{Iface } I) \text{ accessible-in } P$   
**by** (*simp add: is-acc-iface-def*)

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

**lemma** *is-acc-reftypeD*:  
*is-acc-reftype*  $G\ P\ T \implies isrtype\ G\ T \wedge G \vdash T\ accessible-in'\ P$   
**by** (*simp add: is-acc-reftype-def*)

## 18 accessibility of members

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

### Various technical conversion and selection functions

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

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

**instance** *acc-modi*::*has-accmodi* ..

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

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

**instance** *decl-ext-type*:: (*type*) *has-accmodi* ..

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

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

**instance** \* :: (*type*,*has-accmodi*) *has-accmodi* ..

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

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

**instance** *memberdecl* :: *has-accmodi* ..

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

```

lemma memberdecl-fdecl-acc-modi-simp[simp]:
  accmodi (fdecl m) = accmodi m
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

```

axclass has-declclass < type
consts declclass:: 'a::has-declclass  $\Rightarrow$  qname

```

```

instance qname-ext-type::(type) has-declclass ..

```

```

defs (overloaded)
qname-declclass-def: declclass (q::qname)  $\equiv$  q

```

```

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

```

```

instance * :: (has-declclass,type) has-declclass ..

```

```

defs (overloaded)
pair-declclass-def: declclass p  $\equiv$  declclass (fst p)

```

```

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

```

axclass has-static < type
consts is-static:: 'a::has-static  $\Rightarrow$  bool

```

```

instance decl-ext-type:: (has-static) has-static ..

```

```

defs (overloaded)
decl-is-static-def:
  is-static (m::('a::has-static) decl-scheme)  $\equiv$  is-static (Decl.decl.more m)

```

```

instance member-ext-type:: (type) has-static ..

```

```

defs (overloaded)
static-field-type-is-static-def:
  is-static (m::('b::type) member-ext-type)  $\equiv$  static-sel m

```

```

lemma member-is-static-simp: is-static (m::'a member-scheme) = static m
apply (cases m)
apply (simp add: static-field-type-is-static-def
               decl-is-static-def Decl.member.dest-convs)
done

```

```

instance * :: (type,has-static) has-static ..

```

```

defs (overloaded)
pair-is-static-def: is-static p  $\equiv$  is-static (snd p)

```



**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*  $p = \text{is-static } (\text{snd } p)$   
**by** (*simp add: pair-is-static-def*)

```
instance memberdecl:: has-static ..
```

```

defs (overloaded)
memberdecl-is-static-def:
  is-static m  $\equiv$  ( case m of
                     fdecl f  $\Rightarrow$  is-static f
                     | mdecl m  $\Rightarrow$  is-static m )

```

**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)

**constdefs** — some mnemonic selectors for various pairs

$$\begin{array}{ll} \text{declface}:: (qname \times ('a::type) \text{ decl-scheme}) \Rightarrow qname & \\ \text{declface} \equiv fst & \text{— get the interface component} \end{array}$$
$$\begin{array}{l} mbr:: (qtname \times memberdecl) \Rightarrow memberdecl \\ mbr \equiv snd \quad \text{— get the memberdecl component} \end{array}$$

<i>mthd</i> ::	$('b \times 'a) \Rightarrow 'a$	— also used for mdecl, mhead
<i>mthd</i> $\equiv$ <i>snd</i>		— get the method component

$$\begin{aligned} fld :: & \quad ('b \times ('a :: type) \text{ decl-scheme}) \Rightarrow ('a :: type) \text{ decl-scheme} \\ & \quad \text{--- also used for } ((vname \times qtname) \times field) \\ fld \equiv snd & \quad \text{--- get the field component} \end{aligned}$$

**constdefs** — some mnemonic selectors for  $(vname \times qname)$   
 $fname :: (vname \times 'a) \Rightarrow vname$  — also used for `fdecl`  
 $fname \equiv fst$

$$\begin{aligned} \text{declclassf}:: (vname \times qtname) &\Rightarrow qtname \\ \text{declclassf} &\equiv \text{snd} \end{aligned}$$

**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*)

**constdefs** — some mnemonic selectors for (*vname* × *qtname*)  
*fldname* :: (*vname* × *qtname*) ⇒ *vname*  
*fldname* ≡ *fst*  
  
*fldclass* :: (*vname* × *qtname*) ⇒ *qtname*  
*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-fldname-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*

**constdefs**  
*methdMembr* :: (*qtname* × *mdecl*) ⇒ (*qtname* × *memberdecl*)  
*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*

**constdefs**  
*method* :: *sig* ⇒ (*qtname* × *methd*) ⇒ (*qtname* × *memberdecl*)  
*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*)

**constdefs**

*fieldm* :: *vname*  $\Rightarrow$  (*qtname*  $\times$  *field*)  $\Rightarrow$  (*qtname*  $\times$  *memberdecl*)  
*fieldm* *n* *f*  $\equiv$  (*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*

**constdefs** *msig*:: (*qtname*  $\times$  *mdecl*)  $\Rightarrow$  *sig*  
*msig* *m*  $\equiv$  *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*

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

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

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

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

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

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

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

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

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

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

**instance** *decl-ext-type* :: (*has-resTy*) *has-resTy* ..

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

**instance** *member-ext-type* :: (*has-resTy*) *has-resTy* ..

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

**instance** *mhead-ext-type* :: (*type*) *has-resTy* ..

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

**lemma** *mhead-resTy-simp*: *resTy* (*m*::'a *mhead-scheme*) = *resT m*  
**apply** (*cases m*)  
**apply** (*simp add: decl-resTy-def member-ext-type-resTy-def*  
*mhead-ext-type-resTy-def*  
*member.dest-convs mhead.dest-convs*)

**done**

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

**instance** \* :: (*type*, *has-resTy*) *has-resTy* ..

**defs** (**overloaded**)  
*pair-resTy-def*: *resTy* *p*  $\equiv$  *resTy* (*snd* *p*)

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

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

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

### inheritable-in

$G \vdash m$  *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.

### constdefs

*inheritable-in*::  
 $prog \Rightarrow (qname \times memberdecl) \Rightarrow pname \Rightarrow bool$   
 $(- \vdash - \textit{inheritable}'\textit{-in} - [61,61,61] \ 60)$   
 $G \vdash membr \textit{inheritable-in} \ pack$   
 $\equiv (case \ (accmodi \ membr) \ of$   
 $\quad Private \Rightarrow False$   
 $\quad | \ Package \Rightarrow (pid \ (declclass \ membr)) = pack$   
 $\quad | \ Protected \Rightarrow True$   
 $\quad | \ Public \Rightarrow True)$

### syntax

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

### translations

$G \vdash Method \ m \textit{inheritable-in} \ p == G \vdash methdMembr \ m \textit{inheritable-in} \ p$

### syntax

*Method-inheritable-in*::  
 $prog \Rightarrow sig \Rightarrow (qname \times methd) \Rightarrow pname \Rightarrow bool$   
 $(- \vdash Method - \textit{inheritable}'\textit{-in} - [61,61,61,61] \ 60)$

### translations

$G \vdash Method \ s \ m \textit{inheritable-in} \ p == G \vdash (method \ s \ m) \textit{inheritable-in} \ p$

**declared-in/undeclared-in**

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

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

**constdefs**  
*declared-in*:: *prog*  $\Rightarrow$  *memberdecl*  $\Rightarrow$  *qname*  $\Rightarrow$  *bool*  
 ( $\vdash$  - *declared'-in* - [61,61,61] 60)  
*G*  $\vdash$  *m* *declared-in* *C*  $\equiv$  (case *m* of  
*fdecl* (*fn*,*f*)  $\Rightarrow$  *cdeclaredfield* *G* *C* *fn* = Some *f*  
 | *mdecl* (*sig*,*m*)  $\Rightarrow$  *cdeclaredmethd* *G* *C* *sig* = Some *m*)

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

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

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

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

**lemma** *declared-in-classD*:  
*G*  $\vdash$  *m* *declared-in* *C*  $\implies$  *is-class* *G* *C*  
**by** (cases *m*)  
 (auto simp add: *declared-in-def* *cdeclaredmethd-def* *cdeclaredfield-def*)

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

*G*  $\vdash$  *m* *undeclared-in* *C*  $\equiv$  (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*  
 ( $\vdash$  - *member'-of* - [61,61,61] 60)  
**for** *G* :: *prog*  
**where**

*Immediate:*  $\llbracket G \vdash \text{mbr } m \text{ declared-in } C; \text{declclass } m = C \rrbracket \implies G \vdash m \text{ member-of } C$   
*Inherited:*  $\llbracket G \vdash m \text{ inheritable-in } (pid \ C); G \vdash \text{memberid } m \text{ undeclared-in } C;$   
 $G \vdash C \prec_{C1} S; G \vdash (Class \ S) \text{ accessible-in } (pid \ C); G \vdash m \text{ member-of } S$   
 $\rrbracket \implies G \vdash m \text{ member-of } C$

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.

#### syntax

*method-member-of::*  $prog \Rightarrow (qname \times mdecl) \Rightarrow qname \Rightarrow bool$   
 $(- \vdash Method - \text{member'-of} - [61,61,61] \ 60)$

#### translations

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

#### syntax

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

#### translations

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

#### syntax

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

#### translations

$G \vdash Field \ n \ f \text{ member-of } C \Leftrightarrow G \vdash fieldm \ n \ f \text{ member-of } C$

#### constdefs

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

$G \vdash C \text{ inherits } m$

$\equiv G \vdash m \text{ inheritable-in } (pid \ C) \wedge G \vdash \text{memberid } m \text{ undeclared-in } C \wedge$   
 $(\exists \ S. G \vdash C \prec_{C1} S \wedge G \vdash (Class \ S) \text{ accessible-in } (pid \ C) \wedge G \vdash m \text{ member-of } S)$

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

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

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

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

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

#### syntax

*method-member-in::*  $prog \Rightarrow (qname \times mdecl) \Rightarrow qname \Rightarrow bool$   
 $(- \vdash Method - \text{member'-in} - [61,61,61] \ 60)$

#### translations

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



**syntax**

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

**translations**

$G \vdash Methd\ s\ m\ member-in\ C \Leftrightarrow G \vdash (method\ s\ m)\ member-in\ C$

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

$\Rightarrow \exists provC. G \vdash C \preceq_C provC \wedge G \vdash m\ member-of\ provC$

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

**lemma** *member-inI*:  $\llbracket G \vdash m\ member-of\ provC; G \vdash C \preceq_C provC \rrbracket \Rightarrow G \vdash m\ member-in\ C$

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

**lemma** *member-of-to-member-in*:  $G \vdash m\ member-of\ C \Rightarrow G \vdash m\ member-in\ C$

**by** (*auto intro: member-inI*)

**overriding**

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

Static overriding (used during the typecheck of the compiler)

**inductive**

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

**for**  $G :: prog$

**where**

*Direct*:  $\llbracket \neg is-static\ new; msig\ new = msig\ old;$   
 $G \vdash Method\ new\ declared-in\ (declclass\ new);$   
 $G \vdash Method\ old\ declared-in\ (declclass\ old);$   
 $G \vdash Method\ old\ inheritable-in\ pid\ (declclass\ new);$   
 $G \vdash (declclass\ new) \prec_{C1}\ superNew;$   
 $G \vdash Method\ old\ member-of\ superNew$   
 $\rrbracket \Rightarrow G \vdash new\ overrides_S\ old$

| *Indirect*:  $\llbracket G \vdash new\ overrides_S\ inter; G \vdash inter\ overrides_S\ old \rrbracket$   
 $\Rightarrow G \vdash new\ overrides_S\ old$

Dynamic overriding (used during the typecheck of the compiler)

**inductive**

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

**for**  $G :: prog$

**where**

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

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

### **syntax**

*sig-stat-overrides*::

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

### **translations**

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

### **syntax**

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

### **translations**

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

## **Hiding**

### **constdefs** *hides*::

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

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

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

### **syntax**

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

### **translations**

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

### **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*:

$\llbracket G \vdash \text{new overrides old} \rrbracket \implies$   
 $\text{declclass new} \neq \text{Object} \wedge \neg \text{is-static new} \wedge \neg \text{is-static old} \wedge$   
 $\text{accmodi new} \neq \text{Private} \wedge$

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

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

**lemma** *overrides-eq-sigD*:  
 $\llbracket G \vdash new\ overrides\ old \rrbracket \implies msig\ old = msig\ new$   
**by** (auto dest: overrides-commonD)

**lemma** *hides-eq-sigD*:  
 $\llbracket G \vdash new\ hides\ old \rrbracket \implies msig\ old = msig\ new$   
**by** (auto simp add: hides-def)

## permits access

### constdefs

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

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

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

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

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

**inductive**

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

**where**

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

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

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

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

**syntax**

$methd\text{-}accessible\text{-}from ::$

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

**translations**

$G \vdash Method \ s \ m \text{ of } cls \text{ accessible}\text{-}from \text{ acc}\text{-}class$   
 $\rightleftharpoons G \vdash (method \ s \ m) \text{ of } cls \text{ accessible}\text{-}from \text{ acc}\text{-}class$

**syntax**

$field\text{-}accessible\text{-}from ::$

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

**translations**

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

**inductive**

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

**where**

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

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

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

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

**syntax**

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

**translations**

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

**syntax**

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

**translations**

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

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

**lemma** *unique-declaration*:

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

**lemma** *declared-not-undeclared*:

$G \vdash m \text{ declared-in } C \Rightarrow \neg G \vdash \text{ memberid } m \text{ undeclared-in } C$   
**by** (*cases m*) (*auto simp add: declared-in-def undeclared-in-def*)

**lemma** *undeclared-not-declared*:

$G \vdash \text{ memberid } m \text{ undeclared-in } C \Rightarrow \neg G \vdash m \text{ declared-in } C$   
**by** (*cases m*) (*auto simp add: declared-in-def undeclared-in-def*)

**lemma** *not-undeclared-declared*:

$\neg G \vdash \text{ membr-id } \text{ undeclared-in } C \Rightarrow (\exists m. G \vdash m \text{ declared-in } C \wedge$   
 $\text{ membr-id } = \text{ memberid } m)$

**proof** –

**assume** *not-undecl*:  $\neg G \vdash \text{ membr-id } \text{ undeclared-in } C$   
**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 fld 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*:  $G \vdash m \text{ member-of } C$  **and**  
*eqid*:  $\text{memberid } n = \text{memberid } m$

**shows**  $n=m$

**proof** –

**from** *n m eqid*

**show**  $n=m$

**proof** (*induct*)

**case** (*Immediate n C*)

**assume** *member-n*:  $G \vdash \text{mbr } n \text{ declared-in } C \text{ declclass } n = C$

**assume** *eqid*:  $\text{memberid } n = \text{memberid } m$

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

**then show**  $n=m$

**proof** (*cases*)

**case** (*Immediate m' -*)

**with** *eqid*

**have**  $m=m'$

$\text{memberid } n = \text{memberid } m$

$G \vdash \text{mbr } m \text{ declared-in } C$

$\text{declclass } m = C$

**by** *auto*

**with** *member-n*

**show** *?thesis*

**by** (*cases n, cases m*)

$(\text{auto simp add: declared-in-def}$   
 $\text{cdeclaredmethd-def cdeclaredfield-def}$   
 $\text{split: memberdecl.splits})$

**next**

**case** (*Inherited m' -*)

**then have**  $G \vdash \text{memberid } m \text{ undeclared-in } C$

**by** *simp*

```

  with eqid member-n
  show ?thesis
  by (cases n) (auto dest: declared-not-undeclared)
qed
next
case (Inherited n C S)
assume undecl:  $G \vdash \text{memberid } n \text{ undeclared-in } C$ 
assume super:  $G \vdash C \prec_{C1} S$ 
assume hyp:  $\llbracket G \vdash m \text{ member-of } S; \text{memberid } n = \text{memberid } m \rrbracket \implies n = m$ 
assume eqid:  $\text{memberid } n = \text{memberid } m$ 
assume  $G \vdash m \text{ member-of } C$ 
then show  $n=m$ 
proof (cases)
case Immediate
then have  $G \vdash \text{mbr } m \text{ declared-in } C$  by simp
with eqid undecl
show ?thesis
by (cases m) (auto dest: declared-not-undeclared)
next
case Inherited
with super have  $G \vdash m \text{ member-of } S$ 
by (auto dest!: subcls1D)
with eqid hyp
show ?thesis
by blast
qed
qed
qed

```

```

lemma member-of-is-classD:  $G \vdash m \text{ member-of } C \implies \text{is-class } G \ C$ 
proof (induct set: members)
case (Immediate m C)
assume  $G \vdash \text{mbr } m \text{ declared-in } C$ 
then show  $\text{is-class } G \ C$ 
by (cases mbr m)
(auto simp add: declared-in-def cdeclaredmethd-def cdeclaredfield-def)
next
case (Inherited m C S)
assume  $G \vdash C \prec_{C1} S$  and  $\text{is-class } G \ S$ 
then show  $\text{is-class } G \ C$ 
by - (rule subcls-is-class2, auto)
qed

```

```

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

```

```

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

```

```

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

```

```

proof (induct set: members)
  case (Immediate m C)
  then show  $G \vdash C \preceq_C \text{declclass } m$  by simp
next
  case (Inherited m C S)
  then show  $G \vdash C \preceq_C \text{declclass } m$ 
    by (auto dest: r-into-rtrancl intro: rtrancl-trans)
qed

```

```

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

```

```

lemma stat-override-declclasses-relation:
   $\llbracket G \vdash (\text{declclass new}) \prec_{C1} \text{superNew}; G \vdash \text{Method old member-of superNew} \rrbracket$ 
 $\implies G \vdash (\text{declclass new}) \prec_C (\text{declclass old})$ 
apply (rule trancl-rtrancl-trancl)
apply (erule r-into-trancl)
apply (cases old)
apply (auto dest: member-of-class-relation)
done

```

```

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

```

```

lemma member-of-Package:
   $\llbracket G \vdash m \text{ member-of } C; \text{accmodi } m = \text{Package} \rrbracket$ 
 $\implies \text{pid } (\text{declclass } m) = \text{pid } C$ 
proof -
  assume member:  $G \vdash m \text{ member-of } C$ 
  then show  $\text{accmodi } m = \text{Package} \implies ?thesis$  (is PROP ?P m C)
  proof (induct rule: members.induct)
    fix C m
    assume C:  $\text{declclass } m = C$ 
    then show  $\text{pid } (\text{declclass } m) = \text{pid } C$ 
      by simp
  next
    fix C S m
    assume inheritable:  $G \vdash m \text{ inheritable-in pid } C$ 
    assume hyp: PROP ?P m S and
      package-acc:  $\text{accmodi } m = \text{Package}$ 
    with inheritable package-acc hyp
    show  $\text{pid } (\text{declclass } m) = \text{pid } C$ 
      by (auto simp add: inheritable-in-def)
  qed
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**  $G \vdash m \text{ member-of declclass } m$

**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*:

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

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

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

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

**lemma** *permits-acc-inheritance*:

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

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

**by** (*cases accmodi m*)

(*auto simp add: permits-acc-def*)

*intro: subclseq-trans*)

**lemma** *permits-acc-static-declC*:

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

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

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

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

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

*static*: *is-static m*

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

**proof** –

**from** *acc-C static*

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

**proof** (*induct*)

```

case (Immediate m C)
then show ?case
  by (auto intro!: dyn-accessible-fromR.Immediate
      dest: member-in-declC permits-acc-static-declC)
next
  case (Overriding m C declCNew new old sup)
  then have  $\neg$  is-static m
    by (auto dest: overrides-commonD)
  moreover
    assume is-static m
    ultimately show ?case
      by contradiction
qed
qed

```

**lemma** *field-accessible-fromD*:

$$\llbracket G \vdash \text{membr of } C \text{ accessible-from } \text{accC}; \text{is-field membr} \rrbracket$$

$$\implies G \vdash \text{membr member-of } C \wedge$$

$$G \vdash (\text{Class } C) \text{ accessible-in } (\text{pid accC}) \wedge$$

$$G \vdash \text{membr in } C \text{ permits-acc-from accC}$$

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

**lemma** *field-accessible-from-permits-acc-inheritance*:

$$\llbracket G \vdash \text{membr of statC accessible-from accC}; \text{is-field membr}; G \vdash \text{dynC} \preceq_C \text{statC} \rrbracket$$

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

**by** (*auto dest*: *field-accessible-fromD intro*: *permits-acc-inheritance*)

**lemma** *accessible-fieldD*:

$$\llbracket G \vdash \text{membr of } C \text{ accessible-from accC}; \text{is-field membr} \rrbracket$$

$$\implies G \vdash \text{membr member-of } C \wedge$$

$$G \vdash (\text{Class } C) \text{ accessible-in } (\text{pid accC}) \wedge$$

$$G \vdash \text{membr in } C \text{ permits-acc-from accC}$$

**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**  $m \text{ subclseq-}D\text{-}C \text{ subclseq-}C\text{-}m$   
**show** *?thesis*  
**proof** (*induct*)  
**case** (*Immediate*  $m \ D$ )  
**assume**  $\text{declclass } m = D$  **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**  $G \vdash m \text{ member-of } C$   
**by** (*auto intro: members.Immediate*)  
**next**  
**case** (*Inherited*  $m \ D \ S$ )  
**assume** *member-of-D-props*:  
 $G \vdash m \text{ inheritable-in pid } D$   
 $G \vdash \text{memberid } m \text{ undeclared-in } D$   
 $G \vdash \text{Class } S \text{ accessible-in pid } D$   
 $G \vdash m \text{ member-of } S$   
**assume** *super*:  $G \vdash D \prec_{C1} S$   
**assume** *hyp*:  $\llbracket G \vdash S \preceq_C C; G \vdash C \preceq_C \text{declclass } m \rrbracket \implies G \vdash m \text{ member-of } C$   
**assume** *subclseq-C-m*:  $G \vdash C \preceq_C \text{declclass } m$   
**assume**  $G \vdash D \preceq_C C$   
**then show**  $G \vdash m \text{ member-of } C$   
**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*:  $G \vdash \text{old} \text{ member-of } C$  **and**  
*new*:  $G \vdash \text{new} \text{ member-of } D$  **and**  
*eqid*:  $\text{memberid new} = \text{memberid old}$  **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**  
 $ws$ :  $ws\text{-prog } G$   
**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 neq-new-old: new ≠ 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 new subclseq-D-C ws
  have  $G \vdash \text{new member-of } C$ 
    by (blast intro: member-of-inheritance)
  with eqid old
  have new=old
    by (blast intro: unique-member-of)
  with neq-new-old
  show ?thesis
    by contradiction
qed
then show ?thesis
proof (cases rule: subclseq-cases)
  case Eq
    assume  $\text{declclass new} = C$ 
    with new have  $G \vdash \text{new member-of } C$ 
      by (auto dest: member-of-member-of-declC)
    with eqid old
    have new=old
      by (blast intro: unique-member-of)
    with neq-new-old
    show ?thesis
      by contradiction
  next
    case Subcls
    assume  $G \vdash \text{declclass new} \prec_C C$ 
    with subclseq-D-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** *stat-acc:  $G \vdash \text{memb of statC accessible-from accC}$*  **and**

```

    is-field: is-field membr and
    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 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:  $G \vdash m \text{ of stat}C \text{ accessible-from acc}C$  and
    subclseq:  $G \vdash \text{dyn}C \preceq_C \text{stat}C$  and
    member-dynC:  $G \vdash m \text{ member-of dyn}C$  and
    dynC-acc:  $G \vdash (\text{Class dyn}C) \text{ accessible-in (pid acc}C)$ 
shows  $G \vdash m \text{ of dyn}C \text{ accessible-from acc}C$ 
proof –
  from stat-acc
  have member-statC:  $G \vdash m \text{ member-of stat}C$ 
    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

### types

$f_{\text{spec}} = v_{\text{name}} \times q_{\text{name}}$

### translations

$f_{\text{spec}} \leq (\text{type}) \ v_{\text{name}} \times q_{\text{name}}$

### constdefs

$\text{imethds} :: \text{prog} \Rightarrow q_{\text{name}} \Rightarrow (\text{sig}, q_{\text{name}} \times m_{\text{head}}) \text{ tables}$   
 $\text{imethds } G \ I$   
 $\equiv \text{iface-rec } (G, I)$   
 $\quad (\lambda I \ i \ ts. (\text{Un-tables } ts) \oplus \oplus$   
 $\quad \quad (\text{o2s} \circ \text{table-of } (\text{map } (\lambda(s, m). (s, I, m)) (\text{imethds } i))))$

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

### constdefs

$\text{accimethds} :: \text{prog} \Rightarrow p_{\text{name}} \Rightarrow q_{\text{name}} \Rightarrow (\text{sig}, q_{\text{name}} \times m_{\text{head}}) \text{ tables}$   
 $\text{accimethds } G \ \text{pack } I$   
 $\equiv \text{if } G \vdash \text{Iface } I \text{ accessible-in pack}$   
 $\quad \text{then imethds } G \ I$

*else*  $\lambda k. \{\}$

only returns imethds if the interface is accessible

### constdefs

*methd*:: *prog*  $\Rightarrow$  *qname*  $\Rightarrow$  (*sig*, *qname*  $\times$  *methd*) *table*

*methd* *G* *C*

$\equiv$  *class-rec* (*G*, *C*) *empty*  
 $(\lambda C\ c\ \text{subcls-mthds}.$   
 $\text{filter-tab } (\lambda \text{sig } m. G \vdash C \text{ inherits method sig } m)$   
 $\text{subcls-mthds}$   
 $++$   
 $\text{table-of } (\text{map } (\lambda (s, m). (s, C, m)) (\text{methods } c)))$

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

### constdefs

*accmethd*:: *prog*  $\Rightarrow$  *qname*  $\Rightarrow$  *qname*  $\Rightarrow$  (*sig*, *qname*  $\times$  *methd*) *table*

*accmethd* *G* *S* *C*

$\equiv$  *filter-tab* ( $\lambda \text{sig } m. G \vdash \text{method sig } m \text{ of } C \text{ accessible-from } S$ )  
 $(\text{methd } G\ C)$

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

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

### constdefs

*dynmethd*:: *prog*  $\Rightarrow$  *qname*  $\Rightarrow$  *qname*  $\Rightarrow$  (*sig*, *qname*  $\times$  *methd*) *table*

*dynmethd* *G* *statC* *dynC*

$\equiv$   $\lambda \text{sig}.$   
 $(\text{if } G \vdash \text{dynC} \preceq_C \text{statC}$   
 $\text{then } (\text{case } \text{methd } G\ \text{statC}\ \text{sig of}$   
 $\text{None} \Rightarrow \text{None}$   
 $| \text{Some statM}$   
 $\Rightarrow (\text{class-rec } (G, \text{dynC})\ \text{empty}$   
 $(\lambda C\ c\ \text{subcls-mthds}.$   
 $\text{subcls-mthds}$   
 $++$   
 $(\text{filter-tab}$   
 $(\lambda - \text{dynM}. G, \text{sig} \vdash \text{dynM overrides statM} \vee \text{dynM} = \text{statM})$   
 $(\text{methd } G\ C))$   
 $)\ \text{sig}$   
 $)$   
 $\text{else None})$

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

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

### constdefs

*dynimethd*:: *prog*  $\Rightarrow$  *qname*  $\Rightarrow$  *qname*  $\Rightarrow$  (*sig*, *qname*  $\times$  *methd*) *table*

*dynimethd* *G* *I* *dynC*

$\equiv \lambda \text{sig}. \text{if imethds } G\ I\ \text{sig} \neq \{\}$

*then methd G dynC sig*  
*else dynmethd G Object dynC sig*

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

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

#### constdefs

*dynlookup::prog*  $\Rightarrow$  *ref-ty*  $\Rightarrow$  *qname*  $\Rightarrow$  (*sig*, *qname*  $\times$  *methd*) *table*  
*dynlookup G statT dynC*  
 $\equiv$  (*case statT of*  
     *NullT*  $\Rightarrow$  *empty*  
     | *IfaceT I*  $\Rightarrow$  *dynimethd G I dynC*  
     | *ClassT statC*  $\Rightarrow$  *dynmethd G statC dynC*  
     | *ArrayT ty*  $\Rightarrow$  *dynmethd G Object dynC*)

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

#### constdefs

*fields::prog*  $\Rightarrow$  *qname*  $\Rightarrow$  ((*vname*  $\times$  *qname*)  $\times$  *field*) *list*  
*fields G C*  
 $\equiv$  *class-rec* (*G*, *C*) [] ( $\lambda C$  *c ts*. *map* ( $\lambda(n, t)$ . ((*n*, *C*), *t*)) (*cfields c*) @ *ts*)

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

#### constdefs

*accfield::prog*  $\Rightarrow$  *qname*  $\Rightarrow$  *qname*  $\Rightarrow$  (*vname*, *qname*  $\times$  *field*) *table*  
*accfield G S C*  
 $\equiv$  *let* *field-tab* = *table-of*((*map* ( $\lambda((n, d), f)$ . (*n*, (*d*, *f*)))) (*fields G C*)  
     *in* *filter-tab* ( $\lambda n$  (*declC*, *f*). *G*  $\vdash$  (*declC*, *fdecl* (*n*, *f*)) of *C* accessible-from *S*)

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

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

#### constdefs

*is-methd :: prog*  $\Rightarrow$  *qname*  $\Rightarrow$  *sig*  $\Rightarrow$  *bool*  
*is-methd G*  $\equiv$   $\lambda C$  *sig*. *is-class G C*  $\wedge$  *methd G C sig*  $\neq$  *None*

**constdefs** *efname::* ((*vname*  $\times$  *qname*)  $\times$  *field*)  $\Rightarrow$  (*vname*  $\times$  *qname*)  
*efname*  $\equiv$  *fst*

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

## 19 imethds

**lemma** *imethds-rec*:  $\llbracket \text{iface } G \ I = \text{Some } i; \text{ws-prog } G \rrbracket \implies$   
 $\text{imethds } G \ I = \text{Un-tables } ((\lambda J. \text{imethds } G \ J) \text{'set } (\text{isuperIfs } i)) \oplus \oplus$   
 $(o2s \circ \text{table-of } (\text{map } (\lambda(s, mh). (s, I, mh)) (\text{imethds } i)))$   
**apply** (*unfold imethds-def*)  
**apply** (*rule iface-rec [THEN trans]*)  
**apply** *auto*  
**done**

**lemma** *imethds-norec*:  
 $\llbracket \text{iface } G \ md = \text{Some } i; \text{ws-prog } G; \text{table-of } (\text{imethds } i) \text{ sig} = \text{Some } mh \rrbracket \implies$   
 $(md, mh) \in \text{imethds } G \ md \text{ sig}$   
**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*:  $\llbracket m \in \text{imethds } G \ I \text{ sig}; \text{ws-prog } G; \text{is-iface } G \ I \rrbracket \implies$   
 $(\exists i. \text{iface } G \ (\text{decliface } m) = \text{Some } i \wedge$   
 $\text{table-of } (\text{imethds } i) \text{ sig} = \text{Some } (mthd \ m)) \wedge$   
 $(I, \text{decliface } m) \in (\text{subint1 } G) \wedge m \in \text{imethds } G \ (\text{decliface } m) \text{ sig}$   
**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* [*consumes 3, case-names NewMethod InheritedMethod*]:  
**assumes** *im*:  $im \in \text{imethds } G \ I \text{ sig}$  **and**  
*ifI*:  $\text{iface } G \ I = \text{Some } i$  **and**  
*ws*:  $\text{ws-prog } G$  **and**  
*hyp-new*:  $\text{table-of } (\text{map } (\lambda(s, mh). (s, I, mh)) (\text{imethds } i)) \text{ sig}$   
 $= \text{Some } im \implies P$  **and**  
*hyp-inh*:  $\bigwedge J. \llbracket J \in \text{set } (\text{isuperIfs } i); im \in \text{imethds } G \ J \text{ sig} \rrbracket \implies P$   
**shows** *P*  
**proof** –  
**from** *ifI ws im hyp-new hyp-inh*  
**show** *P*  
**by** (*auto simp add: imethds-rec*)  
**qed**

## 20 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*:  
 $im \in \text{accimethds } G \text{ pack } I \text{ sig}$



$\Rightarrow im \in imethds\ G\ I\ sig \wedge G \vdash I\text{face } I\text{ accessible-in pack}$   
**by** (auto simp add: accimethds-def)

**lemma** accimethdsI:

$\llbracket im \in imethds\ G\ I\ sig; G \vdash I\text{face } I\text{ accessible-in pack} \rrbracket$

$\Rightarrow im \in accimethds\ G\ pack\ I\ sig$

**by** simp

## 21 methd

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

$= (if\ C = Object$   
 $\quad then\ empty$   
 $\quad else\ filter-tab\ (\lambda sig\ m. G \vdash C\ inherits\ method\ sig\ m)$   
 $\quad \quad (methd\ G\ (super\ c)))$   
 $++\ table-of\ (map\ (\lambda(s,m). (s,C,m))\ (methods\ c))$

**apply** (unfold methd-def)

**apply** (erule class-rec [THEN trans], assumption)

**apply** (simp)

**done**

**lemma** methd-norec:

$\llbracket class\ G\ declC = Some\ c; ws-prog\ G; table-of\ (methods\ c)\ sig = Some\ m \rrbracket$

$\Rightarrow methd\ G\ declC\ sig = Some\ (declC, m)$

**apply** (simp only: methd-rec)

**apply** (rule disjI1 [THEN map-add-Some-iff [THEN iffD2]])

**apply** (auto elim: table-of-map-SomeI)

**done**

**lemma** methd-declC:

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

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

**apply** (erule rev-mp)

**apply** (rule ws-subcls1-induct, assumption, assumption)

**apply** (subst methd-rec, assumption)

**apply** (case-tac Ca=Object)

**apply** (force elim: methd-norec)

**apply** simp

**apply** (case-tac table-of (map ( $\lambda(s, m). (s, Ca, m)$ ) (methods c)) sig)

**apply** (force intro: rtrancl-into-rtrancl2)

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

**done**

**lemma** methd-inheritedD:

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

$\Rightarrow (declclass\ m \neq C \longrightarrow G \vdash C\ inherits\ method\ sig\ m)$

**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*:

$\llbracket \text{methd } G \ C \ \text{sig} = \text{Some } m; \text{ws-prog } G; \text{is-class } G \ C \rrbracket \\ \implies G \vdash \text{Methd } \text{sig } m \ \text{declared-in } (\text{declclass } m)$   
**by** (*auto dest: methd-declC method-declared-inI*)

**lemma** *member-methd*:

**assumes** *member-of*:  $G \vdash \text{Methd } \text{sig } m \ \text{member-of } C$  **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**  $G \vdash \text{Methd } \text{sig } m \ \text{member-of } \text{Object}$

**then have**  $G \vdash \text{Methd } \text{sig } m \ \text{declared-in } \text{Object} \wedge \text{declclass } m = \text{Object}$

**by** (*cases set: members*) (*cases m, auto dest: subcls1D*)

**with** *ws Object*

**show** *?Methd*  $\text{Object}$

**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*  $G \ C = \text{Some } c$  **and**

*neq-C-Obj*:  $C \neq \text{Object}$  **and**

*hyp*:  $G \vdash \text{Methd } \text{sig } m \ \text{member-of } \text{super } c \implies \text{?Methd } (\text{super } c)$  **and**

*member-of*:  $G \vdash \text{Methd } \text{sig } m \ \text{member-of } C$

**from** *member-of*

**show** *?Methd*  $C$

**proof** (*cases*)

**case** (*Immediate membr*  $C a$ )

**then have**  $C a = C \ \text{membr} = \text{method } \text{sig } m$  **and**

$G \vdash \text{Methd } \text{sig } m \ \text{declared-in } C \ \text{declclass } m = C$

**by** (*cases m, auto*)

**with** *clsC*

**have** *table-of* (*map* ( $\lambda(s, m). (s, C, m)$ ) (*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 membr Ca S)
with clsC
have eq-Ca-C: Ca=C and
  undecl:  $G \vdash \text{mid sig undeclared-in } C$  and
  super:  $G \vdash \text{Methd sig m member-of (super c)}$ 
  by (auto dest: subcls1D)
from eq-Ca-C clsC undecl
have table-of (map ( $\lambda(s, m). (s, C, m)$ ) (methods c)) sig = None
  by (auto simp add: undeclared-in-def cdeclaredmethd-def
    intro: table-of-mapconst-NoneI)
moreover
from Inherited have  $G \vdash C \text{ inherits (method sig m)}$ 
  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  $G \implies \text{finite } \{\text{methd } G \ C \ \text{sig} \mid \text{sig } C. \text{is-class } G \ C\}$ 
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:
 $\llbracket \text{ws-prog } G; \text{is-class } G \ C \rrbracket \implies \text{finite } (\text{dom } (\text{methd } G \ C))$ 
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

```

## 22 accmethd

```

lemma accmethd-SomeD:
accmethd  $G \ S \ C \ \text{sig} = \text{Some } m$ 
 $\implies \text{methd } G \ C \ \text{sig} = \text{Some } m \wedge G \vdash \text{method sig } m \text{ of } C \text{ accessible-from } S$ 
by (auto simp add: accmethd-def dest: filter-tab-SomeD)

```

```

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

## 23 dynmethd

**lemma** *dynmethd-rec*:

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

**proof** –

**assume** *clsDynC*:  $\text{class } G \ \text{dynC} = \text{Some } c$  **and**

*ws*:  $\text{ws-prog } G$

**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*:  $\text{statC} = \text{Object} \ ..$

**show** *thesis*

**proof** (cases  $\text{methd } G \ \text{statC } \text{sig}$ )

**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**  $? \text{Dynmethd-def } \text{dynC } \text{sig} = ? \text{Dynmethd-rec } \text{dynC } c \ \text{sig}$

```

proof (cases  $G \vdash \text{dyn}C \preceq_C \text{stat}C$ )
  case False
  then show ?thesis by (simp add: dynmethod-def)
next
  case True
  note subclseq-dynC-statC = True
  show ?thesis
  proof (cases method G statC sig)
    case None then show ?thesis by (simp add: dynmethod-def)
  next
    case (Some statM)
    note statM = Some
    let ?filter C =
      filter-tab
      ( $\lambda$ - dynM.  $G, \text{sig} \vdash \text{dyn}M \text{ overrides } \text{stat}M \vee \text{dyn}M = \text{stat}M$ )
      (method G C)
    let ?class-rec C =
      (class-rec (G, C) empty
      ( $\lambda C$  c subcls-mthds. subcls-mthds ++ (?filter C)))
    from statM Subcls ws subclseq-dynC-statC
    have dynmethod-dynC-def:
      ?Dynmethod-def dynC sig =
        ((?class-rec (super c))
        ++
        (?filter dynC)) sig
    by (simp (no-asm-simp) only: dynmethod-def class-rec
      auto)
    show ?thesis
  proof (cases dynC = statC)
    case True
    with subclseq-dynC-statC statM dynmethod-dynC-def
    have ?Dynmethod-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:  $G \vdash (\text{super } c) \preceq_C \text{stat}C$ 
      by (blast dest: subclseq-superD)
    show ?thesis
  proof (cases method G dynC sig)
    case None
    then have ?filter dynC sig = None
      by (rule filter-tab-None)
    then have ?Dynmethod-def dynC sig = ?class-rec (super c) sig
      by (simp add: dynmethod-dynC-def)
    with subclseq-super-statC statM None
    have ?Dynmethod-def dynC sig = ?Dynmethod-def (super c) sig
      by (auto simp add: empty-def dynmethod-def)
    with None subclseq-dynC-statC statM
    show ?thesis
      by simp
  next
    case (Some dynM)
    note dynM = Some
    let ?Termination =  $G \vdash \text{qmdecl } \text{sig } \text{dyn}M \text{ overrides } \text{qmdecl } \text{sig } \text{stat}M \vee$ 
       $\text{dyn}M = \text{stat}M$ 

```

```

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:  $\llbracket \text{is-class } G \ C; \text{ws-prog } G \rrbracket$ 
 $\implies$  dynmethd  $G \ C \ C \text{ sig} = \text{methd } G \ C \text{ sig}$ 
apply (auto simp add: dynmethd-rec)
done

```

```

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

```

```

lemma dynmethd-Some-cases [consumes 3, case-names Static Overrides]:
  assumes      dynM: dynmethd  $G \ \text{statC } \text{dynC } \text{sig} = \text{Some } \text{dynM}$  and
    is-cls-dynC: is-class  $G \ \text{dynC}$  and
    ws: ws-prog  $G$  and
    hyp-static: methd  $G \ \text{statC } \text{sig} = \text{Some } \text{dynM} \implies P$  and
    hyp-override:  $\bigwedge \text{statM}. \llbracket \text{methd } G \ \text{statC } \text{sig} = \text{Some } \text{statM}; \text{dynM} \neq \text{statM};$ 
       $G, \text{sig} \vdash \text{dynM} \text{ overrides } \text{statM} \rrbracket \implies P$ 
  shows  $P$ 
proof -
  from is-cls-dynC obtain dc where clsDynC: class  $G \ \text{dynC} = \text{Some } dc$  by blast
  from clsDynC ws dynM hyp-static hyp-override
  show  $P$ 
proof (induct rule: ws-class-induct)
  case (Object co)
  with ws have statC = Object

```

```

  by (auto simp add: dynmethd-rec)
with ws Object show ?thesis by (auto simp add: dynmethd-C-C)
next
  case (Subcls C c)
  with ws show ?thesis
  by (auto simp add: dynmethd-rec)
qed
qed

```

**lemma** *no-override-in-Object*:

```

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

```

**lemma** *dynmethd-Some-rec-cases* [consumes 3,

*case-names Static Override Recursion*]:

```

  assumes
    dynM: dynmethd G statC dynC sig = Some dynM and
    clsDynC: class G dynC = Some c and
    ws: ws-prog G and
    hyp-static: methd G statC sig = Some dynM  $\implies$  P and
    hyp-override:  $\bigwedge$  statM.  $\llbracket$ methd G statC sig = Some statM;
      methd G dynC sig = Some dynM; statM ≠ dynM;
      G, sig  $\vdash$  dynM overrides statM $\rrbracket \implies$  P and
    hyp-recursion:  $\llbracket$ dynC ≠ Object;
      dynmethd G statC (super c) sig = Some dynM $\rrbracket \implies$  P
  shows P
proof -
  from clsDynC have is-class G dynC by simp
  note no-override-in-Object' = no-override-in-Object [OF dynM this ws]
  from ws clsDynC dynM hyp-static hyp-override hyp-recursion
  show ?thesis
  by (auto simp add: dynmethd-rec dest: no-override-in-Object')
qed

```

**lemma** *dynmethd-declC*:

$\llbracket$ dynmethd G statC dynC sig = Some m;

```

is-class G statC;ws-prog G
]] ==>
(∃ d. class G (declclass m)=Some d ∧ table-of (methods d) sig=Some (methd m)) ∧
G⊢ dynC ⊆C (declclass m) ∧ methd G (declclass m) sig = Some m
proof -
  assume is-cls-statC: is-class G statC
  assume ws: ws-prog G
  assume m: dynmethd G statC dynC sig = Some m
  from m
  have G⊢ dynC ⊆C statC by (auto simp add: dynmethd-def)
  from this is-cls-statC
  have is-cls-dynC: is-class G dynC by (rule subcls-is-class2)
  from is-cls-dynC ws m
  show ?thesis (is ?P dynC)
proof (induct rule: ws-class-induct')
  case (Object co)
  with ws have statC=Object by (auto simp add: dynmethd-rec)
  with ws Object
  show ?P Object
    by (auto simp add: dynmethd-C-C dest: methd-declC)
next
  case (Subcls dynC c)
  assume hyp: dynmethd G statC (super c) sig = Some m ==> ?P (super c) and
    clsDynC: class G dynC = Some c and
    m': dynmethd G statC dynC sig = Some m and
    neq-dynC-Obj: dynC ≠ Object
  from ws this obtain statM where
    subclseq-dynC-statC: G⊢ dynC ⊆C statC and
    statM: methd G statC sig = Some statM
  by (blast dest: dynmethdSomeD)
  from clsDynC neq-dynC-Obj
  have subclseq-dynC-super: G⊢ dynC ⊆C (super c)
  by (auto intro: subcls1I)
  from m' clsDynC ws
  show ?P dynC
proof (cases rule: dynmethd-Some-rec-cases)
  case Static
  with is-cls-statC ws subclseq-dynC-statC
  show ?thesis
    by (auto intro: rtrancl-trans dest: methd-declC)
next
  case Override
  with clsDynC ws
  show ?thesis
    by (auto dest: methd-declC)
next
  case Recursion
  with hyp subclseq-dynC-super
  show ?thesis
    by (auto intro: rtrancl-trans)
qed
qed
qed

```

**lemma** methd-Some-dynmethd-Some:

```

assumes statM: methd G statC sig = Some statM and
  subclseq: G⊢ dynC ⊆C statC and
  is-cls-statC: is-class G statC and

```



```

ws: ws-prog G
shows  $\exists \text{ dynM. dynmethd } G \text{ statC dynC sig} = \text{Some dynM}$ 
(is ?P dynC)
proof -
  from subclseq is-cls-statC
  have is-cl-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:  $G \vdash \text{super } dc \preceq_C \text{ statC} \implies ?P \text{ (super } dc)$ 
  assume subclseq':  $G \vdash \text{dynC} \preceq_C \text{ statC}$ 
  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  $G \vdash \text{dynC} \prec_C \text{ statC}$ 
  from this clsDynC'
  have  $G \vdash \text{super } dc \preceq_C \text{ statC}$  by (rule subcls-superD)
  with hyp ws clsDynC' subclseq' statM
  show ?thesis
  by (auto simp add: dynmethd-rec)
qed
qed
qed

```

**lemma** *dynmethd-cases* [consumes 4, case-names Static Overrides]:

```

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 and
  hyp-static:  $\text{dynmethd } G \text{ statC dynC sig} = \text{Some statM} \implies P$  and
  hyp-override:  $\bigwedge \text{ dynM. } \llbracket \text{dynmethd } G \text{ statC dynC sig} = \text{Some dynM};$ 
                  $\text{dynM} \neq \text{statM};$ 
                  $G, \text{sig} \vdash \text{dynM overrides statM} \rrbracket \implies P$ 

```

shows P

```

proof -
  from subclseq is-cls-statC
  have is-cl-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-cl-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

```

## 24 dynlookup

```

lemma dynlookup-cases [consumes 1, case-names NullT IfaceT ClassT ArrayT]:
   $\llbracket \text{dynlookup } G \text{ statT dynC sig} = x; \quad \begin{array}{l} \llbracket \text{statT} = \text{NullT} \quad ; \text{ empty sig} = x \\ \wedge I. \quad \llbracket \text{statT} = \text{IfaceT } I \quad ; \text{ dynimethd } G \text{ } I \quad \text{dynC sig} = x \rrbracket \implies P; \\ \wedge \text{statC}. \llbracket \text{statT} = \text{ClassT statC}; \text{ dynmethd } G \text{ statC dynC sig} = x \rrbracket \implies P; \\ \wedge \text{ty}. \llbracket \text{statT} = \text{ArrayT ty} \quad ; \text{ dynmethd } G \text{ Object dynC sig} = x \rrbracket \implies P \end{array} \rrbracket \implies P$ 
by (cases statT) (auto simp add: dynlookup-def)

```

## 25 fields

```

lemma fields-rec:  $\llbracket \text{class } G \text{ } C = \text{Some } c; \text{ ws-prog } G \rrbracket \implies$ 
   $\text{fields } G \text{ } C = \text{map } (\lambda(fn, ft). ((fn, C), ft)) \text{ (cfields } c) \text{ @}$ 
  (if C = Object then  $\llbracket$  else fields G (super c))
apply (simp only: fields-def)
apply (erule class-rec [THEN trans])
apply assumption
apply clarsimp

```

done

**lemma** *fields-norec*:

$\llbracket \text{class } G \text{ fd} = \text{Some } c; \text{ws-prog } G; \text{table-of } (\text{cfields } c) \text{ fn} = \text{Some } f \rrbracket$

$\implies \text{table-of } (\text{fields } G \text{ fd}) (\text{fn}, \text{fd}) = \text{Some } f$

**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*:

$\text{table-of } (\text{map } (\lambda(\text{fn}, \text{ft}). ((\text{fn}, C), \text{ft})) (\text{cfields } c)) \text{ efn} = \text{Some } f$

$\implies (\text{declclassf efn}) = C \wedge \text{table-of } (\text{cfields } c) (\text{fname efn}) = \text{Some } f$

**apply** (*case-tac efn*)

**by** *auto*

**lemma** *fields-declC*:

$\llbracket \text{table-of } (\text{fields } G \text{ C}) \text{ efn} = \text{Some } f; \text{ws-prog } G; \text{is-class } G \text{ C} \rrbracket \implies$

$(\exists d. \text{class } G (\text{declclassf efn}) = \text{Some } d \wedge$

$\text{table-of } (\text{cfields } d) (\text{fname efn}) = \text{Some } f) \wedge$

$G \vdash C \preceq_C (\text{declclassf efn}) \wedge \text{table-of } (\text{fields } G (\text{declclassf efn})) \text{ efn} = \text{Some } f$

**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 (split ( $\lambda \text{fn}. \text{Pair } (\text{fn}, \text{Ca}))) (\text{cfields } c)) \text{ efn}$* )

**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 (split ( $\lambda \text{fn}. \text{Pair } (\text{fn}, \text{Ca}))) (\text{cfields } c))$* )

**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*:  $\bigwedge y. \llbracket \text{ws-prog } G; \text{class } G \text{ C} = \text{Some } c; \text{cfields } c = [];$

$C \neq \text{Object} \longrightarrow \text{class } G (\text{super } c) = \text{Some } y \wedge \text{fields } G (\text{super } c) = [] \rrbracket \implies$

$\text{fields } G \text{ C} = []$

**apply** (*subst fields-rec*)

**apply** *assumption*

**apply** *auto*

done

**lemma** *fields-mono-lemma*:

```

 $\llbracket x \in \text{set } (\text{fields } G \ C); G \vdash D \preceq_C C; \text{ws-prog } G \rrbracket$ 
 $\implies x \in \text{set } (\text{fields } G \ D)$ 
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:
 $\llbracket (efn, fd) \in \text{set } (\text{fields } G \ (\text{super } c)); fc \in \text{set } (cfields \ c); \text{ws-prog } G;$ 
 $fname \ efn = fname \ fc; \text{declclassf } efn = C;$ 
 $\text{class } G \ C = \text{Some } c; C \neq \text{Object}; \text{class } G \ (\text{super } c) = \text{Some } d \rrbracket \implies R$ 
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:  $\llbracket \text{is-class } G \ C; \text{ws-prog } G;$ 
 $\bigwedge C \ c. \llbracket \text{class } G \ C = \text{Some } c \rrbracket \implies \text{unique } (cfields \ c) \rrbracket \implies$ 
 $\text{unique } (\text{fields } G \ C)$ 
apply (rule ws-subcls1-induct, assumption, assumption)
apply (subst fields-rec, assumption)
apply (auto intro!: unique-map-inj inj-onI
elim!: unique-append ws-unique-fields-lemma fields-norec)
done

```

## 26 accfield

```

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

```

```

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

```

```

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

```

## 27 is methd

**lemma** *is-methdI*:

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

**lemma** *is-methdD*:

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

**lemma** *finite-is-methd*:

$\text{ws-prog } G \implies \text{finite } (\text{Collect } (\text{split } (\text{is-methd } G)))$   
**apply** (*unfold is-methd-def*)  
**apply** (*subst SetCompr-Sigma-eq*)  
**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

**constdefs**

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

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

$\text{superclasses } G \ C$   
 $= (\text{if } (C = \text{Object})$   
 $\quad \text{then } \{\}$   
 $\quad \text{else insert } (\text{super } c) (\text{superclasses } G (\text{super } c)))$   
**apply** (*unfold superclasses-def*)  
**apply** (*erule class-rec [THEN trans], assumption*)  
**apply** (*simp*)  
**done**

**lemma** *superclasses-mono*:

$\llbracket G \vdash C \prec_C D; \text{ws-prog } G; \text{class } G \ C = \text{Some } c;$   
 $\quad \wedge \ C \ c. \llbracket \text{class } G \ C = \text{Some } c; C \neq \text{Object} \rrbracket \implies \exists \ sc. \text{class } G \ (\text{super } c) = \text{Some } sc;$   
 $\quad x \in \text{superclasses } G \ D$   
 $\rrbracket \implies x \in \text{superclasses } G \ C$   
**proof** –

**assume**  $\text{ws: ws-prog } G$  **and**  
 $\text{cls-C: class } G \ C = \text{Some } c$  **and**  
 $\text{wf: } \bigwedge \ C \ c. \llbracket \text{class } G \ C = \text{Some } c; C \neq \text{Object} \rrbracket$   
 $\implies \exists \ sc. \text{class } G \ (\text{super } c) = \text{Some } sc$

```

assume clsrel:  $G \vdash C \prec_C D$ 
thus  $\bigwedge c. \llbracket \text{class } G \ C = \text{Some } c; x \in \text{superclasses } G \ D \rrbracket \implies$ 
 $x \in \text{superclasses } G \ C$  (is PROP ?P C
 $\text{is } \bigwedge c. ?\text{CLS } C \ c \implies ?\text{SUP } D \implies ?\text{SUP } C$ )
proof (induct ?P C rule: converse-trancl-induct)
  fix C c
  assume  $G \vdash C \prec_{C1} D$  class G C = Some c  $x \in \text{superclasses } G \ D$ 
  with wf ws show ?SUP C
    by (auto intro: no-subcls1-Object
      simp add: superclasses-rec subcls1-def)
next
  fix C S c
  assume clsrel':  $G \vdash C \prec_{C1} S$   $G \vdash S \prec_C D$ 
    and hyp :  $\bigwedge s. \llbracket \text{class } G \ S = \text{Some } s; x \in \text{superclasses } G \ D \rrbracket$ 
 $\implies x \in \text{superclasses } G \ S$ 
    and cls-C': class G C = Some c
    and  $x: x \in \text{superclasses } G \ D$ 
  moreover note wf ws
  moreover from calculation
  have ?SUP 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 ?SUP C
    by (auto intro: no-subcls1-Object simp add: superclasses-rec)
qed
qed

lemma subclsEval:
 $\llbracket G \vdash C \prec_C D; \text{ws-prog } G; \text{class } G \ C = \text{Some } c;$ 
 $\bigwedge C \ c. \llbracket \text{class } G \ C = \text{Some } c; C \neq \text{Object} \rrbracket \implies \exists \text{sc. class } G \ (\text{super } c) = \text{Some } \text{sc}$ 
 $\rrbracket \implies D \in \text{superclasses } G \ C$ 
proof –
  note converse-trancl-induct
    = converse-trancl-induct [consumes 1, case-names Single Step]
  assume
    ws: ws-prog G and
    cls-C: class G C = Some c and
    wf:  $\bigwedge C \ c. \llbracket \text{class } G \ C = \text{Some } c; C \neq \text{Object} \rrbracket$ 
 $\implies \exists \text{sc. class } G \ (\text{super } c) = \text{Some } \text{sc}$ 
  assume clsrel:  $G \vdash C \prec_C D$ 
  thus  $\bigwedge c. \text{class } G \ C = \text{Some } c \implies D \in \text{superclasses } G \ C$ 
    (is PROP ?P C is  $\bigwedge c. ?\text{CLS } C \ c \implies ?\text{SUP } C$ )
  proof (induct ?P C rule: converse-trancl-induct)
    fix C c
    assume  $G \vdash C \prec_{C1} D$  class G C = Some c
    with ws wf show ?SUP C
      by (auto intro: no-subcls1-Object simp add: superclasses-rec subcls1-def)
  next
  fix C S c
  assume  $G \vdash C \prec_{C1} S$   $G \vdash S \prec_C D$ 
 $\bigwedge s. \text{class } G \ S = \text{Some } s \implies D \in \text{superclasses } G \ S$ 
 $\text{class } G \ C = \text{Some } c$ 
  with ws wf show ?SUP C
    by – (rule superclasses-mono,
      auto dest: no-subcls1-Object simp add: subcls1-def )
qed

```

**qed**

**end**





## Chapter 11

# WellType

## 28 Well-typedness of Java programs

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

improvements over Java Specification 1.0:

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

simplifications:

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

design issues:

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

**types** *lenv*

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

**record** *env* =

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

**translations**

*lenv* <= (*type*) (*lname*, *ty*) *table*  
*lenv* <= (*type*) *lname*  $\Rightarrow$  *ty option*  
*env* <= (*type*) (*prg*::*prog*, *cls*::*qtname*, *lcl*::*lenv*)  
*env* <= (*type*) (*prg*::*prog*, *cls*::*qtname*, *lcl*::*lenv*, ...::'*a*)

**syntax**

*pkg* :: *env*  $\Rightarrow$  *pname* — select the current package from an environment

**translations**

*pkg e* == *pid (cls e)*

### Static overloading: maximally specific methods

**types**

*emhead* = *ref-ty*  $\times$  *mhead*

— Some mnemonic selectors for *emhead*

**constdefs**

*declrefT* :: *emhead*  $\Rightarrow$  *ref-ty*  
*declrefT*  $\equiv$  *fst*

*mhd* :: *emhead*  $\Rightarrow$  *mhead*  
*mhd*  $\equiv$  *snd*

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

**by** (*simp add: declrefT-def*)

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

**by** (*simp add: mhd-def*)

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

**by** (*cases m*) (*simp add: member-is-static-simp mhd-def*)

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

**by** (*cases m*) *simp*

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

**by** (*cases m*) *simp*

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

**by** (*cases m*) *simp*

**consts**

*cmheads* :: *prog*  $\Rightarrow$  *qtname*  $\Rightarrow$  *qtname*  $\Rightarrow$  *sig*  $\Rightarrow$  *emhead set*

*Objectmheads* :: *prog*  $\Rightarrow$  *qtname*  $\Rightarrow$  *sig*  $\Rightarrow$  *emhead set*

*accObjectmheads*:: *prog*  $\Rightarrow$  *qtname*  $\Rightarrow$  *ref-ty*  $\Rightarrow$  *sig*  $\Rightarrow$  *emhead set*

*mheads* :: *prog*  $\Rightarrow$  *qtname*  $\Rightarrow$  *ref-ty*  $\Rightarrow$  *sig*  $\Rightarrow$  *emhead set*

**defs**

*cmheads-def*:

*cmheads G S C*

$\equiv \lambda sig. (\lambda (Cls, mthd). (ClassT Cls, (mhead mthd))) \text{ 'o2s (accmethd G S C sig)}$

*Objectmheads-def*:

*Objectmheads G S*

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

$\text{ 'o2s (filter-tab } (\lambda sig m. accmodi m \neq Private) (accmethd G S Object) sig)$

*accObjectmheads-def*:

*accObjectmheads G S T*

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

$\text{ then } Objectmheads G S$

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

**primrec**

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

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

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

$\text{ accObjectmheads G S (IfaceT I) sig)$

*mheads G S (ClassT C)* = *cmheads G S C*

*mheads G S (ArrayT T)* = *accObjectmheads G S (ArrayT T)*

**constdefs**

— applicable methods, cf. 15.11.2.1

*appl-methds* :: *prog*  $\Rightarrow$  *qtname*  $\Rightarrow$  *ref-ty*  $\Rightarrow$  *sig*  $\Rightarrow$  (*emhead*  $\times$  *ty list*) *set*

*appl-methds G S rt*  $\equiv \lambda sig.$

$\{(mh, pTs') \mid mh \ pTs'. mh \in mheads G S rt \ (name = name \ sig, parTs = pTs') \} \wedge$   
 $G \vdash (parTs \ sig) [\preceq] pTs' \}$

— more specific methods, cf. 15.11.2.2

*more-spec* :: *prog*  $\Rightarrow$  *emhead*  $\times$  *ty list*  $\Rightarrow$  *emhead*  $\times$  *ty list*  $\Rightarrow$  *bool*

*more-spec G*  $\equiv \lambda (mh, pTs). \lambda (mh', pTs'). G \vdash pTs [\preceq] pTs'$

— maximally specific methods, cf. 15.11.2.2

$max-spec \quad :: prog \Rightarrow qname \Rightarrow ref-ty \Rightarrow sig \Rightarrow (emhead \times ty\ list) \quad set$

$max-spec\ G\ S\ rt\ sig \equiv \{m. m \in appl-methds\ G\ S\ rt\ sig \wedge$   
 $(\forall m' \in appl-methds\ G\ S\ rt\ sig. more-spec\ G\ m'\ m \longrightarrow m' = m)\}$

**lemma** *max-spec2appl-meths*:

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

**by** (*auto simp: max-spec-def*)

**lemma** *appl-methsD*:  $(mh, pTs') \in appl-methds\ G\ S\ T\ (\llbracket name = mn, parTs = pTs \rrbracket) \implies$   
 $mh \in mheads\ G\ S\ T\ (\llbracket name = mn, parTs = pTs \rrbracket) \wedge G \vdash pTs[\preceq] pTs'$

**by** (*auto simp: appl-meths-def*)

**lemma** *max-spec2mheads*:

$max-spec\ G\ S\ rt\ (\llbracket name = mn, parTs = pTs \rrbracket) = insert\ (mh, pTs')\ A$

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

**apply** (*auto dest: equalityD2 subsetD max-spec2appl-meths appl-methsD*)

**done**

**constdefs**

*empty-dt* :: *dyn-ty*

*empty-dt*  $\equiv \lambda a. None$

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

*invmode*  $m\ e \equiv if\ is-static\ m$

*then Static*

*else if*  $e = Super$  *then SuperM* *else IntVir*

**lemma** *invmode-nonstatic* [*simp*]:

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

**apply** (*unfold invmode-def*)

**apply** (*simp (no-asm) add: member-is-static-simp*)

**done**

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

**apply** (*unfold invmode-def*)

**apply** (*simp (no-asm)*)

**done**

**lemma** *invmode-IntVir-eq*:  $(invmode\ m\ e = IntVir) = (\neg(is-static\ m) \wedge e \neq Super)$

**apply** (*unfold invmode-def*)

**apply** (*simp (no-asm)*)

**done**

**lemma** *Null-staticD*:

$a' = Null \longrightarrow (is-static\ m) \implies invmode\ m\ e = IntVir \longrightarrow a' \neq Null$

**apply** (*clarsimp simp add: invmode-IntVir-eq*)  
**done**

### Typing for unary operations

**consts** *unop-type* :: *unop*  $\Rightarrow$  *prim-ty*

**primrec**

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

**consts** *wt-unop* :: *unop*  $\Rightarrow$  *ty*  $\Rightarrow$  *bool*

**primrec**

*wt-unop* *UPlus* *t* = (*t* = *PrimT Integer*)  
*wt-unop* *UMinus* *t* = (*t* = *PrimT Integer*)  
*wt-unop* *UBitNot* *t* = (*t* = *PrimT Integer*)  
*wt-unop* *UNot* *t* = (*t* = *PrimT Boolean*)

### Typing for binary operations

**consts** *binop-type* :: *binop*  $\Rightarrow$  *prim-ty*

**primrec**

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

**consts** *wt-binop* :: *prog*  $\Rightarrow$  *binop*  $\Rightarrow$  *ty*  $\Rightarrow$  *ty*  $\Rightarrow$  *bool*

**primrec**

*wt-binop* *G Mul* *t1 t2* = ((*t1* = *PrimT Integer*)  $\wedge$  (*t2* = *PrimT Integer*))  
*wt-binop* *G Div* *t1 t2* = ((*t1* = *PrimT Integer*)  $\wedge$  (*t2* = *PrimT Integer*))  
*wt-binop* *G Mod* *t1 t2* = ((*t1* = *PrimT Integer*)  $\wedge$  (*t2* = *PrimT Integer*))  
*wt-binop* *G Plus* *t1 t2* = ((*t1* = *PrimT Integer*)  $\wedge$  (*t2* = *PrimT Integer*))  
*wt-binop* *G Minus* *t1 t2* = ((*t1* = *PrimT Integer*)  $\wedge$  (*t2* = *PrimT Integer*))  
*wt-binop* *G LShift* *t1 t2* = ((*t1* = *PrimT Integer*)  $\wedge$  (*t2* = *PrimT Integer*))  
*wt-binop* *G RShift* *t1 t2* = ((*t1* = *PrimT Integer*)  $\wedge$  (*t2* = *PrimT Integer*))  
*wt-binop* *G RShiftU* *t1 t2* = ((*t1* = *PrimT Integer*)  $\wedge$  (*t2* = *PrimT Integer*))  
*wt-binop* *G Less* *t1 t2* = ((*t1* = *PrimT Integer*)  $\wedge$  (*t2* = *PrimT Integer*))  
*wt-binop* *G Le* *t1 t2* = ((*t1* = *PrimT Integer*)  $\wedge$  (*t2* = *PrimT Integer*))  
*wt-binop* *G Greater* *t1 t2* = ((*t1* = *PrimT Integer*)  $\wedge$  (*t2* = *PrimT Integer*))

$wt\text{-}binop\ G\ Ge\quad t1\ t2 = ((t1 = PrimT\ Integer) \wedge (t2 = PrimT\ Integer))$   
 $wt\text{-}binop\ G\ Eq\quad t1\ t2 = (G \vdash t1 \preceq t2 \vee G \vdash t2 \preceq t1)$   
 $wt\text{-}binop\ G\ Neq\quad t1\ t2 = (G \vdash t1 \preceq t2 \vee G \vdash t2 \preceq t1)$   
 $wt\text{-}binop\ G\ BitAnd\ t1\ t2 = ((t1 = PrimT\ Integer) \wedge (t2 = PrimT\ Integer))$   
 $wt\text{-}binop\ G\ And\quad t1\ t2 = ((t1 = PrimT\ Boolean) \wedge (t2 = PrimT\ Boolean))$   
 $wt\text{-}binop\ G\ BitXor\ t1\ t2 = ((t1 = PrimT\ Integer) \wedge (t2 = PrimT\ Integer))$   
 $wt\text{-}binop\ G\ Xor\quad t1\ t2 = ((t1 = PrimT\ Boolean) \wedge (t2 = PrimT\ Boolean))$   
 $wt\text{-}binop\ G\ BitOr\quad t1\ t2 = ((t1 = PrimT\ Integer) \wedge (t2 = PrimT\ Integer))$   
 $wt\text{-}binop\ G\ Or\quad t1\ t2 = ((t1 = PrimT\ Boolean) \wedge (t2 = PrimT\ Boolean))$   
 $wt\text{-}binop\ G\ CondAnd\ t1\ t2 = ((t1 = PrimT\ Boolean) \wedge (t2 = PrimT\ Boolean))$   
 $wt\text{-}binop\ G\ CondOr\ t1\ t2 = ((t1 = PrimT\ Boolean) \wedge (t2 = PrimT\ Boolean))$

## Typing for terms

**types**  $tys = ty + ty\ list$

**translations**

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

## inductive

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

**where**

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

— well-typed statements

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

$| Expr: \llbracket E, dt \models e :: - T \rrbracket \Longrightarrow E, dt \models Expr\ e :: \surd$   
— cf. 14.6

$| Lab: E, dt \models c :: \surd \Longrightarrow E, dt \models l \cdot c :: \surd$

$| Comp: \llbracket E, dt \models c1 :: \surd; E, dt \models c2 :: \surd \rrbracket \Longrightarrow E, dt \models c1 ;; c2 :: \surd$

— cf. 14.8

$| If: \llbracket E, dt \models e :: - PrimT\ Boolean; E, dt \models c1 :: \surd; E, dt \models c2 :: \surd \rrbracket \Longrightarrow E, dt \models If(e)\ c1\ Else\ c2 :: \surd$

— cf. 14.10

$| Loop: \llbracket E, dt \models e :: - PrimT\ Boolean; E, dt \models c :: \surd \rrbracket \Longrightarrow E, dt \models l \cdot While(e)\ c :: \surd$   
— cf. 14.13, 14.15, 14.16

$| Jmp: E, dt \models Jmp\ jump :: \surd$

- cf. 14.16
- | *Throw*:  $\llbracket E, dt \models e :: - \text{Class } tn; \text{prg } E \vdash tn \preceq_C \text{SXcpt Throwable} \rrbracket \implies E, dt \models \text{Throw } e :: \checkmark$
- cf. 14.18
- | *Try*:  $\llbracket E, dt \models c1 :: \checkmark; \text{prg } E \vdash tn \preceq_C \text{SXcpt Throwable}; \text{lcl } E (V\text{Name } vn) = \text{None}; E (\text{lcl} := \text{lcl } E (V\text{Name } vn \mapsto \text{Class } tn)) \rrbracket, dt \models c2 :: \checkmark \rrbracket \implies E, dt \models \text{Try } c1 \text{ Catch}(tn \text{ } vn) \text{ } c2 :: \checkmark$
- cf. 14.18
- | *Fin*:  $\llbracket E, dt \models c1 :: \checkmark; E, dt \models c2 :: \checkmark \rrbracket \implies E, dt \models c1 \text{ Finally } c2 :: \checkmark$
- | *Init*:  $\llbracket \text{is-class } (\text{prg } E) \text{ } C \rrbracket \implies E, dt \models \text{Init } C :: \checkmark$ 
  - *Init* is created on the fly during evaluation (see Eval.thy). The class isn't necessarily accessible from the points *Init* is called. Therefor we only demand *is-class* and not *is-acc-class* here.
- well-typed expressions
- cf. 15.8
- | *NewC*:  $\llbracket \text{is-acc-class } (\text{prg } E) (\text{pkg } E) \text{ } C \rrbracket \implies E, dt \models \text{NewC } C :: - \text{Class } C$
- cf. 15.9
- | *NewA*:  $\llbracket \text{is-acc-type } (\text{prg } E) (\text{pkg } E) \text{ } T; E, dt \models i :: - \text{PrimT Integer} \rrbracket \implies E, dt \models \text{New } T[i] :: - T.[]$
- cf. 15.15
- | *Cast*:  $\llbracket E, dt \models e :: - T; \text{is-acc-type } (\text{prg } E) (\text{pkg } E) \text{ } T'; \text{prg } E \vdash T \preceq ? T' \rrbracket \implies E, dt \models \text{Cast } T' \text{ } e :: - T'$
- 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*:  $\llbracket \text{typeof } dt \text{ } x = \text{Some } T \rrbracket \implies E, dt \models \text{Lit } x :: - T$
- | *UnOp*:  $\llbracket E, dt \models e :: - T; \text{wt-unop unop } Te; T = \text{PrimT } (\text{unop-type unop}) \rrbracket \implies E, dt \models \text{UnOp unop } e :: - T$
- | *BinOp*:  $\llbracket E, dt \models e1 :: - T1; E, dt \models e2 :: - T2; \text{wt-binop } (\text{prg } E) \text{ binop } T1 \text{ } T2; T = \text{PrimT } (\text{binop-type binop}) \rrbracket \implies E, dt \models \text{BinOp binop } e1 \text{ } e2 :: - T$
- cf. 15.10.2, 15.11.1
- | *Super*:  $\llbracket \text{lcl } E \text{ This} = \text{Some } (\text{Class } C); C \neq \text{Object}; \text{class } (\text{prg } E) \text{ } C = \text{Some } c \rrbracket \implies E, dt \models \text{Super} :: - \text{Class } (\text{super } c)$
- cf. 15.13.1, 15.10.1, 15.12

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

$$E, dt \models \text{Acc } va ::= T$$
- cf. 15.25, 15.25.1
- | *Ass*:  $\llbracket E, dt \models va ::= T; va \neq \text{LVar } This;$   

$$E, dt \models v ::= T';$$
  

$$\text{prg } E \vdash T' \preceq T \rrbracket \implies$$
  

$$E, dt \models va ::= v ::= T'$$
- 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*:  $\llbracket E, dt \models e ::= \text{RefT statT};$   

$$E, dt \models ps ::= pTs;$$
  

$$\text{max-spec } (\text{prg } E) (\text{cls } E) \text{ statT } (\text{name} = mn, \text{parTs} = pTs)$$
  

$$= \{((\text{statDeclT}, m), pTs')\}$$
  

$$\rrbracket \implies$$
  

$$E, dt \models \{ \text{cls } E, \text{statT}, \text{invmode } m \} e \cdot mn(\{pTs'\}ps) ::= (\text{resTy } m)$$
- | *Method*:  $\llbracket \text{is-class } (\text{prg } E) \text{ } C;$   

$$\text{methd } (\text{prg } E) \text{ } C \text{ sig} = \text{Some } m;$$
  

$$E, dt \models \text{Body } (\text{declclass } m) (\text{stmt } (\text{mbody } (\text{methd } m))) ::= T \rrbracket \implies$$
  

$$E, dt \models \text{Method } C \text{ sig} ::= T$$
- 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 (ensured 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*:  $\llbracket \text{is-class } (\text{prg } E) \text{ } D;$   

$$E, dt \models \text{blk} ::= \sqrt{};$$
  

$$(\text{lcl } E) \text{ Result} = \text{Some } T;$$
  

$$\text{is-type } (\text{prg } E) \text{ } T \rrbracket \implies$$
  

$$E, dt \models \text{Body } D \text{ blk} ::= T$$
- 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 (ensured 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 *Method*.
- well-typed variables
- cf. 15.13.1
- | *LVar*:  $\llbracket \text{lcl } E \text{ } vn = \text{Some } T; \text{is-acc-type } (\text{prg } E) (\text{pkg } E) \text{ } T \rrbracket \implies$   

$$E, dt \models \text{LVar } vn ::= T$$
- cf. 15.10.1
- | *FVar*:  $\llbracket E, dt \models e ::= \text{Class } C;$   

$$\text{accfield } (\text{prg } E) (\text{cls } E) \text{ } C \text{ fn} = \text{Some } (\text{statDeclC}, f) \rrbracket \implies$$
  

$$E, dt \models \{ \text{cls } E, \text{statDeclC}, \text{is-static } f \} e \cdot \text{fn} ::= (\text{type } f)$$
- 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.???

| *Nil*:  $E, dt \models [] :: \div []$

— cf. 15.11.???

| *Cons*:  $\llbracket E, dt \models e :: -T; E, dt \models es :: \div Ts \rrbracket \implies E, dt \models e \# es :: \div T \# Ts$

**syntax**

-*wt*  $:: env \Rightarrow [term, tys] \Rightarrow bool \ (-|-::- [51, 51, 51] \ 50)$

-*wt-stmt*  $:: env \Rightarrow stmt \Rightarrow bool \ (-|-::<> [51, 51] \ 50)$

-*ty-expr*  $:: env \Rightarrow [expr, ty] \Rightarrow bool \ (-|-::- [51, 51, 51] \ 50)$

-*ty-var*  $:: env \Rightarrow [var, ty] \Rightarrow bool \ (-|-::- [51, 51, 51] \ 50)$

-*ty-exprs*  $:: env \Rightarrow [expr \ list, ty \ list] \Rightarrow bool \ (-|-::\#- [51, 51, 51] \ 50)$

**syntax** (*xsymbols*)

-*wt*  $:: env \Rightarrow [term, tys] \Rightarrow bool \ (+-::- [51, 51, 51] \ 50)$

-*wt-stmt*  $:: env \Rightarrow stmt \Rightarrow bool \ (+-::\surd [51, 51] \ 50)$

-*ty-expr*  $:: env \Rightarrow [expr, ty] \Rightarrow bool \ (+-::- [51, 51, 51] \ 50)$

-*ty-var*  $:: env \Rightarrow [var, ty] \Rightarrow bool \ (+-::- [51, 51, 51] \ 50)$

-*ty-exprs*  $:: env \Rightarrow [expr \ list, ty \ list] \Rightarrow bool \ (+-::\div- [51, 51, 51] \ 50)$

**translations**

$E \vdash t :: T == E, empty\_dt \models t :: T$

$E \vdash s :: \surd == E \vdash In1r \ s :: Inl \ (PrimT \ Void)$

$E \vdash e :: -T == E \vdash In1l \ e :: Inl \ T$

$E \vdash e :: T == E \vdash In2 \ e :: Inl \ T$

$E \vdash e :: \div T == E \vdash In3 \ e :: Inr \ T$

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

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

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

**declaration**  $\ll K \ (Simplifier.map\_ss \ (fn \ ss \Rightarrow ss \ delloop \ split\_all\_tac)) \gg$

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

$E, dt \models In2 \ (LVar \ vn) :: T$

$E, dt \models In2 \ (\{accC, statDeclC, s\}e..fn) :: T$

$E, dt \models In2 \ (e.[i]) :: T$

$E, dt \models In1l \ (NewC \ C) :: T$

$E, dt \models In1l \ (New \ T'[i]) :: T$

$E, dt \models In1l \ (Cast \ T' \ e) :: T$

$E, dt \models In1l \ (e \ InstOf \ T') :: T$

$E, dt \models In1l \ (Lit \ x) :: T$

$E, dt \models In1l \ (UnOp \ unop \ e) :: T$

$E, dt \models In1l \ (BinOp \ binop \ e1 \ e2) :: T$

$E, dt \models In1l \ (Super) :: T$

$E, dt \models In1l \ (Acc \ va) :: T$

$E, dt \models In1l \ (Ass \ va \ v) :: T$

$E, dt \models In1l \ (e0 \ ? \ e1 : e2) :: T$

$E, dt \models In1l \ (\{accC, statT, mode\}e.mn(\{pT^{\wedge}p\})) :: T$

$E, dt \models In1l \ (Methd \ C \ sig) :: T$

$E, dt \models In1l \ (Body \ D \ blk) :: T$

$E, dt \models In3 \ ([]) :: Ts$

$E, dt \models In3 \ (e \# es) :: Ts$

$E, dt \models In1r \ Skip :: x$

```

    E,dt|=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 split-if [split] split-if-asm [split]
declare split-paired-All [simp] split-paired-Ex [simp]
declaration << K (Simplifier.map-ss (fn ss => ss addloop (split-all-tac, split-all-tac))) >>

```

**lemma** *is-acc-class-is-accessible*:  
*is-acc-class G P C*  $\implies$  *G*⊢(*Class C*) *accessible-in P*  
**by** (auto simp add: *is-acc-class-def*)

**lemma** *is-acc-iface-is-iface*: *is-acc-iface G P I*  $\implies$  *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*⊢(*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*⊢*T* *accessible-in P*  
**by** (auto simp add: *is-acc-type-def*)

**lemma** *wt-Methd-is-methd*:  
*E*⊢*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*:  
 $\llbracket E,dt|=e::\text{--RefT statT}; E,dt|=ps::\text{--}pTs;$   
 $\text{max-spec (prg E) (cls E) statT } (\text{name}=mn, \text{parTs}=pTs)$   
 $= \{((\text{statDeclC}, m), pTs')\}; rT=(\text{resTy } m); \text{accC}=\text{cls } E;$   
 $\text{mode} = \text{invmode } m \text{ } e \rrbracket \implies E,dt|=\{\text{accC}, \text{statT}, \text{mode}\} e.mn(\{pTs'\}ps)::\text{--}rT$   
**by** (auto elim: *wt.Call*)

**lemma** *invocationTypeExpr-noClassD*:  
 $\llbracket E \vdash e::\text{--RefT statT} \rrbracket$

```

 $\impl (\forall \text{ stat } C. \text{ stat } T \neq \text{Class } T \text{ stat } C) \longrightarrow \text{invmode } m \ e \neq \text{Super } M$ 
proof –
  assume  $wt: E \vdash e :: -\text{Ref } T \text{ stat } T$ 
  show  $?thesis$ 
  proof ( $\text{cases } e = \text{Super}$ )
    case True
      with  $wt$  obtain  $C$  where  $\text{stat } T = \text{Class } T \ C$  by ( $\text{blast elim: wt-elim-cases}$ )
      then show  $?thesis$  by blast
    next
      case False then show  $?thesis$ 
      by ( $\text{auto simp add: invmode-def split: split-if-asm}$ )
  qed
qed

```

```

lemma wt-Super:
 $\llbracket \text{lcl } E \text{ This} = \text{Some } (\text{Class } C); C \neq \text{Object}; \text{class } (\text{prg } E) \ C = \text{Some } c; D = \text{super } c \rrbracket$ 
 $\impl E, dt \models \text{Super} :: -\text{Class } D$ 
by ( $\text{auto elim: wt.Super}$ )

```

```

lemma wt-FVar:
 $\llbracket E, dt \models e :: -\text{Class } C; \text{accfield } (\text{prg } E) \ (\text{cls } E) \ C \text{ fn} = \text{Some } (\text{statDecl } C, f);$ 
 $\text{sf} = \text{is-static } f; fT = (\text{type } f); \text{accC} = \text{cls } E \rrbracket$ 
 $\impl E, dt \models \{ \text{accC}, \text{statDecl } C, \text{sf} \} e.. \text{fn} :: = fT$ 
by ( $\text{auto dest: wt.FVar}$ )

```

```

lemma wt-init [iff]:  $E, dt \models \text{Init } C :: \surd = \text{is-class } (\text{prg } E) \ C$ 
by ( $\text{auto elim: wt-elim-cases intro: wt.Init}$ )

```

```

declare wt.Skip [iff]

```

```

lemma wt-StatRef:
 $\text{is-acc-type } (\text{prg } E) \ (\text{pkg } E) \ (\text{Ref } T \text{ rt}) \impl E \vdash \text{StatRef } \text{rt} :: -\text{Ref } T \text{ rt}$ 
apply ( $\text{rule wt.Cast}$ )
apply ( $\text{rule wt.Lit}$ )
apply ( $\text{simp (no-asm)}$ )
apply ( $\text{simp (no-asm-simp)}$ )
apply ( $\text{rule cast.widen}$ )
apply ( $\text{simp (no-asm)}$ )
done

```

```

lemma wt-Inj-elim:
 $\bigwedge E. E, dt \models t :: U \impl \text{case } t \text{ of}$ 
   $\text{In1 } ec \Rightarrow (\text{case } ec \text{ of}$ 
     $\text{Inl } e \Rightarrow \exists T. U = \text{Inl } T$ 
     $\mid \text{Inr } s \Rightarrow U = \text{Inl } (\text{Prim } T \text{ Void}))$ 
   $\mid \text{In2 } e \Rightarrow (\exists T. U = \text{Inl } T)$ 
   $\mid \text{In3 } e \Rightarrow (\exists T. U = \text{Inr } T)$ 
apply ( $\text{erule wt.induct}$ )
apply auto
done

```

— In the special syntax to distinguish the typing judgements for expressions, statements, variables and expression lists the kind of term corresponds to the kind of type in the end e.g. An statement (injection

*In3* into terms, always has type void (injection *Inl* into the generalised types. The following simplification procedures establish these kinds of correlation.

**lemma** *wt-expr-eq*:  $E, dt \models In1l\ t :: U = (\exists T. U = Inl\ T \wedge E, dt \models t :: -T)$   
**by** (*auto*, *frule wt-Inj-elim*, *auto*)

**lemma** *wt-var-eq*:  $E, dt \models In2\ t :: U = (\exists T. U = Inl\ T \wedge E, dt \models t :: T)$   
**by** (*auto*, *frule wt-Inj-elim*, *auto*)

**lemma** *wt-exprs-eq*:  $E, dt \models In3\ t :: U = (\exists Ts. U = Inr\ Ts \wedge E, dt \models t :: Ts)$   
**by** (*auto*, *frule wt-Inj-elim*, *auto*)

**lemma** *wt-stmt-eq*:  $E, dt \models In1r\ t :: U = (U = Inl(PrimT\ Void) \wedge E, dt \models t :: \surd)$   
**by** (*auto*, *frule wt-Inj-elim*, *auto*, *frule wt-Inj-elim*, *auto*)

**simplproc-setup** *wt-expr* ( $E, dt \models In1l\ t :: U$ ) =  $\langle\langle$   
 $fn\ - \Rightarrow fn\ - \Rightarrow fn\ ct \Rightarrow$   
 $(case\ Thm.term-of\ ct\ of$   
 $(-\ \$\ -\ \$\ -\ \$\ -\ \$\ (Const\ -\ \$\ -)) \Rightarrow NONE$   
 $| - \Rightarrow SOME\ (mk-meta-eq\ @\{thm\ wt-expr-eq\}) \rangle\rangle$

**simplproc-setup** *wt-var* ( $E, dt \models In2\ t :: U$ ) =  $\langle\langle$   
 $fn\ - \Rightarrow fn\ - \Rightarrow fn\ ct \Rightarrow$   
 $(case\ Thm.term-of\ ct\ of$   
 $(-\ \$\ -\ \$\ -\ \$\ -\ \$\ (Const\ -\ \$\ -)) \Rightarrow NONE$   
 $| - \Rightarrow SOME\ (mk-meta-eq\ @\{thm\ wt-var-eq\}) \rangle\rangle$

**simplproc-setup** *wt-exprs* ( $E, dt \models In3\ t :: U$ ) =  $\langle\langle$   
 $fn\ - \Rightarrow fn\ - \Rightarrow fn\ ct \Rightarrow$   
 $(case\ Thm.term-of\ ct\ of$   
 $(-\ \$\ -\ \$\ -\ \$\ -\ \$\ (Const\ -\ \$\ -)) \Rightarrow NONE$   
 $| - \Rightarrow SOME\ (mk-meta-eq\ @\{thm\ wt-exprs-eq\}) \rangle\rangle$

**simplproc-setup** *wt-stmt* ( $E, dt \models In1r\ t :: U$ ) =  $\langle\langle$   
 $fn\ - \Rightarrow fn\ - \Rightarrow fn\ ct \Rightarrow$   
 $(case\ Thm.term-of\ ct\ of$   
 $(-\ \$\ -\ \$\ -\ \$\ -\ \$\ (Const\ -\ \$\ -)) \Rightarrow NONE$   
 $| - \Rightarrow SOME\ (mk-meta-eq\ @\{thm\ wt-stmt-eq\}) \rangle\rangle$

**lemma** *wt-elim-BinOp*:  
 $\llbracket E, dt \models In1l\ (BinOp\ binop\ e1\ e2) :: T;$   
 $\bigwedge T1\ T2\ T3.$   
 $\llbracket E, dt \models e1 :: -T1; E, dt \models e2 :: -T2; wt-binop\ (prg\ E)\ binop\ T1\ T2;$   
 $E, dt \models (if\ b\ then\ In1l\ e2\ else\ In1r\ Skip) :: T3;$   
 $T = Inl\ (PrimT\ (binop-type\ binop)) \rrbracket$   
 $\implies P \rrbracket$   
 $\implies P$   
**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)]:  
 $\forall S \ T. G \vdash S \preceq T \longrightarrow G \vdash T \preceq S \longrightarrow S = T \implies E, dt \models t :: T \implies$   
 $G = \text{prg } E \longrightarrow (\forall T'. E, dt \models t :: T' \longrightarrow T = T')$   
**apply** (cases *E*, erule *wt.induct*)  
**apply** (safe del: *disjE*)  
**apply** (simp-all (no-asm-use) split del: *split-if-asm*)  
**apply** (safe del: *disjE*)

**apply** (tactic  $\ll \text{ALLGOALS } (\text{fn } i \Rightarrow \text{if } i = 11 \text{ then } \text{EVERY} [\text{thin-tac } ?E, dt \models e0 :: - \text{PrimT Boolean}, \text{thin-tac } ?E, dt \models e1 :: - ?T1, \text{thin-tac } ?E, dt \models e2 :: - ?T2] \ i \text{ else thin-tac All } ?P \ i) \gg)$

**apply** (tactic  $\ll \text{ALLGOALS } (\text{eresolve-tac } (\text{thms wt-elim-cases})) \gg)$   
**apply** (simp-all (no-asm-use) split del: *split-if-asm*)  
**apply** (erule-tac [12]  $V = \text{All } ?P \text{ in thin-rl}$ )  
**apply** ((blast del: *equalityCE* dest: *sym* [THEN *trans*])+)  
**done**

**lemma** *single-valued-tys*:  
 $\text{ws-prog } (\text{prg } E) \implies \text{single-valued } \{(t, T). E, dt \models t :: T\}$   
**apply** (unfold *single-valued-def*)  
**apply** *clarsimp*  
**apply** (rule *single-valued-tys-lemma*)  
**apply** (auto intro!: *widen-antisym*)  
**done**

**lemma** *typeof-empty-is-type* [rule-format (no-asm)]:  
 $\text{typeof } (\lambda a. \text{None}) \ v = \text{Some } T \longrightarrow \text{is-type } G \ T$   
**apply** (rule *val.induct*)  
**apply** *auto*  
**done**

**lemma** *typeof-is-type* [rule-format (no-asm)]:  
 $(\forall a. v \neq \text{Addr } a) \longrightarrow (\exists T. \text{typeof } dt \ v = \text{Some } T \wedge \text{is-type } G \ T)$   
**apply** (rule *val.induct*)  
**prefer** 5  
**apply** *fast*  
**apply** (simp-all (no-asm))  
**done**

**end**



## Chapter 12

# DefiniteAssignment

## 29 Definite Assignment

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

Definite Assignment Analysis (cf. 16)

The definite assignment analysis approximates the sets of local variables that will be assigned at a certain point of evaluation, and ensures that we will only read variables which previously were assigned. It should conform to the following idea: If the evaluation of a term completes normally (no abrupton (exception, break, continue, return) appeared) , the set of local variables calculated by the analysis is a subset of the variables that were actually assigned during evaluation.

To get more precise information about the sets of assigned variables the analysis includes the following optimisations:

- Inside of a while loop we also take care of the variables assigned before break statements, since the break causes the while loop to continue normally.
- For conditional statements we take care of constant conditions to statically determine the path of evaluation.
- Inside a distinct path of a conditional statements we know to which boolean value the condition has evaluated to, and so can retrieve more information about the variables assigned during evaluation of the boolean condition.

Since in our model of Java the return values of methods are stored in a local variable we also ensure that every path of (normal) evaluation will assign the result variable, or in the sense of real Java every path ends up in and return instruction.

Not covered yet:

- analysis of definite unassigned
- special treatment of final fields

### Correct nesting of jump statements

For definite assignment it becomes crucial, that jumps (break, continue, return) are nested correctly i.e. a continue jump is nested in a matching while statement, a break jump is nested in a proper label statement, a class initialiser does not terminate abruptly with a return. With this we can for example ensure that evaluation of an expression will never end up with a jump, since no breaks, continues or returns are allowed in an expression.

**consts** *jumpNestingOkS* :: *jump set*  $\Rightarrow$  *stmt*  $\Rightarrow$  *bool*

**primrec**

*jumpNestingOkS jmps (Skip)* = *True*

*jumpNestingOkS jmps (Expr e)* = *True*

*jumpNestingOkS jmps (j• s)* = *jumpNestingOkS* ( $\{j\} \cup$  *jmps*) *s*

*jumpNestingOkS jmps (c1;;c2)* = (*jumpNestingOkS jmps c1*  $\wedge$   
*jumpNestingOkS jmps c2*)

*jumpNestingOkS jmps (If(e) c1 Else c2)* = (*jumpNestingOkS jmps c1*  $\wedge$   
*jumpNestingOkS jmps c2*)

*jumpNestingOkS jmps (l• While(e) c)* = *jumpNestingOkS* ( $\{Cont\ l\} \cup$  *jmps*) *c*

— The label of the while loop only handles continue jumps. Breaks are only handled by *Lab*

*jumpNestingOkS jmps (Jmp j)* = ( $j \in$  *jmps*)

*jumpNestingOkS jmps (Throw e)* = *True*

*jumpNestingOkS jmps (Try c1 Catch(C vn) c2)* = (*jumpNestingOkS jmps c1*  $\wedge$   
*jumpNestingOkS jmps c2*)

*jumpNestingOkS jmps (c1 Finally c2)* = (*jumpNestingOkS jmps c1*  $\wedge$



*jumpNestingOkS jmps c2)*

*jumpNestingOkS jmps (Init C) = True*  
 — wellformedness of the program must enshure that for all initializers *jumpNestingOkS* holds  
 — Dummy analysis for intermediate smallstep term *FinA*  
*jumpNestingOkS jmps (FinA a c) = False*

**constdefs** *jumpNestingOk* :: *jump set*  $\Rightarrow$  *term*  $\Rightarrow$  *bool*  
*jumpNestingOk jmps t*  $\equiv$  (case *t* of  
     *In1 se*  $\Rightarrow$  (case *se* of  
         *Inl e*  $\Rightarrow$  *True*  
         | *Inr s*  $\Rightarrow$  *jumpNestingOkS jmps s*)  
     | *In2 v*  $\Rightarrow$  *True*  
     | *In3 es*  $\Rightarrow$  *True*)

**lemma** *jumpNestingOk-expr-simp* [*simp*]: *jumpNestingOk jmps (In1l e) = True*  
**by** (*simp add: jumpNestingOk-def*)

**lemma** *jumpNestingOk-expr-simp1* [*simp*]: *jumpNestingOk jmps (e::expr) = True*  
**by** (*simp add: inj-term-simps*)

**lemma** *jumpNestingOk-stmt-simp* [*simp*]:  
*jumpNestingOk jmps (In1r s) = jumpNestingOkS jmps s*  
**by** (*simp add: jumpNestingOk-def*)

**lemma** *jumpNestingOk-stmt-simp1* [*simp*]:  
*jumpNestingOk jmps (s::stmt) = jumpNestingOkS jmps s*  
**by** (*simp add: inj-term-simps*)

**lemma** *jumpNestingOk-var-simp* [*simp*]: *jumpNestingOk jmps (In2 v) = True*  
**by** (*simp add: jumpNestingOk-def*)

**lemma** *jumpNestingOk-var-simp1* [*simp*]: *jumpNestingOk jmps (v::var) = True*  
**by** (*simp add: inj-term-simps*)

**lemma** *jumpNestingOk-expr-list-simp* [*simp*]: *jumpNestingOk jmps (In3 es) = True*  
**by** (*simp add: jumpNestingOk-def*)

**lemma** *jumpNestingOk-expr-list-simp1* [*simp*]:  
*jumpNestingOk jmps (es::expr list) = True*  
**by** (*simp add: inj-term-simps*)

## Calculation of assigned variables for boolean expressions

### 30 Very restricted calculation fallback calculation

**consts** *the-LVar-name*:: *var*  $\Rightarrow$  *lname*  
**primrec**  
*the-LVar-name (LVar n) = n*

**consts** *assignsE* :: *expr*  $\Rightarrow$  *lname set*

$assignsV :: var \Rightarrow lname\ set$   
 $assignsEs :: expr\ list \Rightarrow lname\ set$

**primrec**

$assignsE\ (NewC\ c) = \{\}$   
 $assignsE\ (NewA\ t\ e) = assignsE\ e$   
 $assignsE\ (Cast\ t\ e) = assignsE\ e$   
 $assignsE\ (e\ InstOf\ r) = assignsE\ e$   
 $assignsE\ (Lit\ val) = \{\}$   
 $assignsE\ (UnOp\ unop\ e) = assignsE\ e$   
 $assignsE\ (BinOp\ binop\ e1\ e2) = (if\ binop=CondAnd\ \vee\ binop=CondOr$   
 $\quad then\ (assignsE\ e1)$   
 $\quad else\ (assignsE\ e1) \cup (assignsE\ e2))$   
 $assignsE\ (Super) = \{\}$   
 $assignsE\ (Acc\ v) = assignsV\ v$   
 $assignsE\ (v:=e)$   
 $\quad = (assignsV\ v) \cup (assignsE\ e) \cup$   
 $\quad (if\ \exists\ n.\ v=(LVar\ n)\ then\ \{the-LVar-name\ v\}$   
 $\quad \quad else\ \{\})$

$assignsE\ (b?\ e1 : e2) = (assignsE\ b) \cup ((assignsE\ e1) \cap (assignsE\ e2))$   
 $assignsE\ (\{accC, statT, mode\}objRef.mn(\{pTs\}args))$   
 $\quad = (assignsE\ objRef) \cup (assignsEs\ args)$

— Only dummy analysis for intermediate expressions *Method*, *Body*, *InsInitE* and *Callee*

$assignsE\ (Method\ C\ sig) = \{\}$   
 $assignsE\ (Body\ C\ s) = \{\}$   
 $assignsE\ (InsInitE\ s\ e) = \{\}$   
 $assignsE\ (Callee\ l\ e) = \{\}$

$assignsV\ (LVar\ n) = \{\}$   
 $assignsV\ (\{accC, statDeclC, stat\}objRef..fn) = assignsE\ objRef$   
 $assignsV\ (e1.[e2]) = assignsE\ e1 \cup assignsE\ e2$

$assignsEs\ [] = \{\}$   
 $assignsEs\ (e\#es) = assignsE\ e \cup assignsEs\ es$

**constdefs**  $assigns :: term \Rightarrow lname\ set$ 

$assigns\ t \equiv (case\ t\ of$   
 $\quad In1\ se \Rightarrow (case\ se\ of$   
 $\quad \quad Inl\ e \Rightarrow assignsE\ e$   
 $\quad \quad | Inr\ s \Rightarrow \{\})$   
 $\quad | In2\ v \Rightarrow assignsV\ v$   
 $\quad | In3\ es \Rightarrow assignsEs\ es)$

**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 (<v>) = assignsV v*  
**by** (*simp add: inj-term-simps*)

**lemma** *assigns-expr-list-simp* [simp]: *assigns (In3 es) = assignsEs es*  
**by** (*simp add: assigns-def*)

**lemma** *assigns-expr-list-simp1* [simp]: *assigns (<es>) = assignsEs es*  
**by** (*simp add: inj-term-simps*)

### 31 Analysis of constant expressions

**consts** *constVal* :: *expr*  $\Rightarrow$  *val option*

**primrec**

```

constVal (NewC c)      = None
constVal (NewA t e)    = None
constVal (Cast t e)    = None
constVal (Inst e r)    = None
constVal (Lit val)     = Some val
constVal (UnOp unop e) = (case (constVal e) of
  None  $\Rightarrow$  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*:  $\bigwedge v. P (Lit v)$  **and**

*hyp-UnOp*:  $\bigwedge unop e'. P e' \Longrightarrow P (UnOp unop e')$  **and**

*hyp-BinOp*:  $\bigwedge binop e1 e2. [P e1; P e2] \Longrightarrow P (BinOp binop e1 e2)$  **and**

```

hyp-CondL:  $\bigwedge b \text{ bv } e1 \ e2. \llbracket \text{constVal } b = \text{Some } bv; \text{the-Bool } bv; P \ b; P \ e1 \rrbracket$ 
            $\implies P \ (b? \ e1 : e2) \text{ and}$ 
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 True and  $\bigwedge v. \text{constVal } e = \text{Some } v \implies P \ e$  and True and True
  proof (induct  $x::\text{var}$  and  $e$  and  $s::\text{stmt}$  and  $es::\text{expr list}$ )
    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)
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

## 32 Main analysis for boolean expressions

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

**consts** *assigns-if*::  $\text{bool} \Rightarrow \text{expr} \Rightarrow \text{lname set}$

**primrec**

<i>assigns-if</i> $b \ (NewC \ c)$	$= UNIV$ — can never evaluate to Boolean
<i>assigns-if</i> $b \ (NewA \ t \ e)$	$= UNIV$ — can never evaluate to Boolean
<i>assigns-if</i> $b \ (Cast \ t \ e)$	$= \text{assigns-if } b \ e$
<i>assigns-if</i> $b \ (Inst \ e \ r)$	$= \text{assignsE } e$ — Inst has type Boolean but $e$ is a reference type
<i>assigns-if</i> $b \ (Lit \ val)$	$= (if \ val = \text{Bool } b \text{ then } \{\} \text{ else } UNIV)$
<i>assigns-if</i> $b \ (UnOp \ unop \ e)$	$= (\text{case } \text{constVal } (UnOp \ unop \ e) \text{ of}$ $None \Rightarrow (if \ unop = UNot$

```

                                then assigns-if ( $\neg b$ ) e
                                else UNIV)
      | Some v  $\Rightarrow$  (if v=Bool b
                      then {}
                      else UNIV))
assigns-if b (BinOp binop e1 e2)
= (case constVal (BinOp binop e1 e2) of
   None  $\Rightarrow$  (if binop=CondAnd then
                  (case b of
                     True  $\Rightarrow$  assigns-if True e1  $\cup$  assigns-if True e2
                     | False  $\Rightarrow$  assigns-if False e1  $\cap$ 
                        (assigns-if True e1  $\cup$  assigns-if False e2))
                  else
                    (if binop=CondOr then
                     (case b of
                        True  $\Rightarrow$  assigns-if True e1  $\cap$ 
                           (assigns-if False e1  $\cup$  assigns-if True e2)
                        | False  $\Rightarrow$  assigns-if False e1  $\cup$  assigns-if False e2)
                     else assignsE e1  $\cup$  assignsE e2))
   | Some v  $\Rightarrow$  (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)  $\cup$ 
                             (case (constVal c) of
                                None  $\Rightarrow$  (assigns-if b e1)  $\cap$ 
                                    (assigns-if b e2)
                                | Some bv  $\Rightarrow$  (case the-Bool bv of
                                    True  $\Rightarrow$  assigns-if b e1
                                    | False  $\Rightarrow$  assigns-if b e2))
assigns-if b ({accC,statT,mode}objRef·mn({pTs}args))
= assignsE ({accC,statT,mode}objRef·mn({pTs}args))
— Only dummy analysis for intermediate expressions Methd, Body, InsInitE and Callee
assigns-if b (Methd C sig) = {}
assigns-if b (Body C s)   = {}
assigns-if b (InsInitE s e) = {}
assigns-if b (Callee l e) = {}

```

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

**assumes** boolConst: constVal e = Some (Bool b) (**is** ?Const b e)

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

**proof** —

**have** True **and**  $\bigwedge b. ?Const b e \implies ?Ass b e$  **and** True **and** True

**proof** (induct - **and** e **and** - **and** - rule: var-expr-stmt.inducts)

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 =  $\langle constVal\ (c\ ?\ e1 : e2) = Some\ (Bool\ b) \rangle$ 
then obtain bv where bv: constVal c = Some bv
  by simp
hence emptyC: assignsE c = {} by (rule assignsE-const-simp)
show ?case
proof (cases the-Bool bv)
  case True
    with const bv
    have ?Const b e1 by simp
    hence ?Ass b e1 by (rule hyp-e1)
    with emptyC bv True
    show ?thesis
    by simp
  next
    case False
    with const bv
    have ?Const b e2 by simp
    hence ?Ass b e2 by (rule hyp-e2)
    with emptyC bv False
    show ?thesis
    by simp
  qed
qed (simp-all)
with boolConst
show ?thesis
  by blast
qed

lemma assigns-if-const-not-b-simp:
  assumes boolConst: constVal e = Some (Bool b) (is ?Const b e)
  shows assigns-if ( $\neg b$ ) e = UNIV (is ?Ass b e)
proof -
  have True and  $\bigwedge b. ?Const\ b\ e \implies ?Ass\ b\ e$  and True and True
  proof (induct - and e and - and - rule: var-expr-stmt.inducts)
    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 =  $\langle constVal\ (c\ ?\ e1 : e2) = Some\ (Bool\ b) \rangle$ 
    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

```

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

```

constdefs
union-ts:: ('a,'b) tables  $\Rightarrow$  ('a,'b) tables  $\Rightarrow$  ('a,'b) tables
  (-  $\Rightarrow \cup$  - [67,67] 65)
A  $\Rightarrow \cup$  B  $\equiv \lambda k. A\ k \cup B\ k$ 

```

```

constdefs
intersect-ts:: ('a,'b) tables  $\Rightarrow$  ('a,'b) tables  $\Rightarrow$  ('a,'b) tables
  (-  $\Rightarrow \cap$  - [72,72] 71)
A  $\Rightarrow \cap$  B  $\equiv \lambda k. A\ k \cap B\ k$ 

```

```

constdefs
all-union-ts:: ('a,'b) tables  $\Rightarrow$  'b set  $\Rightarrow$  ('a,'b) tables
  (infixl  $\Rightarrow \cup_{\forall}$  40)
A  $\Rightarrow \cup_{\forall}$  B  $\equiv \lambda k. A\ k \cup B$ 

```

#### 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  $\implies$  c  $\in$  (A  $\Rightarrow \cup$  B) k
  by simp

```

```

lemma union-tsI2 [elim?]: c  $\in$  B k  $\implies$  c  $\in$  (A  $\Rightarrow \cup$  B) k
  by simp

```

```

lemma union-tsCI [intro!]: (c  $\notin$  B k  $\implies$  c  $\in$  A k)  $\implies$  c  $\in$  (A  $\Rightarrow \cup$  B) k
  by auto

```

```

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

**types** *breakass* = (*label*, *lname*) *tables*

— Mapping from a break label, to the set of variables that will be assigned if the evaluation terminates with this break

**record** *assigned* =

*nrm* :: *lname set* — Definetly assigned variables for normal completion

*brk* :: *breakass* — Definetly assigned variables for abrupt completion with a break

**constdefs** *rmlab* :: '*a*  $\Rightarrow$  ('*a*, '*b*) *tables*  $\Rightarrow$  ('*a*, '*b*) *tables*

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

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



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

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

$da :: env \Rightarrow lname \text{ set} \Rightarrow term \Rightarrow assigned \Rightarrow bool$  (-|-|-|-|- [65,65,65,65] 71)

### where

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

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

$\Rightarrow$

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

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

$\Rightarrow$

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

|  $Comp: \llbracket Env \vdash B \gg \langle c1 \rangle \gg C1; Env \vdash \text{nrm } C1 \gg \langle c2 \rangle \gg C2; \text{nrm } A = \text{nrm } C2; \text{brk } A = (\text{brk } C1) \Rightarrow \cap (\text{brk } C2) \rrbracket$

$\Rightarrow$

$Env \vdash B \gg \langle c1;; c2 \rangle \gg A$

|  $If: \llbracket Env \vdash B \gg \langle e \rangle \gg E;$

$Env \vdash (B \cup \text{assigns-if True } e) \gg \langle c1 \rangle \gg C1;$

$Env \vdash (B \cup \text{assigns-if False } e) \gg \langle c2 \rangle \gg C2;$

$\text{nrm } A = \text{nrm } C1 \cap \text{nrm } C2;$

$\text{brk } A = \text{brk } C1 \Rightarrow \cap \text{brk } C2 \rrbracket$

$\Rightarrow$

$Env \vdash B \gg \langle If(e) c1 Else c2 \rangle \gg A$

— Note that  $E$  is not further used, because we take the specialized sets that also consider if the expression evaluates to true or false. Inside of  $e$  there is no **break** or **finally**, so the break map of  $E$  will be the trivial one. So  $Env \vdash B \gg \langle e \rangle \gg E$  is just used to ensure the definite assignment in expression  $e$ . Notice the implicit analysis of a constant boolean expression  $e$  in this rule. For example, if  $e$  is constantly *True* then *assigns-if False e* = *UNIV* and therefor  $\text{nrm } C2 = UNIV$ . So finally  $\text{nrm } A = \text{nrm } C1$ . For the break maps this trick workd too, because the trival break map will map all labels to *UNIV*. In the example, if no break occurs in  $c2$  the break maps will trivially map to *UNIV* and if a break occurs it will map to *UNIV* too, because *assigns-if False e* = *UNIV*. So in the intersection of the break maps the path  $c2$  will have no contribution.

|  $Loop: \llbracket Env \vdash B \gg \langle e \rangle \gg E;$

$Env \vdash (B \cup \text{assigns-if True } e) \gg \langle c \rangle \gg C;$

$\text{nrm } A = \text{nrm } C \cap (B \cup \text{assigns-if False } e);$

$\text{brk } A = \text{brk } C \rrbracket$

$\Rightarrow$

$Env \vdash B \gg \langle l \cdot While(e) c \rangle \gg A$

— The *Loop* rule resembles some of the ideas of the *If* rule. For the  $\text{nrm } A$  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 ( $\text{nrm } C$ ) or with a break. But in this model, the label  $l$  of the loop only handles continue labels, not break labels. The break label will be handled by an enclosing *Lab* statement. So we don't have to handle the breaks specially.

|  $Jmp: \llbracket \text{jump} = \text{Ret} \longrightarrow \text{Result} \in B;$

$\text{nrm } A = UNIV;$

$\text{brk } A = (\text{case jump of}$

$\text{Break } l \Rightarrow \lambda k. \text{ if } k=l \text{ then } B \text{ else } UNIV$

|  $\text{Cont } l \Rightarrow \lambda k. UNIV$

|  $\text{Ret} \Rightarrow \lambda k. UNIV) \rrbracket$

$$\Rightarrow$$

$$Env \vdash B \gg \langle \text{Jump } jump \rangle \gg A$$

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

$$\begin{aligned} | \text{Throw: } & \llbracket Env \vdash B \gg \langle e \rangle \gg E; nrm\ A = UNIV; brk\ A = (\lambda\ l.\ UNIV) \rrbracket \\ & \Rightarrow Env \vdash B \gg \langle \text{Throw } e \rangle \gg A \end{aligned}$$

$$\begin{aligned} | \text{Try: } & \llbracket Env \vdash B \gg \langle c1 \rangle \gg C1; \\ & Env(\llbracket lcl := lcl\ Env(VName\ vn \mapsto Class\ C) \rrbracket) \vdash (B \cup \{VName\ vn\}) \gg \langle c2 \rangle \gg C2; \\ & nrm\ A = nrm\ C1 \cap nrm\ C2; \\ & brk\ A = brk\ C1 \Rightarrow \cap\ brk\ C2 \rrbracket \\ & \Rightarrow Env \vdash B \gg \langle \text{Try } c1\ \text{Catch}(C\ vn)\ c2 \rangle \gg A \end{aligned}$$

$$\begin{aligned} | \text{Fin: } & \llbracket Env \vdash B \gg \langle c1 \rangle \gg C1; \\ & Env \vdash B \gg \langle c2 \rangle \gg C2; \\ & nrm\ A = nrm\ C1 \cup nrm\ C2; \\ & brk\ A = ((brk\ C1) \Rightarrow \cup_{\vee}\ (nrm\ C2)) \Rightarrow \cap\ (brk\ C2) \rrbracket \\ & \Rightarrow \\ & Env \vdash B \gg \langle c1\ \text{Finally } c2 \rangle \gg A \end{aligned}$$

— The set of assigned variables before execution  $c2$  are the same as before execution  $c1$ , because  $c1$  could throw an exception and so we can't guarantee that any variable will be assigned in  $c1$ . The *Finally* statement completes normally if both  $c1$  and  $c2$  complete normally. If  $c1$  completes abruptly with a break, then  $c2$  also will be executed and may terminate normally or with a break. The overall break map then is the intersection of the maps of both paths. If  $c2$  terminates normally we have to extend all break sets in  $brk\ C1$  with  $nrm\ C2$  ( $\Rightarrow \cup_{\vee}$ ). If  $c2$  exits with a break this break will appear in the overall result state. We don't know if  $c1$  completed normally or abruptly (maybe with an exception not only a break) so  $c1$  has no contribution to the break map following this path.

— Evaluation of expressions and the break sets of definite assignment: Thinking of a Java expression we assume that we can never have a break statement inside of an expression. So for all expressions the break sets could be set to the trivial one:  $\lambda l.\ UNIV$ . But we can't trivially prove, that evaluating an expression will never result in a break, although Java expressions already syntactically don't allow nested statements in them. The reason are the nested class initialization statements which are inserted by the evaluation rules. So to prove the absence of a break we need to ensure, that the initialization statements will never end up in a break. In a wellformed initialization statement, of course, 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*

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

— 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.

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

$$\begin{aligned} | \text{NewA: } & Env \vdash B \gg \langle e \rangle \gg A \\ & \Rightarrow \\ & Env \vdash B \gg \langle \text{New } T[e] \rangle \gg A \end{aligned}$$

$$\begin{aligned} | \text{Cast: } & Env \vdash B \gg \langle e \rangle \gg A \\ & \Rightarrow \\ & Env \vdash B \gg \langle \text{Cast } T\ e \rangle \gg A \end{aligned}$$

- | *Inst*:  $Env \vdash B \gg \langle e \rangle \gg A$   
 $\implies$   
 $Env \vdash B \gg \langle e \text{ InstOf } T \rangle \gg A$
  - | *Lit*:  $Env \vdash B \gg \langle Lit \ v \rangle \gg (\llbracket nrm=B, brk=\lambda l. UNIV \rrbracket)$
  - | *UnOp*:  $Env \vdash B \gg \langle e \rangle \gg A$   
 $\implies$   
 $Env \vdash B \gg \langle UnOp \ unop \ e \rangle \gg A$
  - | *CondAnd*:  $\llbracket Env \vdash B \gg \langle e1 \rangle \gg E1; Env \vdash (B \cup \text{assigns-if True } e1) \gg \langle e2 \rangle \gg E2;$   
 $nrm \ A = B \cup (\text{assigns-if True } (BinOp \ CondAnd \ e1 \ e2) \cap$   
 $\text{assigns-if False } (BinOp \ CondAnd \ e1 \ e2));$   
 $brk \ A = (\lambda l. UNIV) \rrbracket$   
 $\implies$   
 $Env \vdash B \gg \langle BinOp \ CondAnd \ e1 \ e2 \rangle \gg A$
  - | *CondOr*:  $\llbracket Env \vdash B \gg \langle e1 \rangle \gg E1; Env \vdash (B \cup \text{assigns-if False } e1) \gg \langle e2 \rangle \gg E2;$   
 $nrm \ A = B \cup (\text{assigns-if True } (BinOp \ CondOr \ e1 \ e2) \cap$   
 $\text{assigns-if False } (BinOp \ CondOr \ e1 \ e2));$   
 $brk \ A = (\lambda l. UNIV) \rrbracket$   
 $\implies$   
 $Env \vdash B \gg \langle BinOp \ CondOr \ e1 \ e2 \rangle \gg A$
  - | *BinOp*:  $\llbracket Env \vdash B \gg \langle e1 \rangle \gg E1; Env \vdash nrm \ E1 \gg \langle e2 \rangle \gg A;$   
 $binop \neq CondAnd; binop \neq CondOr \rrbracket$   
 $\implies$   
 $Env \vdash B \gg \langle BinOp \ binop \ e1 \ e2 \rangle \gg A$
  - | *Super*:  $This \in B$   
 $\implies$   
 $Env \vdash B \gg \langle Super \rangle \gg (\llbracket nrm=B, brk=\lambda l. UNIV \rrbracket)$
  - | *AccLVar*:  $\llbracket vn \in B;$   
 $nrm \ A = B; brk \ A = (\lambda k. UNIV) \rrbracket$   
 $\implies$   
 $Env \vdash B \gg \langle Acc \ (LVar \ vn) \rangle \gg A$
- To properly access a local variable we have to test the definite assignment here. The variable must occur in the set  $B$
- | *Acc*:  $\llbracket \forall \ vn. v \neq LVar \ vn;$   
 $Env \vdash B \gg \langle v \rangle \gg A \rrbracket$   
 $\implies$   
 $Env \vdash B \gg \langle Acc \ v \rangle \gg A$
  - | *AssLVar*:  $\llbracket Env \vdash B \gg \langle e \rangle \gg E; nrm \ A = nrm \ E \cup \{vn\}; brk \ A = brk \ E \rrbracket$   
 $\implies$   
 $Env \vdash B \gg \langle (LVar \ vn) := e \rangle \gg A$
  - | *Ass*:  $\llbracket \forall \ vn. v \neq LVar \ vn; Env \vdash B \gg \langle v \rangle \gg V; Env \vdash nrm \ V \gg \langle e \rangle \gg A \rrbracket$   
 $\implies$   
 $Env \vdash B \gg \langle v := e \rangle \gg A$
  - | *CondBool*:  $\llbracket Env \vdash (c \ ? \ e1 : e2) :: \neg (PrimT \ Boolean);$   
 $Env \vdash B \gg \langle c \rangle \gg C;$   
 $Env \vdash (B \cup \text{assigns-if True } c) \gg \langle e1 \rangle \gg E1;$   
 $Env \vdash (B \cup \text{assigns-if False } c) \gg \langle e2 \rangle \gg E2;$   
 $nrm \ A = B \cup (\text{assigns-if True } (c \ ? \ e1 : e2) \cap$   
 $\text{assigns-if False } (c \ ? \ e1 : e2)); \rrbracket$

$$\begin{aligned}
& \text{brk } A = (\lambda l. \text{UNIV}) \\
& \implies \\
& \text{Env} \vdash B \gg \langle c \text{ ? } e1 : e2 \rangle \gg A
\end{aligned}$$

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

$$\begin{aligned}
& | \text{Call: } \llbracket \text{Env} \vdash B \gg \langle e \rangle \gg E; \text{Env} \vdash \text{nrm } E \gg \langle \text{args} \rangle \gg A \rrbracket \\
& \implies \\
& \text{Env} \vdash B \gg \langle \{ \text{accC}, \text{statT}, \text{mode} \} e \cdot \text{mn}(\{ pTs \} \text{args}) \rangle \gg A
\end{aligned}$$

— 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.

$$\begin{aligned}
& | \text{Method: } \llbracket \text{method } (\text{prg } \text{Env}) \text{ } D \text{ sig} = \text{Some } m; \\
& \quad \text{Env} \vdash B \gg \langle \text{Body } (\text{declclass } m) (\text{stmt } (\text{mbody } (\text{mthd } m))) \rangle \gg A \\
& \quad \rrbracket \\
& \implies \\
& \text{Env} \vdash B \gg \langle \text{Method } D \text{ sig} \rangle \gg A
\end{aligned}$$

$$\begin{aligned}
& | \text{Body: } \llbracket \text{Env} \vdash B \gg \langle c \rangle \gg C; \text{jumpNestingOkS } \{ \text{Ret} \} c; \text{Result} \in \text{nrm } C; \\
& \quad \text{nrm } A = B; \text{brk } A = (\lambda l. \text{UNIV}) \\
& \implies \\
& \text{Env} \vdash B \gg \langle \text{Body } D c \rangle \gg A
\end{aligned}$$

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

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

$$\begin{aligned}
& | \text{FVar: } \text{Env} \vdash B \gg \langle e \rangle \gg A \\
& \implies \\
& \text{Env} \vdash B \gg \langle \{ \text{accC}, \text{statDeclC}, \text{stat} \} e \cdot \text{fn} \rangle \gg A
\end{aligned}$$

$$\begin{aligned}
& | \text{AVar: } \llbracket \text{Env} \vdash B \gg \langle e1 \rangle \gg E1; \text{Env} \vdash \text{nrm } E1 \gg \langle e2 \rangle \gg A \rrbracket \\
& \implies \\
& \text{Env} \vdash B \gg \langle e1.[e2] \rangle \gg A
\end{aligned}$$

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

$$\begin{aligned}
& | \text{Cons: } \llbracket \text{Env} \vdash B \gg \langle e :: \text{expr} \rangle \gg E; \text{Env} \vdash \text{nrm } E \gg \langle es \rangle \gg A \rrbracket \\
& \implies \\
& \text{Env} \vdash B \gg \langle e \# es \rangle \gg A
\end{aligned}$$

**declare** *inj-term-sym-simps* [*simp*]  
**declare** *assigns-if.simps* [*simp del*]  
**declare** *split-paired-All* [*simp del*] *split-paired-Ex* [*simp del*]  
**declaration**  $\ll K \text{ (Simplifier.map-ss (fn ss => ss delloop split-all-tac))} \gg$

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

$Env \vdash B \gg \langle \text{Skip} \rangle \gg A$   
 $Env \vdash B \gg \text{In1r Skip} \gg A$   
 $Env \vdash B \gg \langle \text{Expr } e \rangle \gg A$   
 $Env \vdash B \gg \text{In1r (Expr } e) \gg A$   
 $Env \vdash B \gg \langle l \cdot c \rangle \gg A$   
 $Env \vdash B \gg \text{In1r (} l \cdot c) \gg A$   
 $Env \vdash B \gg \langle c1;; c2 \rangle \gg A$   
 $Env \vdash B \gg \text{In1r (} c1;; c2) \gg A$   
 $Env \vdash B \gg \langle \text{If}(e) \ c1 \ \text{Else } c2 \rangle \gg A$   
 $Env \vdash B \gg \text{In1r (If}(e) \ c1 \ \text{Else } c2) \gg A$   
 $Env \vdash B \gg \langle l \cdot \text{While}(e) \ c \rangle \gg A$   
 $Env \vdash B \gg \text{In1r (} l \cdot \text{While}(e) \ c) \gg A$   
 $Env \vdash B \gg \langle \text{Jmp jump} \rangle \gg A$   
 $Env \vdash B \gg \text{In1r (Jmp jump)} \gg A$   
 $Env \vdash B \gg \langle \text{Throw } e \rangle \gg A$   
 $Env \vdash B \gg \text{In1r (Throw } e) \gg A$   
 $Env \vdash B \gg \langle \text{Try } c1 \ \text{Catch}(C \ vn) \ c2 \rangle \gg A$   
 $Env \vdash B \gg \text{In1r (Try } c1 \ \text{Catch}(C \ vn) \ c2) \gg A$   
 $Env \vdash B \gg \langle c1 \ \text{Finally } c2 \rangle \gg A$   
 $Env \vdash B \gg \text{In1r (} c1 \ \text{Finally } c2) \gg A$   
 $Env \vdash B \gg \langle \text{Init } C \rangle \gg A$   
 $Env \vdash B \gg \text{In1r (Init } C) \gg A$   
 $Env \vdash B \gg \langle \text{NewC } C \rangle \gg A$   
 $Env \vdash B \gg \text{In1l (NewC } C) \gg A$   
 $Env \vdash B \gg \langle \text{New } T[e] \rangle \gg A$   
 $Env \vdash B \gg \text{In1l (New } T[e]) \gg A$   
 $Env \vdash B \gg \langle \text{Cast } T \ e \rangle \gg A$   
 $Env \vdash B \gg \text{In1l (Cast } T \ e) \gg A$   
 $Env \vdash B \gg \langle e \ \text{InstOf } T \rangle \gg A$   
 $Env \vdash B \gg \text{In1l (} e \ \text{InstOf } T) \gg A$   
 $Env \vdash B \gg \langle \text{Lit } v \rangle \gg A$   
 $Env \vdash B \gg \text{In1l (Lit } v) \gg A$   
 $Env \vdash B \gg \langle \text{UnOp unop } e \rangle \gg A$   
 $Env \vdash B \gg \text{In1l (UnOp unop } e) \gg A$   
 $Env \vdash B \gg \langle \text{BinOp binop } e1 \ e2 \rangle \gg A$   
 $Env \vdash B \gg \text{In1l (BinOp binop } e1 \ e2) \gg A$   
 $Env \vdash B \gg \langle \text{Super} \rangle \gg A$   
 $Env \vdash B \gg \text{In1l (Super)} \gg A$   
 $Env \vdash B \gg \langle \text{Acc } v \rangle \gg A$   
 $Env \vdash B \gg \text{In1l (Acc } v) \gg A$   
 $Env \vdash B \gg \langle v := e \rangle \gg A$   
 $Env \vdash B \gg \text{In1l (} v := e) \gg A$   
 $Env \vdash B \gg \langle c \ ? \ e1 : e2 \rangle \gg A$   
 $Env \vdash B \gg \text{In1l (} c \ ? \ e1 : e2) \gg A$   
 $Env \vdash B \gg \langle \{accC, statT, mode\} e \cdot mn(\{pTs\} args) \rangle \gg A$   
 $Env \vdash B \gg \text{In1l (\{accC, statT, mode\} e \cdot mn(\{pTs\} args)) \gg A$   
 $Env \vdash B \gg \langle \text{Methd } C \ sig \rangle \gg A$   
 $Env \vdash B \gg \text{In1l (Methd } C \ sig) \gg A$   
 $Env \vdash B \gg \langle \text{Body } D \ c \rangle \gg A$   
 $Env \vdash B \gg \text{In1l (Body } D \ c) \gg A$   
 $Env \vdash B \gg \langle \text{LVar } vn \rangle \gg A$

```

Env ⊢ B » In2 (LVar vn) » A
Env ⊢ B » ⟨{accC, statDeclC, stat} e..fn⟩ » A
Env ⊢ B » In2 ({accC, statDeclC, stat} e..fn) » A
Env ⊢ B » ⟨e1.[e2]⟩ » A
Env ⊢ B » In2 (e1.[e2]) » A
Env ⊢ B » ⟨[]::expr list⟩ » A
Env ⊢ B » In3 ([]::expr list) » A
Env ⊢ B » ⟨e#es⟩ » A
Env ⊢ B » In3 (e#es) » A
declare inj-term-sym-simps [simp del]
declare assigns-if.simps [simp]
declare split-paired-All [simp] split-paired-Ex [simp]
declaration ⟨⟨ K (Simplifier.map-ss (fn ss => ss addloop (split-all-tac, split-all-tac))) ⟩⟩

```

```

lemma da-Skip: A = ⟨nrm=B, brk=λ l. UNIV⟩ ⟹ Env ⊢ B » ⟨Skip⟩ » A
by (auto intro: da.Skip)

```

```

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

```

```

lemma da-Lit: A = ⟨nrm=B, brk=λ l. UNIV⟩ ⟹ Env ⊢ B » ⟨Lit v⟩ » A
by (auto intro: da.Lit)

```

```

lemma da-Super: ⟨⟨ This ∈ B; A = ⟨nrm=B, brk=λ l. UNIV⟩ ⟩ ⟩ ⟹ Env ⊢ B » ⟨Super⟩ » A
by (auto intro: da.Super)

```

```

lemma da-Init: A = ⟨nrm=B, brk=λ l. UNIV⟩ ⟹ Env ⊢ B » ⟨Init C⟩ » A
by (auto intro: da.Init)

```

```

lemma assignsE-subseteq-assigns-ifs:
assumes boolEx: E ⊢ e::¬ PrimT Boolean (is ?Boolean e)
shows assignsE e ⊆ assigns-if True e ∩ assigns-if False e (is ?Incl e)
proof -
have True and ?Boolean e ⟹ ?Incl e and True and True
proof (induct - and e and - and - rule: var-expr-stmt.inducts)
case (Cast T e)
have E ⊢ e::¬ (PrimT Boolean)
proof -
from ⟨E ⊢ (Cast T e)::¬ (PrimT Boolean)⟩
obtain Te where E ⊢ e::¬ Te
prg E ⊢ Te ≤? PrimT Boolean
by cases simp
thus ?thesis
by - (drule cast-Boolean2, simp)
qed
with Cast.hyps
show ?case
by simp

```

```

next
  case (Lit val)
  thus ?case
  by - (erule wt-elim-cases, cases val, auto simp add: empty-dt-def)
next
  case (UnOp unop e)
  thus ?case
  by - (erule wt-elim-cases, cases unop,
        (fastsimp simp add: assignsE-const-simp)+)
next
  case (BinOp binop e1 e2)
  from BinOp.premis obtain e1T e2T
  where E1::e1T and E2::e2T and wt-binop (prg E) binop e1T e2T
  and (binop-type binop) = Boolean
  by (elim wt-elim-cases) simp
  with BinOp.hyps
  show ?case
  by - (cases binop, auto simp add: assignsE-const-simp)
next
  case (Cond c e1 e2)
  note hyp-c = ⟨?Boolean c ⟹ ?Incl c⟩
  note hyp-e1 = ⟨?Boolean e1 ⟹ ?Incl e1⟩
  note hyp-e2 = ⟨?Boolean e2 ⟹ ?Incl e2⟩
  note wt = ⟨E1(c ? e1 : e2)::¬PrimT Boolean⟩
  then obtain
    boolean-c: E1c::¬PrimT Boolean and
    boolean-e1: E1e1::¬PrimT Boolean and
    boolean-e2: E1e2::¬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
qed simp-all
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*]:  $l=l' \implies (rmlab\ l\ A)\ l' = UNIV$   
**by** (*simp add: rmlab-def*)

**lemma** *rmlab-other-label* [*simp*]:  $l \neq l' \implies (rmlab\ l\ A)\ l' = A\ l'$   
**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'*:  
 $\llbracket \forall\ k.\ A\ k \subseteq B\ k; x \in \Rightarrow \bigcap A \rrbracket \implies x \in \Rightarrow \bigcap B$   
**by** (*auto simp add: range-inter-ts-def*)

**lemma** *da-monotone*:  
**assumes** *da*:  $Env \vdash B \gg t \gg A$  **and**  
 $B \subseteq B'$  **and**  
 $da'$ :  $Env \vdash B' \gg t \gg A'$   
**shows**  $(nrm\ A \subseteq nrm\ A') \wedge (\forall\ l.\ (brk\ A\ l \subseteq brk\ A'\ l))$   
**proof** –  
**from** *da*  
**show**  $\bigwedge\ B'\ A'. \llbracket Env \vdash B' \gg t \gg A'; B \subseteq B' \rrbracket$   
 $\implies (nrm\ A \subseteq nrm\ A') \wedge (\forall\ l.\ (brk\ A\ l \subseteq brk\ A'\ l))$   
(is *PROP ?Hyp Env B t A*)  
**proof** (*induct*)  
**case** *Skip*  
**from** *Skip.prem*s *Skip.hyps*  
**show** *?case* **by** *cases simp*  
**next**  
**case** *Expr*  
**from** *Expr.prem*s *Expr.hyps*  
**show** *?case* **by** *cases simp*  
**next**  
**case** (*Lab Env B c C A l B' A'*)  
**note**  $A = \langle nrm\ A = nrm\ C \cap brk\ C\ l \rangle \langle brk\ A = rmlab\ l\ (brk\ C) \rangle$   
**note**  $\langle PROP\ ?Hyp\ Env\ B\ \langle c \rangle\ C \rangle$   
**moreover**  
**note**  $\langle B \subseteq B' \rangle$   
**moreover**  
**obtain** *C'*  
**where**  $Env \vdash B' \gg \langle c \rangle \gg C'$   
**and**  $A'$ :  $nrm\ A' = nrm\ C' \cap brk\ C'\ l\ brk\ A' = rmlab\ l\ (brk\ C')$   
**using** *Lab.prem*s  
**by** – (*erule da-elim-cases, simp*)  
**ultimately**  
**have**  $nrm\ C \subseteq nrm\ C'$  **and** *hyp-brk*:  $(\forall\ l.\ brk\ C\ l \subseteq brk\ C'\ l)$  **by** *auto*  
**then**  
**have**  $nrm\ C \cap brk\ C\ l \subseteq nrm\ C' \cap brk\ C'\ l$  **by** *auto*  
**moreover**  
{  
**fix** *l'*  
**from** *hyp-brk*  
**have**  $rmlab\ l\ (brk\ C)\ l' \subseteq rmlab\ l\ (brk\ C')\ l'$   
**by** (*cases l=l'*) *simp-all*



```

}
moreover note A A'
ultimately show ?case
  by simp
next
case (Comp Env B c1 C1 c2 C2 A B' A')
note A = ⟨nrm A = nrm C2⟩ ⟨brk A = brk C1 ⇒ ∩ brk C2⟩
from ⟨Env ⊢ B' ⟩⟨c1;; c2⟩ A'
obtain C1' C2'
  where da-c1: Env ⊢ B' ⟩⟨c1⟩ C1' and
        da-c2: Env ⊢ nrm C1' ⟩⟨c2⟩ C2' and
        A': nrm A' = nrm C2' brk A' = brk C1' ⇒ ∩ brk C2'
  by (rule da-elim-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 (rule da-elim-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 (rule da-elim-cases) auto
note ⟨PROP ?Hyp Env (B ∪ assigns-if True e) ⟨c⟩ C⟩
moreover note B' = ⟨B ⊆ B'⟩
moreover note da-c'

```

```

ultimately obtain  $C'$ :  $nrm\ C \subseteq nrm\ C' (\forall l. brk\ C\ l \subseteq brk\ C'\ l)$ 
  by blast
with  $A\ A'\ B'$ 
have  $nrm\ A \subseteq nrm\ A'$ 
  by blast
moreover
{ fix  $l'$ 
  have  $brk\ A\ l' \subseteq brk\ A'\ l'$ 
  proof (cases constVal e)
    case None
    with  $A\ A'\ C'$ 
    show ?thesis
      by (cases  $l=l'$ ) auto
  next
    case (Some bv)
    with  $A\ A'\ C'$ 
    show ?thesis
      by (cases the-Bool bv, cases  $l=l'$ ) auto
  qed
}
ultimately show ?case
  by auto
next
case (Jmp jump B A Env B' A')
thus ?case by (elim da-elim-cases) (auto split: jump.splits)
next
case Throw thus ?case by - (erule da-elim-cases, auto)
next
case (Try Env B c1 C1 vn C c2 C2 A B' A')
note  $A = \langle nrm\ A = nrm\ C1 \cap nrm\ C2 \rangle \langle brk\ A = brk\ C1 \Rightarrow \cap\ brk\ C2 \rangle$ 
from  $\langle Env \vdash B' \rangle \langle Try\ c1\ Catch(C\ vn)\ c2 \rangle \langle A' \rangle$ 
obtain  $C1'\ C2'$ 
  where  $da-c1'$ :  $Env \vdash B' \rangle \langle c1 \rangle \rangle C1'$  and
         $da-c2'$ :  $Env \langle lcl := lcl\ Env(VName\ vn \mapsto Class\ C) \rangle \vdash B' \cup \{VName\ vn\}$ 
         $\rangle \langle c2 \rangle \rangle C2'$  and
         $A': nrm\ A' = nrm\ C1' \cap nrm\ C2'$ 
         $brk\ A' = brk\ C1' \Rightarrow \cap\ brk\ C2'$ 
  by (rule da-elim-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  $\langle PROP\ ?Hyp\ (Env \langle lcl := lcl\ Env(VName\ vn \mapsto Class\ C) \rangle) \vdash (B \cup \{VName\ vn\}) \langle c2 \rangle\ C2 \rangle$ 
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 (rule da-elim-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 - (erule da-elim-cases, auto)
next
  case NewC thus ?case by - (erule da-elim-cases, auto)
next
  case NewA thus ?case by - (erule da-elim-cases, auto)
next
  case Cast thus ?case by - (erule da-elim-cases, auto)
next
  case Inst thus ?case by - (erule da-elim-cases, auto)
next
  case Lit thus ?case by - (erule da-elim-cases, auto)
next
  case UnOp thus ?case by - (erule 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)  $\rangle$ 
       $\langle brk \ A = (\lambda l. \ UNIV) \rangle$ 
  from  $\langle Env \vdash B' \rangle \langle BinOp \ CondAnd \ e1 \ e2 \rangle \langle A' \rangle$ 
  obtain A': nrm A' = B'  $\cup$ 
      assigns-if True (BinOp CondAnd e1 e2)  $\cap$ 
      assigns-if False (BinOp CondAnd e1 e2)  $\rangle$ 
       $\langle brk \ A' = (\lambda l. \ UNIV) \rangle$ 
    by (rule da-elim-cases) auto
  note B' =  $\langle B \subseteq B' \rangle$ 
  with A A' show ?case
    by auto
next
  case CondOr thus ?case by - (erule da-elim-cases, auto)
next
  case BinOp thus ?case by - (erule da-elim-cases, auto)
next
  case Super thus ?case by - (erule da-elim-cases, auto)
next
  case AccLVar thus ?case by - (erule da-elim-cases, auto)
next
  case Acc thus ?case by - (erule da-elim-cases, auto)
next
  case AssLVar thus ?case by - (erule da-elim-cases, auto)
next
  case Ass thus ?case by - (erule 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 − (erule 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 − (erule da-elim-cases, auto simp add: inj-term-simps)

note ⟨PROP ?Hyp Env (B ∪ assigns-if True c) ⟨e1⟩ E1⟩
moreover note B' = ⟨B ⊆ B'⟩
moreover note da-e1'
ultimately obtain E1': nrm E1 ⊆ nrm E1' (∀ l. brk E1 l ⊆ brk E1' l)
by blast
note ⟨PROP ?Hyp Env (B ∪ assigns-if False c) ⟨e2⟩ E2⟩
with B' da-e2'
obtain nrm E2 ⊆ nrm E2' (∀ l. brk E2 l ⊆ brk E2' l)
by blast
with E1' A A'
show ?case
by auto
next
case Call
from Call.prem and Call.hyps
show ?case by cases auto
next
case Methd thus ?case by − (erule da-elim-cases, auto)
next
case Body thus ?case by − (erule da-elim-cases, auto)
next
case LVar thus ?case by − (erule da-elim-cases, auto)
next
case FVar thus ?case by − (erule da-elim-cases, auto)
next
case AVar thus ?case by − (erule da-elim-cases, auto)
next
case Nil thus ?case by − (erule da-elim-cases, auto)

```

```

next
  case Cons thus ?case by - (erule da-elim-cases, auto)
qed
qed (rule da', rule  $\langle B \subseteq B' \rangle$ )

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
  show  $\bigwedge B'. B \subseteq B' \implies \exists A'. Env \vdash B' \gg t \gg A'$  (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-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-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  $Env \vdash (B' \cup assigns-if\ True\ e) \gg \langle c1 \rangle \gg C1'$ 
    proof -
      from B'

```

```

have (B ∪ assigns-if True e) ⊆ (B' ∪ assigns-if True e)
  by blast
moreover
have PROP ?Hyp Env (B ∪ assigns-if True e) ⟨c1⟩ by (rule If.hyps)
ultimately
show ?thesis using that by iprover
qed
moreover
obtain C2' where Env ⊢ (B' ∪ assigns-if False e) » ⟨c2⟩ C2'
proof -
  from B' have (B ∪ assigns-if False e) ⊆ (B' ∪ assigns-if False e)
    by blast
  moreover
  have PROP ?Hyp Env (B ∪ assigns-if False e) ⟨c2⟩ by (rule If.hyps)
  ultimately
  show ?thesis using that by iprover
qed
ultimately
obtain A' where Env ⊢ B' » ⟨If(e) c1 Else c2⟩ A'
  by (iprover intro: da.If)
thus ?case ..
next
case (Loop Env B e E c C A l B')
note B' = ⟨B ⊆ B'⟩
obtain E' where Env ⊢ B' » ⟨e⟩ E'
proof -
  have PROP ?Hyp Env B ⟨e⟩ by (rule Loop.hyps)
  with B'
  show ?thesis using that by iprover
qed
moreover
obtain C' where Env ⊢ (B' ∪ assigns-if True e) » ⟨c⟩ C'
proof -
  from B'
  have (B ∪ assigns-if True e) ⊆ (B' ∪ assigns-if True e)
    by blast
  moreover
  have PROP ?Hyp Env (B ∪ assigns-if True e) ⟨c⟩ by (rule Loop.hyps)
  ultimately
  show ?thesis using that by iprover
qed
ultimately
obtain A' where Env ⊢ B' » ⟨l. While(e) c⟩ A'
  by (iprover intro: da.Loop)
thus ?case ..
next
case (Jmp jump B A Env B')
note B' = ⟨B ⊆ B'⟩
with Jmp.hyps have jump = Ret ⟶ Result ∈ B'
  by auto
moreover
obtain A'::assigned
  where nrm A' = UNIV
        brk A' = (case jump of
          Break l ⇒ λk. if k = l then B' else UNIV
        | Cont l ⇒ λk. UNIV
        | Ret ⇒ λk. UNIV)

by iprover

```

```

ultimately have Env ⊢ B' »⟨Jump jump⟩ A'
  by (rule da.Jmp)
thus ?case ..
next
  case Throw thus ?case by (iprover intro: da.Throw )
next
  case (Try Env B c1 C1 vn C c2 C2 A B')
  note B' = ⟨B ⊆ B'⟩
  obtain C1' where Env ⊢ B' »⟨c1⟩ C1'
  proof -
    have PROP ?Hyp Env B ⟨c1⟩ by (rule Try.hyps)
    with B'
    show ?thesis using that by iprover
  qed
  moreover
  obtain C2' where
    Env(⟨lcl := lcl Env(VName vn → Class C)⟩) ⊢ B' ∪ {VName vn} »⟨c2⟩ C2'
  proof -
    from B' have B ∪ {VName vn} ⊆ B' ∪ {VName vn} by blast
    moreover
    have PROP ?Hyp (Env(⟨lcl := lcl Env(VName vn → Class C)⟩)
      (B ∪ {VName vn})) ⟨c2⟩
      by (rule Try.hyps)
    ultimately
    show ?thesis using that by iprover
  qed
  ultimately
  obtain A' where Env ⊢ B' »⟨Try c1 Catch(C vn) c2⟩ A'
    by (iprover intro: da.Try )
  thus ?case ..
next
  case (Fin Env B c1 C1 c2 C2 A B')
  note B' = ⟨B ⊆ B'⟩
  obtain C1' where C1': Env ⊢ B' »⟨c1⟩ C1'
  proof -
    have PROP ?Hyp Env B ⟨c1⟩ by (rule Fin.hyps)
    with B'
    show ?thesis using that by iprover
  qed
  moreover
  obtain C2' where Env ⊢ B' »⟨c2⟩ C2'
  proof -
    have PROP ?Hyp Env B ⟨c2⟩ by (rule Fin.hyps)
    with B'
    show ?thesis using that by iprover
  qed
  ultimately
  obtain A' where Env ⊢ B' »⟨c1 Finally c2⟩ A'
    by (iprover intro: da.Fin )
  thus ?case ..
next
  case Init thus ?case by (iprover intro: da.Init)
next
  case NewC thus ?case by (iprover intro: da.NewC)
next
  case NewA thus ?case by (iprover intro: da.NewA)
next
  case Cast thus ?case by (iprover intro: da.Cast)
next

```

```

  case Inst thus ?case by (iprover intro: da.Inst)
next
  case Lit thus ?case by (iprover intro: da.Lit)
next
  case UnOp thus ?case by (iprover intro: da.UnOp)
next
  case (CondAnd Env B e1 E1 e2 E2 A B')
  note  $B' = \langle B \subseteq B' \rangle$ 
  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  $PROP \ ?Hyp \ Env \ (B \cup \text{assigns-if False } e1) \ \langle e2 \rangle$  by (rule CondOr.hyps)
    ultimately show ?thesis using that by iprover
  qed
  ultimately
  obtain  $A'$  where  $Env \vdash B' \gg \langle BinOp \ CondOr \ e1 \ e2 \rangle \gg A'$ 
  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  $PROP \ ?Hyp \ Env \ B \ \langle e1 \rangle$  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  $PROP\ ?Hyp\ Env\ (nrm\ E1)\ \langle e2 \rangle$  by (rule BinOp.hyps)
    ultimately show ?thesis using that by iprover
qed
ultimately
have  $Env \vdash B' \gg \langle BinOp\ binop\ e1\ e2 \rangle \gg A'$ 
  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  $Env \vdash B' \gg \langle c ? e1 : e2 \rangle \gg A'$ 
  by (iprover intro: da.CondBool)
thus ?case ..
next
case (Cond Env c e1 e2 B C E1 E2 A B')
note  $B' = \langle B \subseteq B' \rangle$ 
note  $\langle \neg Env \vdash (c ? e1 : e2) :: \neg (PrimT Boolean) \rangle$ 
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 Cond.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 Cond.hyps)

```

```

ultimately
  show ?thesis using that by iprover
qed
moreover
obtain  $E2'$  where  $Env \vdash B' \cup \text{assigns-if False } c \gg \langle e2 \rangle \gg E2'$ 
proof -
  from  $B'$ 
  have  $(B \cup \text{assigns-if False } c) \subseteq (B' \cup \text{assigns-if False } c)$ 
    by blast
  moreover
  have  $PROP \ ?Hyp \ Env \ (B \cup \text{assigns-if False } c) \ \langle e2 \rangle$  by (rule Cond.hyps)
  ultimately
  show ?thesis using that by iprover
qed
ultimately
obtain  $A'$  where  $Env \vdash B' \gg \langle c \ ? \ e1 : e2 \rangle \gg A'$ 
  by (iprover intro: da.Cond)
thus ?case ..
next
case (Call Env B e E args A accC statT mode mn pTs B')
note  $B' = \langle B \subseteq B' \rangle$ 
obtain  $E'$  where  $E': Env \vdash B' \gg \langle e \rangle \gg E'$ 
proof -
  have  $PROP \ ?Hyp \ Env \ B \ \langle e \rangle$  by (rule Call.hyps)
  with  $B'$ 
  show ?thesis using that by iprover
qed
moreover
obtain  $A'$  where  $Env \vdash nrm \ E' \gg \langle args \rangle \gg A'$ 
proof -
  have  $Env \vdash B \gg \langle e \rangle \gg E$  by (rule Call.hyps)
  from this  $B' \ E'$ 
  have  $nrm \ E \subseteq nrm \ E'$ 
    by (rule da-monotone [THEN conjE])
  moreover
  have  $PROP \ ?Hyp \ Env \ (nrm \ E) \ \langle args \rangle$  by (rule Call.hyps)
  ultimately show ?thesis using that by iprover
qed
ultimately
have  $Env \vdash B' \gg \langle \{accC, statT, mode\} e \cdot mn(\{pTs\} args) \rangle \gg A'$ 
  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  $\langle \text{Result} \in \text{nrm } C \rangle$ 
with  $\text{nrm-}C'$  have  $\text{Result} \in \text{nrm } C'$ 
  by blast
moreover note  $\langle \text{jumpNestingOkS } \{ \text{Ret} \} \ c \rangle$ 
ultimately obtain  $A'$  where
   $\text{Env} \vdash B' \gg \langle \text{Body } D \ c \rangle \gg A'$ 
  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 ( $AVar \ \text{Env } B \ e1 \ E1 \ e2 \ A \ B'$ )
  note  $B' = \langle B \subseteq B' \rangle$ 
  obtain  $E1'$  where  $E1': \text{Env} \vdash B' \gg \langle e1 \rangle \gg E1'$ 
  proof –
    have  $PROP \ ?Hyp \ \text{Env } B \ \langle e1 \rangle$  by (rule AVar.hyps)
    with  $B'$ 
    show ?thesis using that by iprover
  qed
  moreover
  obtain  $A'$  where  $\text{Env} \vdash \text{nrm } E1' \gg \langle e2 \rangle \gg A'$ 
  proof –
    have  $\text{Env} \vdash B \gg \langle e1 \rangle \gg E1$  by (rule AVar.hyps)
    from this  $B' \ E1'$ 
    have  $\text{nrm } E1 \subseteq \text{nrm } E1'$ 
    by (rule da-monotone [THEN conjE])
    moreover
    have  $PROP \ ?Hyp \ \text{Env } (\text{nrm } E1) \ \langle e2 \rangle$  by (rule AVar.hyps)
    ultimately show ?thesis using that by iprover
  qed
  ultimately
  have  $\text{Env} \vdash B' \gg \langle e1.[e2] \rangle \gg A'$ 
  by (iprover intro: da.AVar)
  thus ?case ..
next
  case  $Nil$  thus ?case by (iprover intro: da.Nil)
next
  case ( $Cons \ \text{Env } B \ e \ E \ es \ A \ B'$ )
  note  $B' = \langle B \subseteq B' \rangle$ 
  obtain  $E'$  where  $E': \text{Env} \vdash B' \gg \langle e \rangle \gg E'$ 
  proof –
    have  $PROP \ ?Hyp \ \text{Env } B \ \langle e \rangle$  by (rule Cons.hyps)
    with  $B'$ 
    show ?thesis using that by iprover
  qed
  moreover
  obtain  $A'$  where  $\text{Env} \vdash \text{nrm } E' \gg \langle es \rangle \gg A'$ 
  proof –
    have  $\text{Env} \vdash B \gg \langle e \rangle \gg E$  by (rule Cons.hyps)
    from this  $B' \ E'$ 
    have  $\text{nrm } E \subseteq \text{nrm } E'$ 
    by (rule da-monotone [THEN conjE])
    moreover
    have  $PROP \ ?Hyp \ \text{Env } (\text{nrm } E) \ \langle es \rangle$  by (rule Cons.hyps)
    ultimately show ?thesis using that by iprover
  qed

```

```

ultimately
have  $Env \vdash B' \gg \langle e \# es \rangle \gg A'$ 
  by (iprover intro: da.Cons)
thus ?case ..
qed
qed (rule  $\langle B \subseteq B' \rangle$ )

```

```

corollary da-weakenE [consumes 2]:
  assumes       $da: Env \vdash B \gg t \gg A$  and
                $B': B \subseteq B'$  and
  ex-mono:  $\bigwedge A'. \llbracket Env \vdash B' \gg t \gg A'; nrm A \subseteq nrm A';$ 
            $\bigwedge l. brk A l \subseteq brk A' l \rrbracket \implies P$ 

  shows  $P$ 
proof -
  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-mono
  show ?thesis
    by iprover
qed

end

```



## Chapter 13

# WellForm

### 34 Well-formedness of Java programs

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

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

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

simplifications:

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

#### well-formed field declarations

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

**constdefs**

$$\begin{aligned} wf\_fdecl &:: prog \Rightarrow pname \Rightarrow fdecl \Rightarrow bool \\ wf\_fdecl\ G\ P &\equiv \lambda(fn,f). is\_acc\_type\ G\ P\ (type\ f) \end{aligned}$$

**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

**constdefs**

$$\begin{aligned} wf\_mhead &:: prog \Rightarrow pname \Rightarrow sig \Rightarrow mhead \Rightarrow bool \\ wf\_mhead\ G\ P &\equiv \lambda\ sig\ mh. length\ (parTs\ sig) = length\ (pars\ mh) \wedge \\ &\quad (\forall T \in set\ (parTs\ sig). is\_acc\_type\ G\ P\ T) \wedge \\ &\quad is\_acc\_type\ G\ P\ (resTy\ mh) \wedge \\ &\quad distinct\ (pars\ mh) \end{aligned}$$

A method declaration is wellformed if:

- the method head is wellformed
- the names of the local variables are unique



- the types of the local variables must be accessible
- the local variables don't shadow the parameters
- the class of the method is defined
- the body statement is welltyped with respect to the modified environment of local names, were the local variables, the parameters the special result variable (Res) and This are assoziated with there types.

**constdefs** *callee-lcl* :: *qname*  $\Rightarrow$  *sig*  $\Rightarrow$  *methd*  $\Rightarrow$  *lenv*  
*callee-lcl* *C* *sig* *m*  
 $\equiv \lambda k. (case\ k\ of$   
 $\quad EName\ e$   
 $\quad \Rightarrow (case\ e\ of$   
 $\quad \quad VName\ v$   
 $\quad \quad \Rightarrow (table-of\ (lcls\ (mbody\ m))((pars\ m)[\mapsto](parTs\ sig)))\ v$   
 $\quad \quad | Res \Rightarrow Some\ (resTy\ m))$   
 $\quad | This \Rightarrow if\ is-static\ m\ then\ None\ else\ Some\ (Class\ C))$

**constdefs** *parameters* :: *methd*  $\Rightarrow$  *lname* *set*  
*parameters* *m*  $\equiv set\ (map\ (EName\ \circ\ VName)\ (pars\ m))$   
 $\cup\ (if\ (static\ m)\ then\ \{\}\ else\ \{This\})$

**constdefs**  
*wf-mdecl* :: *prog*  $\Rightarrow$  *qname*  $\Rightarrow$  *mdecl*  $\Rightarrow$  *bool*  
*wf-mdecl* *G* *C*  $\equiv$   
 $\lambda(sig,m).$   
 $wf-mhead\ G\ (pid\ C)\ sig\ (mhead\ m) \wedge$   
 $unique\ (lcls\ (mbody\ m)) \wedge$   
 $(\forall (vn,T) \in set\ (lcls\ (mbody\ m)).\ is-acc-type\ G\ (pid\ C)\ T) \wedge$   
 $(\forall pn \in set\ (pars\ m).\ table-of\ (lcls\ (mbody\ m))\ pn = None) \wedge$   
 $jumpNestingOkS\ \{Ret\}\ (stmt\ (mbody\ m)) \wedge$   
 $is-class\ G\ C \wedge$   
 $(\downarrow prg = G, cls = C, lcl = callee-lcl\ C\ sig\ m) \vdash (stmt\ (mbody\ m)) :: \surd \wedge$   
 $(\exists\ A.\ (\downarrow prg = G, cls = C, lcl = callee-lcl\ C\ sig\ m)$   
 $\quad \vdash\ parameters\ m \gg (stmt\ (mbody\ m)) \gg A$   
 $\quad \wedge\ Result \in nrm\ A)$

**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*:

$\neg \text{is-static } m \implies \text{callee-lcl } C \text{ sig } m \text{ (This)} = \text{Some (Class } C)$

**by** *simp*

**lemma** *wf-mheadI*:

$\llbracket \text{length (parTs sig)} = \text{length (pars m)}; \forall T \in \text{set (parTs sig)}. \text{is-acc-type } G \text{ P } T;$   
 $\text{is-acc-type } G \text{ P (resTy m)}; \text{distinct (pars m)} \rrbracket \implies$   
 $\text{wf-mhead } G \text{ P sig } m$

**apply** (*unfold wf-mhead-def*)

**apply** (*simp (no-asm-simp)*)

**done**

**lemma** *wf-mdeclI*:  $\llbracket$

$\text{wf-mhead } G \text{ (pid } C) \text{ sig (mhead m)}; \text{unique (lcls (mbody m))};$   
 $(\forall pn \in \text{set (pars m)}. \text{table-of (lcls (mbody m)) } pn = \text{None});$   
 $\forall (vn, T) \in \text{set (lcls (mbody m))}. \text{is-acc-type } G \text{ (pid } C) \text{ } T;$   
 $\text{jumpNestingOkS \{Ret\} (stmt (mbody m))};$   
 $\text{is-class } G \text{ } C;$   
 $(\llbracket \text{prg} = G, \text{cls} = C, \text{lcl} = \text{callee-lcl } C \text{ sig } m \rrbracket \vdash \text{stmt (mbody m)})::\checkmark;$   
 $(\exists A. (\llbracket \text{prg} = G, \text{cls} = C, \text{lcl} = \text{callee-lcl } C \text{ sig } m \rrbracket \vdash \text{parameters } m \gg \langle \text{stmt (mbody m)} \rangle \gg A$   
 $\wedge \text{Result} \in \text{nrm } A)$   
 $\rrbracket \implies$

$\text{wf-mdecl } G \text{ } C \text{ (sig, m)}$

**apply** (*unfold wf-mdecl-def*)

**apply** *simp*

**done**

**lemma** *wf-mdeclE* [*consumes 1*]:

$\llbracket \text{wf-mdecl } G \text{ } C \text{ (sig, m)};$   
 $\llbracket \text{wf-mhead } G \text{ (pid } C) \text{ sig (mhead m)}; \text{unique (lcls (mbody m))};$   
 $\forall pn \in \text{set (pars m)}. \text{table-of (lcls (mbody m)) } pn = \text{None};$   
 $\forall (vn, T) \in \text{set (lcls (mbody m))}. \text{is-acc-type } G \text{ (pid } C) \text{ } T;$   
 $\text{jumpNestingOkS \{Ret\} (stmt (mbody m))};$   
 $\text{is-class } G \text{ } C;$   
 $(\llbracket \text{prg} = G, \text{cls} = C, \text{lcl} = \text{callee-lcl } C \text{ sig } m \rrbracket \vdash \text{stmt (mbody m)})::\checkmark;$   
 $(\exists A. (\llbracket \text{prg} = G, \text{cls} = C, \text{lcl} = \text{callee-lcl } C \text{ sig } m \rrbracket \vdash \text{parameters } m \gg \langle \text{stmt (mbody m)} \rangle \gg A$   
 $\wedge \text{Result} \in \text{nrm } A)$   
 $\rrbracket \implies P$

$\rrbracket \implies P$

**by** (*unfold wf-mdecl-def*) *simp*

**lemma** *wf-mdeclD1*:

$\text{wf-mdecl } G \text{ } C \text{ (sig, m)} \implies$

$\text{wf-mhead } G \text{ (pid } C) \text{ sig (mhead m)} \wedge \text{unique (lcls (mbody m))} \wedge$   
 $(\forall pn \in \text{set (pars m)}. \text{table-of (lcls (mbody m)) } pn = \text{None}) \wedge$   
 $(\forall (vn, T) \in \text{set (lcls (mbody m))}. \text{is-acc-type } G \text{ (pid } C) \text{ } T)$

**apply** (*unfold wf-mdecl-def*)

**apply** *simp*

**done**

**lemma** *wf-mdecl-bodyD*:

```

wf-mdecl G C (sig,m)  $\impl$ 
  ( $\exists T. (\text{prg}=G, \text{cls}=C, \text{lcl}=\text{callee-lcl } C \text{ sig } m) \vdash \text{Body } C \text{ (stmt (mbody } m)) :: -T \wedge$ 
     $G \vdash T \preceq (\text{resTy } m)$ )
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  $\impl$  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

### constdefs

```

wf-idecl :: prog  $\Rightarrow$  idecl  $\Rightarrow$  bool
wf-idecl G  $\equiv$ 
   $\lambda(I,i).$ 
    ws-idecl G I (isuperIfs i)  $\wedge$ 
     $\neg$ is-class G I  $\wedge$ 
    ( $\forall (sig,mh) \in \text{set } (\text{imethods } i). \text{wf-mhead } G \text{ (pid } I) \text{ sig } mh \wedge$ 
       $\neg$ is-static mh  $\wedge$ 
      accmodi mh = Public)  $\wedge$ 
    unique (imethods i)  $\wedge$ 
    ( $\forall J \in \text{set } (\text{isuperIfs } i). \text{is-acc-iface } G \text{ (pid } I) J) \wedge$ 
    (table-of (imethods i)
      hiding (methd G Object)
      under ( $\lambda \text{ new old. accmodi old } \neq \text{Private}$ )
      entails ( $\lambda \text{ new old. } G \vdash \text{resTy new} \preceq \text{resTy old} \wedge$ 
         $\text{is-static new} = \text{is-static old}$ ))  $\wedge$ 
    (o2s  $\circ$  table-of (imethods i)
      hidings Un-tables(( $\lambda J. (\text{imethds } G J)$ ) 'set (isuperIfs i))
      entails ( $\lambda \text{ new old. } G \vdash \text{resTy new} \preceq \text{resTy old}$ ))

```

**lemma** *wf-idecl-mhead*:  $\llbracket \text{wf-idecl } G \ (I, i); (sig, mh) \in \text{set } (imethods \ i) \rrbracket \implies$   
 $\text{wf-mhead } G \ (pid \ I) \ sig \ mh \wedge \neg is-static \ mh \wedge accmodi \ mh = Public$   
**apply** (*unfold wf-idecl-def*)  
**apply** *auto*  
**done**

**lemma** *wf-idecl-hidings*:  
 $\text{wf-idecl } G \ (I, i) \implies$   
 $(\lambda s. o2s \ (table-of \ (imethods \ i) \ s))$   
 $hidings \ Un-tables \ ((\lambda J. imethds \ G \ J) \ 'set \ (isuperIfs \ i))$   
 $entails \ \lambda new \ old. \ G \vdash_{resTy} new \preceq_{resTy} old$   
**apply** (*unfold wf-idecl-def o-def*)  
**apply** *simp*  
**done**

**lemma** *wf-idecl-hiding*:  
 $\text{wf-idecl } G \ (I, i) \implies$   
 $(table-of \ (imethods \ i))$   
 $hiding \ (methd \ G \ Object)$   
 $under \ (\lambda new \ old. accmodi \ old \neq Private)$   
 $entails \ (\lambda new \ old. \ G \vdash_{resTy} new \preceq_{resTy} old \wedge$   
 $is-static \ new = is-static \ old))$   
**apply** (*unfold wf-idecl-def*)  
**apply** *simp*  
**done**

**lemma** *wf-idecl-supD*:  
 $\llbracket \text{wf-idecl } G \ (I, i); J \in \text{set } (isuperIfs \ i) \rrbracket$   
 $\implies is-acc-iface \ G \ (pid \ I) \ J \wedge (J, I) \notin (subint1 \ G) \wedge +$   
**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



**lemma** *wf-cdeclE [consumes 1]*:  
 $\llbracket \text{wf-cdecl } G \ (C, c);$   
 $\llbracket \neg \text{is-iface } G \ C;$   
 $(\forall I \in \text{set } (\text{superIfs } c). \text{is-acc-iface } G \ (\text{pid } C) \ I \wedge$   
 $(\forall s. \forall \text{im} \in \text{imethds } G \ I \ s.$   
 $(\exists \text{cm} \in \text{methd } G \ C \ s: G \vdash \text{resTy } \text{cm} \preceq \text{resTy } \text{im} \wedge$   
 $\neg \text{is-static } \text{cm} \wedge$   
 $\text{accmodi } \text{im} \leq \text{accmodi } \text{cm})))$ ;  
 $\forall f \in \text{set } (\text{cfields } c). \text{wf-fdecl } G \ (\text{pid } C) \ f; \text{unique } (\text{cfields } c);$   
 $\forall m \in \text{set } (\text{methods } c). \text{wf-mdecl } G \ C \ m; \text{unique } (\text{methods } c);$   
 $\text{jumpNestingOkS } \{\} \ (\text{init } c);$   
 $\exists A. (\text{prg} = G, \text{cls} = C, \text{lcl} = \text{empty}) \vdash \{\} \gg \langle \text{init } c \rangle \gg A;$   
 $(\text{prg} = G, \text{cls} = C, \text{lcl} = \text{empty}) \vdash (\text{init } c) :: \checkmark;$   
 $\text{ws-cdecl } G \ C \ (\text{super } c);$   
 $(C \neq \text{Object} \longrightarrow$   
 $(\text{is-acc-class } G \ (\text{pid } C) \ (\text{super } c) \wedge$   
 $(\text{table-of } (\text{map } (\lambda (s, m). (s, C, m)) (\text{methods } c))$   
 $\text{entails } (\lambda \text{new}. \forall \text{old sig.}$   
 $(G, \text{sig} \vdash \text{new overrides}_S \text{old}$   
 $\longrightarrow (G \vdash \text{resTy } \text{new} \preceq \text{resTy } \text{old} \wedge$   
 $\text{accmodi } \text{old} \leq \text{accmodi } \text{new} \wedge$   
 $\neg \text{is-static } \text{old})) \wedge$   
 $(G, \text{sig} \vdash \text{new hides } \text{old}$   
 $\longrightarrow (\text{accmodi } \text{old} \leq \text{accmodi } \text{new} \wedge$   
 $\text{is-static } \text{old}))))$   
 $\rrbracket \implies P$   
**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*:  
 $\llbracket \text{wf-cdecl } G \ (C, c); I \in \text{set } (\text{superIfs } c) \rrbracket$   
 $\implies \text{is-acc-iface } G \ (\text{pid } C) \ I \wedge$   
 $(\forall s. \forall \text{im} \in \text{imethds } G \ I \ s.$   
 $(\exists \text{cm} \in \text{methd } G \ C \ s: G \vdash \text{resTy } \text{cm} \preceq \text{resTy } \text{im} \wedge \neg \text{is-static } \text{cm} \wedge$   
 $\text{accmodi } \text{im} \leq \text{accmodi } \text{cm}))$

```

apply (unfold wf-cdecl-def)
apply auto
done

```

```

lemma wf-cdecl-supD:
   $\llbracket wf-cdecl\ G\ (C,c); C \neq Object \rrbracket \implies$ 
   $is-acc-class\ G\ (pid\ C)\ (super\ c) \wedge (super\ c,C) \notin (subcls1\ G)^+ \wedge$ 
   $(table-of\ (map\ (\lambda\ (s,m). (s,C,m))\ (methods\ c)))$ 
   $entails\ (\lambda\ new. \forall\ old\ sig.$ 
     $(G, sig \vdash new\ overrides_S\ old$ 
       $\longrightarrow (G \vdash resTy\ new \preceq resTy\ old \wedge$ 
         $accmodi\ old \leq accmodi\ new \wedge$ 
         $\neg is-static\ old))) \wedge$ 
     $(G, sig \vdash new\ hides\ old$ 
       $\longrightarrow (accmodi\ old \leq accmodi\ new \wedge$ 
         $is-static\ old))))$ 
apply (unfold wf-cdecl-def ws-cdecl-def)
apply auto
done

```

```

lemma wf-cdecl-overrides-SomeD:
   $\llbracket wf-cdecl\ G\ (C,c); C \neq Object; table-of\ (methods\ c)\ sig = Some\ newM;$ 
   $G, sig \vdash (C, newM)\ overrides_S\ old$ 
 $\rrbracket \implies G \vdash resTy\ newM \preceq resTy\ old \wedge$ 
   $accmodi\ old \leq accmodi\ newM \wedge$ 
   $\neg is-static\ old$ 
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:
   $\llbracket wf-cdecl\ G\ (C,c); C \neq Object; table-of\ (methods\ c)\ sig = Some\ newM;$ 
   $G, sig \vdash (C, newM)\ hides\ old$ 
 $\rrbracket \implies accmodi\ old \leq access\ newM \wedge$ 
   $is-static\ old$ 
apply (drule (1) wf-cdecl-supD)
apply (clarify)
apply (drule entailsD)
apply (blast intro: table-of-map-SomeI)
apply (drule-tac x=old in spec)
apply (auto dest: hides-eq-sigD simp add: msig-def)
done

```

```

lemma wf-cdecl-wt-init:
   $wf-cdecl\ G\ (C, c) \implies (\llbracket prg=G, cls=C, lcl=empty \rrbracket) \vdash init\ c :: \checkmark$ 
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 "overridden" by an interface method this is also overridden by the class implementing the the interface (see *implement-dynmethd* and *class-mheadsD*)
- all standard Exceptions are defined
- all defined interfaces are wellformed
- all defined classes are wellformed

### constdefs

```

wf-prog :: prog ⇒ bool
wf-prog G ≡ let is = ifaces G; cs = classes G in
  ObjectC ∈ set cs ∧
  (∀ m∈set Object-mdecls. accmodi m ≠ Package) ∧
  (∀ xn. SXcptC xn ∈ set cs) ∧
  (∀ i∈set is. wf-idecl G i) ∧ unique is ∧
  (∀ c∈set cs. wf-cdecl G c) ∧ unique cs

```

```

lemma wf-prog-idecl: [iface G I = Some i; wf-prog G] ⇒ wf-idecl G (I,i)
apply (unfold wf-prog-def Let-def)
apply simp
apply (fast dest: map-of-SomeD)
done

```

```

lemma wf-prog-cdecl: [class G C = Some c; wf-prog G] ⇒ wf-cdecl G (C,c)
apply (unfold wf-prog-def Let-def)
apply simp
apply (fast dest: map-of-SomeD)
done

```

```

lemma wf-prog-Object-mdecls:
wf-prog G ⇒ (∀ m∈set Object-mdecls. accmodi m ≠ Package)
apply (unfold wf-prog-def Let-def)
apply simp
done

```

```

lemma wf-prog-acc-superD:
[ wf-prog G; class G C = Some c; C ≠ Object ]
⇒ is-acc-class G (pid C) (super c)
by (auto dest: wf-prog-cdecl wf-cdecl-supD)

```

```

lemma wf-ws-prog [elim!,simp]: wf-prog G ⇒ ws-prog G
apply (unfold wf-prog-def Let-def)
apply (rule ws-progI)
apply (simp-all (no-asm))
apply (auto simp add: is-acc-class-def is-acc-iface-def)

```



```

    dest!: wf-idecl-supD wf-cdecl-supD )+
done

```

```

lemma class-Object [simp]:
wf-prog G  $\implies$ 
  class G Object = Some ( $\langle$ access=Public,cfields=[],methods=Object-mdecls,
    init=Skip,super=arbitrary,superIfs=[] $\rangle$ )
apply (unfold wf-prog-def Let-def ObjectC-def)
apply (fast dest!: map-of-SomeI)
done

```

```

lemma methd-Object[simp]: wf-prog G  $\implies$  methd G Object =
  table-of (map ( $\lambda(s,m).$  (s, Object, m)) Object-mdecls)
apply (subst methd-rec)
apply (auto simp add: Let-def)
done

```

```

lemma wf-prog-Object-methd:
 $\llbracket$ wf-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-prog G  $\implies$  is-public G Object
by (auto simp add: is-public-def dest: class-Object)

```

```

lemma class-SXcpt [simp]:
wf-prog G  $\implies$ 
  class G (SXcpt xn) = Some ( $\langle$ access=Public,cfields=[],methods=SXCpt-mdecls,
    init=Skip,
    super=if xn = Throwable then Object
    else SXcpt Throwable,
    superIfs=[] $\rangle$ )
apply (unfold wf-prog-def Let-def SXcptC-def)
apply (fast dest!: map-of-SomeI)
done

```

```

lemma wf-ObjectC [simp]:
  wf-cdecl G ObjectC = ( $\neg$ is-iface G Object  $\wedge$  Ball (set Object-mdecls)
    (wf-mdecl G Object)  $\wedge$  unique Object-mdecls)
apply (unfold wf-cdecl-def ws-cdecl-def ObjectC-def)
apply (auto intro: da.Skip)
done

```

```

lemma Object-is-class [simp,elim!]: wf-prog G  $\implies$  is-class G Object
apply (simp (no-asm-simp))
done

```

```

lemma Object-is-acc-class [simp,elim!]: wf-prog G  $\implies$  is-acc-class G S Object
apply (simp (no-asm-simp) add: is-acc-class-def is-public-def
  accessible-in-RefT-simp)
done

```

**lemma** *SXcpt-is-class* [*simp,elim!*]: *wf-prog G*  $\implies$  *is-class G (SXcpt xn)*  
**apply** (*simp* (*no-asm-simp*))  
**done**

**lemma** *SXcpt-is-acc-class* [*simp,elim!*]:  
*wf-prog G*  $\implies$  *is-acc-class G S (SXcpt xn)*  
**apply** (*simp* (*no-asm-simp*) *add: is-acc-class-def is-public-def*  
*accessible-in-RefT-simp*)  
**done**

**lemma** *fields-Object* [*simp*]: *wf-prog G*  $\implies$  *DeclConcepts.fields G Object* = []  
**by** (*force intro: fields-emptyI*)

**lemma** *accfield-Object* [*simp*]:  
*wf-prog G*  $\implies$  *accfield G S Object* = *empty*  
**apply** (*unfold accfield-def*)  
**apply** (*simp* (*no-asm-simp*) *add: Let-def*)  
**done**

**lemma** *fields-Throwable* [*simp*]:  
*wf-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*  $\implies G \vdash \text{SXcpt } xn \preceq_C \text{SXcpt } Throwable$   
**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*:  
 $\llbracket \text{table-of } (\text{DeclConcepts.fields } G \ C) \text{ fn} = \text{Some } f; G \vdash D \preceq_C C; \rrbracket$

```

  is-class G D; wf-prog G]]
  ==> table-of (DeclConcepts.fields G D) fn = Some f
apply (rule map-of-SomeI)
apply (erule (1) unique-fields)
apply (erule (1) map-of-SomeD [THEN fields-mono-lemma])
apply (erule wf-ws-prog)
done

```

```

lemma fields-is-type [elim]:
  [[table-of (DeclConcepts.fields G C) m = Some f; wf-prog G; is-class G C]] ==>
    is-type G (type f)
apply (frule wf-ws-prog)
apply (force dest: fields-declC [THEN conjunct1]
        wf-prog-cdecl [THEN wf-cdecl-fdecl]
        simp add: wf-fdecl-def2 is-acc-type-def)
done

```

```

lemma imethds-wf-mhead [rule-format (no-asm)]:
  [[m ∈ imethds G I sig; wf-prog G; is-iface G I]] ==>
    wf-mhead G (pid (decliface m)) sig (mthd m) ∧
    ¬ is-static m ∧ accmodi m = Public
apply (frule wf-ws-prog)
apply (drule (2) imethds-declI [THEN conjunct1])
apply clarify
apply (frule-tac I=(decliface m) in wf-prog-idecl,assumption)
apply (drule wf-idecl-mhead)
apply (erule map-of-SomeD)
apply (cases m, simp)
done

```

```

lemma methd-wf-mdecl:
  [[methd G C sig = Some m; wf-prog G; class G C = Some y]] ==>
    G ⊢ C ≤C (declclass m) ∧ is-class G (declclass m) ∧
    wf-mdecl G (declclass m) (sig,(mthd m))
apply (frule wf-ws-prog)
apply (drule (1) methd-declC)
apply fast
apply clarsimp
apply (frule (1) wf-prog-cdecl, erule wf-cdecl-mdecl, erule map-of-SomeD)
done

```

```

lemma methd-rT-is-type:
  [[wf-prog G; methd G C sig = Some m;
    class G C = Some y]]
  ==> is-type G (resTy m)
apply (drule (2) methd-wf-mdecl)
apply clarify
apply (drule wf-mdeclD1)
apply clarify
apply (drule rT-is-acc-type)
apply (cases m, simp add: is-acc-type-def)

```

done

**lemma** *accmethd-rT-is-type*:  
 $\llbracket wf\text{-prog } G; accmethd\ G\ S\ C\ sig = Some\ m;$   
 $\quad class\ G\ C = Some\ y \rrbracket$   
 $\implies is\text{-type } G\ (resTy\ m)$   
**by** (auto simp add: accmethd-def  
intro: methd-rT-is-type)

**lemma** *methd-Object-SomeD*:  
 $\llbracket wf\text{-prog } G; methd\ G\ Object\ sig = Some\ m \rrbracket$   
 $\implies declclass\ m = Object$   
**by** (auto dest: class-Object simp add: methd-rec )

**lemma** *wf-imethdsD*:  
 $\llbracket im \in imethds\ G\ I\ sig; wf\text{-prog } G; is\text{-iface } G\ I \rrbracket$   
 $\implies \neg is\text{-static } im \wedge accmodi\ im = Public$   
**proof** –  
**assume** *asm*:  $wf\text{-prog } G\ is\text{-iface } G\ I\ im \in imethds\ G\ I\ sig$   
**have**  $wf\text{-prog } G \longrightarrow$   
 $(\forall\ i\ im. iface\ G\ I = Some\ i \longrightarrow im \in imethds\ G\ I\ sig$   
 $\longrightarrow \neg is\text{-static } im \wedge accmodi\ im = Public) \text{ (is } ?P\ G\ I)$   
**proof** (rule iface-rec.induct,intro allI impI)  
**fix**  $G\ I\ i\ im$   
**assume** *hyp*:  $\forall\ J\ i. J \in set\ (isuperIfs\ i) \wedge ws\text{-prog } G \wedge iface\ G\ I = Some\ i$   
 $\longrightarrow ?P\ G\ J$   
**assume** *wf*:  $wf\text{-prog } G$  **and** *if-I*:  $iface\ G\ I = Some\ i$  **and**  
 $im: im \in imethds\ G\ I\ sig$   
**show**  $\neg is\text{-static } im \wedge accmodi\ im = Public$   
**proof** –  
**let**  $?inherited = Un\text{-tables } (imethds\ G\ 'set\ (isuperIfs\ i))$   
**let**  $?new = (o2s \circ table\text{-of } (map\ (\lambda(s, mh). (s, I, mh))\ (imethds\ i)))$   
**from** *if-I* **wf** *im* **have**  $imethds: im \in (?inherited \oplus \oplus ?new)\ sig$   
**by** (simp add: imethds-rec)  
**from** *wf if-I* **have**  
 $wf\text{-supI}: \forall\ J. J \in set\ (isuperIfs\ i) \longrightarrow (\exists\ j. iface\ G\ J = Some\ j)$   
**by** (blast dest: wf-prog-idecl wf-idecl-supD is-acc-ifaceD)  
**from** *wf if-I* **have**  
 $\forall\ im \in set\ (imethds\ i). \neg is\text{-static } im \wedge accmodi\ im = Public$   
**by** (auto dest!: wf-prog-idecl wf-idecl-mhead)  
**then have** *new-ok*:  $\forall\ im. table\text{-of } (imethds\ i)\ sig = Some\ im$   
 $\longrightarrow \neg is\text{-static } im \wedge accmodi\ im = Public$   
**by** (auto dest!: table-of-Some-in-set)  
**show** *?thesis*  
**proof** (cases *?new sig* = {})  
**case** *True*  
**from** *True wf wf-supI if-I imethds hyp*  
**show** *?thesis* **by** (auto simp del: split-paired-All)  
**next**  
**case** *False*  
**from** *False wf wf-supI if-I imethds new-ok hyp*  
**show** *?thesis* **by** (auto dest: wf-idecl-hidings hidings-entailsD)  
**qed**  
**qed**  
**qed**  
**with** *asm* **show** *?thesis* **by** (auto simp del: split-paired-All)

qed

**lemma** *wf-prog-hidesD*:

**assumes** *hides*:  $G \vdash \text{new hides old}$  **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**

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

**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*  $G (\text{declclass new}, c)$  **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 wellfromed program. Dynamic overriding has no restrictions on the access modifiers but enforces confrom result types as precondition. But with some efford we can guarantee the access modifier restriction for dynamic overriding, too. See lemma *wf-prog-dyn-override-prop*.

**lemma** *wf-prog-stat-overridesD*:

**assumes** *stat-override*:  $G \vdash \text{new overrides}_S \text{ old}$  **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*:  $\text{class } G (\text{declclass new}) = \text{Some } c$  **and**

*neqObj*:  $\text{declclass new} \neq \text{Object}$

**by** (*auto dest*: *stat-overrides-commonD* *declared-in-classD*)

**with** *stat-override* **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**

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

**by** (*cases new, cases old*)

```

      (auto dest: stat-overrides-commonD
        simp add: cdeclaredmethd-def declared-in-def)
with stat-override
have stat-override':
   $G, (msig\ new) \vdash (declclass\ new, newM) \text{ overrides}_S (declclass\ old, oldM)$ 
  by auto
from clsNew wf
have wf-cdecl  $G (declclass\ new, c)$  by (blast intro: wf-prog-cdecl)
note wf-cdecl-overrides-SomeD [OF this neqObj newM stat-override']
with new old
show ?thesis
  by (cases new, cases old) auto
qed

```

**lemma** static-to-dynamic-overriding:

```

  assumes stat-override:  $G \vdash new \text{ overrides}_S old$  and wf : wf-prog  $G$ 
  shows  $G \vdash new \text{ overrides } old$ 
proof –
  from stat-override
  show ?thesis (is ?Overrides new old)
proof (induct)
  case (Direct new old superNew)
  then have stat-override:  $G \vdash new \text{ overrides}_S old$ 
    by (rule stat-overridesR.Direct)
  from stat-override wf
  have resTy-widen:  $G \vdash resTy\ new \preceq resTy\ old$  and
    not-static-old:  $\neg is-static\ old$ 
    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:  $G \vdash Method\ old\ inheritable-in\ (pid\ C)$  and
     $accmodi\ old: accmodi\ old \neq Package$  and
    instance-method:  $\neg is-static\ old$  and
    subcls:  $G \vdash C \prec_C declclass\ old$  and
    old-declared:  $G \vdash Method\ old\ declared-in\ (declclass\ old)$  and

```

```

wf: wf-prog G
shows  $G \vdash \text{Method old member-of } C \vee$ 
  ( $\exists \text{ new. } G \vdash \text{new overrides}_S \text{ old} \wedge G \vdash \text{Method new member-of } C$ )
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: class G C = Some c and
      neg-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 neg-C-Obj
    have super:  $G \vdash C \prec_{C1} \text{super c}$ 
      by (rule subcls1I)
    from wf cls-C neg-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)
    {
      fix old
      assume member-super:  $G \vdash \text{Method old member-of (super c)}$ 
      assume inheritable:  $G \vdash \text{Method old inheritable-in pid C}$ 
      assume instance-method:  $\neg \text{is-static old}$ 
      from member-super
      have old-declared:  $G \vdash \text{Method old declared-in (declclass old)}$ 
        by (cases old) (auto dest: member-of-declC)
      have ?P C old
      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
           $\text{mid (msig old)} = \text{memberid new-member}$ 
          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
} note hyp-member-super = this
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  $G \vdash \text{Method old member-of (super } c)$ 
      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 (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  $?P \ C \ \text{super-new}$ 
  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 [consumes 6,
  case-names Inheritance Overriding]:
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 and
  inheritance:  $G \vdash \text{Method old member-of } C \implies P$  and
  overriding:  $\bigwedge \text{new.}$ 
     $\llbracket G \vdash \text{new overrides}_S \text{ old}; G \vdash \text{Method new member-of } C \rrbracket$ 
     $\implies P$ 

shows  $P$ 
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*  $\neq \text{Package}$  **and**  
 $\neg \text{is-static old}$  **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*:  $\text{declclass new'} = \text{declclass new}$

**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**  $\text{new} = \text{new'}$

**by** (*cases new, cases new'*) (*auto dest: unique-declared-in*)

**with** *stat-override-new'*

**show** *?thesis*

**by** *simp*

**next**

```

    case Inherited
    then have  $G \vdash \text{mid } (\text{msig } \text{new}') \text{ undeclared-in declclass new}$ 
      by (cases  $\text{new}'$ ) (auto)
    with  $\text{eq-sig-new-new}' \text{ new-declared}$ 
    show ?thesis
      by (cases  $\text{new}, \text{cases new}'$ ) (auto dest!: declared-not-undeclared)
  qed
qed
next
case (Indirect new inter old)
assume  $\text{accmodi-old}: \text{accmodi old} \neq \text{Package}$ 
assume  $\text{accmodi old} \neq \text{Package} \implies G \vdash \text{inter overrides}_S \text{ old}$ 
with  $\text{accmodi-old}$ 
have  $\text{stat-override-inter-old}: G \vdash \text{inter overrides}_S \text{ old}$ 
  by blast
moreover
assume  $\text{hyp-inter}: \text{accmodi inter} \neq \text{Package} \implies G \vdash \text{new overrides}_S \text{ inter}$ 
moreover
have  $\text{accmodi inter} \neq \text{Package}$ 
proof -
  from  $\text{stat-override-inter-old wf}$ 
  have  $\text{accmodi old} \leq \text{accmodi inter}$ 
    by (auto dest: wf-prog-stat-overridesD)
  with  $\text{stat-override-inter-old accmodi-old}$ 
  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  $\text{dyn-override}: G \vdash \text{new overrides old}$  and
     $\text{wf}: \text{wf-prog } G$ 
  shows  $\text{accmodi old} \leq \text{accmodi new}$ 
proof (cases  $\text{accmodi old} = \text{Package}$ )
case True
note  $\text{old-Package} = \text{this}$ 
show ?thesis
proof (cases  $\text{accmodi old} \leq \text{accmodi new}$ )
case True then show ?thesis .
next
case False
with  $\text{old-Package}$ 
have  $\text{accmodi new} = \text{Private}$ 
  by (cases  $\text{accmodi new}$ ) (auto simp add: le-acc-def less-acc-def)
with  $\text{dyn-override}$ 
show ?thesis
  by (auto dest: overrides-commonD)
qed
next
case False
with  $\text{dyn-override wf}$ 
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  $\text{accmodi old} = \text{Package}$ 
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  $\text{pid}(\text{declclass old}) = \text{pid}(\text{declclass new})$ 
proof –
  from dyn-override accmodi-old accmodi-new
  show ?thesis (is ?EqPid old new)
proof (induct rule: overridesR.induct)
  case (Direct new old)
    assume  $\text{accmodi old} = \text{Package}$ 
     $G \vdash \text{Method old inheritable-in pid}(\text{declclass new})$ 
    then show  $\text{pid}(\text{declclass old}) = \text{pid}(\text{declclass new})$ 
    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  $\exists \text{ inter. } G \vdash \text{new overrides inter} \wedge G \vdash \text{inter overrides old} \wedge$ 
    pid (declclass old) = pid (declclass inter)  $\wedge$ 
    Protected  $\leq$  accmodi inter

```

**proof** –

```

from dyn-override accmodi-old outside-pack
show ?thesis (is ?P new old)
proof (induct rule: overridesR.induct)
case (Direct new old)
assume accmodi-old: accmodi old = Package
assume outside-pack: pid (declclass old)  $\neq$  pid (declclass new)
assume  $G \vdash \text{Method old inheritable-in pid (declclass new)}$ 
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:  $\llbracket \text{accmodi inter} = \text{Package};$ 
    pid (declclass inter)  $\neq$  pid (declclass new)  $\rrbracket$ 
     $\implies ?P \text{ new inter}$ 
assume hyp-inter-old:  $\llbracket \text{accmodi old} = \text{Package};$ 
    pid (declclass old)  $\neq$  pid (declclass inter)  $\rrbracket$ 
     $\implies ?P \text{ inter old}$ 

```

**show**  $?P \text{ new old}$

**proof** (cases pid (declclass old) = pid (declclass inter))

case True

note same-pack-old-inter = this

**show**  $?thesis$

**proof** (cases pid (declclass inter) = pid (declclass new))

case True

with same-pack-old-inter outside-pack

**show**  $?thesis$

by auto

**next**

```

case False
note diff-pack-inter-new = this
show ?thesis
proof (cases accmodi inter = Package)
  case True
  with diff-pack-inter-new hyp-new-inter
  obtain newinter where
    over-new-newinter: G ⊢ new overrides newinter and
    over-newinter-inter: G ⊢ newinter overrides inter and
    eq-pid: pid (declclass inter) = pid (declclass newinter) and
    accmodi-newinter: Protected ≤ accmodi newinter
    by auto
  from over-newinter-inter override-inter-old
  have G ⊢ newinter overrides old
    by (rule overridesR.Indirect)
  moreover
  from eq-pid same-pack-old-inter
  have pid (declclass old) = pid (declclass newinter)
    by simp
  moreover
  note over-new-newinter accmodi-newinter
  ultimately show ?thesis
    by blast
next
  case False
  with override-new-inter
  have Protected ≤ accmodi inter
    by (cases accmodi inter) (auto dest: no-Private-override)
  with override-new-inter override-inter-old same-pack-old-inter
  show ?thesis
    by blast
  qed
qed
next
  case False
  with accmodi-old hyp-inter-old
  obtain newinter where
    over-inter-newinter: G ⊢ inter overrides newinter and
    over-newinter-old: G ⊢ newinter overrides old and
    eq-pid: pid (declclass old) = pid (declclass newinter) and
    accmodi-newinter: Protected ≤ accmodi newinter
    by auto
  from override-new-inter over-inter-newinter
  have G ⊢ new overrides newinter
    by (rule overridesR.Indirect)
  with eq-pid over-newinter-old accmodi-newinter
  show ?thesis
    by blast
  qed
qed
qed

```

**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 (rule class-rec.induct,intro allI impI)
  fix G C c m

```

```

assume Hyp:  $\forall c. C \neq \text{Object} \wedge \text{ws-prog } G \wedge \text{class } G \ C = \text{Some } c$ 
            $\longrightarrow ?P \ G \ (\text{super } c)$ 
assume wf: wf-prog  $G$  and cls-C: class  $G \ C = \text{Some } c$  and
           m: methd  $G \ C \ \text{sig} = \text{Some } m$ 
show  $G \vdash C \preceq_C \text{declclass } m$ 
proof (cases  $C = \text{Object}$ )
  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: ( $?filter \ (\text{methd } G \ (\text{super } c)) \ ++ \ ?table$ )  $\text{sig} = \text{Some } m$ 
        by (simp add: methd-rec)
      show ?thesis
      proof (cases ?table 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  $G \ (\text{super } c) = \text{Some } \text{sup}$ 
              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  $G \vdash C \prec_{C1} \text{super } c$  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  $G \ C \ \text{sig} = \text{Some } m$ 
            $\implies G \vdash (\text{mdecl } (\text{sig}, \text{methd } m)) \text{ declared-in } (\text{declclass } m)$ 
           (is PROP ?P C)
proof (induct ?P C rule: ws-class-induct')
  case Object
    assume methd  $G \ \text{Object} \ \text{sig} = \text{Some } m$ 
    with wf show ?thesis
      by - (rule method-declared-inI, auto)
  next
    case Subcls

```

```

fix C c
assume clsC: class G C = Some c
and      m: methd G C sig = Some m
and      hyp: methd G (super c) sig = Some m  $\implies$  ?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* [consumes 4, case-names NewMethod InheritedMethod]:

```

assumes methd-C: methd G C sig = Some m and
  ws: ws-prog G and
  clsC: class G C = Some c and
  neq-C-Obj: C  $\neq$  Object
shows
   $\llbracket \text{table-of (map } (\lambda(s, m). (s, C, m)) \text{ (methods c)) sig = Some m} \rrbracket \implies P$ ;
   $\llbracket G \vdash C \text{ inherits (method sig m); methd G (super c) sig = Some m} \rrbracket \implies P$ 
proof –
  let ?inherited = filter-tab ( $\lambda \text{sig } m. G \vdash C \text{ inherits method sig } m$ )
    (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)
  assume NewMethod: ?new sig = Some m  $\implies$  P
  assume InheritedMethod:  $\llbracket G \vdash C \text{ inherits (method sig m);$ 
    methd G (super c) sig = Some m  $\rrbracket \implies$  P
  show P
  proof (cases ?new sig)
    case None
    with methd-unfold have ?inherited sig = Some m
      by (auto)
    with InheritedMethod show P by blast
  next
    case Some
    with methd-unfold have ?new sig = Some m
      by auto
    with NewMethod show P by blast
  qed
qed

```

**lemma** *methd-member-of*:

```

assumes wf: wf-prog G
shows
   $\llbracket \text{is-class } G \ C; \text{ methd } G \ C \text{ sig = Some } m \rrbracket \implies G \vdash \text{Methd sig } m \text{ member-of } C$ 
  (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:  $G \vdash \text{Methd sig old member-of (super c)}$ 
  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:  $G \vdash C \prec_{C1} \text{super c}$ 
  by (rule subcls1I)
then have  $G \vdash C \prec_C \text{super c} ..$ 
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 sig = 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  $(\exists \text{new. } (? \text{Constraint } C \ \text{new } \text{old})))$ )
proof -
  from subclseq-C-D is-cls-D
  have is-cls-C: is-class G C by (rule subcls-is-class2)
  from wf
  have ws: ws-prog G ..
  from is-cls-C ws subclseq-C-D

```

```

show  $\exists \text{ new. } ?\text{Constraint } C \text{ new old}$ 
proof (induct rule: ws-class-induct')
  case (Object co)
  then have eq-D-Obj: D=Object by auto
  with old
  have ?Constraint Object old old
    by auto
  with eq-D-Obj
  show  $\exists \text{ new. } ?\text{Constraint Object new old}$  by auto
next
  case (Subcls C c)
  assume hyp:  $G \vdash \text{super } c \preceq_C D \implies \exists \text{ new. } ?\text{Constraint (super } c) \text{ new old}$ 
  assume clsC: class G C = Some c
  assume neq-C-Obj: C  $\neq$  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:  $G \vdash (\text{Class (super } c)) \text{ accessible-in (pid } C)$  and
    subcls1-C-super:  $G \vdash C \prec_{C1} \text{super } c$ 
    by (auto dest: wf-prog-cdecl wf-cdecl-supD is-acc-classD
      intro: subcls1I)
  show  $\exists \text{ new. } ?\text{Constraint } C \text{ new old}$ 
  proof (cases  $G \vdash \text{super } c \preceq_C D$ )
    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  $\exists \text{ new. } ?\text{Constraint } C \text{ new old}$  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  $\leq$  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:  $G \vdash C \prec_C$  (declclass super-method)
  by (auto intro: rtranc1-into-tranc12 dest: methd-declC)
show  $\exists$  new. ?Constraint C new old
proof (cases methd G C sig)
  case None
  have methd G (super c) sig = None
  proof -
    from clsC ws None
    have no-new: table-of (methods c) sig = None
      by (auto simp add: methd-rec)
    with clsC
    have undeclared:  $G \vdash \text{mid sig undeclared-in } C$ 
      by (auto simp add: undeclared-in-def cdeclaredmethd-def)
    with inheritable member superAccessible subcls1-C-super
    have inherits:  $G \vdash C$  inherits (method sig super-method)
      by (auto simp add: inherits-def)
    with clsC ws no-new super neq-C-Obj
    have methd G C sig = Some super-method
      by (auto simp add: methd-rec map-add-def intro: filter-tab-SomeI)
    with None show ?thesis
      by simp
  qed
  with super show ?thesis by auto
next
case (Some new)
from this ws clsC neq-C-Obj
show ?thesis
proof (cases rule: methd-rec-Some-cases)
  case InheritedMethod
  with super Some show ?thesis
    by auto
next
case NewMethod
assume new: table-of (map ( $\lambda(s, m). (s, C, m)$ ) (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:  $\neg$  is-static new
  by (auto dest: wf-prog-staticD)
from clsC new
have decl-new:  $G \vdash \text{Methd sig new declared-in } C$ 
  by (auto simp add: declared-in-def cdeclaredmethd-def)
from not-static-new decl-new decl-super-method
  member subcls1-C-super inheritable declcls-new subcls-C-super
have  $G, \text{sig} \vdash$  new overridesS 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 [consumes 6

, case-names *Inheritance Overriding*]:

**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*  $\leq \text{accmodi old}$  **and**  
*not-static-old*:  $\neg \text{is-static old}$  **and**  
*inheritance*: *methd*  $G C \text{ sig} = \text{Some old} \implies P$  **and**  
*overriding*:  $\bigwedge \text{new. } \llbracket \text{methd } G C \text{ sig} = \text{Some new};$   
 $G, \text{sig} \vdash \text{new overrides}_S \text{ old} \rrbracket \implies P$

**shows**  $P$

**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*  $G D \text{ sig} = \text{Some old}$  **and**  
*accmodi-old*: *Protected*  $\leq \text{accmodi old}$  **and**  
*not-static-old*:  $\neg \text{is-static old}$

**shows**  
 $\exists \text{new. } \text{methd } G C \text{ sig} = \text{Some new} \wedge$   
 $\neg \text{is-static new} \wedge G \vdash \text{resTy new} \preceq_{\text{resTy}} \text{old} \wedge \text{accmodi old} \leq \text{accmodi new}$   
(is ( $\exists \text{new. } (? \text{Constraint } C \text{ new old}))$ )

**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**  $\neg \text{is-static new}$  **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*:

$\llbracket G \vdash I \preceq I J; \text{wf-prog } G; \text{is-iface } G J; \text{jm} \in \text{imethds } G J \text{ sig} \rrbracket \implies$   
 $\exists \text{im} \in \text{imethds } G I \text{ sig. } \text{is-static im} = \text{is-static jm} \wedge$   
 $\text{accmodi im} = \text{accmodi jm} \wedge$   
 $G \vdash \text{resTy im} \preceq_{\text{resTy}} \text{jm}$

**proof** –

**assume** *irel*:  $G \vdash I \preceq I J$  **and**

```

    wf: wf-prog G and
    is-iface: is-iface G J
from irel show  $jm \in imethds\ G\ J\ sig \implies ?thesis$ 
    (is PROP ?P I is PROP ?Prem J  $\implies$  ?Concl I)
proof (induct ?P I rule: converse-rtrancl-induct)
  case Id
    assume  $jm \in imethds\ G\ J\ sig$ 
    then show ?Concl J by (blast elim: bexI')
  next
    case Step
    fix I SI
    assume subint1-I-SI:  $G \vdash I \prec I1\ SI$  and
      subint-SI-J:  $G \vdash SI \preceq I\ J$  and
      hyp: PROP ?P SI and
       $jm: jm \in imethds\ G\ J\ sig$ 
    from subint1-I-SI
    obtain i where
      ifI: iface G I = Some i and
      SI: SI  $\in set\ (isuperIfs\ i)$ 
      by (blast dest: subint1D)

    let ?newMethods
      = (o2s  $\circ$  table-of (map ( $\lambda(sig, mh).$  (sig, I, mh)) (imethods i)))
    show ?Concl I
    proof (cases ?newMethods sig = {})
      case True
        with ifI SI hyp wf jm
        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:  $im \in ?newMethods\ sig$ 
          by (blast)
        with imethds
        have im:  $im \in imethds\ G\ I\ sig$ 
          by (blast)
        with im wf ifI
        obtain
          imStatic:  $\neg is-static\ im$  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 jm hyp
        obtain sim::qtname  $\times$  mhead where
          sim:  $sim \in imethds\ G\ SI\ sig$  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:  $G \vdash \text{resTy } \text{sim} \preceq \text{resTy } \text{jm}$ 
by blast
with wf-I SI indef sim
have  $G \vdash \text{resTy } \text{im} \preceq \text{resTy } \text{sim}$ 
    by (auto dest!: wf-idecl-hidings hidings-entailsD)
with wf resTy-widen-sim-jm
have resTy-widen-im-jm:  $G \vdash \text{resTy } \text{im} \preceq \text{resTy } \text{jm}$ 
    by (blast intro: widen-trans)
from sim wf ifSI
obtain
    simStatic:  $\neg \text{is-static } \text{sim}$  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
qed

```

```

lemma implmt1-methd:
 $\bigwedge \text{sig. } \llbracket G \vdash C \rightsquigarrow 1I; \text{wf-prog } G; \text{im} \in \text{imethds } G \text{ I sig} \rrbracket \implies$ 
 $\exists \text{cm} \in \text{methd } G \text{ C sig: } \neg \text{is-static } \text{cm} \wedge \neg \text{is-static } \text{im} \wedge$ 
 $G \vdash \text{resTy } \text{cm} \preceq \text{resTy } \text{im} \wedge$ 
 $\text{accmodi } \text{im} = \text{Public} \wedge \text{accmodi } \text{cm} = \text{Public}$ 
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)]:
 $\llbracket \text{wf-prog } G; G \vdash C \rightsquigarrow I \rrbracket \implies \text{is-iface } G \text{ I} \longrightarrow$ 
 $(\forall \text{im} \in \text{imethds } G \text{ I sig.}$ 
 $\exists \text{cm} \in \text{methd } G \text{ C sig: } \neg \text{is-static } \text{cm} \wedge \neg \text{is-static } \text{im} \wedge$ 
 $G \vdash \text{resTy } \text{cm} \preceq \text{resTy } \text{im} \wedge$ 
 $\text{accmodi } \text{im} = \text{Public} \wedge \text{accmodi } \text{cm} = \text{Public})$ 
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-rtrancl [THEN inheritable-instance-methd-props,
                                OF - implmt-is-class])
apply  auto
done

```

```

lemma mheadsD [rule-format (no-asm)]:
emh ∈ mheads G S t sig ⟶ wf-prog G ⟶
(∃ C D m. t = ClassT C ∧ declrefT emh = ClassT D ∧
  accmethd G S C sig = Some m ∧
  (declclass m = D) ∧ mhead (methd m) = (mhd emh)) ∨
(∃ I. t = IfaceT I ∧ ((∃ im. im ∈ accimethds G (pid S) I sig ∧
  methd im = mhd emh) ∨
  (∃ m. G⊢Iface I accessible-in (pid S) ∧ accmethd G S Object sig = Some m ∧
  accmodi m ≠ Private ∧
  declrefT emh = ClassT Object ∧ mhead (methd m) = mhd emh))) ∨
(∃ T m. t = ArrayT T ∧ G⊢Array T accessible-in (pid S) ∧
  accmethd G S Object sig = Some m ∧ accmodi m ≠ Private ∧
  declrefT emh = ClassT Object ∧ mhead (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 [consumes 2, case-names Class-methd
                    Iface-methd Iface-Object-methd Array-Object-methd]:
⌊emh ∈ mheads G S t sig; wf-prog G;
  ∧ C D m. ⌊t = ClassT C; declrefT emh = ClassT D; accmethd G S C sig = Some m;
  (declclass m = D); mhead (methd m) = (mhd emh)⌋ ⟹ P emh;
  ∧ I im. ⌊t = IfaceT I; im ∈ accimethds G (pid S) I sig; methd im = mhd emh⌋
    ⟹ P emh;
  ∧ I m. ⌊t = IfaceT I; G⊢Iface I accessible-in (pid S);
  accmethd G S Object sig = Some m; accmodi m ≠ Private;
  declrefT emh = ClassT Object; mhead (methd m) = mhd emh⌋ ⟹ P emh;
  ∧ T m. ⌊t = ArrayT T; G⊢Array T accessible-in (pid S);
  accmethd G S Object sig = Some m; accmodi m ≠ Private;
  declrefT emh = ClassT Object; mhead (methd m) = mhd emh⌋ ⟹ P emh
⌋ ⟹ P emh
by (blast dest!: mheadsD)

```

```

lemma declclassD[rule-format]:
⌊wf-prog G; class G C = Some c; methd G C sig = Some m;
  class G (declclass m) = Some d⌋
  ⟹ table-of (methods d) sig = Some (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  $\bigwedge m d. \llbracket \text{methd } G \ C \ \text{sig} = \text{Some } m; \text{class } G \ (\text{declclass } m) = \text{Some } d \rrbracket$ 
   $\implies \text{table-of } (\text{methods } d) \ \text{sig} = \text{Some } (\text{methd } m)$ 
  (is PROP ?P C)
proof (induct ?P C rule: ws-class-induct)
  case Object
  fix m d
  assume methd G Object sig = Some m
    class G (declclass m) = Some d
  with wf show ?thesis m d by auto
next
  case Subcls
  fix C c m d
  assume hyp: PROP ?P (super c)
  and m: methd G C sig = Some m
  and declC: class G (declclass m) = Some d
  and clsC: class G C = Some c
  and nObj: C  $\neq$  Object
  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 clsC nObj ws m declC
    show ?thesis by (auto simp add: methd-rec) (rule hyp)
  next
    case Some
    from Some clsC nObj ws m declC
    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
  {
    fix I i new
    assume ifI: iface G I = Some i
    assume new: table-of (imethds i) sig = Some new
    from ifI new not-private wf old
    have ?P (I,new)
      by (auto dest!: wf-prog-idecl wf-idecl-hiding cond-hiding-entailsD
          simp del: methd-Object)
  } note hyp-newmethod = this
  from is-if-I ws new
  show ?thesis
proof (induct rule: ws-interface-induct)
  case (Step I i)
  assume ifI: iface G I = Some i
  assume new: new  $\in$  imethds G I sig
  from Step
  have hyp:  $\forall J \in \text{set (isuperIfs i)}. (new \in \text{imethds G J sig} \longrightarrow ?P \text{ new})$ 
    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  $\text{statT}$  and a dynamic class  $\text{dynC}$ . Between both of these types the widening relation holds  $G \vdash \text{Class dynC} \preceq \text{statT}$ . 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 Array / Object Object

In most cases we can lookup the member in the dynamic class. But as an interface can't declare new static methods, nor an array can define new methods at all, we have to lookup methods in the base class `Object`.

The limitation to classes in the field column is artificial and comes out of the typing rule for the field access (see rule *FVar* in the welltyping relation *wt* in theory *WellType*). It stems out of the fact, that `Object` indeed has no non private fields. So interfaces and arrays can actually have no fields at all and a field access would be senseless. (In Java interfaces are allowed to declare new fields but in current Bali not!). So there is no principal reason why we should not allow `Objects` to declare non private fields. Then we would get the following column:

statT field ————— NullT / Iface Object Class dynC Array Object

**consts** *valid-lookup-clis*:: *prog*  $\Rightarrow$  *ref-ty*  $\Rightarrow$  *qtname*  $\Rightarrow$  *bool*  $\Rightarrow$  *bool*  
 ( $\cdot, - \vdash - \text{valid}'\text{-lookup}'\text{-cls}'\text{-for} - [61, 61, 61, 61] \ 60$ )

**primrec**

$G, \text{NullT} \vdash \text{dynC } \text{valid-lookup-clis-for static-membr} = \text{False}$

$G, \text{IfaceT } I \vdash \text{dynC } \text{valid-lookup-clis-for static-membr}$   
 $= (\text{if static-membr}$   
      $\text{then dynC} = \text{Object}$   
      $\text{else } G \vdash \text{Class dynC} \preceq \text{Iface } I)$

$G, \text{ClassT } C \vdash \text{dynC } \text{valid-lookup-clis-for static-membr} = G \vdash \text{Class dynC} \preceq \text{Class } C$

$G, \text{ArrayT } T \vdash \text{dynC } \text{valid-lookup-clis-for static-membr} = (\text{dynC} = \text{Object})$

**lemma** *valid-lookup-clis-is-class*:

**assumes** *dynC*:  $G, \text{statT} \vdash \text{dynC } \text{valid-lookup-clis-for static-membr}$  **and**  
     *ty-statT*: *isrtype*  $G \text{ statT}$  **and**  
     *wf*: *wf-prog*  $G$

**shows** *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*]

**declaration**  $\ll K (\text{Simplifier.map-ss } (fn \text{ ss } \Rightarrow \text{ ss delloop split-all-tac})) \gg$

**declaration**  $\ll K (\text{Classical.map-cs } (fn \text{ cs } \Rightarrow \text{ cs delSWrapper split-all-tac})) \gg$

**lemma** *dynamic-mheadsD*:

$\ll emh \in mheads \ G \ S \ \text{statT} \ \text{sig};$

```

   $G, statT \vdash dynC \text{ valid-lookup-clsf-for } (is-static \ emh);$ 
   $isrtype \ G \ statT; wf\text{-prog } G$ 
 $\Downarrow \implies \exists m \in dynlookup \ G \ statT \ dynC \ sig:$ 
   $is-static \ m = is-static \ emh \wedge G \vdash resTy \ m \preceq resTy \ emh$ 
proof –
  assume     $emh: emh \in mheads \ G \ S \ statT \ sig$ 
  and       $wf: wf\text{-prog } G$ 
  and     $dynC\text{-Prop}: G, statT \vdash dynC \text{ valid-lookup-clsf-for } (is-static \ emh)$ 
  and     $istype: isrtype \ G \ statT$ 
  from  $dynC\text{-Prop} \ istype \ wf$ 
  obtain  $y$  where
     $dynC: class \ G \ dynC = Some \ y$ 
    by  $(auto \ dest: \text{valid-lookup-clsf-is-class})$ 
  from  $emh \ wf$  show  $?thesis$ 
proof  $(cases \ rule: mheads\text{-cases})$ 
  case  $Class\text{-methd}$ 
  fix  $statC \ statDeclC \ sm$ 
  assume     $statC: statT = ClassT \ statC$ 
  assume     $accmethd \ G \ S \ statC \ sig = Some \ sm$ 
  then have     $sm: methd \ G \ statC \ sig = Some \ sm$ 
    by  $(blast \ dest: accmethd\text{-SomeD})$ 
  assume  $eq\text{-mheads}: mhead \ (methd \ sm) = mhd \ emh$ 
  from  $statC$ 
  have  $dynlookup: dynlookup \ G \ statT \ dynC \ sig = dynmethd \ G \ statC \ dynC \ sig$ 
    by  $(simp \ add: dynlookup\text{-def})$ 
  from  $wf \ statC \ istype \ dynC\text{-Prop} \ sm$ 
  obtain  $dm$  where
     $dynmethd \ G \ statC \ dynC \ sig = Some \ dm$ 
     $is-static \ dm = is-static \ sm$ 
     $G \vdash resTy \ dm \preceq resTy \ sm$ 
    by  $(force \ dest!: ws\text{-dynmethd} \ accmethd\text{-SomeD})$ 
  with  $dynlookup \ eq\text{-mheads}$ 
  show  $?thesis$ 
    by  $(cases \ emh \ type: *) \ (auto)$ 
next
  case  $Iface\text{-methd}$ 
  fix  $I \ im$ 
  assume     $statI: statT = IfaceT \ I$  and
     $eq\text{-mheads}: methd \ im = mhd \ emh$  and
     $im \in accimethds \ G \ (pid \ S) \ I \ sig$ 
  then have  $im: im \in imethds \ G \ I \ sig$ 
    by  $(blast \ dest: accimethdsD)$ 
  with  $istype \ statI \ eq\text{-mheads} \ wf$ 
  have  $not\text{-static-emh}: \neg is-static \ emh$ 
    by  $(cases \ emh) \ (auto \ dest: wf\text{-prog-idecl} \ imethds\text{-wf-mhead})$ 
  from  $statI \ im$ 
  have  $dynlookup: dynlookup \ G \ statT \ dynC \ sig = methd \ G \ dynC \ sig$ 
    by  $(auto \ simp \ add: dynlookup\text{-def} \ dynimethd\text{-def})$ 
  from  $wf \ dynC\text{-Prop} \ statI \ istype \ im \ not\text{-static-emh}$ 
  obtain  $dm$  where
     $methd \ G \ dynC \ sig = Some \ dm$ 
     $is-static \ dm = is-static \ im$ 
     $G \vdash resTy \ (methd \ dm) \preceq resTy \ (methd \ im)$ 
    by  $(force \ dest: implmt\text{-methd})$ 
  with  $dynlookup \ eq\text{-mheads}$ 
  show  $?thesis$ 
    by  $(cases \ emh \ type: *) \ (auto)$ 
next
  case  $Iface\text{-Object-methd}$ 

```

```

fix  $I$   $sm$ 
assume  $statI: statT = IfaceT\ I$  and
          $sm: accmethd\ G\ S\ Object\ sig = Some\ sm$  and
          $eq\ mheads: mhead\ (mthd\ sm) = mhd\ emh$  and
          $nPriv: accmodi\ sm \neq Private$ 
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  $subclsObj: G \vdash dynC \preceq_C Object$ 
    by (auto intro: subcls-ObjectI)
  from  $wf\ dynC\ dynC\ Prop\ istype\ sm\ subclsObj$ 
  obtain  $dm$  where
     $dynmethd\ G\ Object\ dynC\ sig = Some\ dm$ 
     $is-static\ dm = is-static\ sm$ 
     $G \vdash resTy\ (mthd\ dm) \preceq resTy\ (mthd\ sm)$ 
    by (auto dest!: ws-dynmethd accmethd-SomeD
          intro: class-Object [OF wf] intro: that)
  with  $dynlookup\ eq\ mheads$ 
  show  $?thesis$ 
    by (cases  $emh\ type: *$ ) (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
     $im: im \in imethds\ G\ I\ sig$  and
     $eq-stat: is-static\ im = is-static\ sm$  and
     $resProp: G \vdash resTy\ (mthd\ im) \preceq resTy\ (mthd\ sm)$ 
    by (auto dest: wf-imethds-hiding-objmethodsD accmethd-SomeD)
  from  $im\ wf\ statI\ istype\ eq-stat\ eq\ mheads$ 
  have  $not-static-sm: \neg is-static\ emh$ 
    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
     $methd\ G\ dynC\ sig = Some\ dm$ 
     $is-static\ dm = is-static\ im$ 
     $G \vdash resTy\ (mthd\ dm) \preceq resTy\ (mthd\ im)$ 
    by (auto dest: implmt-methd)
  with  $wf\ eq-stat\ resProp\ dynlookup\ eq\ mheads$ 
  show  $?thesis$ 
    by (cases  $emh\ type: *$ ) (auto intro: widen-trans)
  qed
next
  case  $Array-Object-methd$ 
  fix  $T\ sm$ 
  assume  $statArr: statT = ArrayT\ T$  and
          $sm: accmethd\ G\ S\ Object\ sig = Some\ sm$  and
          $eq\ mheads: mhead\ (mthd\ sm) = mhd\ emh$ 
  from  $statArr\ dynC\ Prop\ wf$ 
  have  $dynlookup: dynlookup\ G\ statT\ dynC\ sig = methd\ G\ Object\ sig$ 

```

```

    by (auto simp add: dynlookup-def dynmethd-C-C)
  with sm eq-mheads sm
  show ?thesis
    by (cases emh type: *) (auto dest: accmethd-SomeD)
qed
qed
declare split-paired-All [simp] split-paired-Ex [simp]
declaration  $\ll K (Classical.map\text{-}cs (fn cs => cs addSbefore (split\text{-}all\text{-}tac, split\text{-}all\text{-}tac))) \gg$ 
declaration  $\ll K (Simplifier.map\text{-}ss (fn ss => ss addloop (split\text{-}all\text{-}tac, split\text{-}all\text{-}tac))) \gg$ 

```

**lemma** *methd-declclass*:

$\ll class\ G\ C = Some\ c; wf\text{-}prog\ G; methd\ G\ C\ sig = Some\ m \gg$

$\implies methd\ G\ (declclass\ m)\ sig = Some\ m$

**proof** –

**assume** *asm*:  $class\ G\ C = Some\ c\ wf\text{-}prog\ G\ methd\ G\ C\ sig = Some\ m$

**have**  $wf\text{-}prog\ G \longrightarrow$

$(\forall\ c\ m. class\ G\ C = Some\ c \longrightarrow methd\ G\ C\ sig = Some\ m$   
 $\longrightarrow methd\ G\ (declclass\ m)\ sig = Some\ m) \quad (is\ ?P\ G\ C)$

**proof** (rule *class-rec.induct,intro allI impI*)

**fix**  $G\ C\ c\ m$

**assume** *hyp*:  $\forall\ c. C \neq Object \wedge ws\text{-}prog\ G \wedge class\ G\ C = Some\ c \longrightarrow$   
 $?P\ G\ (super\ c)$

**assume** *wf*:  $wf\text{-}prog\ G$  **and** *cls-C*:  $class\ G\ C = Some\ c$  **and**

$m: methd\ G\ C\ sig = Some\ m$

**show**  $methd\ G\ (declclass\ m)\ sig = Some\ m$

**proof** (cases  $C=Object$ )

**case** *True*

**with** *wf m* **show** *?thesis* **by** (auto intro: *table-of-map-SomeI*)

**next**

**let** *?filter*=*filter-tab* ( $\lambda sig\ m. G \vdash C\ inherits\ method\ sig\ m$ )

**let** *?table* = *table-of* ( $\lambda (s, m). (s, C, m)$ ) (*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** **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]

**declaration**  $\ll K \text{ (Simplifier.map-ss (fn ss => ss delloop split-all-tac))} \gg$

**declaration**  $\ll K \text{ (Classical.map-cs (fn cs => cs delSWrapper split-all-tac))} \gg$

**lemma** *wt-is-type*:  $E, dt \models v :: T \Rightarrow \text{wf-prog (prg } E) \longrightarrow$

```

  dt = empty-dt  $\longrightarrow$  (case T of
    Inl T  $\Rightarrow$  is-type (prg E) T
  | Inr Ts  $\Rightarrow$  Ball (set Ts) (is-type (prg E)))

```

**apply** (unfold empty-dt-def)

**apply** (erule wt.induct)

**apply** (auto split del: split-if-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]

**declaration**  $\ll K (Classical.map\text{-}cs (fn cs => cs addSbefore (split\text{-}all\text{-}tac, split\text{-}all\text{-}tac))) \gg$   
**declaration**  $\ll K (Simplifier.map\text{-}ss (fn ss => ss addloop (split\text{-}all\text{-}tac, split\text{-}all\text{-}tac))) \gg$

**lemma** *ty-expr-is-type*:  
 $\ll E \vdash e :: -T; wf\text{-}prog (prg E) \gg \implies is\text{-}type (prg E) T$   
**by** (auto dest!: wt-is-type)

**lemma** *ty-var-is-type*:  
 $\ll E \vdash v :: T; wf\text{-}prog (prg E) \gg \implies is\text{-}type (prg E) T$   
**by** (auto dest!: wt-is-type)

**lemma** *ty-exprs-is-type*:  
 $\ll E \vdash es :: Ts; wf\text{-}prog (prg E) \gg \implies Ball (set Ts) (is\text{-}type (prg E))$   
**by** (auto dest!: wt-is-type)

**lemma** *static-mheadsD*:  
 $\ll emh \in mheads G S t sig; wf\text{-}prog G; E \vdash e :: -RefT t; prg E = G ;$   
 $invmode (mhd emh) e \neq IntVir$   
 $\gg \implies \exists m. ( (\exists C. t = ClassT C \wedge accmethd G S C sig = Some m)$   
 $\vee (\forall C. t \neq ClassT C \wedge accmethd G S Object sig = Some m)) \wedge$   
 $declrefT emh = ClassT (declclass m) \wedge mhead (mthd m) = (mhd emh)$   
**apply** (subgoal-tac is-static emh  $\vee e = Super$ )  
**defer apply** (force simp add: invmode-def)  
**apply** (frule ty-expr-is-type)  
**apply** simp  
**apply** (case-tac is-static emh)  
**apply** (frule (1) mheadsD)  
**apply** clarsimp  
**apply** safe  
**apply** blast  
**apply** (auto dest!: imethds-wf-mhead  
 $accmethd\text{-}SomeD$   
 $accimethdsD$   
 $simp add: accObjectmheads\text{-}def Objectmheads\text{-}def$ )  
**apply** (erule wt-elim-cases)  
**apply** (force simp add: cmheads-def)  
**done**

**lemma** *wt-MethdI*:  
 $\ll methd G C sig = Some m; wf\text{-}prog G;$   
 $class G C = Some c \gg \implies$   
 $\exists T. (\ll prg = G, cls = (declclass m),$   
 $lcl = callee\text{-}lcl (declclass m) sig (mthd m) \gg \vdash Methd C sig :: -T \wedge G \vdash T \preceq_{resTy} m$   
**apply** (frule (2) methd-wf-mdecl, clarify)  
**apply** (force dest!: wf-mdecl-bodyD intro!: wt.Methd)  
**done**

### 35 accessibility concerns

**lemma** *mheads-type-accessible*:  
 $\ll emh \in mheads G S T sig; wf\text{-}prog G \gg$   
 $\implies G \vdash RefT T accessible\text{-}in (pid S)$   
**by** (erule mheads-cases)  
 $(auto dest: accmethd\text{-}SomeD accessible\text{-}from\text{-}commonD accimethdsD)$



```

lemma static-to-dynamic-accessible-from-aux:
   $\llbracket G \vdash m \text{ of } C \text{ accessible-from } accC; wf\text{-prog } G \rrbracket$ 
   $\implies G \vdash m \text{ in } C \text{ dyn-accessible-from } accC$ 
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:  $G \vdash m \text{ of } statC \text{ accessible-from } accC$  and
    subclseq:  $G \vdash dynC \preceq_C statC$  and
    wf: wf-prog G
  shows  $G \vdash m \text{ in } dynC \text{ dyn-accessible-from } accC$ 
proof -
  from stat-acc subclseq
  show ?thesis (is ?Dyn-accessible m)
  proof (induct rule: accessible-fromR.induct)
    case (Immediate m statC)
    then show ?Dyn-accessible m
    by (blast intro: dyn-accessible-fromR.Immediate
          member-inI
          permits-acc-inheritance)
  next
    case (Overriding m -)
    with wf show ?Dyn-accessible m
    by (blast intro: dyn-accessible-fromR.Overriding
          member-inI
          static-to-dynamic-overriding
          rtrancl-trancl-trancl
          static-to-dynamic-accessible-from-aux)
  qed
qed

```

```

lemma static-to-dynamic-accessible-from-static:
  assumes stat-acc:  $G \vdash m \text{ of } statC \text{ accessible-from } accC$  and
    static: is-static m and
    wf: wf-prog G
  shows  $G \vdash m \text{ in } (declclass m) \text{ dyn-accessible-from } accC$ 
proof -
  from stat-acc wf
  have  $G \vdash m \text{ in } statC \text{ dyn-accessible-from } accC$ 
  by (auto intro: static-to-dynamic-accessible-from)
  from this static
  show ?thesis
  by (rule dyn-accessible-from-static-declC)
qed

```

```

lemma dynmethd-member-in:
  assumes m: dynmethd G statC dynC sig = Some m and
    iscls-statC: is-class G statC and
    wf: wf-prog G
  shows  $G \vdash Methd \text{ sig } m \text{ member-in } dynC$ 
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
     $\text{pub: accmodi dyn}M = \text{Public}$ 
  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  $\neq \{\}$  and  
implements:  $G \vdash \text{dynC} \rightsquigarrow I$  and  
isif-I: is-iface G I and  
wf: wf-prog G  
shows  $\exists \text{ dynM}. \text{dynimethd } G \text{ I } \text{dynC sig} = \text{Some dynM} \wedge$   
 $G \vdash \text{Methd sig dynM in dynC dyn-accessible-from accC}$

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:  $\text{emh} \in \text{mheads } G \text{ accC statT sig}$  and  
dynM:  $\text{dynlookup } G \text{ statT dynC sig} = \text{Some dynM}$  and  
dynC-prop:  $G, \text{statT} \vdash \text{dynC valid-lookup-cls-for is-static emh}$  and  
isT-statT:  $\text{isrtype } G \text{ statT}$  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:  $G \vdash \text{RefT statT accessible-in (pid accC)}$   
by (rule mheads-type-accessible)  
from dynC-prop isT-statT wf  
have iscls-dynC:  $\text{is-class } G \text{ dynC}$   
by (rule valid-lookup-cls-is-class)  
from emh dynC-prop isT-statT wf dynM  
have eq-static:  $\text{is-static emh} = \text{is-static dynM}$   
by (auto dest: dynamic-mheadsD)  
from emh wf show ?thesis  
proof (cases rule: mheads-cases)  
case (Class-methd statC - statM)  
assume statT:  $\text{statT} = \text{ClassT statC}$   
assume accmethd  $G \text{ accC statC sig} = \text{Some statM}$   
then have  $\text{statM: methd } G \text{ statC sig} = \text{Some statM}$  and  
 $\text{stat-acc: } G \vdash \text{Methd sig statM of statC accessible-from accC}$   
by (auto dest: accmethd-SomeD)  
from dynM statT  
have dynM':  $\text{dynmethd } G \text{ statC dynC sig} = \text{Some dynM'}$   
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:  $\text{imethds } G \text{ I sig} \neq \{\}$  and  
 $\text{statT-acc: } G \vdash \text{RefT statT accessible-in (pid accC)}$   
by (auto dest: accimethdsD)  
assume  $\text{statT: statT} = \text{IfaceT I}$

```

assume    im: im ∈ accimethds G (pid accC) I sig
assume eq-mhds: mthd im = mhd emh
from dynM statT
have dynM': dynimethd G I dynC sig = Some dynM
  by (simp add: dynlookup-def)
from isT-statT statT
have isif-I: is-iface G I
  by simp
show ?thesis
proof (cases is-static emh)
  case False
  with statT dynC-prop
  have widen-dynC:  $G \vdash \text{Class } \text{dynC} \preceq \text{RefT } \text{statT}$ 
    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: statT = IfaceT I
assume accmethd G accC Object sig = Some statM
then have    statM: methd G Object sig = Some statM and
      stat-acc:  $G \vdash \text{Methd } \text{sig } \text{statM} \text{ of } \text{Object} \text{ accessible-from } \text{accC}$ 
    by (auto dest: accmethd-SomeD)
assume not-Private-statM: accmodi statM ≠ Private
assume eq-mhds: mhead (mthd statM) = mhd 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': dynmethd G Object dynC sig = Some 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:  $\neg$  is-static emh
  proof –
    from im statM statT isT-statT wf not-Private-statM
    have is-static im = is-static statM
      by (fastsimp dest: wf-imethds-hiding-objmethdsD)
    with wf isT-statT statT im
    have  $\neg$  is-static statM
      by (auto dest: wf-prog-idecl imethds-wf-mhead)
    with eq-mhds
    show ?thesis
      by (cases emh) auto
  qed
  with statT dynC-prop
  have implmnt:  $G \vdash \text{dynC} \rightsquigarrow I$ 
    by simp
  with isT-statT statT
  have isif-I: is-iface G I
    by simp
  from dynM statT
  have dynM': dynimethd G I dynC sig = Some dynM
    by (simp add: dynlookup-def)
  from False implmnt isif-I wf dynM'
  show ?thesis
    by – (drule implmt-dynimethd-access, auto)
  qed
next
  case (Array-Object-methd T statM)
  assume statT: statT = ArrayT T
  assume accmethd G accC Object sig = Some statM
  then have statM: methd G Object sig = Some statM and
    stat-acc:  $G \vdash \text{Methd } sig \text{ statM of Object accessible-from } accC$ 
    by (auto dest: accmethd-SomeD)
  from statT dynC-prop
  have dynC-Obj: dynC = Object
    by simp
  then
  have widen-dynC-Obj:  $G \vdash \text{Class } dynC \preceq \text{Class } Object$ 
    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:  $G, statT \vdash dynC \text{ valid-lookup-cls-for } (is-static \text{ emh})$  and
    isT-statT: isrtype G statT and
    wf: wf-prog G

```

```

shows  $\exists \text{ dynM}. \text{dynlookup } G \text{ statT dynC sig} = \text{Some dynM} \wedge$ 
 $G \vdash \text{Methd sig dynM in dynC dyn-accessible-from accC}$ 
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  $\langle \text{accmodi } m = \text{Private} \rangle$ 
    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:
   $\llbracket G \vdash m \text{ in } C \text{ dyn-accessible-from accC}; \text{accmodi } m = \text{Package};$ 
 $\text{wf-prog } G \rrbracket$ 
 $\implies \text{pid accC} = \text{pid } (\text{declclass } m)$ 
proof –
  assume wf: wf-prog G

```

```

assume accessible:  $G \vdash m \text{ in } C \text{ dyn-accessible-from } accC$ 
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  $G \vdash m \text{ member-in } C$ 
     $G \vdash m \text{ in } C \text{ permits-acc-from } accC$ 
     $accmodi\ m = Package$ 
    then show  $?P\ m$ 
    by (auto simp add: permits-acc-def)
  next
    case (Overriding  $new\ C\ declC\ newm\ old\ Sup$ )
    assume member-new:  $G \vdash new \text{ member-in } C$  and
      new:  $new = (declC, mdecl\ newm)$  and
      override:  $G \vdash (declC, newm) \text{ overrides } old$  and
      subcls-C-Sup:  $G \vdash C \prec_C Sup$  and
      acc-old:  $G \vdash methdMembr\ old \text{ in } Sup \text{ dyn-accessible-from } accC$  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:  $G \vdash f \text{ in } C \text{ dyn-accessible-from } accC$  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  $\langle G \vdash f \text{ in } C \text{ permits-acc-from } accC \rangle$  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:  $G \vdash f \text{ in } C \text{ dyn-accessible-from } accC$  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 (declclass f)  $\neq$  pid accC
    ultimately
    show  $G \vdash C \preceq_C accC$ 
      by (auto simp add: permits-acc-def)
  next
    case Overriding
    then show ?case by (simp add: is-field-def)
  qed
qed

```

**lemma** *dyn-accessible-static-field-Protected:*

```

  assumes dyn-acc:  $G \vdash f \text{ in } C \text{ dyn-accessible-from } accC$  and
    prot: accmodi f = Protected and
    field: is-field f and
    static-field: is-static f and
    outside: pid (declclass f)  $\neq$  pid accC
  shows  $G \vdash accC \preceq_C declclass f \wedge G \vdash C \preceq_C declclass f$ 
proof -
  from dyn-acc prot field static-field outside
  show ?thesis
  proof (induct)
    case (Immediate f C)
    assume accmodi f = Protected and is-field f and is-static f and
      pid (declclass f)  $\neq$  pid accC
    moreover
    note  $\langle G \vdash f \text{ in } C \text{ permits-acc-from } accC \rangle$ 
    ultimately
    have  $G \vdash accC \preceq_C declclass f$ 
      by (auto simp add: permits-acc-def)
    moreover
    from  $\langle G \vdash f \text{ member-in } C \rangle$ 
    have  $G \vdash C \preceq_C declclass f$ 
      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

### 36 State for evaluation of Java expressions and statements

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

design issues:

- all kinds of objects (class instances, arrays, and class objects) are handled via a general object abstraction
- the heap and the map for class objects are combined into a single table (*recall* (*loc*, *obj*) *table*  $\times$  (*qname*, *obj*) *table*  $\sim =$  (*loc* + *qname*, *obj*) *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)

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

#### translations

*fspec* <= (*type*) *vname*  $\times$  *qname*  
*vn* <= (*type*) *fspec* + *int*  
*obj* <= (*type*) (*tag*::*obj-tag*, *values*::*vn*  $\Rightarrow$  *val option*)  
*obj* <= (*type*) (*tag*::*obj-tag*, *values*::*vn*  $\Rightarrow$  *val option*,...::'*a*)

#### constdefs

*the-Arr* :: *obj option*  $\Rightarrow$  *ty*  $\times$  *int*  $\times$  (*vn*, *val*) *table*  
*the-Arr obj*  $\equiv$  *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*]:  
 $\llbracket \text{tag } obj = \text{Arr } T \ k \rrbracket \Longrightarrow \text{the-Arr } (\text{Some } obj) = (T,k,\text{values } obj)$   
**apply** (*auto simp add: the-Arr-def*)  
**done**

#### constdefs

*upd-obj* :: *vn*  $\Rightarrow$  *val*  $\Rightarrow$  *obj*  $\Rightarrow$  *obj*  
*upd-obj n v*  $\equiv$   $\lambda$  *obj* . *obj* (*values:=*(*values obj*)(*n* $\mapsto$ *v*))

**lemma** *upd-obj-def2* [*simp*]:  
 $\text{upd-obj } n \ v \ obj = obj \ (\text{values:=}(\text{values } obj)(n \mapsto v))$   
**apply** (*auto simp: upd-obj-def*)  
**done**

**constdefs**

$$\begin{aligned}
obj\text{-}ty &:: obj \Rightarrow ty \\
obj\text{-}ty\ obj &\equiv \text{case tag } obj \text{ of} \\
&\quad CInst\ C \Rightarrow Class\ C \\
&\quad | Arr\ T\ k \Rightarrow T.[]
\end{aligned}$$

**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 add: obj-tag.split-asm*)  
**done**

**constdefs**

$$\begin{aligned}
obj\text{-}class &:: obj \Rightarrow qname \\
obj\text{-}class\ obj &\equiv \text{case tag } obj \text{ of} \\
&\quad CInst\ C \Rightarrow C \\
&\quad | Arr\ T\ k \Rightarrow Object
\end{aligned}$$

**lemma** *obj-class-CInst* [simp]:  $obj\text{-}class\ (\llbracket tag=CInst\ C, values=vs \rrbracket) = C$   
**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* ( $\text{tag}=\text{Arr } T \text{ } k, \text{values}=vs$ ) = *Object*  
**by** (*auto simp: obj-class-def*)

**lemma** *obj-class-Arr1* [*simp*,*intro!*,*dest*]:  
 $\text{tag } \text{obj} = \text{Arr } T \text{ } k \implies \text{obj-class } \text{obj} = \text{Object}$   
**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

**types** *oref* = *loc* + *qname* — generalized object reference

### syntax

*Heap* :: *loc*  $\Rightarrow$  *oref*  
*Stat* :: *qname*  $\Rightarrow$  *oref*

### translations

*Heap*  $\Rightarrow$  *Inl*  
*Stat*  $\Rightarrow$  *Inr*  
*oref*  $\leq$  (*type*) *loc* + *qname*

### constdefs

*fields-table*::  
 $\text{prog} \Rightarrow \text{qname} \Rightarrow (\text{fspec} \Rightarrow \text{field} \Rightarrow \text{bool}) \Rightarrow (\text{fspec}, \text{ty}) \text{ table}$   
*fields-table* *G C P*  
 $\equiv \text{option-map } \text{type} \circ \text{table-of } (\text{filter } (\text{split } P) (\text{DeclConcepts.fields } G \text{ } C))$

**lemma** *fields-table-SomeI*:  
 $\llbracket \text{table-of } (\text{DeclConcepts.fields } G \text{ } C) \text{ } n = \text{Some } f; P \text{ } n \text{ } f \rrbracket$   
 $\implies \text{fields-table } G \text{ } C \text{ } P \text{ } n = \text{Some } (\text{type } f)$   
**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*  $\implies$   
 $\exists f. (fn, f) \in \text{set}(\text{DeclConcepts.fields } G \text{ } C) \wedge \text{type } f = T$   
**apply** (*unfold fields-table-def*)

```

apply clarsimp
apply (drule 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

```

```

constdefs
in-bounds :: int  $\Rightarrow$  int  $\Rightarrow$  bool          ((-/ in'-bounds -) [50, 51] 50)
i in-bounds k  $\equiv 0 \leq i \wedge i < k$ 

```

```

arr-comps :: 'a  $\Rightarrow$  int  $\Rightarrow$  int  $\Rightarrow$  'a option
arr-comps T k  $\equiv \lambda i. \text{if } i \text{ in-bounds } k \text{ then } \text{Some } T \text{ else } \text{None}$ 

```

```

var-tys      :: prog  $\Rightarrow$  obj-tag  $\Rightarrow$  oref  $\Rightarrow$  (vn, ty) table
var-tys G oi r
 $\equiv \text{case } r \text{ of}$ 
  Heap a  $\Rightarrow$  (case oi of
    CInst C  $\Rightarrow$  fields-table G C ( $\lambda n f. \neg \text{static } f$ ) (+) empty
    | Arr T k  $\Rightarrow$  empty (+) arr-comps T k)
  | Stat C  $\Rightarrow$  fields-table G C ( $\lambda fn f. \text{declclassf } fn = C \wedge \text{static } f$ )
    (+) empty

```

```

lemma var-tys-Some-eq:
var-tys G oi r n = Some T
 $= (\text{case } r \text{ of}$ 
  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 add: sum.split-asm sum.split obj-tag.split)
done

```

## stores

```

types  globs          — global variables: heap and static variables
      = (oref , obj) table
      heap
      = (loc , obj) table

```

## translations

```

globs <= (type) (oref , obj) table

```

*heap* ≤ (type) (loc , obj) table

**datatype** *st* =  
     *st* globs locals

### 37 access

**constdefs**

*globs* :: *st* ⇒ *globs*  
*globs* ≡ *st-case* (λ*g l. g*)

*locals* :: *st* ⇒ *locals*  
*locals* ≡ *st-case* (λ*g l. l*)

*heap* :: *st* ⇒ *heap*  
*heap* *s* ≡ *globs* *s* ∘ *Heap*

**lemma** *globs-def2* [*simp*]: *globs* (*st* *g l*) = *g*  
**by** (*simp* *add: globs-def*)

**lemma** *locals-def2* [*simp*]: *locals* (*st* *g l*) = *l*  
**by** (*simp* *add: locals-def*)

**lemma** *heap-def2* [*simp*]: *heap* *s* *a* = *globs* *s* (*Heap* *a*)  
**by** (*simp* *add: heap-def*)

**syntax**

*val-this* :: *st* ⇒ *val*  
*lookup-obj* :: *st* ⇒ *val* ⇒ *obj*

**translations**

*val-this* *s* == *the* (*locals* *s* *This*)  
*lookup-obj* *s* *a'* == *the* (*heap* *s* (*the-Addr* *a'*))

### 38 memory allocation

**constdefs**

*new-Addr* :: *heap* ⇒ *loc option*  
*new-Addr* *h* ≡ *if* (∀ *a. h* *a* ≠ *None*) *then* *None* *else* *Some* (*SOME* *a. h* *a* = *None*)

**lemma** *new-AddrD*: *new-Addr* *h* = *Some* *a* ⇒ *h* *a* = *None*  
**apply** (*auto simp add: new-Addr-def*)  
**apply** (*erule someI*)  
**done**

**lemma** *new-AddrD2*: *new-Addr* *h* = *Some* *a* ⇒ ∀ *b. h* *b* ≠ *None* → *b* ≠ *a*  
**apply** (*drule new-AddrD*)  
**apply** *auto*  
**done**



```

lemma new-Addr-SomeI:  $h\ a = \text{None} \implies \exists b. \text{new-Addr } h = \text{Some } b \wedge h\ b = \text{None}$ 
apply (simp add: new-Addr-def)
apply (fast intro: someI2)
done

```

### 39 initialization

#### syntax

$\text{init-vals} \quad :: ('a, ty)\ \text{table} \Rightarrow ('a, val)\ \text{table}$

#### translations

$\text{init-vals}\ vs \quad == \text{CONST option-map default-val} \circ vs$

```

lemma init-arr-comps-base [simp]:  $\text{init-vals}\ (\text{arr-comps}\ T\ 0) = \text{empty}$ 
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

```

### 40 update

#### constdefs

$\text{gupd} \quad :: \text{oref} \Rightarrow \text{obj} \Rightarrow \text{st} \Rightarrow \text{st} \quad (\text{gupd}'(\mapsto)[10,10]1000)$   
 $\text{gupd}\ r\ \text{obj} \equiv \text{st-case}\ (\lambda g\ l.\ \text{st}\ (g(r \mapsto \text{obj})))\ l$

$\text{lupd} \quad :: \text{lname} \Rightarrow \text{val} \Rightarrow \text{st} \Rightarrow \text{st} \quad (\text{lupd}'(\mapsto)[10,10]1000)$   
 $\text{lupd}\ vn\ v \equiv \text{st-case}\ (\lambda g\ l.\ \text{st}\ (g(l(vn \mapsto v))))$

$\text{upd-gobj} \quad :: \text{oref} \Rightarrow \text{vn} \Rightarrow \text{val} \Rightarrow \text{st} \Rightarrow \text{st}$   
 $\text{upd-gobj}\ r\ n\ v \equiv \text{st-case}\ (\lambda g\ l.\ \text{st}\ (\text{chg-map}\ (\text{upd-obj}\ n\ v)\ r\ g)\ l)$

$\text{set-locals} \quad :: \text{locals} \Rightarrow \text{st} \Rightarrow \text{st}$   
 $\text{set-locals}\ l \equiv \text{st-case}\ (\lambda g\ l'.\ \text{st}\ g\ l)$

$\text{init-obj} \quad :: \text{prog} \Rightarrow \text{obj-tag} \Rightarrow \text{oref} \Rightarrow \text{st} \Rightarrow \text{st}$   
 $\text{init-obj}\ G\ oi\ r \equiv \text{gupd}(r \mapsto \langle \text{tag}=oi, \text{values}=\text{init-vals}\ (\text{var-tys}\ G\ oi\ r) \rangle)$

#### syntax

$\text{init-class-obj} :: \text{prog} \Rightarrow \text{qtname} \Rightarrow \text{st} \Rightarrow \text{st}$

#### translations

$\text{init-class-obj}\ G\ C == \text{init-obj}\ G\ \text{arbitrary}\ (\text{Inr}\ C)$

```

lemma gupd-def2 [simp]:  $\text{gupd}(r \mapsto \text{obj})\ (\text{st}\ g\ l) = \text{st}\ (g(r \mapsto \text{obj}))\ l$ 
apply (unfold gupd-def)
apply (simp (no-asm))

```

done

**lemma** *lupd-def2* [simp]:  $\text{lupd}(vn \mapsto v) (st\ g\ l) = st\ g\ (l(vn \mapsto v))$   
**apply** (*unfold lupd-def*)  
**apply** (*simp (no-asm)*)  
**done**

**lemma** *globs-gupd* [simp]:  $\text{globs}\ (\text{gupd}(r \mapsto \text{obj})\ s) = \text{globs}\ s(r \mapsto \text{obj})$   
**apply** (*induct s*)  
**by** (*simp add: gupd-def*)

**lemma** *globs-lupd* [simp]:  $\text{globs}\ (\text{lupd}(vn \mapsto v)\ s) = \text{globs}\ s$   
**apply** (*induct s*)  
**by** (*simp add: lupd-def*)

**lemma** *locals-gupd* [simp]:  $\text{locals}\ (\text{gupd}(r \mapsto \text{obj})\ s) = \text{locals}\ s$   
**apply** (*induct s*)  
**by** (*simp add: gupd-def*)

**lemma** *locals-lupd* [simp]:  $\text{locals}\ (\text{lupd}(vn \mapsto v)\ s) = \text{locals}\ s(vn \mapsto v)$   
**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]:  $st\ (globs\ s)\ (locals\ s) = s$   
**apply** (induct s)  
**by** auto

**lemma** *surjective-st-init-obj*:  
 $st\ (globs\ (init-obj\ G\ oi\ r\ s))\ (locals\ s) = init-obj\ G\ oi\ r\ s$   
**apply** (subst locals-init-obj [THEN sym])  
**apply** (rule surjective-st)  
**done**

**lemma** *heap-heap-upd* [simp]:  
 $heap\ (st\ (g(Inl\ a \mapsto obj))\ l) = heap\ (st\ g\ l)(a \mapsto obj)$   
**apply** (rule ext)  
**apply** (simp (no-asm))  
**done**

**lemma** *heap-stat-upd* [simp]:  $heap\ (st\ (g(Inr\ C \mapsto obj))\ l) = heap\ (st\ g\ l)$   
**apply** (rule ext)  
**apply** (simp (no-asm))  
**done**

**lemma** *heap-local-upd* [simp]:  $heap\ (st\ g\ (l(vn \mapsto v))) = heap\ (st\ g\ l)$   
**apply** (rule ext)  
**apply** (simp (no-asm))  
**done**

**lemma** *heap-gupd-Heap* [simp]:  $heap\ (gupd(Heap\ a \mapsto obj)\ s) = heap\ s(a \mapsto obj)$   
**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

**consts**

```

the-Xcpt :: abrupt  $\Rightarrow$  xcpt
the-Jump :: abrupt  $\Rightarrow$  jump
the-Loc  :: xcpt  $\Rightarrow$  loc
the-Std  :: xcpt  $\Rightarrow$  xname

```

```

primrec the-Xcpt (Xcpt x) = x
primrec the-Jump (Jump j) = j
primrec the-Loc (Loc a) = a
primrec the-Std (Std x) = x

```

**constdefs**

```

abrupt-if  :: bool  $\Rightarrow$  abopt  $\Rightarrow$  abopt  $\Rightarrow$  abopt
abrupt-if c x' x  $\equiv$  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]:  $x \neq \text{None} \implies \text{abrupt-if True } x \ y \neq \text{None}$   
**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]:  $y \neq \text{None} \implies \text{abrupt-if } c \ x \ y = y$   
**apply** (*simp add: abrupt-if-def*)  
**by** *auto*

**lemma** *split-abrupt-if*:  
 $P (\text{abrupt-if } c \ x' \ x) =$   
 $((c \wedge x = \text{None} \longrightarrow P \ x') \wedge (\neg (c \wedge x = \text{None}) \longrightarrow P \ x))$   
**apply** (*unfold abrupt-if-def*)  
**apply** (*split split-if*)  
**apply** *auto*  
**done**

## syntax

*raise-if* :: *bool*  $\Rightarrow$  *xname*  $\Rightarrow$  *abopt*  $\Rightarrow$  *abopt*  
*np* :: *val*  $\Rightarrow$  *abopt*  $\Rightarrow$  *abopt*  
*check-neg* :: *val*  $\Rightarrow$  *abopt*  $\Rightarrow$  *abopt*  
*error-if* :: *bool*  $\Rightarrow$  *error*  $\Rightarrow$  *abopt*  $\Rightarrow$  *abopt*

## translations

*raise-if c xn* == *abrupt-if c (Some (Xcpt (Std xn)))*  
*np v* == *raise-if (v = Null) NullPointer*  
*check-neg i'* == *raise-if (the-Intg i' < 0) NegArrSize*  
*error-if c e* == *abrupt-if c (Some (Error e))*

**lemma** *raise-if-None* [simp]:  $(\text{raise-if } c \ x \ y = \text{None}) = (\neg c \wedge y = \text{None})$   
**apply** (*simp add: abrupt-if-def*)  
**by** *auto*  
**declare** *raise-if-None* [THEN iffD1, dest!]

**lemma** *if-raise-if-None* [simp]:  
 $((\text{if } b \text{ then } y \text{ else } \text{raise-if } c \ x \ y) = \text{None}) = ((c \longrightarrow b) \wedge y = \text{None})$   
**apply** (*simp add: abrupt-if-def*)  
**apply** *auto*  
**done**

**lemma** *raise-if-SomeD* [dest!]:

```

    raise-if c x y = Some z  $\implies$  c  $\wedge$  z=(Xcpt (Std x))  $\wedge$  y=None  $\vee$  (y=Some z)
  apply (case-tac y)
  apply (case-tac c)
  apply (simp add: abrupt-if-def)
  apply (simp add: abrupt-if-def)
  apply auto
done

```

```

lemma error-if-None [simp]: (error-if c e y = None) = ( $\neg$ c  $\wedge$  y = None)
apply (simp add: abrupt-if-def)
by auto
declare error-if-None [THEN iffD1, dest!]

```

```

lemma if-error-if-None [simp]:
  ((if b then y else error-if c e y) = None) = ((c  $\longrightarrow$  b)  $\wedge$  y = None)
apply (simp add: abrupt-if-def)
apply auto
done

```

```

lemma error-if-SomeD [dest!]:
  error-if c e y = Some z  $\implies$  c  $\wedge$  z=(Error e)  $\wedge$  y=None  $\vee$  (y=Some z)
apply (case-tac y)
apply (case-tac c)
apply (simp add: abrupt-if-def)
apply (simp add: abrupt-if-def)
apply auto
done

```

```

constdefs
  absorb :: jump  $\Rightarrow$  abopt  $\Rightarrow$  abopt
  absorb j a  $\equiv$  if a=Some (Jump j) then None else a

```

```

lemma absorb-SomeD [dest!]: absorb j a = Some x  $\implies$  a = Some x
by (auto simp add: absorb-def)

```

```

lemma absorb-same [simp]: absorb j (Some (Jump j)) = None
by (auto simp add: absorb-def)

```

```

lemma absorb-other [simp]: a  $\neq$  Some (Jump j)  $\implies$  absorb j a = a
by (auto simp add: absorb-def)

```

```

lemma absorb-Some-NoneD: absorb j (Some abr) = None  $\implies$  abr = Jump j
by (simp add: absorb-def)

```

```

lemma absorb-Some-JumpD: absorb j s = Some (Jump j')  $\implies$  j'  $\neq$  j
by (simp add: absorb-def)

```

## full program state

### types

state = abopt  $\times$  st — state including abruption information

**syntax**

$Norm \quad :: \quad st \Rightarrow state$   
 $abrupt \quad :: \quad state \Rightarrow abopt$   
 $store \quad :: \quad state \Rightarrow st$

**translations**

$Norm \ s \quad == \ (None, s)$   
 $abrupt \quad ==> \ fst$   
 $store \quad ==> \ snd$   
 $abopt \quad <= \ (type) \ State.abrupt \ option$   
 $abopt \quad <= \ (type) \ abrupt \ option$   
 $state \quad <= \ (type) \ abopt \times \ State.st$   
 $state \quad <= \ (type) \ abopt \times \ st$

**lemma** *single-stateE*:  $\forall Z. Z = (s::state) \implies False$

**apply** (*erule-tac*  $x = (Some \ k, y)$  **in** *all-dupE*)

**apply** (*erule-tac*  $x = (None, y)$  **in** *allE*)

**apply** *clarify*

**done**

**lemma** *state-not-single*:  $All \ (op = (x::state)) \implies R$

**apply** (*drule-tac*  $x = (if \ abrupt \ x = None \ then \ Some \ ?x \ else \ None, ?y)$  **in** *spec*)

**apply** *clarsimp*

**done**

**constdefs**

$normal \quad :: \quad state \Rightarrow bool$   
 $normal \equiv \lambda s. \ abrupt \ s = None$

**lemma** *normal-def2* [*simp*]:  $normal \ s = (abrupt \ s = None)$

**apply** (*unfold* *normal-def*)

**apply** (*simp* (*no-asm*))

**done**

**constdefs**

$heap-free \quad :: \quad nat \Rightarrow state \Rightarrow bool$   
 $heap-free \ n \equiv \lambda s. \ atleast-free \ (heap \ (store \ s)) \ n$

**lemma** *heap-free-def2* [*simp*]:  $heap-free \ n \ s = atleast-free \ (heap \ (store \ s)) \ n$

**apply** (*unfold* *heap-free-def*)

**apply** *simp*

**done**

**41 update****constdefs**

$abupd \quad :: \quad (abopt \Rightarrow abopt) \Rightarrow state \Rightarrow state$   
 $abupd \ f \equiv prod-fun \ f \ id$

$supd \quad :: (st \Rightarrow st) \Rightarrow state \Rightarrow state$   
 $supd \equiv prod\_fun \ id$

**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]:  
 $\bigwedge s. supd \ (lupd \ vn \ v) \ s = (abrupt \ s, lupd \ vn \ v \ (store \ s))$   
**apply** (simp (no-asm-simp) only: *split-tupled-all*)  
**apply** (simp (no-asm))  
**done**

**lemma** *supd-gupd* [simp]:  
 $\bigwedge s. supd \ (gupd \ r \ obj) \ s = (abrupt \ s, gupd \ r \ obj \ (store \ s))$   
**apply** (simp (no-asm-simp) only: *split-tupled-all*)  
**apply** (simp (no-asm))  
**done**

**lemma** *supd-init-obj* [simp]:  
 $supd \ (init\_obj \ G \ oi \ r) \ s = (abrupt \ s, init\_obj \ G \ oi \ r \ (store \ s))$   
**apply** (unfold *init-obj-def*)  
**apply** (simp (no-asm))  
**done**

**lemma** *abupd-store-invariant* [simp]:  $store \ (abupd \ f \ s) = store \ s$   
**by** (cases *s*) *simp*

**lemma** *supd-abrupt-invariant* [simp]:  $abrupt \ (supd \ f \ s) = abrupt \ s$   
**by** (cases *s*) *simp*

**syntax**

$set\_lvars \quad :: locals \Rightarrow state \Rightarrow state$   
 $restore\_lvars :: state \Rightarrow state \Rightarrow state$

**translations**

$set\_lvars \ l == supd \ (set\_locals \ l)$   
 $restore\_lvars \ s' \ s == set\_lvars \ (locals \ (store \ s')) \ s$

**lemma** *set-set-lvars* [simp]:  $\bigwedge s. set\_lvars \ l \ (set\_lvars \ l' \ s) = set\_lvars \ l \ s$   
**apply** (simp (no-asm-simp) only: *split-tupled-all*)  
**apply** (simp (no-asm))



done

**lemma** *set-lvars-id* [simp]:  $\bigwedge s. \text{set-lvars } (\text{locals } (\text{store } s)) \ s = s$   
**apply** (simp (no-asm-simp) only: split-tupled-all)  
**apply** (simp (no-asm))  
**done**

## initialisation test

### constdefs

*initd* :: *qname*  $\Rightarrow$  *globs*  $\Rightarrow$  *bool*  
*initd* *C* *g*  $\equiv$  *g* (*Stat* *C*)  $\neq$  *None*

*initd* :: *qname*  $\Rightarrow$  *state*  $\Rightarrow$  *bool*  
*initd* *C*  $\equiv$  *initd* *C*  $\circ$  *globs*  $\circ$  *store*

**lemma** *not-initd-empty* [simp]:  $\neg \text{initd } C \text{ empty}$   
**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 add: st.split)  
**done**

**lemma** *initd-init-class-obj* [intro!]:  $\text{initd } C \ (\text{globs } (\text{init-class-obj } G \ C \ s))$   
**apply** (unfold initd-def)  
**apply** (simp (no-asm))  
**done**

**lemma** *not-initdD*:  $\neg \text{initd } C \ g \implies g \ (\text{Stat } C) = \text{None}$   
**apply** (unfold initd-def)  
**apply** (erule notnotD)  
**done**

**lemma** *initdD*:  $\text{initd } C \ g \implies \exists \text{ obj. } g \ (\text{Stat } C) = \text{Some obj}$   
**apply** (unfold initd-def)  
**apply** auto  
**done**

**lemma** *initd-def2* [simp]:  $\text{initd } C \ s = \text{initd } C \ (\text{globs } (\text{store } s))$   
**apply** (unfold initd-def)  
**apply** (simp (no-asm))  
**done**

*error-free*

**constdefs** *error-free*:: *state*  $\Rightarrow$  *bool*  
*error-free* *s*  $\equiv$   $\neg (\exists \text{ err. abrupt } s = \text{Some } (\text{Error err}))$

**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]:  
 $\neg (\exists \text{ err. } x = \text{Error err}) \implies \text{error-free } ((\text{Some } x),s)$   
**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: split-if-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: split-if-asm*)

**lemma** *error-free-abrupt-if* [simp,intro]:  
 $\llbracket \text{error-free } s; \neg (\exists \text{ err. } x = \text{Error err}) \rrbracket$   
 $\implies \text{error-free } (\text{abupd } (\text{abrupt-if } p \text{ (Some } x)) \text{ } s)$   
**by** (*cases s*)  
 (*auto simp add: abrupt-if-def*  
*split: split-if*)

**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: split-if*)

**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*)  
 $\implies$  *error-free* (*abrupt-if* *p* (*Some* (*Xcpt* *x*)) *a*, *s*)  
**by** *simp*

**lemma** *error-free-abrupt-if-Jump* [*simp,intro*]:  
*error-free* *s*  
 $\implies$  *error-free* (*abupd* (*abrupt-if* *p* (*Some* (*Jump* *j*))) *s*)  
**by** *simp*

**lemma** *error-free-abrupt-if-Jump1* [*simp,intro*]:  
*error-free* (*a,s*)  
 $\implies$  *error-free* (*abrupt-if* *p* (*Some* (*Jump* *j*)) *a*, *s*)  
**by** *simp*

**lemma** *error-free-raise-if* [*simp,intro*]:  
*error-free* *s*  $\implies$  *error-free* (*abupd* (*raise-if* *p* *x*) *s*)  
**by** *simp*

**lemma** *error-free-raise-if1* [*simp,intro*]:  
*error-free* (*a,s*)  $\implies$  *error-free* ((*raise-if* *p* *x* *a*), *s*)  
**by** *simp*

**lemma** *error-free-supd* [*simp,intro*]:  
*error-free* *s*  $\implies$  *error-free* (*supd* *f* *s*)  
**by** (*cases* *s*) (*simp* *add: error-free-def*)

**lemma** *error-free-supd1* [*simp,intro*]:  
*error-free* (*a,s*)  $\implies$  *error-free* (*a,f* *s*)  
**by** (*simp* *add: error-free-def*)

**lemma** *error-free-set-lvars* [*simp,intro*]:  
*error-free* *s*  $\implies$  *error-free* ((*set-lvars* *l*) *s*)  
**by** (*cases* *s*) *simp*

**lemma** *error-free-set-locals* [*simp,intro*]:  
*error-free* (*x*, *s*)  
 $\implies$  *error-free* (*x*, *set-locals* *l* *s*')

**by** (*simp* *add: error-free-def*)

**end**



## Chapter 15

### Eval

## 42 Operational evaluation (big-step) semantics of Java expressions and statements

theory *Eval* imports *State DeclConcepts* begin

improvements over Java Specification 1.0:

- dynamic method lookup does not need to consider the return type (cf.15.11.4.4)
- throw raises a NullPointerException if a null reference is given, and each throw of a standard exception yield a fresh exception object (was not specified)
- if there is not enough memory even to allocate an OutOfMemory exception, evaluation/execution fails, i.e. simply stops (was not specified)
- array assignment checks lhs (and may throw exceptions) before evaluating rhs
- fixed exact positions of class initializations (immediate at first active use)

design issues:

- evaluation vs. (single-step) transition semantics evaluation semantics chosen, because:
  - ++ less verbose and therefore easier to read (and to handle in proofs)
  - + more abstract
  - + intermediate values (appearing in recursive rules) need not be stored explicitly, e.g. no call body construct or stack of invocation frames containing local variables and return addresses for method calls needed
  - + convenient rule induction for subject reduction theorem
    - no interleaving (for parallelism) can be described
    - stating a property of infinite executions requires the meta-level argument that this property holds for any finite prefixes of it (e.g. stopped using a counter that is decremented to zero and then throwing an exception)
- unified evaluation for variables, expressions, expression lists, statements
- the value entry in statement rules is redundant
- the value entry in rules is irrelevant in case of exceptions, but its full inclusion helps to make the rule structure independent of exception occurrence.
- as irrelevant value entries are ignored, it does not matter if they are unique. For simplicity, (fixed) arbitrary values are preferred over "free" values.
- the rule format is such that the start state may contain an exception.
  - ++ facilitates exception handling
  - + symmetry
- the rules are defined carefully in order to be applicable even in not type-correct situations (yielding undefined values), e.g. *the-Addr (Val (Bool b)) = arbitrary*.
  - ++ fewer rules
    - less readable because of auxiliary functions like *the-Addr*

Alternative: "defensive" evaluation throwing some InternalError exception in case of (impossible, for correct programs) type mismatches

- there is exactly one rule per syntactic construct
  - + no redundancy in case distinctions
- `halloc` fails iff there is no free heap address. When there is only one free heap address left, it returns an `OutOfMemory` exception. In this way it is guaranteed that when an `OutOfMemory` exception is thrown for the first time, there is a free location on the heap to allocate it.
- the allocation of objects that represent standard exceptions is deferred until execution of any enclosing catch clause, which is transparent to the program.
  - requires an auxiliary execution relation
  - ++ avoids copies of allocation code and awkward case distinctions (whether there is enough memory to allocate the exception) in evaluation rules
- unfortunately `new-Addr` is not directly executable because of Hilbert operator.

simplifications:

- local variables are initialized with default values (no definite assignment)
- garbage collection not considered, therefore also no finalizers
- stack overflow and memory overflow during class initialization not modelled
- exceptions in initializations not replaced by `ExceptionInInitializerError`

**types**  $vvar = val \times (val \Rightarrow state \Rightarrow state)$   
 $vals = (val, vvar, val\ list)\ sum3$

**translations**

$vvar \leq (type)\ val \times (val \Rightarrow state \Rightarrow state)$   
 $vals \leq (type)(val, vvar, val\ list)\ sum3$

To avoid redundancy and to reduce the number of rules, there is only one evaluation rule for each syntactic term. This is also true for variables (e.g. see the rules below for *LVar*, *FVar* and *AVar*). So evaluation of a variable must capture both possible further uses: read (rule *Acc*) or write (rule *Ass*) to the variable. 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.

**syntax** (*xsymbols*)  
 $dummy-res :: vals\ (\Diamond)$

**translations**

$\Diamond == In1\ Unit$

**syntax**

$val-inj-vals :: expr \Rightarrow term\ ([\_]\_e\ 1000)$   
 $var-inj-vals :: var \Rightarrow term\ ([\_]\_v\ 1000)$   
 $lst-inj-vals :: expr\ list \Rightarrow term\ ([\_]\_l\ 1000)$

**translations**

$$\begin{aligned} [e]_e &\rightarrow In1\ e \\ [v]_v &\rightarrow In2\ v \\ [es]_l &\rightarrow In3\ es \end{aligned}$$
**constdefs**

$$\begin{aligned} arbitrary3 &:: ('al + 'ar, 'b, 'c)\ sum3 \Rightarrow vals \\ arbitrary3 &\equiv sum3\text{-}case\ (In1 \circ sum\text{-}case\ (\lambda x. arbitrary))\ (\lambda x. Unit)) \\ &\quad (\lambda x. In2\ arbitrary)\ (\lambda x. In3\ arbitrary) \end{aligned}$$

**lemma** [simp]:  $arbitrary3\ (In1\ x) = In1\ arbitrary$   
**by** (simp add: arbitrary3-def)

**lemma** [simp]:  $arbitrary3\ (In1r\ x) = \Diamond$   
**by** (simp add: arbitrary3-def)

**lemma** [simp]:  $arbitrary3\ (In2\ x) = In2\ arbitrary$   
**by** (simp add: arbitrary3-def)

**lemma** [simp]:  $arbitrary3\ (In3\ x) = In3\ arbitrary$   
**by** (simp add: arbitrary3-def)

**exception throwing and catching****constdefs**

$$\begin{aligned} throw &:: val \Rightarrow abopt \Rightarrow abopt \\ throw\ a'\ x &\equiv abrupt\text{-}if\ True\ (Some\ (Xcpt\ (Loc\ (the\text{-}Addr\ a'))))\ (np\ a'\ x) \end{aligned}$$
**lemma** throw-def2:
$$\begin{aligned} throw\ a'\ x &= abrupt\text{-}if\ True\ (Some\ (Xcpt\ (Loc\ (the\text{-}Addr\ a'))))\ (np\ a'\ x) \\ \mathbf{apply}\ (unfold\ throw\text{-}def) \\ \mathbf{apply}\ (simp\ (no\text{-}asm)) \\ \mathbf{done} \end{aligned}$$
**constdefs**

$$\begin{aligned} fits &:: prog \Rightarrow st \Rightarrow val \Rightarrow ty \Rightarrow bool\ (\neg, \vdash\text{-} fits\ [61, 61, 61, 61] 60) \\ G, s \vdash a' fits\ T &\equiv (\exists rt. T = RefT\ rt) \longrightarrow a' = Null \vee G \vdash obj\text{-}ty(lookup\text{-}obj\ s\ a') \preceq T \end{aligned}$$

**lemma** fits-Null [simp]:  $G, s \vdash Null\ fits\ T$   
**by** (simp add: fits-def)

**lemma** fits-Addr-RefT [simp]:
$$\begin{aligned} G, s \vdash Addr\ a\ fits\ RefT\ t &= G \vdash obj\text{-}ty\ (the\ (heap\ s\ a)) \preceq RefT\ t \\ \mathbf{by}\ (simp\ add: fits\text{-}def) \end{aligned}$$

**lemma** fitsD:  $\bigwedge X. G, s \vdash a' fits\ T \implies (\exists pt. T = PrimT\ pt) \vee$   
 $(\exists t. T = RefT\ t) \wedge a' = Null \vee$   
 $(\exists t. T = RefT\ t) \wedge a' \neq Null \wedge G \vdash obj\text{-}ty\ (lookup\text{-}obj\ s\ a') \preceq T$   
**apply** (unfold fits-def)  
**apply** (case-tac  $\exists pt. T = PrimT\ pt$ )  
**apply** simp-all



```

apply (case-tac  $T$ )
defer
apply (case-tac  $a' = \text{Null}$ )
apply simp-all
apply iprover
done

```

```

constdefs
  catch :: prog  $\Rightarrow$  state  $\Rightarrow$  qtname  $\Rightarrow$  bool    ( $\neg, \vdash$  catch  $-[61,61,61]60$ )
   $G, s \vdash \text{catch } C \equiv \exists xc. \text{ abrupt } s = \text{Some } (Xcpt\ xc) \wedge$ 
     $G, \text{store } s \vdash \text{Addr } (\text{the-Loc } xc) \text{ fits Class } C$ 

```

```

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]:
   $G, (\text{Some } (Xcpt (\text{Loc } a)), s) \vdash \text{catch } C = G, s \vdash \text{Addr } a \text{ fits Class } C$ 
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

```

```

constdefs
  new-xcpt-var :: vname  $\Rightarrow$  state  $\Rightarrow$  state
  new-xcpt-var vn  $\equiv$ 
     $\lambda(x, s). \text{Norm } (\text{lupd}(VName\ vn \mapsto \text{Addr } (\text{the-Loc } (\text{the-Xcpt } (\text{the } x)))) s)$ 

```

```

lemma new-xcpt-var-def2 [simp]:
  new-xcpt-var vn  $(x, s) =$ 
     $\text{Norm } (\text{lupd}(VName\ vn \mapsto \text{Addr } (\text{the-Loc } (\text{the-Xcpt } (\text{the } x)))) s)$ 
apply (unfold new-xcpt-var-def)
apply (simp (no-asm))
done

```

**misc**

**constdefs**

```

  assign    :: ('a  $\Rightarrow$  state  $\Rightarrow$  state)  $\Rightarrow$  'a  $\Rightarrow$  state  $\Rightarrow$  state
  assign f v  $\equiv \lambda(x, s). \text{let } (x', s') = (\text{if } x = \text{None then } f\ v \text{ else id})\ (x, s)$ 
    in  $(x', \text{if } x' = \text{None then } s' \text{ else } s)$ 

```

**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]:  
 $assign\ (\lambda v. supd\ (f\ v))\ v\ (x, s)$   
 $= (x, if\ x = None\ then\ f\ v\ s\ else\ s)$   
**apply** auto  
**done**

**lemma** *assign-raise-if* [simp]:  
 $assign\ (\lambda v\ (x, s). ((raise-if\ (b\ s\ v)\ xcpt)\ x, f\ v\ s))\ v\ (x, s) =$   
 $(raise-if\ (b\ s\ v)\ xcpt\ x, if\ x = None \wedge \neg b\ s\ v\ then\ f\ v\ s\ else\ s)$   
**apply** (case-tac  $x = None$ )  
**apply** auto  
**done**

**constdefs**

*init-comp-ty* ::  $ty \Rightarrow stmt$   
 $init-comp-ty\ T \equiv if\ (\exists C. T = Class\ C)\ then\ Init\ (the-Class\ T)\ else\ Skip$

**lemma** *init-comp-ty-PrimT* [simp]:  $init-comp-ty\ (PrimT\ pt) = Skip$   
**apply** (unfold init-comp-ty-def)  
**apply** (simp (no-asm))  
**done**

**constdefs**

*invocation-class* ::  $inv-mode \Rightarrow st \Rightarrow val \Rightarrow ref-ty \Rightarrow qname$   
 $invocation-class\ m\ s\ a'\ statT$   
 $\equiv (case\ m\ of$   
 $Static \Rightarrow if\ (\exists statC. statT = ClassT\ statC)$

```

      then the-Class (RefT statT)
      else Object
| SuperM ⇒ the-Class (RefT statT)
| IntVir ⇒ obj-class (lookup-obj s a')

```

```

invocation-declclass::prog ⇒ inv-mode ⇒ st ⇒ val ⇒ ref-ty ⇒ sig ⇒ qtname
invocation-declclass G m s a' statT sig
  ≡ declclass (the (dynlookup G statT
                     (invocation-class m s a' statT)
                     sig))

```

**lemma** *invocation-class-IntVir* [simp]:  
*invocation-class IntVir s a' statT* = *obj-class (lookup-obj s a')*  
**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)

#### constdefs

```

init-lvars :: prog ⇒ qtname ⇒ sig ⇒ inv-mode ⇒ val ⇒ val list ⇒
              state ⇒ state
init-lvars G C sig mode a' pvs
  ≡ λ (x,s).
    let m = mthd (the (methd G C sig));
    l = λ k.
      (case k of
       EName e
       ⇒ (case e of
          VNam v ⇒ (empty ((pars m)[↦]pvs)) v
          | Res   ⇒ None)
       | This
       ⇒ (if mode=Static then None else Some a'))
    in set-lvars l (if mode = Static then x else np a' x,s)

```

**lemma** *init-lvars-def2*: — better suited for simplification

```

init-lvars G C sig mode a' pvs (x,s) =
  set-lvars
    (λ k.
     (case k of
      EName e
      ⇒ (case e of
         VNam v
         ⇒ (empty ((pars (mthd (the (methd G C sig))))[↦]pvs)) v
         | Res ⇒ None)
      | This
      ⇒ (if mode=Static then None else Some a'))
    )

```

```

    (if mode = Static then x else np a' x,s)
apply (unfold init-lvars-def)
apply (simp (no-asm) add: Let-def)
done

```

### constdefs

```

  body :: prog  $\Rightarrow$  qtname  $\Rightarrow$  sig  $\Rightarrow$  expr
  body G C sig  $\equiv$  let m = the (methd G C sig)
                  in Body (declclass m) (stmt (mbody (methd m)))

```

**lemma** body-def2: — better suited for simplification

```

  body G C sig = Body (declclass (the (methd G C sig)))
                  (stmt (mbody (methd (the (methd G C sig)))))
apply (unfold body-def Let-def)
apply auto
done

```

### variables

#### constdefs

```

  lvar :: lname  $\Rightarrow$  st  $\Rightarrow$  vvar
  lvar vn s  $\equiv$  (the (locals s vn),  $\lambda v$ . supd (lupd(vn $\mapsto$ v)))

  fvar :: qtname  $\Rightarrow$  bool  $\Rightarrow$  vname  $\Rightarrow$  val  $\Rightarrow$  state  $\Rightarrow$  vvar  $\times$  state
  fvar C stat fn a' s
     $\equiv$  let (oref,xf) = if stat then (Stat C,id)
                    else (Heap (the-Addr a'),np a');
        n = Inl (fn,C);
        f = ( $\lambda v$ . supd (upd-gobj oref n v))
    in ((the (values (the (globs (store s) oref)) n),f),abupd xf s)

  avar :: prog  $\Rightarrow$  val  $\Rightarrow$  val  $\Rightarrow$  state  $\Rightarrow$  vvar  $\times$  state
  avar G i' a' s
     $\equiv$  let oref = Heap (the-Addr a');
        i = the-Intg i';
        n = Inr i;
        (T,k,cs) = the-Arr (globs (store s) oref);
        f = ( $\lambda v$  (x,s). (raise-if ( $\neg G, s \vdash v$  fits T)
                                   ArrStore x
                                   ,upd-gobj oref n v s))
    in ((the (cs n),f)
        ,abupd (raise-if ( $\neg i$  in-bounds k) IndOutBound  $\circ$  np a') s)

```

**lemma** fvar-def2: — better suited for simplification

```

  fvar C stat fn a' s =
    ((the
      (values
        (the (globs (store s) (if stat then Stat C else Heap (the-Addr a'))))
        (Inl (fn,C))))
      ,( $\lambda v$ . supd (upd-gobj (if stat then Stat C else Heap (the-Addr a'))
                  (Inl (fn,C))
                  v)))
      ,abupd (if stat then id else np a') s)

```

```

apply (unfold fvar-def)
apply (simp (no-asm) add: Let-def split-beta)

```

**done**

**lemma** *avar-def2*: — better suited for simplification

*avar*  $G\ i'\ a'\ s =$   
 $((the\ ((snd(snd(the-Arr\ (globs\ (store\ s)\ (Heap\ (the-Addr\ a'))))))$   
 $(Inr\ (the-Intg\ i'))$   
 $,(\lambda v\ (x,s').\ (raise-if\ (\neg G,s \vdash v\ fits\ (fst(the-Arr\ (globs\ (store\ s)$   
 $(Heap\ (the-Addr\ a'))))))$   
 $ArrStore\ x$   
 $,upd-gobj\ (Heap\ (the-Addr\ a'))$   
 $(Inr\ (the-Intg\ i'))\ v\ s'))$   
 $,abupd\ (raise-if\ (\neg(the-Intg\ i')\ in-bounds\ (fst(snd(the-Arr\ (globs\ (store\ s)$   
 $(Heap\ (the-Addr\ a'))))))\ IndOutBound\ o\ np\ a')$   
 $s)$   
**apply** *(unfold avar-def)*  
**apply** *(simp (no-asm) add: Let-def split-beta)*  
**done**

**constdefs**

*check-field-access*::  
 $prog \Rightarrow qname \Rightarrow qname \Rightarrow vname \Rightarrow bool \Rightarrow val \Rightarrow state \Rightarrow state$   
*check-field-access*  $G\ accC\ statDeclC\ fn\ stat\ a'\ s$   
 $\equiv let\ oref = if\ stat\ then\ Stat\ statDeclC$   
 $else\ Heap\ (the-Addr\ a');$   
 $dynC = case\ oref\ of$   
 $Heap\ a \Rightarrow obj-class\ (the\ (globs\ (store\ s)\ oref))$   
 $| Stat\ C \Rightarrow C;$   
 $f = (the\ (table-of\ (DeclConcepts.fields\ G\ dynC)\ (fn,statDeclC)))$   
*in* *abupd*  
 $(error-if\ (\neg G \vdash Field\ fn\ (statDeclC,f)\ in\ dynC\ dyn-accessible-from\ accC)$   
 $AccessViolation)$   
 $s$

**constdefs**

*check-method-access*::  
 $prog \Rightarrow qname \Rightarrow ref-ty \Rightarrow inv-mode \Rightarrow sig \Rightarrow val \Rightarrow state \Rightarrow state$   
*check-method-access*  $G\ accC\ statT\ mode\ sig\ a'\ s$   
 $\equiv let\ invC = invocation-class\ mode\ (store\ s)\ a'\ statT;$   
 $dynM = the\ (dynlookup\ G\ statT\ invC\ sig)$   
*in* *abupd*  
 $(error-if\ (\neg G \vdash Methd\ sig\ dynM\ in\ invC\ dyn-accessible-from\ accC)$   
 $AccessViolation)$   
 $s$

**evaluation judgments**

**inductive**

*halloc* ::  $[prog, state, obj-tag, loc, state] \Rightarrow bool\ (\vdash -\ halloc\ \rightarrow - \rightarrow [61,61,61,61,61]60)$  **for**  $G::prog$   
**where** — allocating objects on the heap, cf. 12.5

*Abrupt*:

$G \vdash (Some\ x,s)\ -halloc\ oi \succ arbitrary \rightarrow (Some\ x,s)$

| *New*:  $\llbracket new-Addr\ (heap\ s) = Some\ a;$   
 $(x,oi') = (if\ atleast-free\ (heap\ s)\ (Suc\ (Suc\ 0))\ then\ (None,oi)$   
 $else\ (Some\ (Xcpt\ (Loc\ a)),CInst\ (SXcpt\ OutOfMemory))) \rrbracket$   
 $\Rightarrow$   
 $G \vdash Norm\ s\ -halloc\ oi \succ a \rightarrow (x,init-obj\ G\ oi'\ (Heap\ a)\ s)$

**inductive** *sxalloc* ::  $[prog, state, state] \Rightarrow bool$  ( $\vdash - \text{sxalloc} \rightarrow -[61, 61, 61]60$ ) **for**  $G::prog$   
**where** — allocating exception objects for standard exceptions (other than OutOfMemory)

*Norm*:  $G \vdash Norm \quad s \text{ --sxalloc--} Norm \quad s$

| *Jmp*:  $G \vdash (Some (Jump j), s) \text{ --sxalloc--} (Some (Jump j), s)$

| *Error*:  $G \vdash (Some (Error e), s) \text{ --sxalloc--} (Some (Error e), s)$

| *XcptL*:  $G \vdash (Some (Xcpt (Loc a)), s) \text{ --sxalloc--} (Some (Xcpt (Loc a)), s)$

| *SXcpt*:  $\llbracket G \vdash Norm s0 \text{ --halloc--} (CInst (SXcpt xn)) \succ a \rightarrow (x, s1) \rrbracket \Longrightarrow$   
 $G \vdash (Some (Xcpt (Std xn)), s0) \text{ --sxalloc--} (Some (Xcpt (Loc a)), s1)$

**inductive**

*eval* ::  $[prog, state, term, vals, state] \Rightarrow bool$  ( $\vdash - \text{eval} \rightarrow '(-, -)' [61, 61, 80, 0, 0]60$ )  
**and** *exec* ::  $[prog, state, stmt, state] \Rightarrow bool$  ( $\vdash - \text{exec} \rightarrow - [61, 61, 65, 61]60$ )  
**and** *evar* ::  $[prog, state, var, vvar, state] \Rightarrow bool$  ( $\vdash - \text{evar} \rightarrow - [61, 61, 90, 61, 61]60$ )  
**and** *eval'* ::  $[prog, state, expr, val, state] \Rightarrow bool$  ( $\vdash - \text{eval'} \rightarrow - [61, 61, 80, 61, 61]60$ )  
**and** *evals* ::  $[prog, state, expr list, val list, state] \Rightarrow bool$  ( $\vdash - \text{evals} \rightarrow - [61, 61, 61, 61, 61]60$ )

**for**  $G::prog$   
**where**

$G \vdash s - c \rightarrow s' \equiv G \vdash s - In1r c \rightarrow (\Diamond, s')$

|  $G \vdash s - e \rightarrow v \rightarrow s' \equiv G \vdash s - In1l e \rightarrow (In1 v, s')$

|  $G \vdash s - e \rightarrow vf \rightarrow s' \equiv G \vdash s - In2 e \rightarrow (In2 vf, s')$

|  $G \vdash s - e \rightarrow v \rightarrow s' \equiv G \vdash s - In3 e \rightarrow (In3 v, s')$

— propagation of abrupt completion

— cf. 14.1, 15.5

| *Abrupt*:  
 $G \vdash (Some xc, s) - t \rightarrow (arbitrary3 t, (Some xc, s))$

— execution of statements

— cf. 14.5

| *Skip*:  $G \vdash Norm s - Skip \rightarrow Norm s$

— cf. 14.7

| *Expr*:  $\llbracket G \vdash Norm s0 - e \rightarrow v \rightarrow s1 \rrbracket \Longrightarrow$   
 $G \vdash Norm s0 - Expr e \rightarrow s1$

| *Lab*:  $\llbracket G \vdash Norm s0 - c \rightarrow s1 \rrbracket \Longrightarrow$   
 $G \vdash Norm s0 - l \cdot c \rightarrow abupd (absorb l) s1$

— cf. 14.2

| *Comp*:  $\llbracket G \vdash Norm s0 - c1 \rightarrow s1; G \vdash s1 - c2 \rightarrow s2 \rrbracket \Longrightarrow$   
 $G \vdash Norm s0 - c1;; c2 \rightarrow s2$

— cf. 14.8.2

| *If*:  $\llbracket G \vdash Norm s0 - e \rightarrow b \rightarrow s1; G \vdash s1 - (if the-Bool b then c1 else c2) \rightarrow s2 \rrbracket \Longrightarrow$   
 $G \vdash Norm s0 - If(e) c1 Else c2 \rightarrow s2$

— cf. 14.10, 14.10.1

— A continue jump from the while body  $c$  is handled by this rule. If a continue jump with the proper label was invoked inside  $c$  this label (Cont  $l$ ) is deleted out of the abrupt component of the state before the iterative evaluation of the while statement. A break jump is handled by the Lab Statement *Lab*  $l$  (*while*...).

| *Loop*:  $\llbracket G \vdash \text{Norm } s0 \text{ } -e \rightarrow b \rightarrow s1;$   
     *if the-Bool*  $b$   
     *then*  $(G \vdash s1 \text{ } -c \rightarrow s2 \wedge$   
          $G \vdash (\text{abupd } (\text{absorb } (\text{Cont } l)) \text{ } s2) \text{ } -l \cdot \text{While}(e) \text{ } c \rightarrow s3)$   
     *else*  $s3 = s1 \rrbracket \Rightarrow$   
      $G \vdash \text{Norm } s0 \text{ } -l \cdot \text{While}(e) \text{ } c \rightarrow s3$

| *Jmp*:  $G \vdash \text{Norm } s \text{ } -\text{Jmp } j \rightarrow (\text{Some } (\text{Jump } j), s)$

— cf. 14.16

| *Throw*:  $\llbracket G \vdash \text{Norm } s0 \text{ } -e \rightarrow a' \rightarrow s1 \rrbracket \Rightarrow$   
      $G \vdash \text{Norm } s0 \text{ } -\text{Throw } e \rightarrow \text{abupd } (\text{throw } a') \text{ } s1$

— cf. 14.18.1

| *Try*:  $\llbracket G \vdash \text{Norm } s0 \text{ } -c1 \rightarrow s1; G \vdash s1 \text{ } -\text{salloc} \rightarrow s2;$   
     *if*  $G, s2 \vdash \text{catch } C \text{ then } G \vdash \text{new-} \text{xcept-var } vn \text{ } s2 \text{ } -c2 \rightarrow s3 \text{ else } s3 = s2 \rrbracket \Rightarrow$   
      $G \vdash \text{Norm } s0 \text{ } -\text{Try } c1 \text{ } \text{Catch}(C \text{ } vn) \text{ } c2 \rightarrow s3$

— cf. 14.18.2

| *Fin*:  $\llbracket G \vdash \text{Norm } s0 \text{ } -c1 \rightarrow (x1, s1);$   
      $G \vdash \text{Norm } s1 \text{ } -c2 \rightarrow s2;$   
      $s3 = (\text{if } (\exists \text{ err. } x1 = \text{Some } (\text{Error } \text{err}))$   
         *then*  $(x1, s1)$   
         *else*  $\text{abupd } (\text{abrupt-if } (x1 \neq \text{None}) \text{ } x1) \text{ } s2) \rrbracket$   
      $\Rightarrow$   
      $G \vdash \text{Norm } s0 \text{ } -c1 \text{ } \text{Finally } c2 \rightarrow s3$

— cf. 12.4.2, 8.5

| *Init*:  $\llbracket \text{the } (\text{class } G \text{ } C) = c;$   
     *if* *inited*  $C$  (*globs*  $s0$ ) *then*  $s3 = \text{Norm } s0$   
     *else*  $(G \vdash \text{Norm } (\text{init-class-obj } G \text{ } C \text{ } s0)$   
          $-(\text{if } C = \text{Object then Skip else Init } (\text{super } c)) \rightarrow s1 \wedge$   
          $G \vdash \text{set-lvars empty } s1 \text{ } -\text{init } c \rightarrow s2 \wedge s3 = \text{restore-lvars } s1 \text{ } s2) \rrbracket$   
      $\Rightarrow$   
      $G \vdash \text{Norm } s0 \text{ } -\text{Init } C \rightarrow s3$

— 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*:  $\llbracket G \vdash \text{Norm } s0 \text{ } -\text{Init } C \rightarrow s1;$   
      $G \vdash s1 \text{ } -\text{halloc } (C \text{Inst } C) \rightarrow a \rightarrow s2 \rrbracket \Rightarrow$   
      $G \vdash \text{Norm } s0 \text{ } -\text{NewC } C \rightarrow \text{Addr } a \rightarrow s2$

— cf. 15.9.1, 12.4.1

| *NewA*:  $\llbracket G \vdash \text{Norm } s0 \text{ } -\text{init-comp-ty } T \rightarrow s1; G \vdash s1 \text{ } -e \rightarrow i' \rightarrow s2;$   
      $G \vdash \text{abupd } (\text{check-neg } i') \text{ } s2 \text{ } -\text{halloc } (\text{Arr } T \text{ } (\text{the-Intg } i')) \rightarrow a \rightarrow s3 \rrbracket \Rightarrow$

$$G \vdash \text{Norm } s0 \text{ --New } T[e] \text{--}\succ \text{Addr } a \rightarrow s3$$

— cf. 15.15

$$\begin{aligned} | \text{Cast: } & \llbracket G \vdash \text{Norm } s0 \text{ --e--}\succ v \rightarrow s1; \\ & s2 = \text{abupd } (\text{raise-if } (\neg G, \text{store } s1 \vdash v \text{ fits } T) \text{ ClassCast}) s1 \rrbracket \implies \\ & G \vdash \text{Norm } s0 \text{ --Cast } T \text{ e--}\succ v \rightarrow s2 \end{aligned}$$

— cf. 15.19.2

$$\begin{aligned} | \text{Inst: } & \llbracket G \vdash \text{Norm } s0 \text{ --e--}\succ v \rightarrow s1; \\ & b = (v \neq \text{Null} \wedge G, \text{store } s1 \vdash v \text{ fits } \text{RefT } T) \rrbracket \implies \\ & G \vdash \text{Norm } s0 \text{ --e InstOf } T \text{--}\succ \text{Bool } b \rightarrow s1 \end{aligned}$$

— cf. 15.7.1

$$| \text{Lit: } G \vdash \text{Norm } s \text{ --Lit } v \text{--}\succ v \rightarrow \text{Norm } s$$

$$\begin{aligned} | \text{UnOp: } & \llbracket G \vdash \text{Norm } s0 \text{ --e--}\succ v \rightarrow s1 \rrbracket \\ & \implies G \vdash \text{Norm } s0 \text{ --UnOp } \text{unop } e \text{--}\succ (\text{eval-unop } \text{unop } v) \rightarrow s1 \end{aligned}$$

$$\begin{aligned} | \text{BinOp: } & \llbracket G \vdash \text{Norm } s0 \text{ --e1--}\succ v1 \rightarrow s1; \\ & G \vdash s1 \text{ --(if need-second-arg binop } v1 \text{ then (In1l } e2) \text{ else (In1r Skip))} \\ & \text{--}\succ \rightarrow (\text{In1 } v2, s2) \\ & \rrbracket \\ & \implies G \vdash \text{Norm } s0 \text{ --BinOp } \text{binop } e1 \text{ } e2 \text{--}\succ (\text{eval-binop } \text{binop } v1 \text{ } v2) \rightarrow s2 \end{aligned}$$

— cf. 15.10.2

$$| \text{Super: } G \vdash \text{Norm } s \text{ --Super--}\succ \text{val-this } s \rightarrow \text{Norm } s$$

— cf. 15.2

$$\begin{aligned} | \text{Acc: } & \llbracket G \vdash \text{Norm } s0 \text{ --va=}\succ (v, f) \rightarrow s1 \rrbracket \implies \\ & G \vdash \text{Norm } s0 \text{ --Acc } va \text{--}\succ v \rightarrow s1 \end{aligned}$$

— cf. 15.25.1

$$\begin{aligned} | \text{Ass: } & \llbracket G \vdash \text{Norm } s0 \text{ --va=}\succ (w, f) \rightarrow s1; \\ & G \vdash s1 \text{ --e--}\succ v \rightarrow s2 \rrbracket \implies \\ & G \vdash \text{Norm } s0 \text{ --va:=e--}\succ v \rightarrow \text{assign } f \text{ } v \text{ } s2 \end{aligned}$$

— cf. 15.24

$$\begin{aligned} | \text{Cond: } & \llbracket G \vdash \text{Norm } s0 \text{ --e0--}\succ b \rightarrow s1; \\ & G \vdash s1 \text{ --(if the-Bool } b \text{ then } e1 \text{ else } e2) \text{--}\succ v \rightarrow s2 \rrbracket \implies \\ & G \vdash \text{Norm } s0 \text{ --e0 ? } e1 : e2 \text{--}\succ v \rightarrow s2 \end{aligned}$$

— 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

$$\begin{aligned} | \text{Call: } & \llbracket G \vdash \text{Norm } s0 \text{ --e--}\succ a' \rightarrow s1; G \vdash s1 \text{ --args=}\succ vs \rightarrow s2; \\ & D = \text{invocation-declclass } G \text{ mode } (\text{store } s2) \text{ } a' \text{ statT } (\llbracket \text{name=mn, parTs=pTs} \rrbracket); \\ & s3 = \text{init-lvars } G \text{ } D \text{ } (\llbracket \text{name=mn, parTs=pTs} \rrbracket) \text{ mode } a' \text{ } vs \text{ } s2; \\ & s3' = \text{check-method-access } G \text{ accC statT mode } (\llbracket \text{name=mn, parTs=pTs} \rrbracket) \text{ } a' \text{ } s3; \end{aligned}$$



$G \vdash s3' - \text{Methd } D \llbracket \text{name} = mn, \text{parTs} = pTs \rrbracket - \succ v \rightarrow s4 \rrbracket$   
 $\implies$   
 $G \vdash \text{Norm } s0 - \{accC, statT, mode\} e \cdot mn(\{pTs\} args) - \succ v \rightarrow (\text{restore-lvars } s2 \ s4)$   
 — The accessibility check is after *init-lvars*, to keep it simple. *init-lvars* already tests for the absence of a null-pointer reference in case of an instance method invocation.

| *Methd*:  $\llbracket G \vdash \text{Norm } s0 - \text{body } G \ D \ \text{sig} - \succ v \rightarrow s1 \rrbracket \implies$   
 $G \vdash \text{Norm } s0 - \text{Methd } D \ \text{sig} - \succ v \rightarrow s1$

| *Body*:  $\llbracket G \vdash \text{Norm } s0 - \text{Init } D \rightarrow s1; G \vdash s1 - c \rightarrow s2;$   
 $s3 = (\text{if } (\exists l. \text{abrupt } s2 = \text{Some } (\text{Jump } (\text{Break } l))) \vee$   
 $\text{abrupt } s2 = \text{Some } (\text{Jump } (\text{Cont } l)))$   
 $\text{then } \text{abupd } (\lambda x. \text{Some } (\text{Error CrossMethodJump})) \ s2$   
 $\text{else } s2 \rrbracket \implies$   
 $G \vdash \text{Norm } s0 - \text{Body } D \ c - \succ \text{the } (\text{locals } (\text{store } s2) \ \text{Result})$   
 $\rightarrow \text{abupd } (\text{absorb } \text{Ret}) \ s3$

— cf. 14.15, 12.4.1

— We filter out a break/continue in *s2*, so that we can proof definite assignment correct, without the need of conformance of the state. By this the different parts of the typesafety proof can be disentangled a little.

— evaluation of variables

— cf. 15.13.1, 15.7.2

| *LVar*:  $G \vdash \text{Norm } s - \text{LVar } vn = \succ \text{lvar } vn \ s \rightarrow \text{Norm } s$

— cf. 15.10.1, 12.4.1

| *FVar*:  $\llbracket G \vdash \text{Norm } s0 - \text{Init } \text{statDeclC} \rightarrow s1; G \vdash s1 - e - \succ a \rightarrow s2;$   
 $(v, s2') = \text{fvar } \text{statDeclC} \ \text{stat fn } a \ s2;$   
 $s3 = \text{check-field-access } G \ \text{accC } \text{statDeclC} \ \text{fn } \text{stat } a \ s2' \rrbracket \implies$   
 $G \vdash \text{Norm } s0 - \{accC, \text{statDeclC}, \text{stat}\} e..fn = \succ v \rightarrow s3$

— The accessibility check is after *fvar*, to keep it simple. *fvar* already tests for the absence of a null-pointer reference in case of an instance field

— cf. 15.12.1, 15.25.1

| *AVar*:  $\llbracket G \vdash \text{Norm } s0 - e1 - \succ a \rightarrow s1; G \vdash s1 - e2 - \succ i \rightarrow s2;$   
 $(v, s2') = \text{avar } G \ i \ a \ s2 \rrbracket \implies$   
 $G \vdash \text{Norm } s0 - e1.[e2] = \succ v \rightarrow s2'$

— evaluation of expression lists

— cf. 15.11.4.2

| *Nil*:  $G \vdash \text{Norm } s0 - [] \doteq \succ [] \rightarrow \text{Norm } s0$

— cf. 15.6.4

| *Cons*:  $\llbracket G \vdash \text{Norm } s0 - e - \succ v \rightarrow s1;$   
 $G \vdash \quad s1 - es \doteq \succ vs \rightarrow s2 \rrbracket \implies$   
 $G \vdash \text{Norm } s0 - e \# es \doteq \succ v \# vs \rightarrow s2$

**ML**  $\llbracket$   
*bind-thm* (*eval-induct*-, *rearrange-prems*  
 $[0, 1, 2, 8, 4, 30, 31, 27, 15, 16,$   
 $17, 18, 19, 20, 21, 3, 5, 25, 26, 23, 6,$   
 $7, 11, 9, 13, 14, 12, 22, 10, 28,$   
 $29, 24] \ @ \{ \text{thm } \text{eval.induct} \}$   
 $\rrbracket$

$$G \vdash \text{Norm } s \rightarrow \text{In1l } (e0 \text{ ? } e1 : e2) \quad \gamma \rightarrow (v, s')$$

$$\begin{array}{ll}
G \vdash \text{Norm } s - \text{In1r } (\text{If}(e) \ c1 \ \text{Else } c2) & \succ \rightarrow (x, s') \\
G \vdash \text{Norm } s - \text{In1r } (l \bullet \text{While}(e) \ c) & \succ \rightarrow (x, s') \\
G \vdash \text{Norm } s - \text{In1r } (c1 \ \text{Finally } c2) & \succ \rightarrow (x, s') \\
G \vdash \text{Norm } s - \text{In1r } (\text{Throw } e) & \succ \rightarrow (x, s') \\
G \vdash \text{Norm } s - \text{In1l } (\text{NewC } C) & \succ \rightarrow (v, s') \\
G \vdash \text{Norm } s - \text{In1l } (\text{New } T[e]) & \succ \rightarrow (v, s') \\
G \vdash \text{Norm } s - \text{In1l } (\text{Ass } va \ e) & \succ \rightarrow (v, s') \\
G \vdash \text{Norm } s - \text{In1r } (\text{Try } c1 \ \text{Catch}(tn \ vn) \ c2) & \succ \rightarrow (x, s') \\
G \vdash \text{Norm } s - \text{In2 } (\{accC, statDeclC, stat\}e..fn) & \succ \rightarrow (v, s') \\
G \vdash \text{Norm } s - \text{In2 } (e1.[e2]) & \succ \rightarrow (v, s') \\
G \vdash \text{Norm } s - \text{In1l } (\{accC, statT, mode\}e.mn(\{pT\}p)) & \succ \rightarrow (v, s') \\
G \vdash \text{Norm } s - \text{In1r } (\text{Init } C) & \succ \rightarrow (x, s')
\end{array}$$

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

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

**declaration**  $\ll K \ (\text{Simplifier.map-ss } (fn \ ss \Rightarrow ss \ \text{addloop } (\text{split-all-tac}, \text{split-all-tac}))) \gg$

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

**lemma** *eval-Inj-elim*:

$G \vdash s - t \succ \rightarrow (w, s')$

$\Rightarrow$  *case* *t* *of*

*In1* *ec*  $\Rightarrow$  (*case* *ec* *of*  
*Inl* *e*  $\Rightarrow$  ( $\exists v. w = \text{In1 } v$ )  
| *Inr* *c*  $\Rightarrow w = \Diamond$ )  
| *In2* *e*  $\Rightarrow$  ( $\exists v. w = \text{In2 } v$ )  
| *In3* *e*  $\Rightarrow$  ( $\exists v. w = \text{In3 } v$ ))

**apply** (*erule* *eval-cases*)

**apply** *auto*

**apply** (*induct-tac* *t*)

**apply** (*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*:  $G \vdash s - \text{In1l } t \succ \rightarrow (w, s') = (\exists v. w = \text{In1 } v \wedge G \vdash s - t \succ v \rightarrow s')$   
**by** (*auto*, *frule* *eval-Inj-elim*, *auto*)

**lemma** *eval-var-eq*:  $G \vdash s - \text{In2 } t \succ \rightarrow (w, s') = (\exists vf. w = \text{In2 } vf \wedge G \vdash s - t = vf \rightarrow s')$   
**by** (*auto*, *frule* *eval-Inj-elim*, *auto*)

**lemma** *eval-exprs-eq*:  $G \vdash s - \text{In3 } t \succ \rightarrow (w, s') = (\exists vs. w = \text{In3 } vs \wedge G \vdash s - t \doteq vs \rightarrow s')$   
**by** (*auto*, *frule* *eval-Inj-elim*, *auto*)

**lemma** *eval-stmt-eq*:  $G \vdash s - \text{In1r } t \succ \rightarrow (w, s') = (w = \Diamond \wedge G \vdash s - t \rightarrow s')$   
**by** (*auto*, *frule* *eval-Inj-elim*, *auto*, *frule* *eval-Inj-elim*, *auto*)

**simproc-setup** *eval-expr* ( $G \vdash s - \text{In1l } t \succ \rightarrow (w, s')$ ) =  $\ll$   
*fn* -  $\Rightarrow$  *fn* -  $\Rightarrow$  *fn* *ct*  $\Rightarrow$   
(*case* *Thm.term-of* *ct* *of*  
(- \$ - \$ - \$ (Const - \$ -) \$ -)  $\Rightarrow$  NONE  
| -  $\Rightarrow$  SOME (*mk-meta-eq* @{*thm eval-expr-eq*})  $\gg$

```

simproc-setup eval-var ( $G \vdash s -In2 t \rightarrow (w, s')$ ) =  $\ll$ 
  fn - => fn - => fn ct =>
    (case Thm.term-of ct of
      (- $ - $ - $ (Const - $ -) $ -) => NONE
      | - => SOME (mk-meta-eq @{thm eval-var-eq}))  $\gg$ 

```

```

simproc-setup eval-exprs ( $G \vdash s -In3 t \rightarrow (w, s')$ ) =  $\ll$ 
  fn - => fn - => fn ct =>
    (case Thm.term-of ct of
      (- $ - $ - $ (Const - $ -) $ -) => NONE
      | - => SOME (mk-meta-eq @{thm eval-exprs-eq}))  $\gg$ 

```

```

simproc-setup eval-stmt ( $G \vdash s -In1r t \rightarrow (w, s')$ ) =  $\ll$ 
  fn - => fn - => fn ct =>
    (case Thm.term-of ct of
      (- $ - $ - $ (Const - $ -) $ -) => NONE
      | - => SOME (mk-meta-eq @{thm eval-stmt-eq}))  $\gg$ 

```

```

ML  $\ll$ 
  bind-thms (AbruptIs, sum3-instantiate @{thm eval.Abrupt})
 $\gg$ 

```

**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*:  $G \vdash Norm\ s -Callee\ l\ e \rightarrow v \rightarrow s' = False$

**proof** –

```

{ fix s t v s'
  assume eval:  $G \vdash s -t \rightarrow (v, s')$  and
    normal: normal s and
    callee:  $t = In1l\ (Callee\ l\ e)$ 
  then have False by induct auto
}
then show ?thesis
by (cases s') fastsimp
qed

```

**lemma** *eval-InsInitE*:  $G \vdash Norm\ s -InsInitE\ c\ e \rightarrow v \rightarrow s' = False$

**proof** –

```

{ fix s t v s'
  assume eval:  $G \vdash s -t \rightarrow (v, s')$  and
    normal: normal s and
    callee:  $t = In1l\ (InsInitE\ c\ e)$ 
  then have False by induct auto
}
then show ?thesis
by (cases s') fastsimp
qed

```

**lemma** *eval-InsInitV*:  $G \vdash Norm\ s -InsInitV\ c\ w \rightarrow v \rightarrow s' = False$

**proof** –

```

{ fix s t v s'
  assume eval:  $G \vdash s -t \rightarrow (v, s')$  and
    normal: normal s and
    callee:  $t = In2\ (InsInitV\ c\ w)$ 

```

```

    then have False by induct auto
  }
  then show ?thesis
    by (cases s') fastsimp
qed

```

**lemma eval-FinA:**  $G \vdash \text{Norm } s \text{--FinA } a \text{--} c \rightarrow s' = \text{False}$

**proof** –

```

{ fix s t v s'
  assume eval:  $G \vdash s \text{--} t \rightarrow (v, s')$  and
    normal: normal s and
    callee:  $t = \text{In1r } (\text{FinA } a \text{--} c)$ 
  then have False by induct auto
}
then show ?thesis
  by (cases s') fastsimp
qed

```

**lemma eval-no-abrupt-lemma:**

$\bigwedge s s'. G \vdash s \text{--} t \rightarrow (w, s') \implies \text{normal } s' \longrightarrow \text{normal } s$   
**by** (erule eval-cases, auto)

**lemma eval-no-abrupt:**

```

 $G \vdash (x, s) \text{--} t \rightarrow (w, \text{Norm } s') =$ 
 $(x = \text{None} \wedge G \vdash \text{Norm } s \text{--} t \rightarrow (w, \text{Norm } s'))$ 
apply auto
apply (frule eval-no-abrupt-lemma, auto)+
done

```

```

simproc-setup eval-no-abrupt ( $G \vdash (x, s) \text{--} e \rightarrow (w, \text{Norm } s')$ ) = <<
  fn - => fn - => 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:**

$G \vdash s \text{--} t \rightarrow (v, s') \implies \text{abrupt } s = \text{Some } xc \longrightarrow s' = s \wedge v = \text{arbitrary3 } t$   
**by** (erule eval-cases, auto)

**lemma eval-abrupt:**

```

 $G \vdash (\text{Some } xc, s) \text{--} t \rightarrow (w, s') =$ 
 $(s' = (\text{Some } xc, s) \wedge w = \text{arbitrary3 } t \wedge$ 
 $G \vdash (\text{Some } xc, s) \text{--} t \rightarrow (\text{arbitrary3 } t, (\text{Some } xc, s)))$ 
apply auto
apply (frule eval-abrupt-lemma, auto)+
done

```

```

simproc-setup eval-abrupt ( $G \vdash (\text{Some } xc, s) \text{--} e \rightarrow (w, s')$ ) = <<
  fn - => fn - => 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:**  $G \vdash s \text{ --Lit } v \text{ --}\succ \text{(if normal } s \text{ then } v \text{ else arbitrary)} \rightarrow s$   
**apply** (case-tac s, case-tac a = None)  
**by** (auto intro!: eval.Lit)

**lemma SkipI** [intro!]:  $G \vdash s \text{ --Skip} \rightarrow s$   
**apply** (case-tac s, case-tac a = None)  
**by** (auto intro!: eval.Skip)

**lemma ExprI:**  $G \vdash s \text{ --}e \text{ --}\succ v \rightarrow s' \implies G \vdash s \text{ --Expr } e \rightarrow s'$   
**apply** (case-tac s, case-tac a = None)  
**by** (auto intro!: eval.Expr)

**lemma CompI:**  $\llbracket G \vdash s \text{ --}c1 \rightarrow s1; G \vdash s1 \text{ --}c2 \rightarrow s2 \rrbracket \implies G \vdash s \text{ --}c1;; c2 \rightarrow s2$   
**apply** (case-tac s, case-tac a = None)  
**by** (auto intro!: eval.Comp)

**lemma CondI:**  
 $\bigwedge s1. \llbracket G \vdash s \text{ --}e \text{ --}\succ b \rightarrow s1; G \vdash s1 \text{ --(if the-Bool } b \text{ then } e1 \text{ else } e2) \text{ --}\succ v \rightarrow s2 \rrbracket \implies$   
 $G \vdash s \text{ --}e ? e1 : e2 \text{ --}\succ \text{(if normal } s1 \text{ then } v \text{ else arbitrary)} \rightarrow s2$   
**apply** (case-tac s, case-tac a = None)  
**by** (auto intro!: eval.Cond)

**lemma IfI:**  $\llbracket G \vdash s \text{ --}e \text{ --}\succ v \rightarrow s1; G \vdash s1 \text{ --(if the-Bool } v \text{ then } c1 \text{ else } c2) \rightarrow s2 \rrbracket$   
 $\implies G \vdash s \text{ --If}(e) \text{ } c1 \text{ Else } c2 \rightarrow s2$   
**apply** (case-tac s, case-tac a = None)  
**by** (auto intro!: eval.If)

**lemma MethdI:**  $G \vdash s \text{ --body } G \text{ } C \text{ sig} \text{ --}\succ v \rightarrow s'$   
 $\implies G \vdash s \text{ --Methd } C \text{ sig} \text{ --}\succ v \rightarrow s'$   
**apply** (case-tac s, case-tac a = None)  
**by** (auto intro!: eval.Methd)

**lemma eval-Call:**  
 $\llbracket G \vdash \text{Norm } s0 \text{ --}e \text{ --}\succ a' \rightarrow s1; G \vdash s1 \text{ --ps} \dot{=} \succ pvs \rightarrow s2;$   
 $D = \text{invocation-declclass } G \text{ mode (store } s2) \text{ } a' \text{ statT } (\llbracket \text{name}=\text{mn}, \text{parTs}=\text{pTs} \rrbracket);$   
 $s3 = \text{init-lvars } G \text{ } D \text{ } (\llbracket \text{name}=\text{mn}, \text{parTs}=\text{pTs} \rrbracket) \text{ mode } a' \text{ pvs } s2;$   
 $s3' = \text{check-method-access } G \text{ accC statT mode } (\llbracket \text{name}=\text{mn}, \text{parTs}=\text{pTs} \rrbracket) \text{ } a' \text{ } s3;$   
 $G \vdash s3' \text{ --Methd } D \text{ } (\llbracket \text{name}=\text{mn}, \text{parTs}=\text{pTs} \rrbracket) \text{ --}\succ v \rightarrow s4;$   
 $s4' = \text{restore-lvars } s2 \text{ } s4 \rrbracket \implies$   
 $G \vdash \text{Norm } s0 \text{ --}\{ \text{accC}, \text{statT}, \text{mode} \} e \cdot \text{mn}(\{ \text{pTs} \} \text{ps}) \text{ --}\succ v \rightarrow s4'$   
**apply** (drule eval.Call, assumption)  
**apply** (rule HOL.refl)  
**apply** simp+  
**done**

**lemma eval-Init:**  
 $\llbracket \text{if inited } C \text{ (globs } s0) \text{ then } s3 = \text{Norm } s0$

```

else  $G \vdash \text{Norm } (\text{init-class-obj } G \ C \ s0)$ 
   $\neg(\text{if } C = \text{Object then Skip else Init } (\text{super } (\text{the } (\text{class } G \ C)))) \rightarrow s1 \wedge$ 
   $G \vdash \text{set-lvars empty } s1 \neg(\text{init } (\text{the } (\text{class } G \ C))) \rightarrow s2 \wedge$ 
   $s3 = \text{restore-lvars } s1 \ s2 \parallel \implies$ 
   $G \vdash \text{Norm } s0 \neg \text{Init } C \rightarrow s3$ 
apply (rule eval.Init)
apply auto
done

```

```

lemma init-done:  $\text{initd } C \ s \implies G \vdash s \neg \text{Init } C \rightarrow s$ 
apply (case-tac s, simp)
apply (case-tac a)
apply safe
apply (rule eval-Init)
apply auto
done

```

```

lemma eval-StatRef:
 $G \vdash s \neg \text{StatRef } rt \neg \succ (\text{if abrupt } s = \text{None then Null else arbitrary}) \rightarrow s$ 
apply (case-tac s, simp)
apply (case-tac a = None)
apply (auto del: eval.Abrupt intro!: eval.intros)
done

```

```

lemma SkipD [dest!]:  $G \vdash s \neg \text{Skip} \rightarrow s' \implies s' = s$ 
apply (erule eval-cases)
by auto

```

```

lemma Skip-eq [simp]:  $G \vdash s \neg \text{Skip} \rightarrow s' = (s = s')$ 
by auto

```

```

lemma init-retains-locals [rule-format (no-asm)]:  $G \vdash s \neg t \succ \rightarrow (w, s') \implies$ 
 $(\forall C. t = \text{In1r } (\text{Init } C) \longrightarrow \text{locals } (\text{store } s) = \text{locals } (\text{store } s'))$ 
apply (erule eval.induct)
apply (simp (no-asm-use) split del: split-if-asm option.split-asm)+
apply auto
done

```

```

lemma halloc-xcpt [dest!]:
 $\bigwedge s'. G \vdash (\text{Some } xc, s) \neg \text{halloc } oi \succ a \rightarrow s' \implies s' = (\text{Some } xc, s)$ 
apply (erule-tac halloc-elim-cases)
by auto

```

```

lemma eval-Methd:
 $G \vdash s \neg \text{In1l}(\text{body } G \ C \ sig) \succ \rightarrow (w, s')$ 
 $\implies G \vdash s \neg \text{In1l}(\text{Methd } C \ sig) \succ \rightarrow (w, s')$ 
apply (case-tac s)
apply (case-tac a)

```

```

apply clarsimp+
apply (erule eval.Methd)
apply (drule eval-abrupt-lemma)
apply force
done

```

```

lemma eval-Body:  $\llbracket G \vdash \text{Norm } s0 \text{ --Init } D \rightarrow s1; G \vdash s1 \text{ --}c \rightarrow s2;$ 
   $\text{res} = \text{the } (\text{locals } (\text{store } s2) \text{ Result});$ 
   $s3 = (\text{if } (\exists l. \text{abrupt } s2 = \text{Some } (\text{Jump } (\text{Break } l))) \vee$ 
     $\text{abrupt } s2 = \text{Some } (\text{Jump } (\text{Cont } l)))$ 
   $\text{then } \text{abupd } (\lambda x. \text{Some } (\text{Error CrossMethodJump})) s2$ 
   $\text{else } s2);$ 
   $s4 = \text{abupd } (\text{absorb Ret}) s3 \rrbracket \implies$ 
 $G \vdash \text{Norm } s0 \text{ --Body } D \text{ --}c \text{ --} \text{res} \rightarrow s4$ 
by (auto elim: eval.Body)

```

```

lemma eval-binop-arg2-indep:
 $\neg \text{need-second-arg binop } v1 \implies \text{eval-binop binop } v1 \ x = \text{eval-binop binop } v1 \ y$ 
by (cases binop)
  (simp-all add: need-second-arg-def)

```

```

lemma eval-BinOp-arg2-indepI:
  assumes eval-e1:  $G \vdash \text{Norm } s0 \text{ --}e1 \text{ --} \text{v1} \rightarrow s1$  and
    no-need:  $\neg \text{need-second-arg binop } v1$ 
  shows  $G \vdash \text{Norm } s0 \text{ --BinOp binop } e1 \ e2 \text{ --} ( \text{eval-binop binop } v1 \ v2 ) \rightarrow s1$ 
    (is ?EvalBinOp v2)
proof –
  from eval-e1
  have ?EvalBinOp Unit
    by (rule eval.BinOp)
    (simp add: no-need)
  moreover
  from no-need
  have  $\text{eval-binop binop } v1 \ \text{Unit} = \text{eval-binop binop } v1 \ v2$ 
    by (simp add: eval-binop-arg2-indep)
  ultimately
  show ?thesis
    by simp
qed

```

### single valued

```

lemma unique-halloc [rule-format (no-asm)]:
   $G \vdash s \text{ --halloc } oi \text{ --}a \rightarrow s' \implies G \vdash s \text{ --halloc } oi \text{ --}a' \rightarrow s'' \longrightarrow a' = a \wedge s'' = s'$ 
apply (erule halloc.induct)
apply (auto elim!: halloc-elim-cases split del: split-if split-if-asm)
apply (drule trans [THEN sym], erule sym)
defer
apply (drule trans [THEN sym], erule sym)
apply auto
done

```

```

lemma single-valued-halloc:

```



*single-valued*  $\{((s, oi), (a, s')). G \vdash s -\text{halloc } oi \succ a \rightarrow s'\}$   
**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*  
*split del: split-if split-if-asm*)  
**done**

**lemma** *single-valued-sxalloc*: *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*  $\ll \text{ALLGOALS } (\text{EVERY'}$   
 $[\text{strip-tac, rotate-tac } \sim 1, \text{eresolve-tac } (\text{thms eval-elim-cases})] \gg)$ )

**prefer** 28  
**apply** (*simp (no-asm-use) only: split add: split-if-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 add: split-if-asm, blast*)

**apply** (*blast dest: unique-sxalloc unique-halloc split-pairD*)  
**done**

**lemma** *single-valued-eval*:  
*single-valued*  $\{((s, t), (v, s')). G \vdash s -t \succ \rightarrow (v, s')\}$   
**apply** (*unfold single-valued-def*)  
**by** (*clarify, drule (1) unique-eval, auto*)

**end**



## Chapter 16

### Example

### 43 Example Bali program

**theory** *Example* **imports** *Eval WellForm* **begin**

The following example Bali program includes:

- class and interface declarations with inheritance, hiding of fields, overriding of methods (with refined result type), array type,
- method call (with dynamic binding), parameter access, return expressions,
- expression statements, sequential composition, literal values, local assignment, local access, field assignment, type cast,
- exception generation and propagation, try and catch statement, throw statement
- instance creation and (default) static initialization

```
package java_lang

public interface HasFoo {
  public Base foo(Base z);
}

public class Base implements HasFoo {
  static boolean arr[] = new boolean[2];
  public HasFoo vee;
  public Base foo(Base z) {
    return z;
  }
}

public class Ext extends Base {
  public int vee;
  public Ext foo(Base z) {
    ((Ext)z).vee = 1;
    return null;
  }
}

public class Main {
  public static void main(String args[]) throws Throwable {
    Base e = new Ext();
    try {e.foo(null); }
    catch(NullPointerException z) {
      while(Ext.arr[2]) ;
    }
  }
}
```

**declare** *widen.null* [*intro*]

**lemma** *wf-fdecl-def2*:  $\bigwedge fd. wf-fdecl\ G\ P\ fd = is-acc-type\ G\ P\ (type\ (snd\ fd))$   
**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'
```

### consts

```
tnam' :: tnam'  $\Rightarrow$  tnam
vnam' :: vnam'  $\Rightarrow$  vname
label' :: label'  $\Rightarrow$  label
```

### axioms

```
inj-tnam' [simp]: (tnam' x = tnam' y) = (x = y)
inj-vnam' [simp]: (vnam' x = vnam' y) = (x = y)
inj-label' [simp]: (label' x = label' y) = (x = y)
```

```
surj-tnam':  $\exists m. n = tnam' m$ 
surj-vnam':  $\exists m. n = vnam' m$ 
surj-label':  $\exists m. n = label' m$ 
```

### abbreviation

```
HasFoo :: qname where
HasFoo == ( $\lambda pid=java-lang, tid=TName (tnam' HasFoo')$ )
```

### abbreviation

```
Base :: qname where
Base == ( $\lambda pid=java-lang, tid=TName (tnam' Base')$ )
```

### abbreviation

```
Ext :: qname where
Ext == ( $\lambda pid=java-lang, tid=TName (tnam' Ext')$ )
```

### abbreviation

```
Main :: qname where
Main == ( $\lambda pid=java-lang, tid=TName (tnam' Main')$ )
```

### abbreviation

```
arr :: vname where
arr == (vnam' arr')
```

### abbreviation

```
vee :: vname where
vee == (vnam' vee')
```

### abbreviation

```
z :: vname where
z == (vnam' z')
```

### abbreviation

```
e :: vname where
e == (vnam' e')
```

**abbreviation**

*lab1*:: *label* **where**  
*lab1* == *label'* *lab1'*

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

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

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

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

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

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

**classes and interfaces****defs**

*Object-mdecls-def*: *Object-mdecls* ≡ []  
*SXcpt-mdecls-def*: *SXcpt-mdecls* ≡ []

**consts**

*foo* :: *mname*

**constdefs**

*foo-sig* :: *sig*  
*foo-sig* ≡ (λ*name=foo,parTs*=[*Class Base*]λ)

*foo-mhead* :: *mhead*  
*foo-mhead* ≡ (λ*access=Public,static=False,par*s=[*z*],*resT=Class Base*λ)

**constdefs**

*Base-foo* :: *mdecl*  
*Base-foo* ≡ (*foo-sig*, (λ*access=Public,static=False,par*s=[*z*],*resT=Class Base*,  
*mbody*=(λ*lcls*=[],*stmt=Return* (!!*z*)λ)))

**constdefs**

*Ext-foo* :: *mdecl*  
*Ext-foo* ≡ (*foo-sig*,  
(λ*access=Public,static=False,par*s=[*z*],*resT=Class Ext*,  
*mbody*=(λ*lcls*=[]

```

,stmt=Expr({Ext,Ext,False} Cast (Class Ext) (!!z)..vee :=
                                   Lit (Intg 1)) ;;
                                   Return (Lit Null))
))

```

**constdefs**

```

arr-viewed-from :: qtname ⇒ qtname ⇒ var
arr-viewed-from accC C ≡ {accC,Base,True}StatRef (ClassT C)..arr

```

```

BaseCl :: class
BaseCl ≡ (|access=Public,
          cfields=[(arr, (|access=Public,static=True ,type=PrimT Boolean.[])),
                    (vee, (|access=Public,static=False,type=Iface HasFoo []))],
          methods=[Base-foo],
          init=Expr(arr-viewed-from Base Base
                    :=New (PrimT Boolean)[Lit (Intg 2)]),
          super=Object,
          superIfs=[HasFoo])

```

```

ExtCl :: class
ExtCl ≡ (|access=Public,
          cfields=[(vee, (|access=Public,static=False,type= PrimT Integer[]))],
          methods=[Ext-foo],
          init=Skip,
          super=Base,
          superIfs=[] )

```

```

MainCl :: class
MainCl ≡ (|access=Public,
          cfields=[],
          methods=[],
          init=Skip,
          super=Object,
          superIfs=[] )

```

**constdefs**

```

HasFooInt :: iface
HasFooInt ≡ (|access=Public,imethods=[(foo-sig, foo-mhead)],isuperIfs=[])

```

```

Ifaces ::idecl list
Ifaces ≡ [(HasFoo,HasFooInt)]

```

```

Classes ::cdecl list
Classes ≡ [(Base,BaseCl),(Ext,ExtCl),(Main,MainCl)]@standard-classes

```

```

lemmas table-classes-defs =
  Classes-def standard-classes-def ObjectC-def SXcptC-def

```

```

lemma table-ifaces [simp]: table-of Ifaces = empty(HasFoo↦HasFooInt)
apply (unfold Ifaces-def)
apply (simp (no-asm))
done

```

```

lemma table-classes-Object [simp]:

```

```

table-of Classes Object = Some ( $\lambda$ access=Public,cfields=[]
                               ,methods=Object-mdecls
                               ,init=Skip,super=arbitrary,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 ( $\lambda$ 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 == ( $\lambda$ ifaces=Ifaces,classes=Classes)

```

### constdefs

```

test   :: (ty)list  $\Rightarrow$  stmt
test pTs  $\equiv$  e:=NewC Ext;
            Try Expr( $\{Main,ClassT Base,IntVir\}!!e.foo(\{pTs\}[Lit Null])$ )
            Catch((SXcpt NullPointer) z)
            (lab1• While(Acc
                       (Acc (arr-viewed-from Main Ext).[Lit (Intg 2)])) Skip)

```



**well-structuredness**

**lemma** *not-Object-subcls-any* [elim!]:  $(Object, C) \in (subcls1\ tprg)^+ \implies R$   
**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!]:  
 $((pid=java-lang, tid=TName\ tn), (pid=java-lang, tid=TName\ tn)) \in (subcls1\ tprg)^+ \implies R$   
**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*:  $ws-idecl\ tprg\ HasFoo\ []$   
**apply** (unfold ws-idecl-def)  
**apply** (simp (no-asm))  
**done**

**lemma** *ws-cdecl-Object*:  $ws-cdecl\ tprg\ Object\ any$   
**apply** (unfold ws-cdecl-def)  
**apply** auto  
**done**

**lemma** *ws-cdecl-Throwable*:  $ws-cdecl\ tprg\ (SXcpt\ Throwable)\ Object$   
**apply** (unfold ws-cdecl-def)  
**apply** auto  
**done**

```

lemma ws-cdecl-SXcpt: ws-cdecl tprg (SXcpt xn) (SXcpt Throwable)
apply (unfold ws-cdecl-def)
apply auto
done

```

```

lemma ws-cdecl-Base: ws-cdecl tprg Base Object
apply (unfold ws-cdecl-def)
apply auto
done

```

```

lemma ws-cdecl-Ext: ws-cdecl tprg Ext Base
apply (unfold ws-cdecl-def)
apply auto
done

```

```

lemma ws-cdecl-Main: ws-cdecl tprg Main Object
apply (unfold ws-cdecl-def)
apply auto
done

```

```

lemmas ws-cdecls = ws-cdecl-SXcpt ws-cdecl-Object ws-cdecl-Throwable
         ws-cdecl-Base ws-cdecl-Ext ws-cdecl-Main

```

```

declare not-Object-subcls-any [rule del]
         not-Throwable-subcls-SXcpt [rule del]
         not-SXcpt-n-subcls-SXcpt-n [rule del]
         not-Base-subcls-Ext [rule del] not-TName-n-subcls-TName-n [rule del]

```

```

lemma ws-idecl-all:
  G=tprg  $\implies (\forall (I,i)\in set Ifaces. ws-idecl G I (isuperIfs i))$ 
apply (simp (no-asm) add: Ifaces-def HasFooInt-def)
apply (auto intro!: ws-idecl-HasFoo)
done

```

```

lemma ws-cdecl-all: G=tprg  $\implies (\forall (C,c)\in set Classes. ws-cdecl G C (super c))$ 
apply (simp (no-asm) add: Classes-def BaseCl-def ExtCl-def MainCl-def)
apply (auto intro!: ws-cdecls simp add: standard-classes-def ObjectC-def
         SXcptC-def)
done

```

```

lemma ws-tprg: ws-prog tprg
apply (unfold ws-prog-def)
apply (auto intro!: ws-idecl-all ws-cdecl-all)
done

```

### **misc program properties (independent of well-structuredness)**

```

lemma single-iface [simp]: is-iface tprg I = (I = HasFoo)
apply (unfold Ifaces-def)
apply (simp (no-asm))
done

```

```

lemma empty-subint1 [simp]: subint1 tprg = {}
apply (unfold subint1-def Ifaces-def HasFooInt-def)
apply auto
done

```

```

lemma unique-ifaces: unique Ifaces
apply (unfold Ifaces-def)
apply (simp (no-asm))
done

```

```

lemma unique-classes: unique Classes
apply (unfold table-classes-defs )
apply (simp )
done

```

```

lemma SXcpt-subcls-Throwable [simp]: tprg ⊢ SXcpt xn ≤C SXcpt Throwable
apply (rule SXcpt-subcls-Throwable-lemma)
apply auto
done

```

```

lemma Ext-subclseq-Base [simp]: tprg ⊢ Ext ≤C Base
apply (rule subcls-direct1)
apply (simp (no-asm) add: ExtCl-def)
apply (simp add: Object-def)
apply (simp (no-asm))
done

```

```

lemma Ext-subcls-Base [simp]: tprg ⊢ Ext <C Base
apply (rule subcls-direct2)
apply (simp (no-asm) add: ExtCl-def)
apply (simp add: Object-def)
apply (simp (no-asm))
done

```

### fields and method lookup

```

lemma fields-tprg-Object [simp]: DeclConcepts.fields tprg Object = []
by (rule ws-tprg [THEN fields-emptyI], force+)

```

```

lemma fields-tprg-Throwable [simp]:
  DeclConcepts.fields tprg (SXcpt Throwable) = []
by (rule ws-tprg [THEN fields-emptyI], force+)

```

```

lemma fields-tprg-SXcpt [simp]: DeclConcepts.fields tprg (SXcpt xn) = []
apply (case-tac xn = Throwable)
apply (simp (no-asm-simp))
by (rule ws-tprg [THEN fields-emptyI], force+)

```

```

lemmas fields-rec' = fields-rec [OF - ws-tprg]

```

```

lemma fields-Base [simp]:

```

```

DeclConcepts.fields tprg Base
  = [((arr,Base), (access=Public,static=True ,type=PrimT Boolean.[])),
      ((vee,Base), (access=Public,static=False,type=Iface HasFoo []))]
apply (subst fields-rec')
apply (auto simp add: BaseCl-def)
done

```

```

lemma fields-Ext [simp]:
  DeclConcepts.fields tprg Ext
    = [((vee,Ext), (access=Public,static=False,type= PrimT Integer))]
    @ DeclConcepts.fields tprg Base
apply (rule trans)
apply (rule fields-rec')
apply (auto simp add: ExtCl-def Object-def)
done

```

```

lemmas imethds-rec' = imethds-rec [OF - ws-tprg]
lemmas methd-rec' = methd-rec [OF - ws-tprg]

```

```

lemma imethds-HasFoo [simp]:
  imethds tprg HasFoo = o2s ∘ empty(foo-sig↦(HasFoo, foo-mhead))
apply (rule trans)
apply (rule imethds-rec')
apply (auto simp add: HasFooInt-def)
done

```

```

lemma methd-tprg-Object [simp]: methd tprg Object = empty
apply (subst methd-rec')
apply (auto simp add: Object-mdecls-def)
done

```

```

lemma methd-Base [simp]:
  methd tprg Base = table-of [(λ(s,m). (s, Base, m)) Base-foo]
apply (rule trans)
apply (rule methd-rec')
apply (auto simp add: BaseCl-def)
done

```

```

lemma memberid-Base-foo-simp [simp]:
  memberid (mdecl Base-foo) = mid foo-sig
by (simp add: Base-foo-def)

```

```

lemma memberid-Ext-foo-simp [simp]:
  memberid (mdecl Ext-foo) = mid foo-sig
by (simp add: Ext-foo-def)

```

```

lemma Base-declares-foo:
  tprg⊢mdecl Base-foo declared-in Base
by (auto simp add: declared-in-def cdeclaredmethd-def BaseCl-def Base-foo-def)

```

```

lemma foo-sig-not-undeclared-in-Base:

```

$\neg \text{tprg} \vdash \text{mid } \text{foo-sig } \text{undeclared-in } \text{Base}$

**proof** –

**from** *Base-declares-foo*

**show** *?thesis*

**by** (*auto dest!:* *declared-not-undeclared* )

**qed**

**lemma** *Ext-declares-foo:*

*tprg*  $\vdash \text{mdecl } \text{Ext-foo } \text{declared-in } \text{Ext}$

**by** (*auto simp add:* *declared-in-def cdeclaredmethd-def ExtCl-def Ext-foo-def*)

**lemma** *foo-sig-not-undeclared-in-Ext:*

$\neg \text{tprg} \vdash \text{mid } \text{foo-sig } \text{undeclared-in } \text{Ext}$

**proof** –

**from** *Ext-declares-foo*

**show** *?thesis*

**by** (*auto dest!:* *declared-not-undeclared* )

**qed**

**lemma** *Base-foo-not-inherited-in-Ext:*

$\neg \text{tprg} \vdash \text{Ext } \text{inherits } (\text{Base}, \text{mdecl } \text{Base-foo})$

**by** (*auto simp add:* *inherits-def foo-sig-not-undeclared-in-Ext*)

**lemma** *Ext-method-inheritance:*

*filter-tab* ( $\lambda \text{sig } m. \text{tprg} \vdash \text{Ext } \text{inherits } \text{method } \text{sig } m$ )

$(\text{empty}(\text{fst } ((\lambda(s, m). (s, \text{Base}, m)) \text{Base-foo}) \mapsto$

$\text{snd } ((\lambda(s, m). (s, \text{Base}, m)) \text{Base-foo})))$

$= \text{empty}$

**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*  $[(\lambda(s, m). (s, \text{Ext}, m)) \text{Ext-foo}]$

**apply** (*rule trans*)

**apply** (*rule methd-rec<sup>^</sup>*)

**apply** (*auto simp add:* *ExtCl-def Object-def Ext-method-inheritance*)

**done**

**accessibility**

**lemma** *classesDefined:*

$\llbracket \text{class } \text{tprg } C = \text{Some } c; C \neq \text{Object} \rrbracket \implies \exists \text{sc. class } \text{tprg } (\text{super } c) = \text{Some } \text{sc}$

**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 = 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  $\implies$  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  $\implies$  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)) overridesS old  $\implies$  P
apply (drule stat-overrides-commonD)
apply (clarsimp)
apply (frule subclsEval)
apply (rule ws-tprg)
apply (simp)
apply (rule classesDefined)
apply assumption+
apply (frule unique-sig-Base-foo)
apply (auto dest!: declared-not-undeclared intro: foo-sig-undeclared-in-Object
  dest: unique-sig-Base-foo)
done

```

```

lemma Base-foo-no-hide:
  tprg, sig ⊢ (Base, (snd Base-foo)) hides old  $\implies$  P
by (auto dest: hidesD simp add: Base-foo-def member-is-static-simp)

```

**lemma** *Ext-foo-no-hide*:

*tp<sub>rg</sub>,sig*⊢(*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<sub>rg</sub>*⊢ *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<sub>rg</sub>,sig*⊢(*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<sub>rg</sub>*)  
**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<sub>rg</sub>,sig*⊢(*Ext*,(*snd Ext-foo*)) *overrides<sub>S</sub> old*  
 $\implies old = (Base, (snd Base-foo))$   
**apply** (*drule stat-overrides-commonD*)  
**apply** (*clarsimp*)  
**apply** (*frule subclsEval*)  
**apply** (*rule ws-tp<sub>rg</sub>*)  
**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<sub>rg</sub>*⊢(*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 expand-fun-eq 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 expand-fun-eq 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 << bind-thms (wt-intros, map (rewrite-rule @{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]
declare member-is-static-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!: wts)

```

```

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) ( $\langle$ name=foo, parTs=[NT] $\rangle$ ) =
  {((ClassT Base, ( $\langle$ access=Public,static=False,pars=[z],resT=Class Base $\rangle$ ))
  ,[Class Base])}
apply (unfold appl-methds-def)
apply (simp (no-asm))
apply (subgoal-tac tprg  $\vdash$  NT  $\preceq$  Class Base)
apply (auto simp add: cmheads-def Base-foo-defs)
done

```

```

lemma max-spec-Base-foo: max-spec tprg S (ClassT Base) ( $\langle$ name=foo,parTs=[NT] $\rangle$ ) =
  {((ClassT Base, ( $\langle$ access=Public,static=False,pars=[z],resT=Class Base $\rangle$ ))
  , [Class Base])}
apply (unfold max-spec-def)

```

```

apply (simp (no-asm) add: appl-methds-Base-foo)
apply auto
done

```

### well-typedness

```

lemma wt-test: ( $\llbracket \text{prg}=\text{tprg}, \text{cls}=\text{Main}, \text{lcl}=\text{empty}(VName\ e \mapsto \text{Class}\ Base) \rrbracket \vdash \text{test } ?pTs::\checkmark$ )
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

```

lemma da-test: ( $\llbracket \text{prg}=\text{tprg}, \text{cls}=\text{Main}, \text{lcl}=\text{empty}(VName\ e \mapsto \text{Class}\ Base) \rrbracket$ )

```



```

       $\vdash \{\} \gg \langle \text{test } ?pTs \rangle \gg (\text{norm} = \{ VName\ e \}, brk = \lambda\ l. UNIV)$ 
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 blast
apply simp
apply (simp add: intersect-ts-def)
done

```

### execution

```

lemma alloc-one:  $\bigwedge a\ obj. \llbracket \text{the } (new\text{-Addr } h) = a; \text{atleast-free } h\ (Suc\ n) \rrbracket \implies$ 
  new-Addr h = Some a  $\wedge$  atleast-free (h(a $\mapsto$ obj)) n
apply (frule atleast-free-SucD)
apply (drule atleast-free-Suc [THEN iffD1])
apply clarsimp
apply (frule new-Addr-SomeI)
apply force
done

```

```

declare fvar-def2 [simp] avar-def2 [simp] init-lvars-def2 [simp]
declare init-obj-def [simp] var-tys-def [simp] fields-table-def [simp]
declare BaseCl-def [simp] ExtCl-def [simp] Ext-foo-def [simp]
  Base-foo-defs [simp]

```

```

ML  $\ll$  bind-thms (eval-intros, map
  (simplify (@{simpset} delsimps @{thms Skip-eq}
    addsimps @{thms lvar-def}) o

```

*rewrite-rule* [*@{thm assign-def}*, *@{thm Let-def}*]) *@{thms eval.intros}*)  $\gg$   
**lemmas** *eval-Is* = *eval-Init eval-StatRef AbruptIs eval-intros*

**consts**

*a* :: *loc*  
*b* :: *loc*  
*c* :: *loc*

**abbreviation** *one* == *Suc 0*

**abbreviation** *two* == *Suc one*

**abbreviation** *tree* == *Suc two*

**abbreviation** *four* == *Suc tree*

**syntax**

*obj-a* :: *obj*  
*obj-b* :: *obj*  
*obj-c* :: *obj*  
*arr-N* :: (*vn*, *val*) *table*  
*arr-a* :: (*vn*, *val*) *table*  
*globs1* :: *globs*  
*globs2* :: *globs*  
*globs3* :: *globs*  
*globs8* :: *globs*  
*locs3* :: *locals*  
*locs4* :: *locals*  
*locs8* :: *locals*  
*s0* :: *state*  
*s0'* :: *state*  
*s9'* :: *state*  
*s1* :: *state*  
*s1'* :: *state*  
*s2* :: *state*  
*s2'* :: *state*  
*s3* :: *state*  
*s3'* :: *state*  
*s4* :: *state*  
*s4'* :: *state*  
*s6'* :: *state*  
*s7'* :: *state*  
*s8* :: *state*  
*s8'* :: *state*

**translations**

*obj-a* <= (*tag*=*Arr (PrimT Boolean) (CONST two)*  
*,values=CONST empty(Inr 0 $\mapsto$ Bool False)(Inr (CONST one) $\mapsto$ Bool False)*)  
*obj-b* <= (*tag*=*CInst (CONST Ext)*  
*,values=(CONST empty(Inl (CONST vee, CONST Base) $\mapsto$ Null* )  
*(Inl (CONST vee, CONST Ext) $\mapsto$ Intg 0))*)  
*obj-c* == (*tag*=*CInst (SXcpt NullPointer),values=CONST empty*)  
*arr-N* == *CONST empty(Inl (CONST arr, CONST Base) $\mapsto$ Null)*  
*arr-a* == *CONST empty(Inl (CONST arr, CONST Base) $\mapsto$ Addr a)*  
*globs1* == *CONST empty(Inr (CONST Ext)  $\mapsto$ (*tag*=*arbitrary, values=CONST empty*)*)  
*(Inr (CONST Base)  $\mapsto$ (*tag*=*arbitrary, values=arr-N*))*  
*(Inr Object $\mapsto$ (*tag*=*arbitrary, values=CONST empty*))*  
*globs2* == *CONST empty(Inr (CONST Ext)  $\mapsto$ (*tag*=*arbitrary, values=CONST empty*)*)  
*(Inr Object $\mapsto$ (*tag*=*arbitrary, values=CONST empty*))*  
*(Inl a $\mapsto$ obj-a)*  
*(Inr (CONST Base)  $\mapsto$ (*tag*=*arbitrary, values=arr-a*))*  
*globs3* == *globs2(Inl b $\mapsto$ obj-b)*

```

globs8 == globs3(Inl c↦obj-c)
locs3  == CONST empty(VName (CONST e)↦Addr b)
locs4  == CONST empty(VName (CONST z)↦Null)(Inr())↦Addr b)
locs8  == locs3(VName (CONST z)↦Addr c)
s0     == st (CONST empty) (CONST empty)
s0'    == Norm s0
s1     == st globs1 (CONST empty)
s1'    == Norm s1
s2     == st globs2 (CONST empty)
s2'    == Norm s2
s3     == st globs3 locs3
s3'    == Norm s3
s4     == st globs3 locs4
s4'    == Norm s4
s6'    == (Some (Xcpt (Std NullPointer)), s4)
s7'    == (Some (Xcpt (Std NullPointer)), s3)
s8     == st globs8 locs8
s8'    == Norm s8
s9'    == (Some (Xcpt (Std IndOutBound)), s8)

```

**declare** *Pair-eq* [*simp del*]

**lemma** *exec-test*:

```

[[the (new-Addr (heap s1)) = a;
 the (new-Addr (heap ?s2)) = b;
 the (new-Addr (heap ?s3)) = c]] ==>
atleast-free (heap s0) four ==>
tprg⊢s0' -test [Class Base]→ ?s9'
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 ?h) = c in thin-rl)
apply (erule-tac [2] V = new-Addr ?h = Some a in thin-rl)
apply (erule-tac [2] V = atleast-free ?h 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 ?h) = b in thin-rl)
apply (erule-tac V = atleast-free ?h tree in thin-rl)
apply (erule-tac [2] V = atleast-free ?h four in thin-rl)
apply (erule-tac [2] V = new-Addr ?h = 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 (rule conjI, rule HOL.refl)+
apply (rule HOL.refl)
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 ?h tree 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: split-if)
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: split-if)
apply (drule alloc-one [THEN conjunct1])
apply (simp (no-asm-simp))
apply (erule-tac V = atleast-free ?h 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 Pair-eq [simp]

end

```



## Chapter 17

# Conform

#### 44 Conformance notions for the type soundness proof for Java

**theory** *Conform* **imports** *State* **begin**

design issues:

- lconf allows for (arbitrary) inaccessible values
- "conforms" does not directly imply that the dynamic types of all objects on the heap are indeed existing classes. Yet this can be inferred for all referenced objs.

**types**  $env' = prog \times (lname, ty) \text{ table}$

**extension of global store**

**constdefs**

$gext \quad :: st \Rightarrow st \Rightarrow bool \quad (-\leq|- \quad [71,71] \quad 70)$   
 $s \leq |s' \equiv \forall r. \forall obj \in globs \ s \ r: \exists obj' \in globs \ s' \ r: tag \ obj' = tag \ obj$

For the the proof of type soundness we will need the property that during execution, objects are not lost and moreover retain the values of their tags. So the object store grows conservatively. Note that if we considered garbage collection, we would have to restrict this property to accessible objects.

**lemma** *gext-objD*:

$\llbracket s \leq |s'; globs \ s \ r = Some \ obj \rrbracket$   
 $\implies \exists obj'. globs \ s' \ r = Some \ obj' \wedge tag \ obj' = tag \ obj$   
**apply** (*simp only: gext-def*)  
**by** *force*

**lemma** *rev-gext-objD*:

$\llbracket globs \ s \ r = Some \ obj; s \leq |s' \rrbracket$   
 $\implies \exists obj'. globs \ s' \ r = Some \ obj' \wedge tag \ obj' = tag \ obj$   
**by** (*auto elim: gext-objD*)

**lemma** *init-class-obj-inited*:

$init-class-obj \ G \ C \ s1 \leq |s2 \implies init-ed \ C \ (globs \ s2)$   
**apply** (*unfold init-ed-def init-obj-def*)  
**apply** (*auto dest!: gext-objD*)  
**done**

**lemma** *gext-refl* [*intro!*, *simp*]:  $s \leq |s$

**apply** (*unfold gext-def*)  
**apply** (*fast del: fst-splitE*)  
**done**

**lemma** *gext-gupd* [*simp*, *elim!*]:  $\bigwedge s. globs \ s \ r = None \implies s \leq |gupd(r \mapsto x)s$   
**by** (*auto simp: gext-def*)

**lemma** *gext-new* [*simp*, *elim!*]:  $\bigwedge s. globs \ s \ r = None \implies s \leq |init-obj \ G \ oi \ r \ s$

**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** *inited-gext*:  $\llbracket \text{inited } C \ (\text{globs } s); s \leq | s' \rrbracket \implies \text{inited } C \ (\text{globs } s')$   
**apply** (*unfold inited-def*)  
**apply** (*auto dest: gext-objD*)  
**done**

## value conformance

### constdefs

*conf* :: *prog*  $\Rightarrow$  *st*  $\Rightarrow$  *val*  $\Rightarrow$  *ty*  $\Rightarrow$  *bool*     $(-, \vdash :: \preceq - \quad [71, 71, 71, 71] \ 70)$   
 $G, s \vdash v :: \preceq T \equiv \exists T' \in \text{typeof} \ (\lambda a. \text{option-map obj-ty} \ (\text{heap } s \ a)) \ v : G \vdash T' \preceq T$

**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*:  
 $G, s \vdash \text{Addr } a :: \preceq T = (\exists \text{obj}. \text{heap } s \ a = \text{Some obj} \wedge G \vdash \text{obj-ty obj} \preceq T)$   
**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*]:  
 $G, s \vdash v :: \preceq T \longrightarrow s \leq |s' \longrightarrow G, s \upharpoonright v \vdash v :: \preceq T$   
**apply** (*unfold gext-def conf-def*)  
**apply** (*rule val.induct*)  
**apply** *force+*  
**done**

**lemma** *conf-list-widen* [*rule-format (no-asm)*]:  
 $\text{ws-prog } G \Longrightarrow$   
 $\forall Ts \ Ts'. \text{list-all2 } (\text{conf } G \ s) \ vs \ Ts$   
 $\longrightarrow G \vdash Ts[\preceq] \ Ts' \longrightarrow \text{list-all2 } (\text{conf } G \ s) \ vs \ Ts'$   
**apply** (*unfold widens-def*)

**apply** (*rule list-all2-trans*)  
**apply** *auto*  
**done**

**lemma** *conf-RefTD* [*rule-format* (*no-asm*)]:  
 $G, s \vdash a' :: \preceq_{RefT} T$   
 $\longrightarrow a' = \text{Null} \vee (\exists a \text{ obj } T'. a' = \text{Addr } a \wedge \text{heap } s \ a = \text{Some obj} \wedge$   
 $\text{obj-ty obj} = T' \wedge G \vdash T' \preceq_{RefT} T)$   
**apply** (*unfold conf-def*)  
**apply** (*induct-tac a'*)  
**apply** (*auto dest: widen-PrimT*)  
**done**

## value list conformance

### constdefs

$lconf :: \text{prog} \Rightarrow \text{st} \Rightarrow ('a, \text{val}) \text{ table} \Rightarrow ('a, \text{ty}) \text{ table} \Rightarrow \text{bool}$   
 $(-, \vdash - :: \preceq) - [71, 71, 71, 71] \ 70)$   
 $G, s \vdash vs :: \preceq Ts \equiv \forall n. \forall T \in Ts \ n: \exists v \in vs \ n: G, s \vdash v :: \preceq T$

**lemma** *lconfD*:  $\llbracket G, s \vdash vs :: \preceq Ts; Ts \ n = \text{Some } T \rrbracket \Longrightarrow G, s \vdash (\text{the } (vs \ n)) :: \preceq T$   
**by** (*force simp: lconf-def*)

**lemma** *lconf-cong* [*simp*]:  $\bigwedge s. G, \text{set-locals } x \ s \vdash l :: \preceq L = G, s \vdash l :: \preceq L$   
**by** (*auto simp: lconf-def*)

**lemma** *lconf-lupd* [*simp*]:  $G, \text{lupd}(vn \mapsto v) s \vdash l :: \preceq L = G, s \vdash l :: \preceq L$   
**by** (*auto simp: lconf-def*)

**lemma** *lconf-new*:  $\llbracket L \ vn = \text{None}; G, s \vdash l :: \preceq L \rrbracket \Longrightarrow G, s \vdash l(vn \mapsto v) :: \preceq L$   
**by** (*auto simp: lconf-def*)

**lemma** *lconf-upd*:  $\llbracket G, s \vdash l :: \preceq L; G, s \vdash v :: \preceq T; L \ vn = \text{Some } T \rrbracket \Longrightarrow$   
 $G, s \vdash l(vn \mapsto v) :: \preceq L$   
**by** (*auto simp: lconf-def*)

**lemma** *lconf-ext*:  $\llbracket G, s \vdash l :: \preceq L; G, s \vdash v :: \preceq T \rrbracket \Longrightarrow G, s \vdash l(vn \mapsto v) :: \preceq L(vn \mapsto T)$   
**by** (*auto simp: lconf-def*)

**lemma** *lconf-map-sum* [*simp*]:  
 $G, s \vdash l1 \ (+) \ l2 :: \preceq L1 \ (+) \ L2 = (G, s \vdash l1 :: \preceq L1 \wedge G, s \vdash l2 :: \preceq L2)$   
**apply** (*unfold lconf-def*)  
**apply** *safe*  
**apply** (*case-tac [3] n*)  
**apply** (*force split add: sum.split*)  
**done**

```

lemma lconf-ext-list [rule-format (no-asm)]:
   $\bigwedge X. \llbracket G, s \vdash l[::\preceq] L \rrbracket \implies$ 
     $\forall vs\ Ts. \text{distinct } vns \longrightarrow \text{length } Ts = \text{length } vns$ 
     $\longrightarrow \text{list-all2 } (\text{conf } G\ s)\ vs\ Ts \longrightarrow G, s \vdash l(vns[\mapsto] vs)[::\preceq] L(vns[\mapsto] Ts)$ 
apply (unfold lconf-def)
apply (induct-tac vns)
apply clarsimp
apply clarify
apply (frule list-all2-lengthD)
apply (clarsimp)
done

```

```

lemma lconf-deallocL:  $\llbracket G, s \vdash l[::\preceq] L(vn \mapsto T); L\ vn = \text{None} \rrbracket \implies G, s \vdash l[::\preceq] L$ 
apply (simp only: lconf-def)
apply safe
apply (drule spec)
apply (drule ospec)
apply auto
done

```

```

lemma lconf-geext [elim]:  $\llbracket G, s \vdash l[::\preceq] L; s \leq |s' \rrbracket \implies G, s' \vdash l[::\preceq] L$ 
apply (simp only: lconf-def)
apply fast
done

```

```

lemma lconf-empty [simp, intro!]:  $G, s \vdash vs[::\preceq] \text{empty}$ 
apply (unfold lconf-def)
apply force
done

```

```

lemma lconf-init-vals [intro!]:
   $\forall n. \forall T \in fs\ n:\text{is-type } G\ T \implies G, s \vdash \text{init-vals } fs[::\preceq] fs$ 
apply (unfold lconf-def)
apply force
done

```

### weak value list conformance

Only if the value is defined it has to conform to its type. This is the contribution of the definite assignment analysis to the notion of conformance. The definite assignment analysis ensures that the program only attempts to access local variables that actually have a defined value in the state. So conformance must only ensure that the defined values are of the right type, and not also that the value is defined.

### constdefs

$$\begin{aligned}
 \text{wlconf} :: \text{prog} \Rightarrow \text{st} \Rightarrow ('a, \text{val}) \text{ table} \Rightarrow ('a, \text{ty}) \text{ table} \Rightarrow \text{bool} \\
 (\neg, \vdash, [\sim::\preceq]) - [71, 71, 71, 71] 70) \\
 G, s \vdash vs[\sim::\preceq] Ts \equiv \forall n. \forall T \in Ts\ n: \forall v \in vs\ n: G, s \vdash v::\preceq T
 \end{aligned}$$

```

lemma wlconfD:  $\llbracket G, s \vdash vs[\sim::\preceq] Ts; Ts\ n = \text{Some } T; vs\ n = \text{Some } v \rrbracket \implies G, s \vdash v::\preceq T$ 
by (auto simp: wlconf-def)

```

**lemma** *wlconf-cong* [simp]:  $\bigwedge s. G, \text{set-locals } x \vdash l[\sim::\preceq] L = G, s \vdash l[\sim::\preceq] L$   
**by** (auto simp: wlconf-def)

**lemma** *wlconf-lupd* [simp]:  $G, \text{lupd}(vn \mapsto v) \vdash l[\sim::\preceq] L = G, s \vdash l[\sim::\preceq] L$   
**by** (auto simp: wlconf-def)

**lemma** *wlconf-upd*:  $\llbracket G, s \vdash l[\sim::\preceq] L; G, s \vdash v::\preceq T; L \text{ vn} = \text{Some } T \rrbracket \implies$   
 $G, s \vdash l(vn \mapsto v)[\sim::\preceq] L$   
**by** (auto simp: wlconf-def)

**lemma** *wlconf-ext*:  $\llbracket G, s \vdash l[\sim::\preceq] L; G, s \vdash v::\preceq T \rrbracket \implies G, s \vdash l(vn \mapsto v)[\sim::\preceq] L(vn \mapsto T)$   
**by** (auto simp: wlconf-def)

**lemma** *wlconf-map-sum* [simp]:  
 $G, s \vdash l1 (+) l2[\sim::\preceq] L1 (+) L2 = (G, s \vdash l1[\sim::\preceq] L1 \wedge G, s \vdash l2[\sim::\preceq] L2)$   
**apply** (unfold wlconf-def)  
**apply** safe  
**apply** (case-tac [3] n)  
**apply** (force split add: sum.split)+  
**done**

**lemma** *wlconf-ext-list* [rule-format (no-asm)]:  
 $\bigwedge X. \llbracket G, s \vdash l[\sim::\preceq] L \rrbracket \implies$   
 $\forall vs \text{ Ts. distinct vns} \longrightarrow \text{length Ts} = \text{length vns}$   
 $\longrightarrow \text{list-all2 (conf } G \text{ s) vs Ts} \longrightarrow G, s \vdash l(\text{vns}[\mapsto] \text{vs})[\sim::\preceq] L(\text{vns}[\mapsto] \text{Ts})$   
**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-gext* [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!*]:  $G, s \vdash vs[\sim::\preceq] \text{empty}$   
**apply** (*unfold wlconf-def*)  
**apply** *force*  
**done**

**lemma** *wlconf-empty-vals*:  $G, s \vdash \text{empty}[\sim::\preceq] ts$   
**by** (*simp add: wlconf-def*)

**lemma** *wlconf-init-vals* [*intro!*]:  
 $\forall n. \forall T \in fs \ n:is\text{-type} \ G \ T \implies G, s \vdash \text{init-vals} \ fs[\sim::\preceq] fs$   
**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

### constdefs

*oconf* :: *prog*  $\Rightarrow$  *st*  $\Rightarrow$  *obj*  $\Rightarrow$  *oref*  $\Rightarrow$  *bool* ( $\sim, \vdash, \preceq, \sqrt{\phantom{x}}$  [71,71,71,71] 70)  
 $G, s \vdash \text{obj}::\preceq\sqrt{r} \equiv G, s \vdash \text{values} \ \text{obj}[\sim::\preceq] \text{var-tys} \ G \ (\text{tag} \ \text{obj}) \ r \wedge$   
     (*case* *r* *of*  
         *Heap* *a*  $\Rightarrow$  *is-type* *G* (*obj-ty* *obj*)  
         | *Stat* *C*  $\Rightarrow$  *True*)

**lemma** *oconf-is-type*:  $G, s \vdash \text{obj}::\preceq\sqrt{\text{Heap } a} \implies \text{is-type } G \ (\text{obj-ty } \text{obj})$   
**by** (*auto simp: oconf-def Let-def*)

**lemma** *oconf-lconf*:  $G, s \vdash \text{obj}::\preceq\sqrt{r} \implies G, s \vdash \text{values} \ \text{obj}[\sim::\preceq] \text{var-tys} \ G \ (\text{tag} \ \text{obj}) \ r$   
**by** (*simp add: oconf-def*)

**lemma** *oconf-cong* [*simp*]:  $G, \text{set-locals } l \ s \vdash \text{obj}::\preceq\sqrt{r} = G, s \vdash \text{obj}::\preceq\sqrt{r}$   
**by** (*auto simp: oconf-def Let-def*)

**lemma** *oconf-init-obj-lemma*:  
 $\llbracket \bigwedge C \ c. \text{class } G \ C = \text{Some } c \implies \text{unique} \ (\text{DeclConcepts.fields } G \ C);$   
 $\bigwedge C \ c \ f \text{fld}. \llbracket \text{class } G \ C = \text{Some } c;$   
 $\text{table-of } (\text{DeclConcepts.fields } G \ C) \ f = \text{Some fld} \rrbracket$   
 $\implies \text{is-type } G \ (\text{type fld});$   
     (*case* *r* *of*  
         *Heap* *a*  $\Rightarrow$  *is-type* *G* (*obj-ty* *obj*)  
         | *Stat* *C*  $\Rightarrow$  *is-class* *G* *C*)  
 $\rrbracket \implies G, s \vdash \text{obj} \ (\llbracket \text{values} := \text{init-vals} \ (\text{var-tys } G \ (\text{tag} \ \text{obj}) \ r) \rrbracket)::\preceq\sqrt{r}$   
**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 obj-ty-CInst1 obj-ty-Arr1
      split add: sum.split-asm obj-tag.split-asm)
done

```

## state conformance

### constdefs

```

conforms :: state  $\Rightarrow$  env'  $\Rightarrow$  bool      (  -:: $\preceq$ -  [71,71]      70)
xs:: $\preceq$ E  $\equiv$  let (G, L) = E; s = snd xs; l = locals s in
  ( $\forall r$ .  $\forall \text{obj} \in \text{globs } s$  r:
     $G, s \vdash \text{obj} :: \preceq \sqrt{r}$ )  $\wedge$ 
     $G, s \vdash l \text{ } [\sim :: \preceq] L$   $\wedge$ 
    ( $\forall a$ . fst xs = Some(Xcpt (Loc a))  $\longrightarrow$   $G, s \vdash \text{Addr } a :: \preceq \text{Class } (\text{SXcpt Throwable})$ )  $\wedge$ 
    (fst xs = Some(Jump Ret)  $\longrightarrow$  l Result  $\neq$  None)

```

### conforms

**lemma** *conforms-globsD*:

$\llbracket (x, s) :: \preceq (G, L); \text{globs } s \text{ } r = \text{Some obj} \rrbracket \Longrightarrow G, s \vdash \text{obj} :: \preceq \sqrt{r}$   
**by** (*auto simp: conforms-def Let-def*)

**lemma** *conforms-localD*:  $\llbracket (x, s) :: \preceq (G, L) \rrbracket \Longrightarrow G, s \vdash \text{locals } s \text{ } [\sim :: \preceq] L$   
**by** (*auto simp: conforms-def Let-def*)

**lemma** *conforms-XcptLocD*:  $\llbracket (x, s) :: \preceq (G, L); x = \text{Some } (\text{Xcpt } (\text{Loc } a)) \rrbracket \Longrightarrow$   
 $G, s \vdash \text{Addr } a :: \preceq \text{Class } (\text{SXcpt Throwable})$   
**by** (*auto simp: conforms-def Let-def*)

**lemma** *conforms-RetD*:  $\llbracket (x, s) :: \preceq (G, L); x = \text{Some } (\text{Jump Ret}) \rrbracket \Longrightarrow$   
 $(\text{locals } s) \text{ Result} \neq \text{None}$   
**by** (*auto simp: conforms-def Let-def*)

**lemma** *conforms-RefTD*:

$\llbracket G, s \vdash a' :: \preceq \text{RefT } t; a' \neq \text{Null}; (x, s) :: \preceq (G, L) \rrbracket \Longrightarrow$   
 $\exists a \text{ obj. } a' = \text{Addr } a \wedge \text{globs } s \text{ } (\text{Inl } a) = \text{Some obj} \wedge$   
 $G \vdash \text{obj-ty obj} \preceq \text{RefT } t \wedge \text{is-type } G \text{ } (\text{obj-ty obj})$

**apply** (*drule-tac conf-RefTD*)

**apply** *clarsimp*

**apply** (*rule conforms-globsD [THEN oconf-is-type]*)

**apply** *auto*

**done**

**lemma** *conforms-Jump [iff]*:

$j = \text{Ret} \longrightarrow \text{locals } s \text{ Result} \neq \text{None}$   
 $\Longrightarrow ((\text{Some } (\text{Jump } j), s) :: \preceq (G, L)) = (\text{Norm } s :: \preceq (G, L))$   
**by** (*auto simp: conforms-def Let-def*)

**lemma** *conforms-StdXcpt [iff]*:

$((\text{Some } (\text{Xcpt } (\text{Std } xn)), s) :: \preceq (G, L)) = (\text{Norm } s :: \preceq (G, L))$   
**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]:  
 $((\text{raise-if } c \text{ } \text{xn } x, s) :: \preceq (G, L)) = ((x, s) :: \preceq (G, L))$   
**by** (auto simp: abrupt-if-def)

**lemma** *conforms-error-if* [iff]:  
 $((\text{error-if } c \text{ } \text{err } x, s) :: \preceq (G, L)) = ((x, s) :: \preceq (G, L))$   
**by** (auto simp: abrupt-if-def split: split-if)

**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]:  
 $(a, b) :: \preceq (G, L) \longrightarrow (\text{absorb } j \ a, b) :: \preceq (G, L)$   
**apply** (rule impI)  
**apply** ( case-tac a)  
**apply** (case-tac absorb j a)  
**apply** auto  
**apply** (case-tac absorb j (Some a), auto)  
**apply** (erule conforms-NormI)  
**done**

**lemma** *conformsI*:  $\llbracket \forall r. \forall \text{obj} \in \text{globs } s \ r: G, s \vdash \text{obj} :: \preceq \sqrt{r};$   
 $G, s \vdash \text{locals } s [\sim :: \preceq] L;$   
 $\forall a. x = \text{Some } (\text{Xcpt } (\text{Loc } a)) \longrightarrow G, s \vdash \text{Addr } a :: \preceq \text{Class } (\text{SXcpt } \text{Throwable});$   
 $x = \text{Some } (\text{Jump } \text{Ret}) \longrightarrow \text{locals } s \ \text{Result} \neq \text{None} \rrbracket \implies$   
 $(x, s) :: \preceq (G, L)$   
**by** (auto simp: conforms-def Let-def)

**lemma** *conforms-xconf*:  $\llbracket (x, s) :: \preceq (G, L);$   
 $\forall a. x' = \text{Some } (\text{Xcpt } (\text{Loc } a)) \longrightarrow G, s \vdash \text{Addr } a :: \preceq \text{Class } (\text{SXcpt } \text{Throwable});$   
 $x' = \text{Some } (\text{Jump } \text{Ret}) \longrightarrow \text{locals } s \ \text{Result} \neq \text{None} \rrbracket \implies$   
 $(x', s) :: \preceq (G, L)$   
**by** (fast intro: conformsI elim: conforms-globsD conforms-localD)

**lemma** *conforms-lupd*:  
 $\llbracket (x, s) :: \preceq (G, L); L \text{ } \text{vn} = \text{Some } T; G, s \vdash v :: \preceq T \rrbracket \implies (x, \text{lupd}(v \mapsto v) s) :: \preceq (G, L)$   
**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*:  
 $\llbracket (x, s) :: \preceq (G, L); G, s \vdash v :: \preceq T \rrbracket \implies (x, \text{lupd}(v \mapsto v) s) :: \preceq (G, L(v \mapsto T))$   
**by** (force intro: conformsI dest: conforms-globsD conforms-RetD)



*elim*: conforms-XcptLocD conforms-allocL-aux  
*simp*: oconf-def)

**lemmas** conforms-deallocL-aux = conforms-localD [THEN wlconf-deallocL]

**lemma** conforms-deallocL:  $\bigwedge s. [s :: \preceq (G, L(vn \mapsto T)); L\ vn = None] \implies s :: \preceq (G, L)$   
**by** (fast intro: conformsI dest: conforms-globsD conforms-RetD  
elim: conforms-XcptLocD conforms-deallocL-aux)

**lemma** conforms-gext:  $\llbracket (x, s) :: \preceq (G, L); s \leq |s' ;$   
 $\forall r. \forall obj \in \text{globs } s' \ r: G, s \vdash obj :: \preceq \sqrt{r};$   
 $\text{locals } s' = \text{locals } s \rrbracket \implies (x, s') :: \preceq (G, L)$   
**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:  $\bigwedge obj. \llbracket (x, s) :: \preceq (G, L); G, s \vdash obj :: \preceq \sqrt{r}; s \leq |gupd(r \mapsto obj) s \rrbracket$   
 $\implies (x, gupd(r \mapsto obj) s) :: \preceq (G, L)$   
**apply** (rule conforms-gext)  
**apply** auto  
**apply** (force dest: conforms-globsD simp add: oconf-def)+  
**done**

**lemma** conforms-upd-gobj:  $\llbracket (x, s) :: \preceq (G, L); \text{globs } s \ r = \text{Some } obj;$   
 $\text{var-ty } G \ (\text{tag } obj) \ r \ n = \text{Some } T; G, s \vdash v :: \preceq T \rrbracket \implies (x, \text{upd-gobj } r \ n \ v \ s) :: \preceq (G, L)$   
**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: sum.weak-case-cong)  
**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*:

$\wedge s'. \llbracket (x,s)::\preceq(G, L); (x',s')::\preceq(G, L'); s \leq |s'; x' \neq \text{Some } (\text{Jump Ret}) \rrbracket \implies$   
 $(x', \text{set-locals } (\text{locals } s) \ s')::\preceq(G, L)$

**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

## 45 Correctness of Definite Assignment

**theory** *DefiniteAssignmentCorrect* **imports** *WellForm Eval begin*

**declare**  $[[simproc\ del:\ wt\text{-}expr\ wt\text{-}var\ wt\text{-}exprs\ wt\text{-}stmt]]$

**lemma** *sxalloc-no-jump*:

**assumes** *sxalloc*:  $G \vdash s0 \text{ --sxalloc--> } s1$  **and**  
*no-jmp*:  $abrupt\ s0 \neq Some\ (Jump\ j)$   
**shows**  $abrupt\ s1 \neq Some\ (Jump\ j)$

**using** *sxalloc no-jmp*

**by** *cases simp-all*

**lemma** *sxalloc-no-jump'*:

**assumes** *sxalloc*:  $G \vdash s0 \text{ --sxalloc--> } s1$  **and**  
*jump*:  $abrupt\ s1 = Some\ (Jump\ j)$   
**shows**  $abrupt\ s0 = Some\ (Jump\ j)$

**using** *sxalloc jump*

**by** *cases simp-all*

**lemma** *halloc-no-jump*:

**assumes** *halloc*:  $G \vdash s0 \text{ --halloc } oi \succ a \rightarrow s1$  **and**  
*no-jmp*:  $abrupt\ s0 \neq Some\ (Jump\ j)$   
**shows**  $abrupt\ s1 \neq Some\ (Jump\ j)$

**using** *halloc no-jmp*

**by** *cases simp-all*

**lemma** *halloc-no-jump'*:

**assumes** *halloc*:  $G \vdash s0 \text{ --halloc } oi \succ a \rightarrow s1$  **and**  
*jump*:  $abrupt\ s1 = Some\ (Jump\ j)$   
**shows**  $abrupt\ s0 = Some\ (Jump\ j)$

**using** *halloc jump*

**by** *cases simp-all*

**lemma** *Body-no-jump*:

**assumes** *eval*:  $G \vdash s0 \text{ --Body } D\ c \text{ --> } v \rightarrow s1$  **and**  
*jump*:  $abrupt\ s0 \neq Some\ (Jump\ j)$   
**shows**  $abrupt\ s1 \neq Some\ (Jump\ j)$

**proof** (*cases normal s0*)

**case** *True*

**with** *eval* **obtain**  $s0'$  **where** *eval'*:  $G \vdash Norm\ s0' \text{ --Body } D\ c \text{ --> } v \rightarrow s1$  **and**  
 $s0: s0 = Norm\ s0'$

**by** (*cases s0*) *simp*

**from** *eval'* **obtain**  $s2$  **where**

$s1: s1 = abupd\ (absorb\ Ret)$

(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  $s2$ )

**by** *cases simp*

**show** *?thesis*

**proof** (*cases*  $\exists l. abrupt\ s2 = Some\ (Jump\ (Break\ l)) \vee$

$abrupt\ s2 = Some\ (Jump\ (Cont\ l))$ )

**case** *True*

**with**  $s1$  **have**  $abrupt\ s1 = Some\ (Error\ CrossMethodJump)$

```

    by (cases s2) simp
  thus ?thesis by simp
next
  case False
  with s1 have s1=abupd (absorb Ret) s2
    by simp
  with False show ?thesis
    by (cases s2,cases j) (auto simp add: absorb-def)
qed
next
  case False
  with eval obtain s0' abr where  $G \vdash (\text{Some } \text{abr}, s0') - \text{Body } D \text{ } c \multimap v \rightarrow s1$ 
     $s0 = (\text{Some } \text{abr}, s0')$ 
    by (cases s0) fastsimp
  with this jump
  show ?thesis
    by (cases) (simp)
qed

```

**lemma** *Method-no-jump*:

```

  assumes eval:  $G \vdash s0 - \text{Method } D \text{ } sig \multimap v \rightarrow s1$  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  $G \vdash \text{Norm } s0' - \text{Method } D \text{ } sig \multimap v \rightarrow s1$ 
     $s0 = \text{Norm } s0'$ 
    by (cases s0) simp
  then obtain D' body where  $G \vdash s0 - \text{Body } D' \text{ } body \multimap v \rightarrow s1$ 
    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  $G \vdash (\text{Some } \text{abr}, s0') - \text{Method } D \text{ } sig \multimap v \rightarrow s1$ 
     $s0 = (\text{Some } \text{abr}, s0')$ 
    by (cases s0) fastsimp
  with this jump
  show ?thesis
    by (cases) (simp)
qed

```

**lemma** *jumpNestingOkS-mono*:

```

  assumes jumpNestingOk-l':  $\text{jumpNestingOkS } j\text{mps}' \text{ } c$ 
    and subset:  $j\text{mps}' \subseteq j\text{mps}$ 
  shows  $\text{jumpNestingOkS } j\text{mps } c$ 
proof -
  have True and True and
     $\bigwedge j\text{mps}' j\text{mps}. [\![\text{jumpNestingOkS } j\text{mps}' \text{ } c; j\text{mps}' \subseteq j\text{mps}]\!] \implies \text{jumpNestingOkS } j\text{mps } c$ 
    and True
  proof (induct rule: var-expr-stmt.inducts)
    case (Lab j c jmps' jmps)
    note jmpOk =  $\langle \text{jumpNestingOkS } j\text{mps}' (j \cdot c) \rangle$ 
    note jmps =  $\langle j\text{mps}' \subseteq j\text{mps} \rangle$ 
    with jmpOk have  $\text{jumpNestingOkS } (\{j\} \cup j\text{mps}') \text{ } c$  by simp
    moreover from jmps have  $(\{j\} \cup j\text{mps}') \subseteq (\{j\} \cup j\text{mps})$  by auto
  qed

```

```

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  $c1$   $c2$   $jmps'$   $jmps$ )
from Comp.prem
have jumpNestingOkS  $jmps$   $c1$  by - (rule Comp.hyps, auto)
moreover from Comp.prem
have jumpNestingOkS  $jmps$   $c2$  by - (rule Comp.hyps, auto)
ultimately show ?case
  by simp
next
case (If'  $e$   $c1$   $c2$   $jmps'$   $jmps$ )
from If'.prem
have jumpNestingOkS  $jmps$   $c1$  by - (rule If'.hyps, auto)
moreover from If'.prem
have jumpNestingOkS  $jmps$   $c2$  by - (rule If'.hyps, auto)
ultimately show ?case
  by simp
next
case (Loop  $l$   $e$   $c$   $jmps'$   $jmps$ )
from jumpNestingOkS  $jmps'$  ( $l \bullet \text{While}(e) c$ )
have jumpNestingOkS ( $\{\text{Cont } l\} \cup jmps'$ )  $c$  by simp
moreover
from  $jmps' \subseteq jmps$ 
have  $\{\text{Cont } l\} \cup jmps' \subseteq \{\text{Cont } l\} \cup jmps$  by auto
ultimately
have jumpNestingOkS ( $\{\text{Cont } l\} \cup jmps$ )  $c$ 
  by (rule Loop.hyps)
thus ?case by simp
next
case (TryC  $c1$   $C$   $vn$   $c2$   $jmps'$   $jmps$ )
from TryC.prem
have jumpNestingOkS  $jmps$   $c1$  by - (rule TryC.hyps, auto)
moreover from TryC.prem
have jumpNestingOkS  $jmps$   $c2$  by - (rule TryC.hyps, auto)
ultimately show ?case
  by simp
next
case (Fin  $c1$   $c2$   $jmps'$   $jmps$ )
from Fin.prem
have jumpNestingOkS  $jmps$   $c1$  by - (rule Fin.hyps, auto)
moreover from Fin.prem
have jumpNestingOkS  $jmps$   $c2$  by - (rule Fin.hyps, auto)
ultimately show ?case
  by simp
qed (simp-all)
with jumpNestingOk-l' subset
show ?thesis
  by iprover
qed

```

corollary *jumpNestingOk-mono*:

```

assumes jmpOk: jumpNestingOk jmps' t
and subset: jmps' ⊆ jmps
shows jumpNestingOk jmps t
proof (cases t)
  case (In1 expr-stmt)
    show ?thesis
    proof (cases expr-stmt)
      case (Inl e)
        with In1 show ?thesis by simp
    next
      case (Inr s)
        with In1 jmpOk subset show ?thesis by (auto intro: jumpNestingOkS-mono)
    qed
qed (simp-all)

```

```

lemma assign-abrupt-propagation:
assumes f-ok: abrupt (f n s) ≠ x
and ass: abrupt (assign f n s) = x
shows abrupt s = x
proof (cases x)
  case None
    with ass show ?thesis
    by (cases s) (simp add: assign-def Let-def)
  next
    case (Some xcpt)
    from f-ok
    obtain xf sf where f n s = (xf, sf)
    by (cases f n s)
    with Some ass f-ok show ?thesis
    by (cases s) (simp add: assign-def Let-def)
qed

```

```

lemma wt-init-comp-ty':
is-acc-type (prg Env) (pid (cls Env)) T ⇒ Env⊢init-comp-ty T::√
apply (unfold init-comp-ty-def)
apply (clarsimp simp add: accessible-in-RefT-simp
        is-acc-type-def is-acc-class-def)
done

```

```

lemma fvar-upd-no-jump:
assumes upd: upd = snd (fst (fvar statDeclC stat fn a s^))
and noJmp: abrupt s ≠ Some (Jump j)
shows abrupt (upd val s) ≠ Some (Jump j)
proof (cases stat)
  case True
    with noJmp upd
    show ?thesis
    by (cases s) (simp add: fvar-def2)
  next
    case False
    with noJmp upd
    show ?thesis
    by (cases s) (simp add: fvar-def2)
qed

```

**lemma** *avar-state-no-jump*:  
**assumes** *jmp*: *abrupt* (*snd* (*avar* *G* *i* *a* *s*)) = *Some* (*Jump* *j*)  
**shows** *abrupt* *s* = *Some* (*Jump* *j*)  
**proof** (*cases normal s*)  
**case** *True* **with** *jmp* **show** ?thesis **by** (*auto simp add: avar-def2 abrupt-if-def*)  
**next**  
**case** *False* **with** *jmp* **show** ?thesis **by** (*auto simp add: avar-def2 abrupt-if-def*)  
**qed**

**lemma** *avar-upd-no-jump*:  
**assumes** *upd*: *upd* = *snd* (*fst* (*avar* *G* *i* *a* *s'*))  
**and** *noJmp*: *abrupt* *s*  $\neq$  *Some* (*Jump* *j*)  
**shows** *abrupt* (*upd* *val* *s*)  $\neq$  *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**  $(\forall j. \text{fst } s1 = \text{Some } (\text{Jump } j) \longrightarrow j \in \text{jmps}) \wedge$   
 $(\text{normal } s1 \longrightarrow$   
 $(\forall w \text{ upd}. v = \text{In2 } (w, \text{upd})$   
 $\longrightarrow (\forall s \text{ j val}.$   
 $\text{abrupt } s \neq \text{Some } (\text{Jump } j) \longrightarrow$   
 $\text{abrupt } (\text{upd val } s) \neq \text{Some } (\text{Jump } j))))$   
**(is ?Jmp jmps s1  $\wedge$  ?Upd v s1)**  
**proof** —  
**let** ?HypObj =  $\lambda t \text{ s0 } s1 \text{ v}.$   
 $(\forall \text{ jmps } T \text{ Env}.$   
 $\text{?Jmp jmps } s0 \longrightarrow \text{jumpNestingOk } \text{jmps } t \longrightarrow \text{Env} \vdash t :: T \longrightarrow \text{prg } \text{Env} = G \longrightarrow$   
 $\text{?Jmp jmps } s1 \wedge \text{?Upd } v \text{ s1})$

— 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  $\wedge$   $\langle \text{jmps } T \text{ Env}. \llbracket ?\text{Jump jmps } s0; \text{jumpNestingOk jmps } t; \text{Env} \vdash t::T; \text{prg Env} = G \rrbracket$ 
 $\implies ?\text{Jump jmps } s1 \wedge ?\text{Upd } v \text{ } s1$ 
(is PROP ?Hyp  $t \text{ } s0 \text{ } s1 \text{ } v$ )

```

— We need to abstract over  $\text{jmps}$  since  $\text{jmps}$  are extended during analysis of  $\text{Lab}$ . Also we need to abstract over  $T$  and  $\text{Env}$  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  $s0 \text{ } c \text{ } s1 \text{ } \text{jmp jmps } T \text{ Env}$ )
  note jmpOK =  $\langle \text{jumpNestingOk jmps } (\text{In1r } (\text{jmp} \cdot c)) \rangle$ 
  note  $G = \langle \text{prg Env} = G \rangle$ 
  have  $\text{wt-}c: \text{Env} \vdash c::\sqrt{\phantom{x}}$ 
  using Lab.premis by (elim wt-elim-cases)
  {
    fix j
    assume  $\text{ab-}s1: \text{abrupt } (\text{abupd } (\text{absorb jmp}) s1) = \text{Some } (\text{Jump } j)$ 
    have  $j \in \text{jmps}$ 
    proof —
      from  $\text{ab-}s1$  have  $\text{jmp-}s1: \text{abrupt } s1 = \text{Some } (\text{Jump } j)$ 
      by (cases  $s1$ ) (simp add: absorb-def)
      note  $\text{hyp-}c = \langle \text{PROP ?Hyp } (\text{In1r } c) (\text{Norm } s0) s1 \diamond \rangle$ 
      from  $\text{ab-}s1$  have  $j \neq \text{jmp}$ 
      by (cases  $s1$ ) (simp add: absorb-def)
      moreover have  $j \in \{\text{jmp}\} \cup \text{jmps}$ 
      proof —
        from jmpOK
        have  $\text{jumpNestingOk } (\{\text{jmp}\} \cup \text{jmps}) (\text{In1r } c) \text{ by simp}$ 
        with  $\text{wt-}c \text{ jmp-}s1 \text{ } G \text{ hyp-}c$ 
        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  $s0 \text{ } c1 \text{ } s1 \text{ } c2 \text{ } s2 \text{ } \text{jmps } T \text{ Env}$ )
  note  $\text{jmpOk} = \langle \text{jumpNestingOk jmps } (\text{In1r } (c1;; c2)) \rangle$ 
  note  $G = \langle \text{prg Env} = G \rangle$ 
  from Comp.premis obtain
     $\text{wt-}c1: \text{Env} \vdash c1::\sqrt{\phantom{x}}$  and  $\text{wt-}c2: \text{Env} \vdash c2::\sqrt{\phantom{x}}$ 
  by (elim wt-elim-cases)
  {
    fix j
    assume  $\text{abr-}s2: \text{abrupt } s2 = \text{Some } (\text{Jump } j)$ 
    have  $j \in \text{jmps}$ 
    proof —
      have  $\text{jmp}: ?\text{Jump jmps } s1$ 
      proof —
        note  $\text{hyp-}c1 = \langle \text{PROP ?Hyp } (\text{In1r } c1) (\text{Norm } s0) s1 \diamond \rangle$ 
        with  $\text{wt-}c1 \text{ jmpOk } G$ 
        show ?thesis by simp
      qed
    qed
  }

```

```

    qed
    moreover note hyp-c2 = ⟨PROP ?Hyp (In1r c2) s1 s2 (◇::vals)⟩
    have jmpOk': jumpNestingOk jmps (In1r c2) using jmpOk by simp
    moreover note wt-c2 G abr-s2
    ultimately show j ∈ jmps
      by (rule hyp-c2 [THEN conjunct1,rule-format (no-asm)])
    qed
  } thus ?case by simp
next
case (If s0 e b s1 c1 c2 s2 jmps T Env)
note jmpOk = ⟨jumpNestingOk jmps (In1r (If(e) c1 Else c2))⟩
note G = ⟨prg Env = G⟩
from If.premis obtain
  wt-e: Env ⊢ e :: − PrimT Boolean and
  wt-then-else: Env ⊢ (if the-Bool b then c1 else c2) :: √
by (elim wt-elim-cases) simp
{
  fix j
  assume jmp: abrupt s2 = Some (Jump j)
  have j ∈ jmps
  proof −
    note ⟨PROP ?Hyp (In1l e) (Norm s0) s1 (In1 b)⟩
    with wt-e G have ?Jmp jmps s1
      by simp
    moreover note hyp-then-else =
      ⟨PROP ?Hyp (In1r (if the-Bool b then c1 else c2)) s1 s2 ◇⟩
    have jumpNestingOk jmps (In1r (if the-Bool b then c1 else c2))
      using jmpOk by (cases the-Bool b) simp-all
    moreover note wt-then-else G jmp
    ultimately show j ∈ jmps
      by (rule hyp-then-else [THEN conjunct1,rule-format (no-asm)])
    qed
  }
thus ?case by simp
next
case (Loop s0 e b s1 c s2 l s3 jmps T Env)
note jmpOk = ⟨jumpNestingOk jmps (In1r (l. While(e) c))⟩
note G = ⟨prg Env = G⟩
note wt = ⟨Env ⊢ In1r (l. While(e) c) :: T⟩
then obtain
  wt-e: Env ⊢ e :: − PrimT Boolean and
  wt-c: Env ⊢ c :: √
by (elim wt-elim-cases)
{
  fix j
  assume jmp: abrupt s3 = Some (Jump j)
  have j ∈ jmps
  proof −
    note ⟨PROP ?Hyp (In1l e) (Norm s0) s1 (In1 b)⟩
    with wt-e G have jmp-s1: ?Jmp jmps s1
      by simp
    show ?thesis
    proof (cases the-Bool b)
    case False
    from Loop.hyps
    have s3=s1
      by (simp (no-asm-use) only: if-False False)
    with jmp-s1 jmp have j ∈ jmps by simp
    thus ?thesis by simp
  }
}

```

```

next
  case True
  from Loop.hyps

  have ?HypObj (In1r c) s1 s2 ( $\Diamond::vals$ )
    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 ( $\{Cont\ l\} \cup jmps$ ) (In1r c)
    by simp
  moreover from jmp-s1 have ?Jmp ( $\{Cont\ l\} \cup jmps$ ) s1 by simp
  ultimately have jmp-s2: ?Jmp ( $\{Cont\ l\} \cup jmps$ ) s2
    using wt-c G by iprover
  have ?Jmp jmps (abupd (absorb (Cont l)) s2)
  proof -
    {
      fix j'
      assume abs: abrupt (abupd (absorb (Cont l)) s2)=Some (Jump j')
      have j'  $\in$  jmps
      proof (cases j' = Cont l)
        case True
        with abs show ?thesis
          by (cases s2) (simp add: absorb-def)
      next
        case False
        with abs have abrupt s2 = Some (Jump j')
          by (cases s2) (simp add: absorb-def)
        with jmp-s2 False show ?thesis
          by simp
      qed
    }
    thus ?thesis by simp
  qed
  moreover
  from Loop.hyps
  have ?HypObj (In1r (l. While(e) c))
    (abupd (absorb (Cont l)) s2) s3 ( $\Diamond::vals$ )
    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  $\in$  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 =  $\langle$ jumpNestingOk jmps (In1r (Throw e)) $\rangle$ 
  note G =  $\langle$ prg Env = G $\rangle$ 
  from Throw.premis obtain Te where
    wt-e: Env  $\vdash$  e :: - Te

```

```

  by (elim wt-elim-cases)
{
  fix j
  assume jmp: abrupt (abupd (throw a) s1) = Some (Jump j)
  have j∈jumps
  proof -
    from ⟨PROP ?Hyp (In1l e) (Norm s0) s1 (In1 a)⟩
    have ?Jump jumps 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 ∈ jumps by simp
  qed
}
thus ?case by simp
next
case (Try s0 c1 s1 s2 C vn c2 s3 jumps T Env)
note jmpOk = ⟨jumpNestingOk jumps (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)
{
  fix j
  assume jmp: abrupt s3 = Some (Jump j)
  have j∈jumps
  proof -
    note ⟨PROP ?Hyp (In1r c1) (Norm s0) s1 (◇::vals)⟩
    with jmpOk wt-c1 G
    have jmp-s1: ?Jump jumps s1 by simp
    note s2 = ⟨G⊢s1 -xalloc→ s2⟩
    show j ∈ jumps
    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 xalloc-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 xalloc-no-jump' [OF s2]
      have ?Jump jumps s2
      by simp
      hence ?Jump jumps (new-xcpt-var vn s2)
      by (cases s2) simp
      moreover have jumpNestingOk jumps (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)
{
  fix j
  assume jmp: abrupt s3 = Some (Jump j)
  have j ∈ jmps
  proof (cases x1 = Some (Jump j))
    case True
    note hyp-c1 = ⟨PROP ?Hyp (In1r c1) (Norm s0) (x1, s1) ◇⟩
    with True jmpOk wt-c1 G show ?thesis
    by - (rule hyp-c1 [THEN conjunct1, rule-format (no-asm)], simp-all)
  next
    case False
    note hyp-c2 = ⟨PROP ?Hyp (In1r c2) (Norm s1) s2 ◇⟩
    note ⟨s3 = (if ∃ err. x1 = Some (Error err) then (x1, s1)
      else abupd (abrupt-if (x1 ≠ None) x1) s2)⟩
    with False jmp have abrupt s2 = Some (Jump j)
    by (cases s2) (simp add: abrupt-if-def)
    with jmpOk wt-c2 G show ?thesis
    by - (rule hyp-c2 [THEN conjunct1, rule-format (no-asm)], simp-all)
  qed
}
thus ?case by simp
next
case (Init C c s0 s3 s1 s2 jmps T Env)
note ⟨jumpNestingOk jmps (In1r (Init C))⟩
note G = ⟨prg Env = G⟩
note ⟨the (class G C) = c⟩
with Init.prem have c: class G C = Some c
by (elim wt-elim-cases) auto
{
  fix j
  assume jmp: abrupt s3 = (Some (Jump j))
  have j ∈ jmps
  proof (cases inited C (globs s0))
    case True
    with Init.hyps have s3 = Norm s0
    by simp
    with jmp
    have False by simp thus ?thesis ..
  next
    case False
    from wf c G
    have wf-cdecl: wf-cdecl G (C, c)
    by (simp add: wf-prog-cdecl)
  }
}

```

```

from Init.hyps
have ?HypObj (In1r (if  $C = \text{Object}$  then Skip else Init (super  $c$ )))
  (Norm ((init-class-obj  $G\ C$ )  $s0$ ))  $s1$  ( $\Diamond::\text{vals}$ )
  apply (simp (no-asm-use) only: False if-False simp-thms)
  apply (erule conjE)+
  apply assumption
  done
note hyp-s1 = this [rule-format (no-asm)]
from wf-cdecl  $G$  have
  wt-super: Env  $\vdash$  (if  $C = \text{Object}$  then Skip else Init (super  $c$ ))  $:: \checkmark$ 
  by (cases  $C = \text{Object}$ )
    (auto dest: wf-cdecl-supD is-acc-classD)
from hyp-s1 [OF - - wt-super  $G$ ]
have ?Jmp jmps  $s1$ 
  by simp
hence jmp-s1: ?Jmp jmps ((set-lvars empty)  $s1$ ) by (cases  $s1$ ) simp
from False Init.hyps
have ?HypObj (In1r (init  $c$ )) ((set-lvars empty)  $s1$ )  $s2$  ( $\Diamond::\text{vals}$ )
  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 = empty)  $\vdash$  init  $c :: \checkmark$ 
  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  $s2 = \text{Some } (\text{Jump } j)$ 
proof -
  from False Init.hyps
  have  $s3 = (\text{set-lvars } (\text{locals } (\text{store } s1)))\ s2$  by simp
  with jmp show ?thesis by (cases  $s2$ ) simp
qed
ultimately show ?thesis
  using hyp-init-c [OF jmp-s1 - wt-init-c]
  by simp
qed
}
thus ?case by simp
next
case (NewC  $s0\ C\ s1\ a\ s2\ jmps\ T\ Env$ )
{
  fix  $j$ 
  assume jmp: abrupt  $s2 = \text{Some } (\text{Jump } j)$ 
  have  $j \in jmps$ 
  proof -
    note  $\langle prg\ Env = G \rangle$ 
    moreover note hyp-init =  $\langle PROP\ ?Hyp\ (In1r\ (Init\ C))\ (Norm\ s0)\ s1\ \Diamond \rangle$ 
    moreover from wf NewC.prems
    have Env  $\vdash$  (Init  $C$ )  $:: \checkmark$ 
    by (elim wt-elim-cases) (drule is-acc-classD, simp)
    moreover
    have abrupt  $s1 = \text{Some } (\text{Jump } j)$ 
    proof -
      from  $\langle G \vdash s1 - \text{halloc } CInst\ C \rangle a \rightarrow s2$  and jmp show ?thesis

```

```

      by (rule halloc-no-jump')
    qed
    ultimately show  $j \in \text{jumps}$ 
      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 jumps T Env)
{
  fix j
  assume jmp: abrupt s3 = Some (Jump j)
  have  $j \in \text{jumps}$ 
  proof -
    note  $G = \langle \text{prg Env} = G \rangle$ 
    from NewA.prem
    obtain wt-init:  $\text{Env} \vdash \text{init-comp-ty elT} :: \surd$  and
      wt-size:  $\text{Env} \vdash e :: -\text{PrimT Integer}$ 
    by (elim wt-elim-cases) (auto dest: wt-init-comp-ty')
    note  $\langle \text{PROP ?Hyp (In1r (init-comp-ty elT)) (Norm s0) s1} \Diamond \rangle$ 
    with wt-init G
    have ?Jmp jumps s1
      by (simp add: init-comp-ty-def)
    moreover
    note hyp-e =  $\langle \text{PROP ?Hyp (In1l e) s1 s2 (In1 i)} \rangle$ 
    have abrupt s2 = Some (Jump j)
    proof -
      note  $\langle G \vdash \text{abupd (check-neg i) s2} \text{--halloc Arr elT (the-Intg i)} \succ a \rightarrow s3 \rangle$ 
      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  $j \in \text{jumps}$  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 jumps T Env)
{
  fix j
  assume jmp: abrupt s2 = Some (Jump j)
  have  $j \in \text{jumps}$ 
  proof -
    note hyp-e =  $\langle \text{PROP ?Hyp (In1l e) (Norm s0) s1 (In1 v)} \rangle$ 
    note  $\langle \text{prg Env} = G \rangle$ 
    moreover from Cast.prem
    obtain eT where  $\text{Env} \vdash e :: -eT$  by (elim wt-elim-cases)
    moreover
    have abrupt s1 = Some (Jump j)
    proof -
      note  $\langle s2 = \text{abupd (raise-if } (\neg G, \text{snd } s1 \vdash v \text{ fits } cT) \text{ ClassCast}) s1 \rangle$ 
      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)
{
  fix j
  assume jmp: abrupt s1 = Some (Jump j)
  have j∈jmps
  proof -
    note hyp-e = ⟨PROP ?Hyp (In1l e) (Norm s0) s1 (In1 v)⟩
    note ⟨prg Env = G⟩
    moreover from Inst.premis
    obtain eT where Env⊢e::-eT by (elim wt-elim-cases)
    moreover note jmp
    ultimately show j∈jmps
      by - (rule hyp-e [THEN conjunct1, rule-format (no-asm)], simp-all)
  qed
}
thus ?case by simp
next
case Lit thus ?case by simp
next
case (UnOp s0 e v s1 unop jmps T Env)
{
  fix j
  assume jmp: abrupt s1 = Some (Jump j)
  have j∈jmps
  proof -
    note hyp-e = ⟨PROP ?Hyp (In1l e) (Norm s0) s1 (In1 v)⟩
    note ⟨prg Env = G⟩
    moreover from UnOp.premis
    obtain eT where Env⊢e::-eT by (elim wt-elim-cases)
    moreover note jmp
    ultimately show j∈jmps
      by - (rule hyp-e [THEN conjunct1, rule-format (no-asm)], simp-all)
  qed
}
thus ?case by simp
next
case (BinOp s0 e1 v1 s1 binop e2 v2 s2 jmps T Env)
{
  fix j
  assume jmp: abrupt s2 = Some (Jump j)
  have j∈jmps
  proof -
    note G = ⟨prg Env = G⟩
    from BinOp.premis
    obtain e1T e2T where
      wt-e1: Env⊢e1::-e1T and
      wt-e2: Env⊢e2::-e2T
    by (elim wt-elim-cases)
    note ⟨PROP ?Hyp (In1l e1) (Norm s0) s1 (In1 v1)⟩
    with G wt-e1 have jmp-s1: ?Jmp jmps s1 by simp
    note hyp-e2 =
      ⟨PROP ?Hyp (if need-second-arg binop v1 then In1l e2 else In1r Skip)
        s1 s2 (In1 v2)⟩
    show j∈jmps
  proof (cases need-second-arg binop v1)

```



```

    case True with jmp-s1 wt-e2 jmp G
    show ?thesis
    by - (rule hyp-e2 [THEN conjunct1,rule-format (no-asm)],simp-all)
next
    case False with jmp-s1 jmp G
    show ?thesis
    by - (rule hyp-e2 [THEN conjunct1,rule-format (no-asm)],auto)
qed
qed
}
thus ?case by simp
next
    case Super thus ?case by simp
next
    case (Acc s0 va v f s1 jmps T Env)
    {
    fix j
    assume jmp: abrupt s1 = Some (Jump j)
    have j∈jmps
    proof -
    note hyp-va = ⟨PROP ?Hyp (In2 va) (Norm s0) s1 (In2 (v,f))⟩
    note ⟨prg Env = G⟩
    moreover from Acc.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)⟩
    {
    fix j
    assume jmp: abrupt (assign f v s2) = Some (Jump j)
    have j∈jmps
    proof -
    have abrupt s2 = Some (Jump j)
    proof (cases normal s2)
    case True
    from ⟨G⊢s1 -e-⋈v→ s2⟩ and True have nrm-s1: normal s1
    by (rule eval-no-abrupt-lemma [rule-format])
    with nrm-s1 wt-va G True
    have abrupt (f v s2) ≠ Some (Jump j)
    using hyp-v [THEN conjunct2,rule-format (no-asm)]
    by simp
    from this jmp
    show ?thesis
    by (rule assign-abrupt-propagation)
    qed
    }
    next
    case False with jmp

```

```

    show ?thesis by (cases s2) (simp add: assign-def Let-def)
  qed
  moreover from wt-va G
  have ?Jmp jmps s1
    by - (rule hyp-v [THEN conjunct1],simp-all)
  ultimately show ?thesis using G
    by - (rule hyp-e [THEN conjunct1,rule-format (no-asm)], simp-all, rule wt-e)
  qed
}
thus ?case by simp
next
case (Cond s0 e0 b s1 e1 e2 v s2 jmps T Env)
note G = ⟨prg Env = G⟩
note hyp-e0 = ⟨PROP ?Hyp (In1l e0) (Norm s0) s1 (In1 b)⟩
note hyp-e1-e2 = ⟨PROP ?Hyp (In1l (if the-Bool b then e1 else e2)) s1 s2 (In1 v)⟩
from Cond.prem
obtain e1T e2T
  where wt-e0: Env ⊢ e0 :: − PrimT Boolean
    and wt-e1: Env ⊢ e1 :: − e1T
    and wt-e2: Env ⊢ e2 :: − e2T
  by (elim wt-elim-cases)
{
  fix j
  assume jmp: abrupt s2 = Some (Jump j)
  have j ∈ jmps
  proof -
    from wt-e0 G
    have jmp-s1: ?Jmp jmps s1
      by - (rule hyp-e0 [THEN conjunct1],simp-all)
    show ?thesis
    proof (cases the-Bool b)
      case True
      with jmp-s1 wt-e1 G jmp
      show ?thesis
        by - (rule hyp-e1-e2 [THEN conjunct1,rule-format (no-asm)],simp-all)
    next
      case False
      with jmp-s1 wt-e2 G jmp
      show ?thesis
        by - (rule hyp-e1-e2 [THEN conjunct1,rule-format (no-asm)],simp-all)
    qed
  qed
}
thus ?case by simp
next
case (Call s0 e a s1 args vs s2 D mode statT mn pTs s3 s3' accC v s4
      jmps T Env)
note G = ⟨prg Env = G⟩
from Call.prem
obtain eT argsT
  where wt-e: Env ⊢ e :: − eT and wt-args: Env ⊢ args :: − argsT
  by (elim wt-elim-cases)
{
  fix j
  assume jmp: abrupt ((set-lvars (locals (store s2))) s4)
    = Some (Jump j)
  have j ∈ jmps
  proof -
    note hyp-e = ⟨PROP ?Hyp (In1l e) (Norm s0) s1 (In1 a)⟩

```

```

from wt-e G
have jmp-s1: ?Jump jumps s1
  by – (rule hyp-e [THEN conjunct1],simp-all)
note hyp-args = ⟨PROP ?Hyp (In3 args) s1 s2 (In3 vs)⟩
have abrupt s2 = Some (Jump j)
proof –
  note ⟨G ⊢ s3' – Methd D (⟦name = mn, parTs = pTs⟧) – v → s4⟩
  moreover
  from jmp have abrupt s4 = Some (Jump j)
    by (cases s4) simp
  ultimately have abrupt s3' = Some (Jump j)
    by – (rule ccontr,drule (1) Methd-no-jump,simp)
  moreover note ⟨s3' = check-method-access G accC statT mode
    (⟦name = mn, parTs = pTs⟧) a s3⟩
  ultimately have abrupt s3 = Some (Jump j)
    by (cases s3)
    (simp add: check-method-access-def abrupt-if-def Let-def)
  moreover
  note ⟨s3 = init-lvars G D (⟦name=mn, parTs=pTs⟧) mode a vs s2⟩
  ultimately show ?thesis
    by (cases s2) (auto simp add: init-lvars-def2)
qed
with jmp-s1 wt-args G
show ?thesis
  by – (rule hyp-args [THEN conjunct1,rule-format (no-asm)], simp-all)
qed
}
thus ?case by simp
next
case (Methd s0 D sig v s1 jumps 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 jumps T Env)
have G ⊢ Norm s0 – Body D c – v → 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 jumps T Env)
note G = ⟨prg Env = G⟩
from wf FVar.prems
obtain statC f where
  wt-e: Env ⊢ e :: – Class statC and
  accfield: accfield (prg Env) accC statC fn = Some (statDeclC.f)
  by (elim wt-elim-cases) simp
have wt-init: Env ⊢ Init statDeclC :: ✓
proof –
  from wf wt-e G
  have is-class (prg Env) statC

```

```

    by (auto dest: ty-expr-is-type type-is-class)
  with wf accfield G
  have is-class (prg Env) statDeclC
    by (auto dest!: accfield-fields dest: fields-declC)
  thus ?thesis
    by simp
qed
note fvar = ⟨(v, s2') = fvar statDeclC stat fn a s2⟩
{
  fix j
  assume jmp: abrupt s3 = Some (Jump j)
  have j∈jmps
  proof -
    note hyp-init = ⟨PROP ?Hyp (In1r (Init statDeclC)) (Norm s0) s1 ◇⟩
    from G wt-init
    have ?Jmp jmps s1
      by - (rule hyp-init [THEN conjunct1], auto)
    moreover
    note hyp-e = ⟨PROP ?Hyp (In1l e) s1 s2 (In1 a)⟩
    have abrupt s2 = Some (Jump j)
    proof -
      note ⟨s3 = check-field-access G accC statDeclC fn stat a s2'⟩
      with jmp have abrupt s2' = Some (Jump j)
      by (cases s2')
      (simp add: check-field-access-def abrupt-if-def Let-def)
      with fvar show abrupt s2 = Some (Jump j)
      by (cases s2) (simp add: fvar-def2 abrupt-if-def)
    qed
    ultimately show ?thesis
      using G wt-e
      by - (rule hyp-e [THEN conjunct1, rule-format (no-asm)], simp-all)
    qed
  }
  moreover
  from fvar obtain upd w
    where upd: upd = snd (fst (fvar statDeclC stat fn a s2)) and
      v: v=(w,upd)
  by (cases fvar statDeclC stat fn a s2) simp
  {
    fix j val fix s::state
    assume normal s3
    assume no-jmp: abrupt s ≠ Some (Jump j)
    with upd
    have abrupt (upd val s) ≠ Some (Jump j)
      by (rule fvar-upd-no-jump)
  }
  ultimately show ?case using v by simp
next
case (AVar s0 e1 a s1 e2 i s2 v s2' jmps T Env)
note G = ⟨prg Env = G⟩
from AVar.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⟩
{
  fix j
  assume jmp: abrupt s2' = Some (Jump j)
  have j∈jmps

```

```

proof –
  note hyp-e1 = ⟨PROP ?Hyp (In1l e1) (Norm s0) s1 (In1 a)⟩
  from G wt-e1
  have ?Jump jumps s1
    by – (rule hyp-e1 [THEN conjunct1], auto)
  moreover
  note hyp-e2 = ⟨PROP ?Hyp (In1l e2) s1 s2 (In1 i)⟩
  have abrupt s2 = Some (Jump j)
  proof –
    from avar have s2' = snd (avar G i a s2)
    by (cases avar G i a s2) simp
    with jmp show ?thesis by – (rule avar-state-no-jump, simp)
  qed
  ultimately show ?thesis
    using wt-e2 G
    by – (rule hyp-e2 [THEN conjunct1, rule-format (no-asm)], simp-all)
  qed
}
moreover
from avar obtain upd w
  where upd: upd = snd (fst (avar G i a s2)) and
    v: v=(w,upd)
  by (cases avar G i a s2) simp
{
  fix j val fix s::state
  assume normal s2'
  assume no-jmp: abrupt s ≠ Some (Jump j)
  with upd
  have abrupt (upd val s) ≠ Some (Jump j)
    by (rule avar-upd-no-jump)
}
ultimately show ?case using v by simp
next
case Nil thus ?case by simp
next
case (Cons s0 e v s1 es vs s2 jumps 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
{
  fix j
  assume jmp: abrupt s2 = Some (Jump j)
  have j ∈ jumps
  proof –
    note hyp-e = ⟨PROP ?Hyp (In1l e) (Norm s0) s1 (In1 v)⟩
    from G wt-e
    have ?Jump 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 fastsimp
    done
qed

```

**lemmas** jumpNestingOk-eval-no-jumpE  
 = jumpNestingOk-eval-no-jump [THEN conjE,rule-format]

```

corollary eval-expression-no-jump:
assumes eval: prg Env ⊢ s0 -e->v→ s1 and
            no-jmp: abrupt s0 ≠ Some (Jump j) and
            wt: Env ⊢ e::¬T and
            wf: wf-prog (prg Env)
shows abrupt s1 ≠ Some (Jump j)
using eval - no-jmp wt wf
by (rule jumpNestingOk-eval-no-jumpE, simp-all)

```

```

corollary eval-var-no-jump:
assumes eval: prg Env ⊢ s0 -var=>(w,upd)→ s1 and
            no-jmp: abrupt s0 ≠ Some (Jump j) and

```

$wt: Env \vdash var ::= T$  **and**  
 $wf: wf\text{-}prog \ (prg \ Env)$   
**shows**  $abrupt \ s1 \neq Some \ (Jump \ j) \wedge$   
 $(normal \ s1 \longrightarrow$   
 $(abrupt \ s \neq Some \ (Jump \ j'))$   
 $\longrightarrow abrupt \ (upd \ val \ s) \neq Some \ (Jump \ j'))$   
**apply**  $(rule\text{-}tac \ upd=upd \ \mathbf{and} \ val=val \ \mathbf{and} \ s=s \ \mathbf{and} \ w=w \ \mathbf{and} \ j'=j')$   
**in**  $jumpNestingOk\text{-}eval\text{-}no\text{-}jumpE \ [OF \ eval \ - \ no\text{-}jmp \ wt \ wf]$   
**by**  $simp\text{-}all$

**lemmas**  $eval\text{-}var\text{-}no\text{-}jumpE = eval\text{-}var\text{-}no\text{-}jump \ [THEN \ conjE, rule\text{-}format]$

**corollary**  $eval\text{-}statement\text{-}no\text{-}jump$ :

**assumes**  $eval: prg \ Env \vdash s0 \dashv\!\rightarrow c \rightarrow s1$  **and**  
 $jmpOk: jumpNestingOkS \ \{\} \ c$  **and**  
 $no\text{-}jmp: abrupt \ s0 \neq Some \ (Jump \ j)$  **and**  
 $wt: Env \vdash c ::= \surd$  **and**  
 $wf: wf\text{-}prog \ (prg \ Env)$   
**shows**  $abrupt \ s1 \neq Some \ (Jump \ j)$   
**using**  $eval \ - \ no\text{-}jmp \ wt \ wf$   
**by**  $(rule \ jumpNestingOk\text{-}eval\text{-}no\text{-}jumpE) \ (simp\text{-}all \ add: \ jmpOk)$

**corollary**  $eval\text{-}expression\text{-}list\text{-}no\text{-}jump$ :

**assumes**  $eval: prg \ Env \vdash s0 \dashv\!\rightarrow es \dashv\!\rightarrow v \rightarrow s1$  **and**  
 $no\text{-}jmp: abrupt \ s0 \neq Some \ (Jump \ j)$  **and**  
 $wt: Env \vdash es ::= T$  **and**  
 $wf: wf\text{-}prog \ (prg \ Env)$   
**shows**  $abrupt \ s1 \neq Some \ (Jump \ j)$   
**using**  $eval \ - \ no\text{-}jmp \ wt \ wf$   
**by**  $(rule \ jumpNestingOk\text{-}eval\text{-}no\text{-}jumpE, \ simp\text{-}all)$

**lemma**  $union\text{-}subteq\text{-}elim \ [elim]: \llbracket A \cup B \subseteq C; \llbracket A \subseteq C; B \subseteq C \rrbracket \Longrightarrow P \rrbracket \Longrightarrow P$   
**by**  $blast$

**lemma**  $dom\text{-}locals\text{-}halloc\text{-}mono$ :

**assumes**  $halloc: G \vdash s0 \text{-}halloc \ oi \dashv\!\rightarrow a \rightarrow s1$   
**shows**  $dom \ (locals \ (store \ s0)) \subseteq dom \ (locals \ (store \ s1))$   
**proof**  $-$   
**from**  $halloc$  **show**  $?thesis$   
**by**  $cases \ simp\text{-}all$   
**qed**

**lemma**  $dom\text{-}locals\text{-}salloc\text{-}mono$ :

**assumes**  $salloc: G \vdash s0 \text{-}salloc \rightarrow s1$   
**shows**  $dom \ (locals \ (store \ s0)) \subseteq dom \ (locals \ (store \ s1))$   
**proof**  $-$   
**from**  $salloc$  **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:  $\text{dom } (\text{locals } (\text{store } s)) \subseteq \text{dom } (\text{locals } (\text{store } (f \ n \ s)))$ 
  shows  $\text{dom } (\text{locals } (\text{store } s)) \subseteq \text{dom } (\text{locals } (\text{store } (\text{assign } f \ n \ s)))$ 
proof (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:
   $\text{dom } (\text{locals } (\text{store } s)) \subseteq \text{dom } (\text{locals } (\text{store } (\text{snd } (\text{lvar } vn \ s') \ \text{val } s)))$ 
by (simp add: lvar-def) blast

```

```

lemma dom-locals-fvar-vvar-mono:
   $\text{dom } (\text{locals } (\text{store } s))$ 
   $\subseteq \text{dom } (\text{locals } (\text{store } (\text{snd } (\text{fst } (\text{fvar } \text{statDeclC } \text{stat } fn \ a \ s') \ \text{val } s))))$ 
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:
   $\text{dom } (\text{locals } (\text{store } s))$ 
   $\subseteq \text{dom } (\text{locals } (\text{store } (\text{snd } (\text{fvar } \text{statDeclC } \text{stat } fn \ a \ s))))$ 
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 ... ⊆ dom (locals (store s2))
    by simp
  finally show ?case by simp
next
  case (If s0 e b s1 c1 c2 s2)
  from If.hyps
  have dom (locals (store ((Norm s0)::state))) ⊆ dom (locals (store s1))
    by simp

```

```

    also
    from If.hyps
    have ...  $\subseteq \text{dom } (\text{locals } (\text{store } s2))$ 
      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:
         $\text{dom } (\text{locals } (\text{store } ((\text{Norm } s0)::\text{state}))) \subseteq \text{dom } (\text{locals } (\text{store } s1))$  and
      s1-s2:  $\text{dom } (\text{locals } (\text{store } s1)) \subseteq \text{dom } (\text{locals } (\text{store } s2))$  and
      s2-s3:  $\text{dom } (\text{locals } (\text{store } s2)) \subseteq \text{dom } (\text{locals } (\text{store } s3))$ 
      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 Jump 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:  $\text{dom } (\text{locals } (\text{store } ((\text{Norm } s0)::\text{state})))$ 
     $\subseteq \text{dom } (\text{locals } (\text{store } s1))$  by simp
  from  $\langle G \vdash s1 -\text{salloc} \rightarrow s2 \rangle$ 
  have s1-s2:  $\text{dom } (\text{locals } (\text{store } s1)) \subseteq \text{dom } (\text{locals } (\text{store } s2))$ 
    by (rule dom-locals-salloc-mono)
  thus ?case
  proof (cases  $G, s2 \vdash \text{catch } C$ )
    case True
    note s0-s1 also note s1-s2
    also
    from True Try.hyps
    have  $\text{dom } (\text{locals } (\text{store } (\text{new-xcpt-var } vn s2)))$ 
       $\subseteq \text{dom } (\text{locals } (\text{store } s3))$ 
      by simp
    hence  $\text{dom } (\text{locals } (\text{store } s2)) \subseteq \text{dom } (\text{locals } (\text{store } s3))$ 
      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  $\exists \text{err}. x1 = \text{Some } (\text{Error } \text{err})$ )
  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 inited C (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 halloc =  $\langle G \vdash s1 \text{ --halloc } C \text{Inst } C \succ a \rightarrow s2 \rangle$ 
  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 =  $\langle G \vdash \text{abupd } (\text{check-neg } i) \ s2 \text{ --halloc } \text{Arr } T \ (\text{the-Intg } i) \succ a \rightarrow s3 \rangle$ 
  from NewA.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 NewA.hyps

```

```

    have ...  $\subseteq$  dom (locals (store s2)) by simp
  also
  from halloc
  have ...  $\subseteq$  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)))  $\subseteq$  dom (locals (store s1))
    by simp
  also
  from BinOp.hyps
  have ...  $\subseteq$  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)))  $\subseteq$  dom (locals (store s1))
    by simp
  show ?case
  proof (cases normal s1)
    case True
    with Ass.hyps
    have ass-ok:
       $\bigwedge s \text{ val. dom (locals (store s))} \subseteq \text{dom (locals (store (f val s)))}$ 
      by simp
    note s0-s1
    also
    from Ass.hyps
    have dom (locals (store s1))  $\subseteq$  dom (locals (store s2))
      by simp
    also
    from ass-ok
    have ...  $\subseteq$  dom (locals (store (assign f v s2)))
      by (rule dom-locals-assign-mono)
    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-s1 False
    have dom (locals (store ((Norm s0)::state)))
       $\subseteq$  dom (locals (store (assign f v s2)))
      by simp

```

```

    thus ?thesis
    by simp
qed
next
case (Cond s0 e0 b s1 e1 e2 v s2)
from Cond.hyps
have dom (locals (store ((Norm s0)::state)))  $\subseteq$  dom (locals (store s1))
  by simp
also
from Cond.hyps
have ...  $\subseteq$  dom (locals (store s2))
  by simp
finally show ?case by simp
next
case (Call s0 e a' s1 args vs s2 D mode statT mn pTs s3 s3' accC v s4)
note s3 =  $\langle s3 = \text{init-lvars } G \ D \ (\text{name} = \text{mn}, \text{parTs} = \text{pTs}) \ \text{mode } a' \ \text{vs } s2 \rangle$ 
from Call.hyps
have dom (locals (store ((Norm s0)::state)))  $\subseteq$  dom (locals (store s1))
  by simp
also
from Call.hyps
have ...  $\subseteq$  dom (locals (store s2))
  by simp
also
have ...  $\subseteq$  dom (locals (store ((set-lvars (locals (store s2))) s4)))
  by (cases s4) simp
finally show ?case by simp
next
case Methd thus ?case by simp
next
case (Body s0 D s1 c s2 s3)
from Body.hyps
have dom (locals (store ((Norm s0)::state)))  $\subseteq$  dom (locals (store s1))
  by simp
also
from Body.hyps
have ...  $\subseteq$  dom (locals (store s2))
  by simp
also
have ...  $\subseteq$  dom (locals (store (abupd (absorb Ret) s2)))
  by simp
also
have ...  $\subseteq$  dom (locals (store (abupd (absorb Ret) s3)))
  by simp
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
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  $s2'$ :  $s2' = \text{snd } (\text{fvar statDeclC stat fn a } s2)$  and
     $v: v = \text{fst } (\text{fvar statDeclC stat fn a } s2)$ 
    by ( $\text{cases fvar statDeclC stat fn a } s2$ ) simp
from  $v$ 
have  $\forall s \text{ val. dom } (\text{locals } (\text{store } s))$ 
     $\subseteq \text{dom } (\text{locals } (\text{store } (\text{snd } v \text{ val } s)))$  (is  $?V\text{-ok}$ )
    by (simp add: dom-locals-fvar-vvar-mono)
hence  $v\text{-ok}: (\forall vv. \text{In2 } v = \text{In2 } vv \wedge \text{normal } s3 \longrightarrow ?V\text{-ok})$ 
    by  $-(\text{intro strip, simp})$ 
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  $v\text{-ok}$ 
    by simp
next
case ( $AVar s0 e1 a s1 e2 i s2 v s2'$ )
from AVar.hyps
obtain  $s2'$ :  $s2' = \text{snd } (\text{avar } G i a s2)$  and
     $v: v = \text{fst } (\text{avar } G i a s2)$ 
    by ( $\text{cases avar } G i a s2$ ) simp
from  $v$ 
have  $\forall s \text{ val. dom } (\text{locals } (\text{store } s))$ 
     $\subseteq \text{dom } (\text{locals } (\text{store } (\text{snd } v \text{ val } s)))$  (is  $?V\text{-ok}$ )
    by (simp add: dom-locals-avar-vvar-mono)
hence  $v\text{-ok}: (\forall vv. \text{In2 } v = \text{In2 } vv \wedge \text{normal } s2' \longrightarrow ?V\text{-ok})$ 
    by  $-(\text{intro strip, simp})$ 
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  $\dots \subseteq \text{dom } (\text{locals } (\text{store } s2'))$ 
    by (simp add: dom-locals-avar-mono)
finally
show  $?case$  using  $v\text{-ok}$  by simp
next
case Nil thus  $?case$  by simp
next
case ( $\text{Cons } s0 e v s1 es vs s2$ )

```

```

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:  $G \vdash s0 -t \rightarrow (v, s1)$ 
  obtains dom (locals (store s0))  $\subseteq$  dom (locals (store s1)) and
     $\bigwedge vv\ s\ val. \llbracket v = \text{In2 } vv; \text{normal } s1 \rrbracket$ 
       $\implies$  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:  $G \vdash s0 -\text{halloc } oi \rightarrow a \rightarrow s1$  and
    normal: normal s1
  shows normal s0
proof -
  from halloc normal show ?thesis
  by cases simp-all
qed

```

```

lemma sxalloc-mono-no-abrupt:
  assumes sxalloc:  $G \vdash s0 -\text{sxalloc} \rightarrow s1$  and
    normal: normal s1
  shows normal s0
proof -
  from sxalloc 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** *Imp* **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**  $\text{halloc} = \langle G \vdash \text{abupd } (\text{check-neg } i) \text{ } s2 \text{ } \text{--halloc } \text{Arr } T \text{ } (\text{the-Intg } i) \succ a \rightarrow s3 \rangle$

**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 assigns (In1l e)  $\subseteq$  dom (locals (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  $\langle G \vdash s1 \text{ -- (if need-second-arg binop v1 then In1l e2} \\ \text{else In1r Skip)} \rangle \rightarrow \langle In1\ v2, s2 \rangle$ 
    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 add: Un-subset-iff)
  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 =  $\langle \text{normal} \ (\text{assign } f\ v\ s2) \rangle$ 
  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 ...  $\subseteq \text{dom } (\text{locals } (\text{store } s2))$ 
  by – (erule dom-locals-eval-mono-elim)
also
from nrm-s2 Ass.hyps
have assigns (In1l e)  $\subseteq \text{dom } (\text{locals } (\text{store } s2))$ 
  by iprover
ultimately
have assigns (In2 va)  $\cup$  assigns (In1l e)  $\subseteq \text{dom } (\text{locals } (\text{store } s2))$ 
  by (rule Un-least)
also
from Ass.hyps nrm-s1
have ...  $\subseteq \text{dom } (\text{locals } (\text{store } (f \ v \ s2)))$ 
  by – (erule dom-locals-eval-mono-elim, cases s2,simp)
then
have  $\text{dom } (\text{locals } (\text{store } s2)) \subseteq \text{dom } (\text{locals } (\text{store } (\text{assign } f \ v \ s2)))$ 
  by (rule dom-locals-assign-mono)
finally
have va-e: assigns (In2 va)  $\cup$  assigns (In1l e)
   $\subseteq \text{dom } (\text{locals } (\text{snd } (\text{assign } f \ v \ s2)))$  .
show ?case
proof (cases  $\exists \ n. \ va = \text{LVar } n$ )
  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  $G \vdash \text{Norm } s0 \multimap \text{LVar } n \multimap (w, f) \rightarrow s1$ 
    by simp
    hence  $(w, f) = \text{lvar } n \ s0$ 
    by (rule eval-elim-cases) simp
    with nrm-ass-s2
    have  $n \in \text{dom } (\text{locals } (\text{store } (\text{assign } f \ v \ s2)))$ 
    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)  $\subseteq \text{dom } (\text{locals } (\text{store } s1))$ 
  by iprover
  also from Cond.hyps
  have ...  $\subseteq \text{dom } (\text{locals } (\text{store } s2))$ 
  by – (erule dom-locals-eval-mono-elim)
  finally have e0: assigns (In1l e0)  $\subseteq \text{dom } (\text{locals } (\text{store } s2))$  .
  show ?case
  proof (cases the-Bool b)
    case True
    with Cond
    have assigns (In1l e1)  $\subseteq \text{dom } (\text{locals } (\text{store } s2))$ 
    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)
     $\subseteq$  dom (locals (store s2))
    by (rule Un-least)
  thus ?thesis using True by simp
next
  case False
  with Cond
  have assigns (In1l e2)  $\subseteq$  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)
     $\subseteq$  dom (locals (store s2))
    by (rule Un-least)
  thus ?thesis using False by simp
qed
next
  case (Call s0 e a' s1 args vs s2 D mode statT mn pTs s3 s3' accC v s4)
  have nrm-s2: normal s2
  proof -
    from (normal ((set-lvars (locals (snd s2))) s4))
    have normal-s4: normal s4 by simp
    hence normal s3' using Call.hyps
      by - (erule eval-no-abrupt-lemma [rule-format])
    moreover note
      (s3' = check-method-access G accC statT mode (name=mn, parTs=pTs) a' s3)
    ultimately have normal s3
      by (cases s3) (simp add: check-method-access-def Let-def)
    moreover
    note s3 = (s3 = init-lvars G D (name = mn, parTs = pTs) mode a' vs s2)
    ultimately show normal s2
      by (cases s2) (simp add: init-lvars-def2)
  qed
  hence normal s1 using Call.hyps
    by - (erule eval-no-abrupt-lemma [rule-format])
  with Call.hyps
  have assigns (In1l e)  $\subseteq$  dom (locals (store s1))
    by iprover
  also from Call.hyps
  have ...  $\subseteq$  dom (locals (store s2))
    by - (erule dom-locals-eval-mono-elim)
  also
  from nrm-s2 Call.hyps
  have assigns (In3 args)  $\subseteq$  dom (locals (store s2))
    by iprover
  ultimately have assigns (In1l e)  $\cup$  assigns (In3 args)  $\subseteq$  ...
    by (rule Un-least)
  also
  have ...  $\subseteq$  dom (locals (store ((set-lvars (locals (store s2))) s4)))
    by (cases s4) simp
  finally show ?case
    by simp
next
  case Methd thus ?case by simp
next
  case Body thus ?case by simp
next

```

```

  case LVar thus ?case by simp
next
  case (FVar s0 statDeclC s1 e a s2 v s2' stat fn s3 accC)
  note s3 = ⟨s3 = check-field-access G accC statDeclC fn stat a s2'⟩
  note avar = ⟨(v, s2') = fvar statDeclC stat fn a s2⟩
  have nrm-s2: normal s2
  proof -
    note ⟨normal s3⟩
    with s3 have normal s2'
      by (cases s2') (simp add: check-field-access-def Let-def)
    with avar show normal s2
      by (cases s2) (simp add: fvar-def2)
  qed
  with FVar.hyps
  have assigns (In1l e) ⊆ dom (locals (store s2))
    by iprover
  also
  have ... ⊆ dom (locals (store s2'))
  proof -
    from avar
    have s2' = snd (fvar statDeclC stat fn a s2)
      by (cases fvar statDeclC stat fn a s2) simp
    thus ?thesis
      by simp (rule dom-locals-fvar-mono)
  qed
  also from s3
  have ... ⊆ dom (locals (store s3))
    by (cases s2') (simp add: check-field-access-def Let-def)
  finally show ?case
    by simp
next
  case (AVar s0 e1 a s1 e2 i s2 v s2')
  note avar = ⟨(v, s2') = avar G i a s2⟩
  have nrm-s2: normal s2
  proof -
    from avar and ⟨normal s2'⟩
    show ?thesis by (cases s2) (simp add: avar-def2)
  qed
  with AVar.hyps
  have normal s1
    by - (erule eval-no-abrupt-lemma [rule-format])
  with AVar.hyps
  have assigns (In1l e1) ⊆ dom (locals (store s1))
    by iprover
  also from AVar.hyps
  have ... ⊆ dom (locals (store s2))
    by - (erule dom-locals-eval-mono-elim)
  also
  from AVar.hyps nrm-s2
  have assigns (In1l e2) ⊆ dom (locals (store s2))
    by iprover
  ultimately
  have assigns (In1l e1) ∪ assigns (In1l e2) ⊆ ...
    by (rule Un-least)
  also
  have dom (locals (store s2)) ⊆ dom (locals (store s2'))
  proof -
    from avar have s2' = snd (avar G i a s2)
      by (cases avar G i a s2) simp

```

```

    thus ?thesis
    by simp (rule dom-locals-avar-mono)
qed
finally
show ?case
  by simp
next
case Nil show ?case by simp
next
case (Cons s0 e v s1 es vs s2)
have assigns (In1l e)  $\subseteq$  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 -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 -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 -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 and True
proof (induct rule: var-expr-stmt.inducts)
  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=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
     $v2: G \vdash s1 \multimap (\text{if need-second-arg } \text{binop } v1 \text{ then } \text{In1l } e2$ 
     $\text{else } \text{In1r } \text{Skip}) \multimap ( \text{In1 } v2, s) \text{ and}$ 
     $v: v = \text{eval-binop } \text{binop } v1 \ v2$ 
    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 vb:  $G \vdash \text{Norm } s0 - b \multimap vb \rightarrow s1$  and
    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  $G \vdash \text{Norm } s1' - e2 \rightarrow v \rightarrow s$ 
      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
     $\bigwedge c. \llbracket \text{constVal } e = \text{Some } c; \text{Env} \vdash e :: \neg \text{PrimT Boolean} \rrbracket$ 
     $\implies \text{typeof empty-dt } c = \text{Some } (\text{PrimT Boolean})$ 
    and True and True
  proof (induct rule: var-expr-stmt.inducts)
    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 ⟨Env ⊢ UnOp unop e :: − PrimT Boolean⟩
  have Boolean = unop-type unop by cases simp
  moreover
  from ⟨constVal (UnOp unop e) = Some c⟩
  obtain ce where c = eval-unop unop ce by auto
  ultimately show ?case by (simp add: eval-unop-type)
next
  case (BinOp binop e1 e2 c)
  from ⟨Env ⊢ BinOp binop e1 e2 :: − PrimT Boolean⟩
  have Boolean = binop-type binop by cases simp
  moreover
  from ⟨constVal (BinOp binop e1 e2) = Some c⟩
  obtain c1 c2 where c = eval-binop binop c1 c2 by auto
  ultimately show ?case by (simp add: eval-binop-type)
next
  case Super hence False by simp thus ?case ..
next
  case Acc hence False by simp thus ?case ..
next
  case Ass hence False by simp thus ?case ..
next
  case (Cond b e1 e2 c)
  note c = ⟨constVal (b ? e1 : e2) = Some c⟩
  then obtain cb c1 c2 where
    cb: constVal b = Some cb and
    c1: constVal e1 = Some c1 and
    c2: constVal e2 = Some c2
  by (auto split: bool.splits)
  note wt = ⟨Env ⊢ b ? e1 : e2 :: − PrimT Boolean⟩
  then
  obtain T1 T2
  where Env ⊢ b :: − PrimT Boolean and
    wt-e1: Env ⊢ e1 :: − PrimT Boolean and
    wt-e2: Env ⊢ e2 :: − PrimT Boolean
  by cases (auto dest: widen-Boolean2)
  show ?case
  proof (cases the-Bool cb)
    case True
    from c1 wt-e1
    have typeof empty-dt c1 = Some (PrimT Boolean)
    by (rule Cond.hyps)
    with True c cb c1 show ?thesis by simp
  next
    case False
    from c2 wt-e2
    have typeof empty-dt c2 = Some (PrimT Boolean)
    by (rule Cond.hyps)
    with False c cb c2 show ?thesis by simp
  qed
next
  case Call hence False by simp thus ?case ..
qed simp-all
with const wt
show ?thesis
by iprover

```

qed

**lemma** *assigns-if-good-approx*:**assumes***eval*:  $\text{prg } \text{Env} \vdash s0 \dashv e \dashv b \rightarrow s1$  **and***normal*: *normal* *s1* **and***bool*:  $\text{Env} \vdash e :: \text{--PrimT Boolean}$ **shows** *assigns-if* (*the-Bool* *b*)  $e \subseteq \text{dom } (\text{locals } (\text{store } s1))$ **proof** –

— To properly perform induction on the evaluation relation we have to generalize the lemma to terms not only expressions.

{ **fix** *t val***assume** *eval'*:  $\text{prg } \text{Env} \vdash s0 \dashv t \rightarrow (val, s1)$ **assume** *bool'*:  $\text{Env} \vdash t :: \text{Inl } (\text{PrimT Boolean})$ **assume** *expr*:  $\exists \text{expr}. t = \text{Inl1 expr}$ **have** *assigns-if* (*the-Bool* (*the-In1 val*)) (*the-In1 t*)  
 $\subseteq \text{dom } (\text{locals } (\text{store } s1))$ **using** *eval'* *normal bool' expr***proof** (*induct*)**case** *Abrupt* **thus** ?*case* **by** *simp***next****case** (*NewC* *s0 C s1 a s2*)**from**  $\langle \text{Env} \vdash \text{NewC } C :: \text{--PrimT Boolean} \rangle$ **have** *False***by** *cases simp***thus** ?*case* ..**next****case** (*NewA* *s0 T s1 e i s2 a s3*)**from**  $\langle \text{Env} \vdash \text{New } T[e] :: \text{--PrimT Boolean} \rangle$ **have** *False***by** *cases simp***thus** ?*case* ..**next****case** (*Cast* *s0 e b s1 s2 T*)**note**  $s2 = \langle s2 = \text{abupd } (\text{raise-if } (\neg \text{prg } \text{Env}, \text{snd } s1 \vdash b \text{ fits } T) \text{ ClassCast}) s1 \rangle$ **have** *assigns-if* (*the-Bool* *b*)  $e \subseteq \text{dom } (\text{locals } (\text{store } s1))$ **proof** –**from** *s2* **and**  $\langle \text{normal } s2 \rangle$ **have** *normal s1***by** (*cases s1*) *simp***moreover****from**  $\langle \text{Env} \vdash \text{Cast } T e :: \text{--PrimT Boolean} \rangle$ **have**  $\text{Env} \vdash e :: \text{--PrimT Boolean}$ **by** *cases (auto dest: cast-Boolean2)***ultimately show** ?*thesis***by** (*rule Cast.hyps [elim-format]*) *auto*

qed

**also from** *s2***have**  $\dots \subseteq \text{dom } (\text{locals } (\text{store } s2))$ **by** *simp***finally show** ?*case* **by** *simp***next****case** (*Inst* *s0 e v s1 b T*)**from**  $\langle \text{prg } \text{Env} \vdash \text{Norm } s0 \dashv e \dashv v \rightarrow s1 \rangle$  **and**  $\langle \text{normal } s1 \rangle$ **have** *assignsE*  $e \subseteq \text{dom } (\text{locals } (\text{store } s1))$ **by** (*rule assignsE-good-approx*)**thus** ?*case***by** *simp*

```

next
  case (Lit s v)
  from (Env ⊢ Lit v :: -PrimT Boolean)
  have typeof empty-dt v = Some (PrimT Boolean)
    by cases simp
  then obtain b where v = Bool b
    by (cases v) (simp-all add: empty-dt-def)
  thus ?case
    by simp
next
  case (UnOp s0 e v s1 unop)
  note bool = (Env ⊢ UnOp unop e :: -PrimT Boolean)
  hence bool-e: Env ⊢ e :: -PrimT Boolean
    by cases (cases unop, simp-all)
  show ?case
  proof (cases constVal (UnOp unop e))
    case None
    note (normal s1)
    moreover note bool-e
    ultimately have assigns-if (the-Bool v) e ⊆ dom (locals (store s1))
      by (rule UnOp.hyps [elim-format]) auto
    moreover
    from bool have unop = UNot
      by cases (cases unop, simp-all)
    moreover note None
    ultimately
    have assigns-if (the-Bool (eval-unop unop v)) (UnOp unop e)
      ⊆ dom (locals (store s1))
      by simp
    thus ?thesis by simp
  next
    case (Some c)
    moreover
    from (prg Env ⊢ Norm s0 - e -> v → s1)
    have prg Env ⊢ Norm s0 - UnOp unop e -> eval-unop unop v → s1
      by (rule eval.UnOp)
    with Some
    have eval-unop unop v = c
      by (rule constVal-eval-elim) simp
    moreover
    from Some bool
    obtain b where c = Bool b
      by (rule constVal-Boolean [elim-format])
      (cases c, simp-all add: empty-dt-def)
    ultimately
    have assigns-if (the-Bool (eval-unop unop v)) (UnOp unop e) = {}
      by simp
    thus ?thesis by simp
  qed
next
  case (BinOp s0 e1 v1 s1 binop e2 v2 s2)
  note bool = (Env ⊢ BinOp binop e1 e2 :: -PrimT Boolean)
  show ?case
  proof (cases constVal (BinOp binop e1 e2))
    case (Some c)
    moreover
    from BinOp.hyps
    have
      prg Env ⊢ Norm s0 - BinOp binop e1 e2 -> eval-binop binop v1 v2 → s2

```

```

    by - (rule eval.BinOp)
  with Some
  have eval-binop binop v1 v2=c
    by (rule constVal-eval-elim) simp
  moreover
  from Some bool
  obtain b where c = Bool b
    by (rule constVal-Boolean [elim-format])
    (cases c, simp-all add: empty-dt-def)
  ultimately
  have assigns-if (the-Bool (eval-binop binop v1 v2)) (BinOp binop e1 e2)
    = {}
    by simp
  thus ?thesis by simp
next
case None
show ?thesis
proof (cases binop=CondAnd ∨ binop=CondOr)
  case True
  from bool obtain bool-e1: Env⊢e1::-PrimT Boolean and
    bool-e2: Env⊢e2::-PrimT Boolean
    using True by cases auto
  have assigns-if (the-Bool v1) e1 ⊆ dom (locals (store s1))
  proof -
    from BinOp have normal s1
      by - (erule eval-no-abrupt-lemma [rule-format])
    from this bool-e1
    show ?thesis
      by (rule BinOp.hyps [elim-format]) auto
  qed
  also
  from BinOp.hyps
  have ... ⊆ dom (locals (store s2))
    by - (erule dom-locals-eval-mono-elim,simp)
  finally
  have e1-s2: assigns-if (the-Bool v1) e1 ⊆ dom (locals (store s2)).
  from True show ?thesis
  proof
    assume condAnd: binop = CondAnd
    show ?thesis
    proof (cases the-Bool (eval-binop binop v1 v2))
      case True
      with condAnd
      have need-second: need-second-arg binop v1
        by (simp add: need-second-arg-def)
      from ⟨normal s2⟩
      have assigns-if (the-Bool v2) e2 ⊆ dom (locals (store s2))
        by (rule BinOp.hyps [elim-format])
        (simp add: need-second bool-e2)+
      with e1-s2
      have assigns-if (the-Bool v1) e1 ∪ assigns-if (the-Bool v2) e2
        ⊆ dom (locals (store s2))
        by (rule Un-least)
      with True condAnd None show ?thesis
        by simp
    end
  end
  next
  case False
  note binop-False = this
  show ?thesis

```

```

proof (cases need-second-arg binop v1)
  case True
  with binop-False condAnd
  obtain the-Bool v1=True and the-Bool v2 = False
    by (simp add: need-second-arg-def)
  moreover
  from ⟨normal s2⟩
  have assigns-if (the-Bool v2) e2 ⊆ dom (locals (store s2))
    by (rule BinOp.hyps [elim-format]) (simp add: True bool-e2)+
  with e1-s2
  have assigns-if (the-Bool v1) e1 ∪ assigns-if (the-Bool v2) e2
    ⊆ dom (locals (store s2))
    by (rule Un-least)
  moreover note binop-False condAnd None
  ultimately show ?thesis
    by auto
next
  case False
  with binop-False condAnd
  have the-Bool v1=False
    by (simp add: need-second-arg-def)
  with e1-s2
  show ?thesis
    using binop-False condAnd None by auto
qed
qed
next
assume condOr: binop = CondOr
show ?thesis
proof (cases the-Bool (eval-binop binop v1 v2))
  case False
  with condOr
  have need-second: need-second-arg binop v1
    by (simp add: need-second-arg-def)
  from ⟨normal s2⟩
  have assigns-if (the-Bool v2) e2 ⊆ dom (locals (store s2))
    by (rule BinOp.hyps [elim-format])
    (simp add: need-second bool-e2)+
  with e1-s2
  have assigns-if (the-Bool v1) e1 ∪ assigns-if (the-Bool v2) e2
    ⊆ dom (locals (store s2))
    by (rule Un-least)
  with False condOr None show ?thesis
    by simp
next
  case True
  note binop-True = this
  show ?thesis
proof (cases need-second-arg binop v1)
  case True
  with binop-True condOr
  obtain the-Bool v1=False and the-Bool v2 = True
    by (simp add: need-second-arg-def)
  moreover
  from ⟨normal s2⟩
  have assigns-if (the-Bool v2) e2 ⊆ dom (locals (store s2))
    by (rule BinOp.hyps [elim-format]) (simp add: True bool-e2)+
  with e1-s2
  have assigns-if (the-Bool v1) e1 ∪ assigns-if (the-Bool v2) e2

```

```

       $\subseteq \text{dom } (\text{locals } (\text{store } s2))$ 
    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  $\langle \neg (\text{binop} = \text{CondAnd} \vee \text{binop} = \text{CondOr}) \rangle$ 
  from BinOp.hyps
  have
    prg Env  $\vdash$  Norm s0  $\neg$  BinOp binop e1 e2  $\neg$  eval-binop binop v1 v2  $\rightarrow$  s2
    by  $\neg$  (rule eval.BinOp)
  moreover note  $\langle \text{normal } s2 \rangle$ 
  ultimately
  have assignsE (BinOp binop e1 e2)  $\subseteq \text{dom } (\text{locals } (\text{store } s2))$ 
    by (rule assignsE-good-approx)
  with False None
  show ?thesis
    by simp
qed
qed
next
  case Super
  note  $\langle \text{Env} \vdash \text{Super} :: \neg \text{PrimT Boolean} \rangle$ 
  hence False
    by cases simp
  thus ?case ..
next
  case (Acc s0 va w f s1)
  from  $\langle \text{prg Env} \vdash \text{Norm } s0 \neg va = \neg (v, f) \rightarrow s1 \rangle$  and  $\langle \text{normal } s1 \rangle$ 
  have assignsV va  $\subseteq \text{dom } (\text{locals } (\text{store } s1))$ 
    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  $\neg va := e \neg v \rightarrow \text{assign } f v s2$ 
    by  $\neg$  (rule eval.Ass)
  moreover note  $\langle \text{normal } (\text{assign } f v s2) \rangle$ 
  ultimately
  have assignsE (va := e)  $\subseteq \text{dom } (\text{locals } (\text{store } (\text{assign } f v s2)))$ 
    by (rule assignsE-good-approx)
  thus ?case by simp
next
  case (Cond s0 e0 b s1 e1 e2 v s2)
  from  $\langle \text{Env} \vdash e0 ? e1 : e2 :: \neg \text{PrimT Boolean} \rangle$ 
  obtain wt-e1:  $\text{Env} \vdash e1 :: \neg \text{PrimT Boolean}$  and
    wt-e2:  $\text{Env} \vdash e2 :: \neg \text{PrimT Boolean}$ 
    by cases (auto dest: widen-Boolean2)

```

```

note eval-e0 = ⟨prg Env ⊢ Norm s0 -e0-> b → s1⟩
have e0-s2: assignsE e0 ⊆ dom (locals (store s2))
proof -
  note eval-e0
  moreover
  from Cond.hyps and ⟨normal s2⟩ have normal s1
    by - (erule eval-no-abrupt-lemma [rule-format],simp)
  ultimately
  have assignsE e0 ⊆ dom (locals (store s1))
    by (rule assignsE-good-approx)
  also
  from Cond
  have ... ⊆ dom (locals (store s2))
    by - (erule dom-locals-eval-mono [elim-format],simp)
  finally show ?thesis .
qed
show ?case
proof (cases constVal e0)
  case None
  have assigns-if (the-Bool v) e1 ∩ assigns-if (the-Bool v) e2
    ⊆ dom (locals (store s2))
  proof (cases the-Bool b)
    case True
    from ⟨normal s2⟩
    have assigns-if (the-Bool v) e1 ⊆ dom (locals (store s2))
      by (rule Cond.hyps [elim-format]) (simp-all add: wt-e1 True)
    thus ?thesis
      by blast
  next
    case False
    from ⟨normal s2⟩
    have assigns-if (the-Bool v) e2 ⊆ dom (locals (store s2))
      by (rule Cond.hyps [elim-format]) (simp-all add: wt-e2 False)
    thus ?thesis
      by blast
  qed
with e0-s2
have assignsE e0 ∪
  (assigns-if (the-Bool v) e1 ∩ assigns-if (the-Bool v) e2)
  ⊆ dom (locals (store s2))
  by (rule Un-least)
with None show ?thesis
  by simp
next
  case (Some c)
  from this eval-e0 have eq-b-c: b=c
    by (rule constVal-eval-elim)
  show ?thesis
  proof (cases the-Bool c)
    case True
    from ⟨normal s2⟩
    have assigns-if (the-Bool v) e1 ⊆ dom (locals (store s2))
      by (rule Cond.hyps [elim-format]) (simp-all add: eq-b-c True wt-e1)
    with e0-s2
    have assignsE e0 ∪ assigns-if (the-Bool v) e1 ⊆ ...
      by (rule Un-least)
    with Some True show ?thesis
      by simp
  next

```

```

    case False
    from ⟨normal s2⟩
    have assigns-if (the-Bool v) e2 ⊆ dom (locals (store s2))
      by (rule Cond.hyps [elim-format]) (simp-all add: eq-b-c False wt-e2)
    with e0-s2
    have assignsE e0 ∪ assigns-if (the-Bool v) e2 ⊆ ...
      by (rule Un-least)
    with Some False show ?thesis
      by simp
  qed
qed
next
case (Call s0 e a s1 args vs s2 D mode statT mn pTs s3 s3' accC v s4)
hence
  prg Env ⊢ Norm s0 - ({accC, statT, mode} e · mn ( {pTs} args )) -> v →
    (set-lvars (locals (store s2)) s4)
  by - (rule eval.Call)
hence assignsE ({accC, statT, mode} e · mn ( {pTs} args ))
  ⊆ dom (locals (store ((set-lvars (locals (store s2))) s4)))
  using (normal ((set-lvars (locals (store s2))) s4))
  by (rule assignsE-good-approx)
thus ?case by simp
next
case Methd show ?case by simp
next
case Body show ?case by simp
qed simp+ — all the statements and variables
}
note generalized = this
from eval bool show ?thesis
  by (rule generalized [elim-format]) simp+
qed

```

```

lemma assigns-if-good-approx':
  assumes eval: G ⊢ s0 - e -> b → s1
    and normal: normal s1
    and bool: (⟦prg=G, cls=C, lcl=L⟧ ⊢ e :: - (PrimT Boolean))
  shows assigns-if (the-Bool b) e ⊆ dom (locals (store s1))
proof -
  from eval have prg (⟦prg=G, cls=C, lcl=L⟧ ⊢ s0 - e -> b → s1) by simp
  from this normal bool show ?thesis
    by (rule assigns-if-good-approx)
qed

```

```

lemma subset-Intl: A ⊆ 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  $\longrightarrow$  (nrm A  $\subseteq$  dom (locals (store s1))))  $\wedge$ 
  ( $\forall$  l. abrupt s1 = Some (Jump (Break l))  $\wedge$  normal s0
     $\longrightarrow$  (brk A l  $\subseteq$  dom (locals (store s1))))  $\wedge$ 
  (abrupt s1 = Some (Jump Ret)  $\wedge$  normal s0
     $\longrightarrow$  Result  $\in$  dom (locals (store s1)))
(is ?NormalAssigned s1 A  $\wedge$  ?BreakAssigned s0 s1 A  $\wedge$  ?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  $\vdash$  s0  $\dashv$  t  $\longrightarrow$  (v, s1) by simp
from G wf have wf: wf-prog G by simp
let ?HypObj =  $\lambda$  t s0 s1.
   $\forall$  Env T A. ?Wt Env t T  $\longrightarrow$  ?Da Env s0 t A  $\longrightarrow$  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  $\vdash$  dom (locals s)  $\gg$  t  $\gg$  A using Abrupt.premis 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  $\vdash$  dom (locals (store (Norm s)))  $\gg$  (Skip)  $\gg$  A
    using Skip.premis 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.premis
  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 = (prg Env = G)
  from Lab.premis
  obtain C l where
    da-c: Env  $\vdash$  dom (locals (snd (Norm s0)))  $\gg$  (c)  $\gg$  C and
    A: nrm A = nrm C  $\cap$  (brk C) l brk A = rmlab l (brk C) and

```

```

    j: j = Break l
  by - (erule da-elim-cases, simp)
from Lab.premis
have wt-c: Env ⊢ c :: √
  by - (erule wt-elim-cases, simp)
from wt-c da-c G and Lab.hyps
have norm-c: ?NormalAssigned s1 C and
  brk-c: ?BreakAssigned (Norm s0) s1 C and
  res-c: ?ResAssigned (Norm s0) s1
  by simp-all
have ?NormalAssigned (abupd (absorb j) s1) A
proof
  assume normal: normal (abupd (absorb j) s1)
  show nrm A ⊆ dom (locals (store (abupd (absorb j) s1)))
  proof (cases abrupt s1)
    case None
    with norm-c A
    show ?thesis
    by auto
  next
    case Some
    with normal j
    have abrupt s1 = Some (Jump (Break l))
    by (auto dest: absorb-Some-NoneD)
    with brk-c A
    show ?thesis
    by auto
  qed
qed
moreover
have ?BreakAssigned (Norm s0) (abupd (absorb j) s1) A
proof -
  {
    fix l'
    assume break: abrupt (abupd (absorb j) s1) = Some (Jump (Break l'))
    with j
    have l ≠ l'
    by (cases s1) (auto dest!: absorb-Some-JumpD)
    hence (rmlab l (brk C)) l' = (brk C) l'
    by (simp)
    with break brk-c A
    have
      (brk A l' ⊆ dom (locals (store (abupd (absorb j) s1))))
    by (cases s1) auto
  }
  then show ?thesis
  by simp
qed
moreover
from res-c have ?ResAssigned (Norm s0) (abupd (absorb j) s1)
  by (cases s1) (simp add: absorb-def)
ultimately show ?case by (intro conjI)
next
case (Comp s0 c1 s1 c2 s2 Env T A)
note G = ⟨prg Env = G⟩
from Comp.premis
obtain C1 C2
  where da-c1: Env ⊢ dom (locals (snd (Norm s0))) » ⟨c1⟩ C1 and
    da-c2: Env ⊢ nrm C1 » ⟨c2⟩ C2 and

```

```

      A: nrm A = nrm C2 brk A = (brk C1)  $\Rightarrow$   $\cap$  (brk C2)
    by (elim da-elim-cases) simp
  from Comp.premis
  obtain wt-c1: Env $\vdash$ c1:: $\surd$  and
    wt-c2: Env $\vdash$ c2:: $\surd$ 
  by (elim wt-elim-cases) simp
  note  $\langle$ PROP ?Hyp (In1r c1) (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 nrm C1  $\subseteq$  dom (locals (snd s1)) by iprover
    with da-c2 obtain C2'
      where da-c2': Env $\vdash$  dom (locals (snd s1))  $\gg$   $\langle$ c2 $\rangle$  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 fastsimp
    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 = ⟨prg Env = G⟩
with If.hyps have eval-e: prg Env ⊢ Norm s0 -e-⋗b→ s1 by simp
from If.prem
obtain E C1 C2 where
  da-e: Env ⊢ dom (locals (store ((Norm s0)::state))) »⟨e⟩ E and
  da-c1: Env ⊢ (dom (locals (store ((Norm s0)::state)))
    ∪ assigns-if True e) »⟨c1⟩ C1 and
  da-c2: Env ⊢ (dom (locals (store ((Norm s0)::state)))
    ∪ assigns-if False e) »⟨c2⟩ C2 and
  A: nrm A = nrm C1 ∩ nrm C2 brk A = brk C1 ⇒ ∩ brk C2
  by (elim da-elim-cases)
from If.prem
obtain
  wt-e: Env ⊢ e::- PrimT Boolean and
  wt-c1: Env ⊢ c1::√ and
  wt-c2: Env ⊢ c2::√
  by (elim wt-elim-cases)
from If.hyps have
  s0-s1: dom (locals (store ((Norm s0)::state))) ⊆ dom (locals (store s1))
  by (elim dom-locals-eval-mono-elim)
show ?case
proof (cases normal s1)
case True
note normal-s1 = this
show ?thesis
proof (cases the-Bool b)
case True
from eval-e normal-s1 wt-e
have assigns-if True e ⊆ dom (locals (store s1))
  by (rule assigns-if-good-approx [elim-format]) (simp add: True)
with s0-s1
have dom (locals (store ((Norm s0)::state))) ∪ assigns-if True e ⊆ ...
  by (rule Un-least)
with da-c1 obtain C1'
  where da-c1': Env ⊢ dom (locals (store s1)) »⟨c1⟩ C1' and
        nrm-c1: nrm C1 ⊆ nrm C1' and
        brk-c1: ∀ l. brk C1 l ⊆ brk C1' l
  by (rule da-weakenE) iprover
from If.hyps True have PROP ?Hyp (In1r c1) s1 s2 by simp
with wt-c1 da-c1'
obtain nrm-c1': ?NormalAssigned s2 C1' and
  brk-c1': ?BreakAssigned s1 s2 C1' and
  res-c1: ?ResAssigned s1 s2
  using G by simp
from nrm-c1' nrm-c1 A
have ?NormalAssigned s2 A
  by blast
moreover from brk-c1' brk-c1 A
have ?BreakAssigned s1 s2 A
  by fastsimp

```

```

with normal-s1
have ?BreakAssigned (Norm s0) s2 A by simp
moreover from res-c1 normal-s1 have ?ResAssigned (Norm s0) s2
  by simp
ultimately show ?thesis by (intro conjI)
next
case False
from eval-e normal-s1 wt-e
have assigns-if False e  $\subseteq$  dom (locals (store s1))
  by (rule assigns-if-good-approx [elim-format]) (simp add: False)
with s0-s1
have dom (locals (store ((Norm s0)::state)))  $\cup$  assigns-if False e  $\subseteq$  ...
  by (rule Un-least)
with da-c2 obtain C2'
  where da-c2': Env $\vdash$  dom (locals (store s1))  $\gg \langle c2 \rangle \gg C2'$  and
    nrm-c2: nrm C2  $\subseteq$  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 fastsimp
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  $\langle G \vdash s1 \text{ --(if the-Bool } b \text{ then } c1 \text{ else } c2) \rightarrow s2 \rangle$ 
  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: prg Env $\vdash$  Norm s0  $-e-\triangleright b \rightarrow s1$ 
  by (simp (no-asm-simp))
from Loop.prems
obtain E C where

```

```

da-e: Env ⊢ dom (locals (store ((Norm s0)::state))) »⟨e⟩ E and
da-c: Env ⊢ (dom (locals (store ((Norm s0)::state)))
  ∪ assigns-if True e) »⟨c⟩ C and
A: nrm A = nrm C ∩
  (dom (locals (store ((Norm s0)::state))) ∪ assigns-if False e)
brk A = brk C
by (elim da-elim-cases)
from Loop.prem
obtain
  wt-e: Env ⊢ e :: ¬PrimT Boolean and
  wt-c: Env ⊢ c :: √
by (elim wt-elim-cases)
from wt-e da-e G
obtain res-s1: ?ResAssigned (Norm s0) s1
by (elim Loop.hyps [elim-format]) simp+
from Loop.hyps have
  s0-s1: dom (locals (store ((Norm s0)::state))) ⊆ dom (locals (store s1))
by (elim dom-locals-eval-mono-elim)
show ?case
proof (cases normal s1)
  case True
  note normal-s1 = this
  show ?thesis
  proof (cases the-Bool b)
    case True
    with Loop.hyps obtain
      eval-c: G ⊢ s1 -c→ s2 and
      eval-while: G ⊢ abrupt (absorb (Cont l)) s2 -l• While(e) c→ s3
    by simp
    from Loop.hyps True
    have ?HypObj (In1r c) s1 s2 by simp
    note hyp-c = this [rule-format]
    from Loop.hyps True
    have ?HypObj (In1r (l• While(e) c)) (abrupt (absorb (Cont l)) s2) s3
    by simp
    note hyp-while = this [rule-format]
    from eval-e normal-s1 wt-e
    have assigns-if True e ⊆ dom (locals (store s1))
    by (rule assigns-if-good-approx [elim-format]) (simp add: True)
    with s0-s1
    have dom (locals (store ((Norm s0)::state))) ∪ assigns-if True e ⊆ ...
    by (rule Un-least)
    with da-c obtain C'
      where da-c': Env ⊢ dom (locals (store s1)) »⟨c⟩ C' and
        nrm-C-C': nrm C ⊆ nrm C' and
        brk-C-C': ∀ l. brk C l ⊆ brk C' l
      by (rule da-weakenE) iprover
    from hyp-c wt-c da-c'
    obtain nrm-C': ?NormalAssigned s2 C' and
      brk-C': ?BreakAssigned s1 s2 C' and
      res-s2: ?ResAssigned s1 s2
    using G by simp
    show ?thesis
    proof (cases normal s2 ∨ abrupt s2 = Some (Jump (Cont l)))
      case True
      from Loop.prem obtain
        wt-while: Env ⊢ In1r (l• While(e) c) :: T and
        da-while: Env ⊢ dom (locals (store ((Norm s0)::state)))
          »⟨l• While(e) c⟩ A

```

```

  by simp
have dom (locals (store ((Norm s0)::state)))
  ⊆ dom (locals (store (abupd (absorb (Cont l)) s2)))
proof -
  note s0-s1
  also from eval-c
  have dom (locals (store s1)) ⊆ dom (locals (store s2))
    by (rule dom-locals-eval-mono-elim)
  also have ... ⊆ dom (locals (store (abupd (absorb (Cont l)) s2)))
    by simp
  finally show ?thesis .
qed
with da-while obtain A'
  where
    da-while': Env⊢ dom (locals (store (abupd (absorb (Cont l)) s2)))
      »⟨l• While(e) c⟩» A'
    and nrm-A-A': nrm A ⊆ nrm A'
    and brk-A-A': ∀ l. brk A l ⊆ brk A' l
    by (rule da-weakenE) simp
with wt-while hyp-while
obtain nrm-A': ?NormalAssigned s3 A' and
  brk-A': ?BreakAssigned (abupd (absorb (Cont l)) s2) s3 A' and
  res-s3: ?ResAssigned (abupd (absorb (Cont l)) s2) s3
  using G by simp
from nrm-A-A' nrm-A'
have ?NormalAssigned s3 A
  by blast
moreover
have ?BreakAssigned (Norm s0) s3 A
proof -
  from brk-A-A' brk-A'
  have ?BreakAssigned (abupd (absorb (Cont l)) s2) s3 A
    by fastsimp
  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 fastsimp
moreover
from eq-s3-s2 brk-C-C' brk-C' normal-s1 A
have ?BreakAssigned (Norm s0) s3 A
  by fastsimp
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  $\subseteq$  dom (locals (store s1))
  by (rule assigns-if-good-approx [elim-format]) (simp add: False)
with s0-s1
have dom (locals (store ((Norm s0)::state)))  $\cup$  assigns-if False e  $\subseteq$  ...
  by (rule Un-least)
hence nrm C  $\cap$ 
  (dom (locals (store ((Norm s0)::state)))  $\cup$  assigns-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:  $\bigwedge j. \text{abrupt } s1 \neq \text{Some } (\text{Jump } j)$ 
  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:  $G \vdash \text{abupd } (\text{absorb } (\text{Cont } l)) \ s2 \rightarrow l \cdot \text{While}(e) \ c \rightarrow s3$ 
  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 (Jump s j Env T A)
have ?NormalAssigned (Some (Jump j),s) A by simp
moreover
from Jump.prem
obtain ret:  $j = \text{Ret} \longrightarrow \text{Result} \in \text{dom} (\text{locals} (\text{store} (\text{Norm } s)))$  and
      brk:  $\text{brk } A = (\text{case } j \text{ of}$ 
        Break  $l \Rightarrow \lambda k. \text{if } k=l$ 
          then  $\text{dom} (\text{locals} (\text{store} ((\text{Norm } s)::\text{state})))$ 
        else UNIV
        | Cont  $l \Rightarrow \lambda k. \text{UNIV}$ 
        | Ret  $\Rightarrow \lambda k. \text{UNIV}$ )
    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  $G = \langle \text{prg } \text{Env} = G \rangle$ 
from Throw.prem obtain E where
  da-e:  $\text{Env} \vdash \text{dom} (\text{locals} (\text{store} ((\text{Norm } s0)::\text{state}))) \gg \langle e \rangle \gg E$ 
  by (elim da-elim-cases)
from Throw.prem
obtain eT where wt-e:  $\text{Env} \vdash e :: -eT$ 
  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:  $\text{prg } \text{Env} \vdash \text{Norm } s0 -e \multimap a \rightarrow s1$ 
    by (simp (no-asm-simp))
  from eval-e - wt-e
  have  $\bigwedge l. \text{abrupt } s1 \neq \text{Some } (\text{Jump } (\text{Break } l))$ 
    by (rule eval-expression-no-jump) (simp-all add: wf G)
  hence  $\bigwedge l. \text{abrupt } (\text{abupd } (\text{throw } a) s1) \neq \text{Some } (\text{Jump } (\text{Break } l))$ 
    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  $G = \langle \text{prg } \text{Env} = G \rangle$ 
from Try.prem obtain C1 C2 where
  da-c1:  $\text{Env} \vdash \text{dom} (\text{locals} (\text{store} ((\text{Norm } s0)::\text{state}))) \gg \langle c1 \rangle \gg C1$  and
  da-c2:
     $\text{Env} \vdash \text{lcl} := \text{lcl } \text{Env} (\text{VName } vn \mapsto \text{Class } C)$ 

```

```

  ⊢ (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 salloc: prg Env ⊢ s1 -salloc→ s2 using Try.hyps G
  by (simp (no-asm-simp))
note ⟨PROP ?Hyp (In1r c1) (Norm s0) s1⟩
with wt-c1 da-c1 G
obtain nrm-C1: ?NormalAssigned s1 C1 and
  brk-C1: ?BreakAssigned (Norm s0) s1 C1 and
  res-s1: ?ResAssigned (Norm s0) s1
  by simp
show ?case
proof (cases normal s1)
  case True
  with nrm-C1 have nrm C1 ∩ nrm C2 ⊆ dom (locals (store s1))
    by auto
  moreover
  have s3=s1
  proof -
    from salloc True have eq-s2-s1: s2=s1
      by (cases s1) (auto elim: salloc-elim-cases)
    with True have ⊢ G, s2 ⊢ catch C
      by (simp add: catch-def)
    with Try.hyps have s3=s2
      by simp
    with eq-s2-s1 show ?thesis by simp
  qed
  ultimately show ?thesis
    using True A res-s1 by simp
next
  case False
  note not-normal-s1 = this
  show ?thesis
  proof (cases ∃ l. abrupt s1 = Some (Jump (Break l)))
    case True
    then obtain l where l: abrupt s1 = Some (Jump (Break l))
      by auto
    with brk-C1 have (brk C1 ⇒ ∩ brk C2) l ⊆ dom (locals (store s1))
      by auto
    moreover have s3=s1
    proof -
      from salloc l have eq-s2-s1: s2=s1
        by (cases s1) (auto elim: salloc-elim-cases)
      with l have ⊢ G, s2 ⊢ catch C
        by (simp add: catch-def)
      with Try.hyps have s3=s2
        by simp
      with eq-s2-s1 show ?thesis by simp
    qed
    ultimately show ?thesis
      using l A res-s1 by simp
  next
    case False
    note abrupt-no-break = this
    show ?thesis

```

```

proof (cases G,s2 ⊢ catch C)
  case True
  with Try.hyps have ?HypObj (In1r c2) (new-xcpt-var vn s2) s3
    by simp
  note hyp-c2 = this [rule-format]
  have (dom (locals (store ((Norm s0)::state))) ∪ {VName vn})
    ⊆ dom (locals (store (new-xcpt-var vn s2)))
  proof –
    from ⟨G ⊢ Norm s0 – c1 → s1⟩
    have dom (locals (store ((Norm s0)::state)))
      ⊆ dom (locals (store s1))
      by (rule dom-locals-eval-mono-elim)
    also
    from xalloc
    have ... ⊆ dom (locals (store s2))
      by (rule dom-locals-xalloc-mono)
    also
    have ... ⊆ dom (locals (store (new-xcpt-var vn s2)))
      by (cases s2) (simp add: new-xcpt-var-def, blast)
    also
    have {VName vn} ⊆ ...
      by (cases s2) simp
    ultimately show ?thesis
      by (rule Un-least)
  qed
with da-c2
obtain C2' where
  da-C2': Env(lcl := lcl Env(VName vn ↦ Class C))
    ⊢ dom (locals (store (new-xcpt-var vn s2))) » ⟨c2⟩ » C2'
  and nrm-C2': nrm C2 ⊆ nrm C2'
  and brk-C2': ∀ l. brk C2 l ⊆ brk C2' l
    by (rule da-weakenE) simp
from wt-c2 da-C2' G and hyp-c2
obtain nrmAss-C2: ?NormalAssigned s3 C2' and
  brkAss-C2: ?BreakAssigned (new-xcpt-var vn s2) s3 C2' and
  resAss-s3: ?ResAssigned (new-xcpt-var vn s2) s3
    by simp
from nrmAss-C2 nrm-C2' A
have ?NormalAssigned s3 A
    by auto
moreover
have ?BreakAssigned (Norm s0) s3 A
proof –
  from brkAss-C2 have ?BreakAssigned (Norm s0) s3 C2'
    by (cases s2) (auto simp add: new-xcpt-var-def)
  with brk-C2' A show ?thesis
    by fastsimp
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 ¬ normal s2
  ∀ l. abrupt s2 ≠ Some (Jump (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 abrupt s2 ≠ Some (Jump 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) ⇒ ∪∇ (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 fastsimp
ultimately
show ?thesis
  using that resAss-s2' by simp
qed
note s3 = ⟨s3 = (if ∃ err. x1 = Some (Error err) then (x1, s1)
  else abrupt (abrupt-if (x1 ≠ None) x1) s2)⟩
have s1-s2: dom (locals s1) ⊆ dom (locals (store s2))
proof -
  from ⟨G ⊢ Norm s1 -c2 → s2⟩
  show ?thesis
    by (rule dom-locals-eval-mono-elim) simp
qed

have ?NormalAssigned s3 A
proof
  assume normal-s3: normal s3
  show nrm A ⊆ dom (locals (snd s3))
  proof -
    have nrm C1 ⊆ dom (locals (snd s3))
    proof -
      from normal-s3 s3
      have normal (x1, s1)
      by (cases s2) (simp add: abrupt-if-def)
      with normal-s3 nrmAss-C1 s3 s1-s2
      show ?thesis
      by fastsimp
    qed
    moreover
    have nrm C2 ⊆ dom (locals (snd s3))
    proof -
      from normal-s3 s3
      have normal s2
      by (cases s2) (simp add: abrupt-if-def)
      with normal-s3 nrmAss-C2 s3 s1-s2
      show ?thesis
      by fastsimp
    qed
    ultimately have nrm C1 ∪ nrm C2 ⊆ ...
    by (rule Un-least)
    with nrm-A show ?thesis
    by simp
  qed
qed
moreover
{
  fix l assume brk-s3: abrupt s3 = Some (Jump (Break l))
  have brk A l ⊆ dom (locals (store s3))
  proof (cases normal s2)
    case True
    with brk-s3 s3
    have s2-s3: dom (locals (store s2)) ⊆ dom (locals (store s3))
    by simp
  }

```

```

have brk C1 l ⊆ dom (locals (store s3))
proof -
  from True brk-s3 s3 have x1=Some (Jump (Break l))
    by (cases s2) (simp add: abrupt-if-def)
  with brkAss-C1 s1-s2 s2-s3
  show ?thesis
    by simp
qed
moreover from True nrmAss-C2 s2-s3
have nrm C2 ⊆ dom (locals (store s3))
  by - (rule subset-trans, simp-all)
ultimately
have ((brk C1) ⇒∪ (nrm C2)) l ⊆ ...
  by blast
with brk-A show ?thesis
  by simp blast
next
case False
note not-normal-s2 = this
have s3=s2
proof (cases normal (x1,s1))
  case True with not-normal-s2 s3 show ?thesis
    by (cases s2) (simp add: abrupt-if-def)
next
  case False with not-normal-s2 s3 brk-s3 show ?thesis
    by (cases s2) (simp add: abrupt-if-def)
qed
with brkAss-C2 brk-s3
have brk C2 l ⊆ dom (locals (store s3))
  by simp
with brk-A show ?thesis
  by simp blast
qed
}
hence ?BreakAssigned (Norm s0) s3 A
  by simp
moreover
{
  assume abr-s3: abrupt s3 = Some (Jump Ret)
  have Result ∈ dom (locals (store s3))
  proof (cases x1 = Some (Jump Ret))
    case True
    note ret-x1 = this
    with resAss-s1 have res-s1: Result ∈ dom (locals s1)
      by simp
    moreover have dom (locals (store ((Norm s1)::state)))
      ⊆ dom (locals (store s2))
      by (rule dom-locals-eval-mono-elim) (rule Fin.hyps)
    ultimately have Result ∈ dom (locals (store s2))
      by - (rule subsetD, auto)
    with res-s1 s3 show ?thesis
      by simp
  next
  case False
  with s3 abr-s3 obtain abrupt s2 = Some (Jump Ret) and s3=s2
    by (cases s2) (simp add: abrupt-if-def)
  with resAss-s2 show ?thesis
    by simp
qed
}

```

```

}
hence ?ResAssigned (Norm s0) s3
  by simp
ultimately show ?case by (intro conjI)
next
case (Init C c s0 s3 s1 s2 Env T A)
note G = ⟨prg Env = G⟩
from Init.hyps
have eval: prg Env ⊢ Norm s0 → Init C → s3
  apply (simp only: G)
  apply (rule eval.Init, assumption)
  apply (cases inited C (globs s0) )
  apply simp
  apply (simp only: if-False )
  apply (elim conjE, intro conjI, assumption+, simp)
done
from Init.premis and ⟨the (class G C) = c⟩
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 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.premis
  obtain A: nrm A = dom (locals (store ((Norm s0)::state)))
    brk A = (λ l. UNIV)
    by (elim da-elim-cases) simp
  from wf NewC.premis
  have wt-init: Env ⊢ (Init C)::√
    by (elim wt-elim-cases) (drule is-acc-classD, simp)
  have dom (locals (store ((Norm s0)::state))) ⊆ dom (locals (store s2))
  proof -
    have dom (locals (store ((Norm s0)::state))) ⊆ dom (locals (store s1))
      by (rule dom-locals-eval-mono-elim) (rule NewC.hyps)
    also
    have ... ⊆ dom (locals (store s2))
      by (rule dom-locals-halloc-mono) (rule NewC.hyps)
    finally show ?thesis .
  qed
  with A have ?NormalAssigned s2 A
    by simp
  moreover
  {
    fix j have abrupt s2 ≠ Some (Jump j)
    proof -
      have eval: prg Env ⊢ Norm s0 -NewC C-⤵ Addr a → s2
        unfolding G by (rule eval.NewC NewC.hyps)+
      from NewC.premis
      obtain T' where T = Inl T'
        by (elim wt-elim-cases) simp
      with NewC.premis 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.premis obtain
    da-e: Env ⊢ dom (locals (store ((Norm s0)::state))) » ⟨e⟩ » A
    by (elim da-elim-cases)
  from NewA.premis 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))  $\subseteq$  dom (locals (store s3))
proof –
  have dom (locals (store s2))
     $\subseteq$  dom (locals (store (abupd (check-neg i) s2)))
    by (simp)
  also have  $\dots \subseteq$  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  $\subseteq$  dom (locals (store s3))
  proof –
    from halloc normal-s3
    have normal (abupd (check-neg i) s2)
      by cases simp-all
    hence normal s2
      by (cases s2) simp
    with nrmAss-A' nrm-A-A' s2-s3 show ?thesis
      by blast
  qed
qed
moreover
{
  fix j have abrupt s3  $\neq$  Some (Jump j)
  proof –
    have eval: prg Env  $\vdash$  Norm s0  $\text{--New elT}[e] \text{--}$  Addr a  $\rightarrow$  s3
      unfolding G by (rule eval.NewA NewA.hyps) +
    from NewA.prems
    obtain T' where T = Inl T'
      by (elim wt-elim-cases) simp
    with NewA.prems have Env  $\vdash$  New elT[e]::  $\text{--}$  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 =  $\langle$ prg Env = G $\rangle$ 
from Cast.prems obtain
  da-e: Env  $\vdash$  dom (locals (store ((Norm s0)::state)))  $\gg$   $\langle$ e $\rangle$  A
  by (elim da-elim-cases)
from Cast.prems obtain eT where
  wt-e: Env  $\vdash$  e:: $\text{--}$ eT
  by (elim wt-elim-cases)
note  $\langle$ PROP ?Hyp (Inl e) (Norm s0) s1 $\rangle$ 
with wt-e da-e G obtain
  nrmAss-A: ?NormalAssigned s1 A and
  brkAss-A: ?BreakAssigned (Norm s0) s1 A

```

```

    by simp
  note s2 = ⟨s2 = abupd (raise-if (¬ G, snd s1 ⊢ v fits cT) ClassCast) s1⟩
  hence s1-s2: dom (locals (store s1)) ⊆ dom (locals (store s2))
    by simp
  have ?NormalAssigned s2 A
  proof
    assume normal s2
    with s2 have normal s1
      by (cases s1) simp
    with nrmAss-A s1-s2
    show nrm A ⊆ dom (locals (store s2))
      by blast
  qed
  moreover
  {
    fix j have abrupt s2 ≠ Some (Jump j)
    proof -
      have eval: prg Env ⊢ Norm s0 -Cast cT e-⤵ v → s2
        unfolding G by (rule eval.Cast Cast.hyps)+
      from Cast.premis
      obtain T' where T=Inl T'
        by (elim wt-elim-cases) simp
      with Cast.premis 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 (Inl 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 ⊆ 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)))
      ∪ (assigns-if True (BinOp CondAnd e1 e2) ∩
        assigns-if False (BinOp CondAnd e1 e2))
    by (elim da-elim-cases) (simp-all add: CondAnd)
  from T BinOp.premis CondAnd
  have Env ⊢ BinOp binop e1 e2 :: - PrimT Boolean
  by (simp)
  with eval normal-s2
  have ass-if: assigns-if (the-Bool (eval-binop binop v1 v2))
    (BinOp binop e1 e2)
    ⊆ dom (locals (store s2))
  by (rule assigns-if-good-approx)
  have (assigns-if True (BinOp CondAnd e1 e2) ∩
    assigns-if False (BinOp CondAnd e1 e2)) ⊆ ...
  proof (cases the-Bool (eval-binop binop v1 v2))
  case True
  with ass-if CondAnd

```

```

    have assigns-if True (BinOp CondAnd e1 e2)
       $\subseteq \text{dom} (\text{locals} (\text{store } s2))$ 
    by simp
    thus ?thesis by blast
  next
    case False
    with ass-if CondAnd
    have assigns-if False (BinOp CondAnd e1 e2)
       $\subseteq \text{dom} (\text{locals} (\text{store } s2))$ 
    by (simp only: False)
    thus ?thesis by blast
  qed
  with s0-s2
  have dom (locals (store ((Norm s0)::state)))
     $\cup (\text{assigns-if True (BinOp CondAnd e1 e2)} \cap \text{assigns-if False (BinOp CondAnd e1 e2)}) \subseteq \dots$ 
    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)))
         $\cup (\text{assigns-if True (BinOp CondOr e1 e2)} \cap \text{assigns-if False (BinOp CondOr e1 e2)})$ 
    by (elim da-elim-cases) (simp-all add: CondOr)
    from T BinOp.premis CondOr
    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)
       $\subseteq \text{dom} (\text{locals} (\text{store } s2))$ 
    by (rule assigns-if-good-approx)
    have (assigns-if True (BinOp CondOr e1 e2)  $\cap$ 
      assigns-if False (BinOp CondOr e1 e2))  $\subseteq \dots$ 
    proof (cases the-Bool (eval-binop binop v1 v2))
      case True
      with ass-if CondOr
      have assigns-if True (BinOp CondOr e1 e2)
         $\subseteq \text{dom} (\text{locals} (\text{store } s2))$ 
      by (simp)
      thus ?thesis by blast
    next
      case False
      with ass-if CondOr
      have assigns-if False (BinOp CondOr e1 e2)
         $\subseteq \text{dom} (\text{locals} (\text{store } s2))$ 
      by (simp)
      thus ?thesis by blast
    qed
  with s0-s2
  have dom (locals (store ((Norm s0)::state)))
     $\cup (\text{assigns-if True (BinOp CondOr e1 e2)} \cap \text{assigns-if False (BinOp CondOr e1 e2)}) \subseteq \dots$ 

```

```

    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
{
  fix j have abrupt s2 ≠ Some (Jump j)
  proof -
    from BinOp.premis T
    have Env⊢In1l (BinOp binop e1 e2)::In1l (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  $\langle G \vdash Norm\ s0 \rightarrow v \Rightarrow (w, upd) \rightarrow s1 \rangle$ 
have  $s1 = Norm\ s0$ 
  by (simp only: vn) (elim eval-elim-cases,simp)
ultimately show ?thesis by simp
next
case False
note  $G = \langle prg\ Env = G \rangle$ 
from False Acc.prems
have  $da-v: Env \vdash dom\ (locals\ (store\ ((Norm\ s0)::state))) \gg \langle v \rangle \gg A$ 
  by (elim da-elim-cases) simp-all
from Acc.prems obtain  $vT$  where
   $wt-v: Env \vdash v::=vT$ 
  by (elim wt-elim-cases)
note  $\langle PROP\ ?Hyp\ (In2\ v)\ (Norm\ s0)\ s1 \rangle$ 
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  $G = \langle prg\ Env = G \rangle$ 
from Ass.prems obtain  $varT\ eT$  where
   $wt-var: Env \vdash var::=varT$  and
   $wt-e: Env \vdash e::=eT$ 
  by (elim wt-elim-cases) simp
have  $eval-var: prg\ Env \vdash Norm\ s0 \rightarrow var \Rightarrow (w, upd) \rightarrow s1$ 
  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  $hyp-var = \langle PROP\ ?Hyp\ (In2\ var)\ (Norm\ s0)\ s1 \rangle$ 
  note  $hyp-e = \langle PROP\ ?Hyp\ (In1l\ e)\ s1\ s2 \rangle$ 
  show  $nrm\ A \subseteq dom\ (locals\ (store\ (assign\ upd\ v\ s2)))$ 
  proof (cases  $\exists\ vn.\ var = LVar\ vn$ )
    case True
    then obtain  $vn$  where  $var = LVar\ vn..$ 
    from Ass.prems obtain  $E$  where
       $da-e: Env \vdash dom\ (locals\ (store\ ((Norm\ s0)::state))) \gg \langle e \rangle \gg E$  and
       $nrm-A: nrm\ A = nrm\ E \cup \{vn\}$ 
      by (elim da-elim-cases) (insert vn,auto)
    obtain  $E'$  where
       $da-e': Env \vdash dom\ (locals\ (store\ s1)) \gg \langle e \rangle \gg E'$  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:  $G \vdash \text{Norm } s0 \text{ --LVar } vn \Rightarrow (w, \text{upd}) \rightarrow s1$ 
  by simp
then have upd:  $\text{upd} = \text{snd } (\text{lvar } vn \text{ (store } s1))$ 
  by cases (cases lvar vn (store s1), simp)
have nrm  $E \subseteq \text{dom } (\text{locals } (\text{store } (\text{assign upd } v \ s2)))$ 
proof -
  from hyp-e wt-e da-e' G normal-s2
  have nrm  $E' \subseteq \text{dom } (\text{locals } (\text{store } s2))$ 
    by simp
  also
  from upd
  have  $\text{dom } (\text{locals } (\text{store } s2)) \subseteq \text{dom } (\text{locals } (\text{store } (\text{upd } v \ s2)))$ 
    by (simp add: lvar-def) blast
  hence  $\text{dom } (\text{locals } (\text{store } s2)) \subseteq \text{dom } (\text{locals } (\text{store } (\text{assign upd } v \ s2)))$ 
    by (rule dom-locals-assign-mono)
  finally
  show ?thesis using E-E'
    by blast
qed
moreover
from upd normal-s2
have  $\{vn\} \subseteq \text{dom } (\text{locals } (\text{store } (\text{assign upd } v \ s2)))$ 
  by (auto simp add: assign-def Let-def lvar-def upd split: split-split)
ultimately
show  $\text{nrm } A \subseteq \dots$ 
  by (rule Un-least [elim-format]) (simp add: nrm-A)
next
case False
from Ass.premis 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) (insert False, simp+)
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
{
  fix j have abrupt (assign upd v s2) ≠ Some (Jump j)
  proof -
    have eval: prg Env ⊢ Norm s0 -var:=e-⋗v→ (assign upd v s2)
      by (simp only: G) (rule eval.Ass Ass.hyps)+
    from Ass.premss
    obtain T' where T=Inl T'
      by (elim wt-elim-cases) simp
    with Ass.premss have Env ⊢ var:=e::- T' by simp
    from eval - this
    show ?thesis
      by (rule eval-expression-no-jump) (simp-all add: G wf)
  qed
}
hence ?BreakAssigned (Norm s0) (assign upd v s2) A
  and ?ResAssigned (Norm s0) (assign upd v s2)
  by simp-all
ultimately show ?case by (intro conjI)
next
case (Cond s0 e0 b s1 e1 e2 v s2 Env T A)
note G = ⟨prg Env = G⟩
have ?NormalAssigned s2 A
proof
  assume normal-s2: normal s2
  show nrm A ⊆ dom (locals (store s2))
  proof (cases Env ⊢ (e0 ? e1 : e2)::-(PrimT Boolean))
    case True
    with Cond.premss
    have nrm-A: nrm A = dom (locals (store ((Norm s0)::state)))
      ∪ (assigns-if True (e0 ? e1 : e2) ∩
        assigns-if False (e0 ? e1 : e2))
    by (elim da-elim-cases) simp-all
    have eval: prg Env ⊢ Norm s0 -(e0 ? e1 : e2)-⋗v→ s2
      unfolding G by (rule eval.Cond Cond.hyps)+
    from eval
    have dom (locals (store ((Norm s0)::state))) ⊆ dom (locals (store s2))
      by (rule dom-locals-eval-mono-elim)
    moreover
    from eval normal-s2 True
    have ass-if: assigns-if (the-Bool v) (e0 ? e1 : e2)
      ⊆ dom (locals (store s2))
    by (rule assigns-if-good-approx)
    have assigns-if True (e0 ? e1:e2) ∩ assigns-if False (e0 ? e1:e2)
      ⊆ dom (locals (store s2))
    proof (cases the-Bool v)
      case True
      from ass-if
      have assigns-if True (e0 ? e1:e2) ⊆ dom (locals (store s2))
        by (simp only: True)
      thus ?thesis by blast
    next
    case False
    from ass-if
    have assigns-if False (e0 ? e1:e2) ⊆ dom (locals (store s2))
      by (simp only: False)
    thus ?thesis by blast
  qed
  qed
next
case False
from ass-if
have assigns-if False (e0 ? e1:e2) ⊆ dom (locals (store s2))
  by (simp only: False)
thus ?thesis by blast
qed

```



```

ultimately show  $nrm\ A \subseteq dom\ (locals\ (store\ s2))$ 
  by (simp only:  $nrm\text{-}A$ ) (rule Un-least)
next
case False
with Cond.premis obtain  $E1\ E2$  where
  da-e1:  $Env \vdash (dom\ (locals\ (store\ ((Norm\ s0)::state)))$ 
     $\cup assigns\text{-}if\ True\ e0) \gg \langle e1 \rangle \gg E1$  and
  da-e2:  $Env \vdash (dom\ (locals\ (store\ ((Norm\ s0)::state)))$ 
     $\cup assigns\text{-}if\ False\ e0) \gg \langle e2 \rangle \gg E2$  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:  $Env \vdash e0 :: \neg PrimT\ Boolean$  and
  wt-e1:  $Env \vdash e1 :: \neg e1T$  and
  wt-e2:  $Env \vdash e2 :: \neg e2T$ 
  by (elim wt-elim-cases)
have  $s0\text{-}s1$ :  $dom\ (locals\ (store\ ((Norm\ s0)::state)))$ 
   $\subseteq dom\ (locals\ (store\ s1))$ 
  by (rule dom-locals-eval-mono-elim) (rule Cond.hyps)
have eval-e0:  $prg\ Env \vdash Norm\ s0 \text{ -- } e0 \text{ -- } \triangleright b \rightarrow s1$ 
  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  $assigns\text{-}if\ True\ e0 \subseteq dom\ (locals\ (store\ s1))$ 
  by (rule assigns-if-good-approx [elim-format]) (simp only: True)
with  $s0\text{-}s1$ 
have  $dom\ (locals\ (store\ ((Norm\ s0)::state)))$ 
   $\cup assigns\text{-}if\ True\ e0 \subseteq \dots$ 
  by (rule Un-least)
with da-e1 obtain  $E1'$  where
  da-e1':  $Env \vdash dom\ (locals\ (store\ s1)) \gg \langle e1 \rangle \gg E1'$  and
  nrm-E1-E1':  $nrm\ E1 \subseteq nrm\ E1'$ 
  by (rule da-weakenE) iprover
ultimately have  $nrm\ E1' \subseteq dom\ (locals\ (store\ s2))$ 
  using wt-e1  $G$  normal-s2 by simp
with nrm-E1-E1' show ?thesis
  by (simp only:  $nrm\text{-}A$ ) 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  $assigns\text{-}if\ False\ e0 \subseteq dom\ (locals\ (store\ s1))$ 
  by (rule assigns-if-good-approx [elim-format]) (simp only: False)
with  $s0\text{-}s1$ 
have  $dom\ (locals\ (store\ ((Norm\ s0)::state)))$ 
   $\cup assigns\text{-}if\ False\ e0 \subseteq \dots$ 
  by (rule Un-least)
with da-e2 obtain  $E2'$  where
  da-e2':  $Env \vdash dom\ (locals\ (store\ s1)) \gg \langle e2 \rangle \gg E2'$  and
  nrm-E2-E2':  $nrm\ E2 \subseteq nrm\ E2'$ 
  by (rule da-weakenE) iprover
ultimately have  $nrm\ E2' \subseteq dom\ (locals\ (store\ s2))$ 

```

```

    using wt-e2 G normal-s2 by simp
  with nrm-E2-E2' show ?thesis
    by (simp only: nrm-A) blast
qed
qed
qed
moreover
{
  fix j have abrupt s2  $\neq$  Some (Jump j)
  proof -
    have eval:  $\text{prg Env} \vdash \text{Norm } s0 \dashv e0 \text{ ? } e1 : e2 \dashv v \rightarrow s2$ 
      unfolding G by (rule eval.Cond Cond.hyps)+
    from Cond.premis
    obtain T' where T = Inl T'
      by (elim wt-elim-cases) simp
    with Cond.premis have  $\text{Env} \vdash e0 \text{ ? } e1 : e2 :: \dashv T'$  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 =  $\langle \text{prg Env} = G \rangle$ 
have ?NormalAssigned (restore-lvars s2 s4) A
  proof
    assume normal-restore-lvars: normal (restore-lvars s2 s4)
    show nrm A  $\subseteq$  dom (locals (store (restore-lvars s2 s4)))
    proof -
      from Call.premis 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:  $\text{Env} \vdash \text{nrm } E \gg \langle \text{args} \rangle \gg A$ 
      by (elim da-elim-cases)
      from Call.premis obtain eT argsT where
        wt-e:  $\text{Env} \vdash e :: \dashv eT$  and
        wt-args:  $\text{Env} \vdash \text{args} :: \dashv \text{argsT}$ 
      by (elim wt-elim-cases)
      note s3 =  $\langle s3 = \text{init-lvars } G D (\text{name} = \text{mn}, \text{parTs} = \text{pTs}) \text{ mode } a \text{ vs } s2 \rangle$ 
      note s3' =  $\langle s3' = \text{check-method-access } G \text{ accC statT mode} \\ (\text{name} = \text{mn}, \text{parTs} = \text{pTs}) \text{ a } s3 \rangle$ 
      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  $\langle \text{PROP ?Hyp (In1 l e) (Norm s0) s1} \rangle$ 
      with da-e wt-e G normal-s1

```

```

have  $nrm\ E \subseteq dom\ (locals\ (store\ s1))$ 
  by simp
with da-args obtain  $A'$  where
   $da-args': Env \vdash dom\ (locals\ (store\ s1)) \gg \langle args \rangle \gg A'$  and
   $nrm-A-A': nrm\ A \subseteq nrm\ A'$ 
  by (rule da-weakenE) iprover
note  $\langle PROP\ ?Hyp\ (In3\ args)\ s1\ s2 \rangle$ 
with da-args' wt-args  $G$  normal-s2
have  $nrm\ A' \subseteq dom\ (locals\ (store\ s2))$ 
  by simp
with nrm-A-A' have  $nrm\ A \subseteq dom\ (locals\ (store\ s2))$ 
  by blast
also have  $\dots \subseteq dom\ (locals\ (store\ (restore-lvars\ s2\ s4)))$ 
  by (cases  $s4$ ) simp
finally show ?thesis .
qed
qed
moreover
{
  fix  $j$  have  $abrupt\ (restore-lvars\ s2\ s4) \neq Some\ (Jump\ j)$ 
  proof –
    have  $eval: prg\ Env \vdash Norm\ s0 - (\{accC, statT, mode\} e \cdot mn(\{pTs\} args)) \multimap v$ 
       $\rightarrow (restore-lvars\ s2\ s4)$ 
    unfolding  $G$  by (rule eval.Call Call)+
    from Call.prems
    obtain  $T'$  where  $T = Inl\ T'$ 
    by (elim wt-elim-cases) simp
    with Call.prems have  $Env \vdash (\{accC, statT, mode\} e \cdot mn(\{pTs\} args)) :: -T'$ 
    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  $G = \langle prg\ Env = G \rangle$ 
from Methd.prems obtain  $m$  where
   $m: methd\ (prg\ Env)\ D\ sig = Some\ m$  and
   $da-body: Env \vdash (dom\ (locals\ (store\ ((Norm\ s0)::state))))$ 
     $\gg \langle Body\ (declclass\ m)\ (stmt\ (mbody\ (methd\ m))) \rangle \gg A$ 
  by – (erule da-elim-cases)
from Methd.prems  $m$  obtain
  isCls: is-class (prg  $Env$ )  $D$  and
   $wt-body: Env \vdash In1l\ (Body\ (declclass\ m)\ (stmt\ (mbody\ (methd\ m)))) :: T$ 
  by – (erule wt-elim-cases, simp)
note  $\langle PROP\ ?Hyp\ (In1l\ (body\ G\ D\ sig))\ (Norm\ s0)\ s1 \rangle$ 
moreover
from wt-body have  $Env \vdash In1l\ (body\ G\ D\ sig) :: T$ 
  using isCls  $m\ G$  by (simp add: body-def2)
moreover
from da-body have  $Env \vdash (dom\ (locals\ (store\ ((Norm\ s0)::state))))$ 
   $\gg \langle body\ G\ D\ sig \rangle \gg A$ 
  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
  qed

```

```

note  $\langle PROP \ ?Hyp \ (In1l \ e) \ s1 \ s2 \rangle$ 
with  $da-e' \ wt-e \ G \ normal-s2$ 
have  $nrm \ A' \subseteq dom \ (locals \ (store \ s2))$ 
  by simp
with  $nrm-A-A' \ \mathbf{have} \ nrm \ A \subseteq dom \ (locals \ (store \ s2))$ 
  by blast
also have  $\dots \subseteq dom \ (locals \ (store \ s3))$ 
proof –
  from fvar have  $s2' = snd \ (fvar \ statDeclC \ stat \ fn \ a \ s2)$ 
  by  $(cases \ fvar \ statDeclC \ stat \ fn \ a \ s2) \ simp$ 
  hence  $dom \ (locals \ (store \ s2)) \subseteq dom \ (locals \ (store \ s2'))$ 
  by  $(simp) \ (rule \ dom-locals-fvar-mono)$ 
  also from s3 have  $\dots \subseteq dom \ (locals \ (store \ s3))$ 
  by  $(cases \ s2') \ (simp \ add: \ check-field-access-def \ Let-def)$ 
  finally show ?thesis .
qed
finally show ?thesis .
qed
moreover
{
  fix j have  $abrupt \ s3 \neq Some \ (Jump \ j)$ 
  proof –
    obtain  $w \ upd \ \mathbf{where} \ v: (w, upd) = v$ 
    by  $(cases \ v) \ auto$ 
    have  $eval: prg \ Env \vdash Norm \ s0 - (\{accC, statDeclC, stat\} e..fn) = \succ (w, upd) \rightarrow s3$ 
    by  $(simp \ only: \ G \ v) \ (rule \ eval.FVar \ FVar.hyps) +$ 
    from FVar.prems
    obtain  $T' \ \mathbf{where} \ T = Inl \ T'$ 
    by  $(elim \ wt-elim-cases) \ simp$ 
    with FVar.prems have  $Env \vdash (\{accC, statDeclC, stat\} e..fn) ::= T'$ 
    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  $G = \langle prg \ Env = G \rangle$ 
have ?NormalAssigned  $s2' \ A$ 
proof
  assume normal-s2': normal  $s2'$ 
  show  $nrm \ A \subseteq dom \ (locals \ (store \ s2'))$ 
  proof –
    note  $avar = \langle (v, s2') = avar \ G \ i \ a \ s2 \rangle$ 
    from AVar.prems obtain E1 where
       $da-e1: Env \vdash (dom \ (locals \ (store \ ((Norm \ s0)::state)))) \rangle \langle e1 \rangle \rangle E1$  and
       $da-e2: Env \vdash nrm \ E1 \rangle \langle e2 \rangle \rangle A$ 
    by  $(elim \ da-elim-cases)$ 
    from AVar.prems obtain  $e1T \ e2T$  where
       $wt-e1: Env \vdash e1 :: -e1T$  and
       $wt-e2: Env \vdash e2 :: -e2T$ 
    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  $\text{nrm } E1 \subseteq \text{dom } (\text{locals } (\text{store } s1))$ 
    using da-e1 wt-e1 G by simp
  with da-e2 obtain A' where
    da-e2':  $\text{Env} \vdash \text{dom } (\text{locals } (\text{store } s1)) \gg \langle e2 \rangle \gg A'$  and
    nrm-A-A':  $\text{nrm } A \subseteq \text{nrm } A'$ 
    by (rule da-weakenE) iprover
  note ⟨PROP ?Hyp (In1l e2) s1 s2⟩
  with da-e2' wt-e2 G normal-s2
  have  $\text{nrm } A' \subseteq \text{dom } (\text{locals } (\text{store } s2))$ 
    by simp
  with nrm-A-A' have  $\text{nrm } A \subseteq \text{dom } (\text{locals } (\text{store } s2))$ 
    by blast
  also have  $\dots \subseteq \text{dom } (\text{locals } (\text{store } s2'))$ 
  proof -
    from avar have  $s2' = \text{snd } (\text{avar } G \text{ i } a \text{ } s2)$ 
    by (cases (avar G i a s2)) simp
    thus  $\text{dom } (\text{locals } (\text{store } s2)) \subseteq \text{dom } (\text{locals } (\text{store } s2'))$ 
      by (simp) (rule dom-locals-avar-mono)
  qed
  finally show ?thesis .
qed
qed
moreover
{
  fix j have abrupt s2'  $\neq$  Some (Jump j)
  proof -
    obtain w upd where  $v: (w, \text{upd}) = v$ 
    by (cases v) auto
    have eval:  $\text{prg } \text{Env} \vdash \text{Norm } s0 - (e1.[e2]) = \succ (w, \text{upd}) \rightarrow s2'$ 
      unfolding G v by (rule eval.AVar AVar.hyps) +
    from AVar.premis
    obtain T' where  $T = \text{Inl } T'$ 
    by (elim wt-elim-cases) simp
    with AVar.premis have  $\text{Env} \vdash (e1.[e2]) ::= T'$ 
    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.premis
have  $\text{nrm } A = \text{dom } (\text{locals } (\text{store } ((\text{Norm } s0)::\text{state})))$ 
  by (elim da-elim-cases) simp
thus ?case by simp
next
case (Cons s0 e v s1 es vs s2 Env T A)
note  $G = \langle \text{prg } \text{Env} = G \rangle$ 
have ?NormalAssigned s2 A
proof
  assume normal-s2: normal s2

```

```

show  $nrm\ A \subseteq dom\ (locals\ (store\ s2))$ 
proof –
  from Cons.prems obtain  $E$  where
     $da-e: Env \vdash (dom\ (locals\ (store\ ((Norm\ s0)::state)))) \gg \langle e \rangle \gg E$  and
     $da-es: Env \vdash nrm\ E \gg \langle es \rangle \gg A$ 
    by (elim da-elim-cases)
  from Cons.prems obtain  $eT\ esT$  where
     $wt-e: Env \vdash e::-eT$  and
     $wt-es: Env \vdash es::\div esT$ 
    by (elim wt-elim-cases)
  have normal s1
    by – (rule eval-no-abrupt-lemma [rule-format], rule Cons.hyps, rule normal-s2)
  moreover note  $\langle PROP\ ?Hyp\ (In1\ e)\ (Norm\ s0)\ s1 \rangle$ 
  ultimately have  $nrm\ E \subseteq dom\ (locals\ (store\ s1))$ 
    using  $da-e\ wt-e\ G$  by simp
  with  $da-es$  obtain  $A'$  where
     $da-es': Env \vdash dom\ (locals\ (store\ s1)) \gg \langle es \rangle \gg A'$  and
     $nrm-A-A': nrm\ A \subseteq nrm\ A'$ 
    by (rule da-weakenE) iprover
  note  $\langle PROP\ ?Hyp\ (In3\ es)\ s1\ s2 \rangle$ 
  with  $da-es'\ wt-es\ G$  normal-s2
  have  $nrm\ A' \subseteq dom\ (locals\ (store\ s2))$ 
    by simp
  with  $nrm-A-A'$  show  $nrm\ A \subseteq dom\ (locals\ (store\ s2))$ 
    by blast
qed
qed
moreover
{
  fix  $j$  have  $abrupt\ s2 \neq Some\ (Jump\ j)$ 
  proof –
    have  $eval: prg\ Env \vdash Norm\ s0 - (e\ \# \ es) \div \succ v\ \# \ vs \rightarrow s2$ 
    unfolding  $G$  by (rule eval.Cons Cons.hyps) +
    from Cons.prems
    obtain  $T'$  where  $T = Inr\ T'$ 
    by (elim wt-elim-cases) simp
    with Cons.prems have  $Env \vdash (e\ \# \ es) :: \div T'$ 
    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**

$prg\ Env \vdash s0 - t \succ \rightarrow (v, s1)$  **and**  $Env \vdash t::T$  **and**  
 $Env \vdash dom\ (locals\ (store\ s0)) \gg t \gg A$  **and**  $wf-prog\ (prg\ Env)$

**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*:  $G \vdash s0 \dashv t \rightarrow (v, s1)$   
**and** *wt*:  $\langle \text{prg}=G, \text{cls}=C, \text{lcl}=L \rangle \vdash t :: T$   
**and** *da*:  $\langle \text{prg}=G, \text{cls}=C, \text{lcl}=L \rangle \vdash \text{dom} (\text{locals} (\text{store } s0)) \gg t \gg A$   
**and** *wf*: *wf-prog* *G*  
**obtains** *normal* *s1*  $\implies \text{nrm } A \subseteq \text{dom} (\text{locals} (\text{store } s1))$  **and**  
 $\bigwedge l. \llbracket \text{abrupt } s1 = \text{Some } (\text{Jump } (\text{Break } l)); \text{normal } s0 \rrbracket$   
 $\implies \text{brk } A \ l \subseteq \text{dom} (\text{locals} (\text{store } s1))$  **and**  
 $\llbracket \text{abrupt } s1 = \text{Some } (\text{Jump } \text{Ret}); \text{normal } s0 \rrbracket$   
 $\implies \text{Result} \in \text{dom} (\text{locals} (\text{store } s1))$

**proof** –

**from** *eval* **have** *prg*  $\langle \text{prg}=G, \text{cls}=C, \text{lcl}=L \rangle \vdash s0 \dashv t \rightarrow (v, s1)$  **by** *simp*  
**moreover** **note** *wt da*  
**moreover** **from** *wf* **have** *wf-prog* (*prg*  $\langle \text{prg}=G, \text{cls}=C, \text{lcl}=L \rangle$ ) **by** *simp*  
**ultimately** **show** *thesis*  
**using** *that* **by** (*rule da-good-approxE*) *iprover*+

**qed**

**declare**  $\llbracket \text{simproc add: wt-expr wt-var wt-exprs wt-stmt} \rrbracket$

**end**



## Chapter 19

# TypeSafe

## 46 The type soundness proof for Java

**theory** *TypeSafe*

**imports** *DefiniteAssignmentCorrect Conform*

**begin**

**error free**

**hide** *const field*

**lemma** *error-free-halloc*:

**assumes** *halloc*:  $G \vdash s0 \text{ --halloc } oi \succ a \rightarrow s1$  **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 split: split-if-asm*)

**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*:

$error\text{-}free\ (check\text{-}field\text{-}access\ G\ accC\ statDeclC\ fn\ stat\ a\ s)$   
 $\implies (check\text{-}field\text{-}access\ G\ accC\ statDeclC\ fn\ stat\ a\ s) = s$   
**apply** (cases s)  
**apply** (auto simp add: check-field-access-def Let-def error-free-def  
 abrupt-if-def  
 split: split-if-asm)  
**done**

**lemma** error-free-check-method-access-eq:  
 $error\text{-}free\ (check\text{-}method\text{-}access\ G\ accC\ statT\ mode\ sig\ a'\ s)$   
 $\implies (check\text{-}method\text{-}access\ G\ accC\ statT\ mode\ sig\ a'\ s) = s$   
**apply** (cases s)  
**apply** (auto simp add: check-method-access-def Let-def error-free-def  
 abrupt-if-def  
 split: split-if-asm)  
**done**

**lemma** error-free-FVar-lemma:  
 $error\text{-}free\ s$   
 $\implies error\text{-}free\ (abupd\ (if\ stat\ then\ id\ else\ np\ a)\ s)$   
**by** (case-tac s) (auto split: split-if)

**lemma** error-free-init-lvars [simp,intro]:  
 $error\text{-}free\ s \implies$   
 $error\text{-}free\ (init\text{-}lvars\ G\ C\ sig\ mode\ a\ pvs\ s)$   
**by** (cases s) (auto simp add: init-lvars-def Let-def split: split-if)

**lemma** error-free-LVar-lemma:  
 $error\text{-}free\ s \implies error\text{-}free\ (assign\ (\lambda v.\ supd\ lupd(vn \mapsto v))\ w\ s)$   
**by** (cases s) simp

**lemma** error-free-throw [simp,intro]:  
 $error\text{-}free\ s \implies error\text{-}free\ (abupd\ (throw\ x)\ s)$   
**by** (cases s) (simp add: throw-def)

## result conformance

### constdefs

$assign\text{-}conforms :: st \Rightarrow (val \Rightarrow state \Rightarrow state) \Rightarrow ty \Rightarrow env' \Rightarrow bool$   
 $(-\leq|-\preceq::\preceq-$   $[71,71,71,71]\ 70)$   
 $s \leq | f \preceq T :: \preceq E \equiv$   
 $(\forall s' w. Norm\ s' :: \preceq E \longrightarrow fst\ E, s \vdash w :: \preceq T \longrightarrow s \leq | s' \longrightarrow assign\ f\ w\ (Norm\ s') :: \preceq E) \wedge$   
 $(\forall s' w. error\text{-}free\ s' \longrightarrow (error\text{-}free\ (assign\ f\ w\ s')))$

### constdefs

$rconf :: prog \Rightarrow lenv \Rightarrow st \Rightarrow term \Rightarrow vals \Rightarrow tys \Rightarrow bool$   
 $(-, -, +, >, - :: \preceq-$   $[71,71,71,71,71,71]\ 70)$   
 $G, L, s \vdash t > v :: \preceq T$   
 $\equiv case\ T\ of$   
 $Inl\ T \Rightarrow if\ (\exists\ var.\ t = In2\ var)$   
 $then\ (\forall\ n.\ (the\text{-}In2\ t) = LVar\ n$   
 $\longrightarrow (fst\ (the\text{-}In2\ v) = the\ (locals\ s\ n)) \wedge$   
 $(locals\ s\ n \neq None \longrightarrow G, s \vdash fst\ (the\text{-}In2\ v) :: \preceq T)) \wedge$

$$\begin{aligned}
& (\neg (\exists n. \text{the-In2 } t = \text{LVar } n) \longrightarrow (G, s \vdash \text{fst } (\text{the-In2 } v) :: \preceq T)) \wedge \\
& (s \leq | \text{snd } (\text{the-In2 } v) \preceq T :: \preceq (G, L)) \\
& \text{else } G, s \vdash \text{the-In1 } v :: \preceq T \\
& | \text{Inr } Ts \Rightarrow \text{list-all2 } (\text{conf } G \ s) (\text{the-In3 } v) \ Ts
\end{aligned}$$

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]*:  
 $va = \text{LVar } n \implies$   
 $G, L, s \vdash \text{In2 } va \succ \text{In2 } vf :: \preceq \text{Inl } T$   
 $= ((\text{fst } vf = \text{the } (\text{locals } s \ n)) \wedge$   
 $(\text{locals } s \ n \neq \text{None} \longrightarrow G, s \vdash \text{fst } vf :: \preceq T) \wedge s \leq | \text{snd } vf \preceq T :: \preceq (G, L))$   
**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*)]:

$G \vdash s \text{ --}t \rightarrow (w, s') \implies \text{snd } s \leq | \text{snd } s' \wedge (\text{case } w \text{ of}$   
 $\quad | \text{In1 } v \Rightarrow \text{True}$   
 $\quad | \text{In2 } vf \Rightarrow \text{normal } s \longrightarrow (\forall v\ x\ s. \text{snd } s \leq | \text{snd } (\text{assign } (\text{snd } vf)\ v\ (x, s)))$   
 $\quad | \text{In3 } vs \Rightarrow \text{True})$

**apply** (*erule eval-induct*)

**prefer** 26

**apply** (*case-tac inited C* (*globs s0*), *clarsimp*, *erule thin-rl*)  
**apply** (*auto del*: *conjI dest!*: *not-initedD gext-new sxalloc-gext halloc-gext*  
*simp add*: *lvar-def fvar-def2 avar-def2 init-lvars-def2*  
*check-field-access-def check-method-access-def Let-def*  
*split del*: *split-if-asm split add*: *sum3.split*)

**apply** *force+*

**done**

**lemma** *eval-gext-f*:

$G \vdash \text{Norm } s1 \text{ --}e \succ vf \rightarrow s2 \implies \text{snd } s1 \leq | \text{snd } (\text{assign } (\text{snd } vf)\ v\ (x, s))$   
**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*:  $G \vdash \text{Norm } s1 \text{ } \neg \text{Init } C \rightarrow s2 \implies \text{initd } C \ s2$   
**apply** (*erule eval-cases* , *auto split del: split-if-asm*)  
**apply** (*case-tac inited C (globs s1)*)  
**apply** (*clarsimp split del: split-if-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*:  
 $\llbracket \text{wf-prog } G; \text{ (case } r \text{ of Heap } a \Rightarrow \text{is-type } G \text{ (obj-ty obj)} \mid \text{Stat } C \Rightarrow \text{is-class } G \ C) \rrbracket \implies G, s \vdash \text{obj } (\text{values} := \text{init-vals } (\text{var-tys } G \text{ (tag obj } r)) :: \preceq \sqrt{r}$   
**apply** (*auto intro!: oconf-init-obj-lemma unique-fields*)  
**done**

**lemma** *conforms-newG*:  $\llbracket \text{globs } s \text{ oref} = \text{None}; (x, s) :: \preceq (G, L); \text{wf-prog } G; \text{ case oref of Heap } a \Rightarrow \text{is-type } G \text{ (obj-ty } (\text{tag} = \text{oi}, \text{values} = \text{vs})) \mid \text{Stat } C \Rightarrow \text{is-class } G \ C \rrbracket \implies (x, \text{init-obj } G \text{ oi oref } s) :: \preceq (G, L)$   
**apply** (*unfold init-obj-def*)  
**apply** (*auto elim!: conforms-gupd dest!: oconf-init-obj*)  
**done**

**lemma** *conforms-init-class-obj*:  
 $\llbracket (x, s) :: \preceq (G, L); \text{wf-prog } G; \text{class } G \ C = \text{Some } y; \neg \text{inited } C \text{ (globs } s) \rrbracket \implies (x, \text{init-class-obj } G \ C \ s) :: \preceq (G, L)$   
**apply** (*rule not-initedD [THEN conforms-newG]*)  
**apply** (*auto*)  
**done**

**lemma** *fst-init-lvars[simp]*:  
 $\text{fst } (\text{init-lvars } G \ C \ \text{sig } (\text{invmode } m \ e) \ a' \ \text{pvs } (x, s)) =$   
 $(\text{if is-static } m \text{ then } x \text{ else } (\text{np } a') \ x)$   
**apply** (*simp (no-asm) add: init-lvars-def2*)  
**done**

**lemma** *halloc-conforms*:  $\bigwedge s1. \llbracket G \vdash s1 \text{ } \neg \text{halloc oi} \succ a \rightarrow s2; \text{wf-prog } G; s1 :: \preceq (G, L);$

```

  is-type G (obj-ty (|tag=oi,values=fs|))  $\impl$  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:
 $\bigwedge s1. \llbracket G \vdash s1 \text{ --halloc oi} \succ a \rightarrow (x,s); \text{wf-prog } G; s1::\preceq(G, L);$ 
   $T = \text{obj-ty } (|tag=oi,values=fs|); \text{is-type } G \ T \rrbracket \impl$ 
   $(x,s)::\preceq(G, L) \wedge (x = \text{None} \longrightarrow G, s \vdash \text{Addr } a::\preceq T)$ 
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 salloc-type-sound:
 $\bigwedge s1 \ s2. \llbracket G \vdash s1 \text{ --salloc} \rightarrow s2; \text{wf-prog } G \rrbracket \impl$ 
  case fst s1 of
    None  $\Rightarrow s2 = s1$ 
  | Some abr  $\Rightarrow$  (case abr of
    Xcpt x  $\Rightarrow (\exists a. \text{fst } s2 = \text{Some}(\text{Xcpt } (\text{Loc } a)) \wedge$ 
       $(\forall L. s1::\preceq(G, L) \longrightarrow s2::\preceq(G, L)))$ 
    | Jump j  $\Rightarrow s2 = s1$ 
    | Error e  $\Rightarrow s2 = s1$ )
apply (simp (no-asm-simp) only: split-tupled-all)
apply (erule salloc.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  $\impl (|prg=G,cls=C,lcl=L|) \vdash \text{init-comp-ty } T::\checkmark$ 
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]

```

```

constdefs
  DynT-prop::[prog,inv-mode,qname,ref-ty]  $\Rightarrow$  bool
  ( $\vdash \longrightarrow \preceq$  [71,71,71,71] 70)
   $G \vdash \text{mode} \rightarrow D \preceq t \equiv \text{mode} = \text{IntVir} \longrightarrow \text{is-class } G \ D \wedge$ 
  (if  $(\exists T. t = \text{ArrayT } T)$  then  $D = \text{Object}$  else  $G \vdash \text{Class } D \preceq \text{RefT } t$ )

```

```

lemma DynT-propI:
 $\llbracket (x,s)::\preceq(G, L); G, s \vdash a'::\preceq \text{RefT } \text{statT}; \text{wf-prog } G; \text{mode} = \text{IntVir} \longrightarrow a' \neq \text{Null} \rrbracket$ 
 $\impl G \vdash \text{mode} \rightarrow \text{invocation-class mode } s \ a' \ \text{statT} \preceq \text{statT}$ 

```

```

proof (unfold DynT-prop-def)
  assume state-conform:  $(x,s)::\leq(G, L)$ 
  and statT-a':  $G, s \vdash a'::\leq_{\text{RefT}} \text{statT}$ 
  and wf: wf-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 =  $\lambda \text{invCls}. \text{is-class } G \text{ invCls} \wedge$ 
     $(\text{if } \exists T. \text{statT} = \text{ArrayT } T$ 
       $\text{then invCls} = \text{Object}$ 
       $\text{else } G \vdash \text{Class invCls} \leq_{\text{RefT}} \text{statT})$ 
  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:  $a' = \text{Addr } a \text{ globs } s \text{ (Inl } a) = \text{Some obj}$ 
       $G \vdash \text{obj-ty obj} \leq_{\text{RefT}} \text{statT is-type } G \text{ (obj-ty obj)}$ 
    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 split add: split-if)
  next
    case Arr
      from Arr obtain T where obj-ty obj = T.[] by (blast dest: obj-ty-Arr1)
      moreover from Arr have obj-class obj = Object
        by (blast dest: obj-class-Arr1)
      moreover note modeIntVir obj-props wf
      ultimately show ?thesis by (auto dest!: widen-Array )
  qed
qed
qed

```

**lemma** invocation-methd:

```

 $\llbracket \text{wf-prog } G; \text{statT} \neq \text{NullT};$ 
 $(\forall \text{statC}. \text{statT} = \text{ClassT statC} \longrightarrow \text{is-class } G \text{ statC});$ 
 $(\forall I. \text{statT} = \text{IfaceT } I \longrightarrow \text{is-iface } G \text{ } I \wedge \text{mode} \neq \text{SuperM});$ 
 $(\forall T. \text{statT} = \text{ArrayT } T \longrightarrow \text{mode} \neq \text{SuperM});$ 
 $G \vdash \text{mode} \longrightarrow \text{invocation-class mode } s \text{ } a' \text{ statT} \leq \text{statT};$ 
 $\text{dynlookup } G \text{ statT (invocation-class mode } s \text{ } a' \text{ statT) sig} = \text{Some } m \rrbracket$ 
 $\implies \text{methd } G \text{ (invocation-declclass } G \text{ mode } s \text{ } a' \text{ statT sig) sig} = \text{Some } m$ 

```

**proof** –

```

assume wf: wf-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:  $G \vdash \text{mode} \longrightarrow \text{invocation-class mode } s \text{ } a' \text{ statT} \leq \text{statT}$ 
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: statT = IfaceT I and
        is-iface: is-iface G I and
        not-SuperM: mode  $\neq$  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 dynimethd-C-C
                    intro: dynimethd-declclass
                    dest!: wf-imethdsD
                    dest: table-of-map-SomeI
                    split: split-if-asm)

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 dynimethd-declclass
                    split: split-if-asm)

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: dynimethd-declclass)

  next
    case SuperM
    with wf ClassT dynlookup statC-prop
    show ?thesis by (auto simp add: invocation-declclass-def dynlookup-def
                                intro: dynimethd-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: dynimethd-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 dynimethd-C-C
                    intro: dynimethd-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 dynmethod-C-C
      intro: dynmethod-declclass
      dest: table-of-map-SomeI)
qed
qed
qed

```

**lemma** *DynT-mheadsD*:

```

[[ G ⊢ invmode sm e → invC ⊑ statT;
  wf-prog G; (|prg=G,cls=C,lcl=L|) ⊢ e :: − RefT statT;
  (statDeclT, sm) ∈ mheads G C statT sig;
  invC = invocation-class (invmode sm e) s a' statT;
  declC = invocation-declclass G (invmode sm e) s a' statT sig
]] ⇒
  ∃ dm.
    methd G declC sig = Some dm ∧ dynlookup G statT invC sig = Some dm ∧
    G ⊢ resTy (methd dm) ⊑ resTy sm ∧
    wf-mdecl G declC (sig, methd dm) ∧
    declC = declclass dm ∧
    is-static dm = is-static sm ∧
    is-class G invC ∧ is-class G declC ∧ G ⊢ invC ⊑C declC ∧
    (if invmode sm e = IntVir
      then (∀ statC. statT = ClassT statC ⟶ G ⊢ invC ⊑C statC)
      else ( (∃ statC. statT = ClassT statC ∧ G ⊢ statC ⊑C declC)
        ∨ (∀ statC. statT ≠ ClassT statC ∧ declC = Object)) ∧
        statDeclT = ClassT (declclass dm))

```

**proof** –

```

  assume invC-prop: G ⊢ invmode sm e → invC ⊑ statT
  and wf: wf-prog G
  and wt-e: (|prg=G,cls=C,lcl=L|) ⊢ e :: − RefT statT
  and sm: (statDeclT, sm) ∈ mheads G C statT sig
  and invC: invC = invocation-class (invmode sm e) s a' statT
  and declC: declC =
    invocation-declclass G (invmode sm e) s a' statT sig
  from wt-e wf have type-statT: is-type G (RefT statT)
    by (auto dest: ty-expr-is-type)
  from sm have not-Null: statT ≠ NullT by auto
  from type-statT
  have wf-C: (∀ statC. statT = ClassT statC ⟶ is-class G statC)
    by (auto)
  from type-statT wt-e
  have wf-I: (∀ I. statT = IfaceT I ⟶ is-iface G I ∧
    invmode sm e ≠ SuperM)
    by (auto dest: invocationTypeExpr-noClassD)
  from wt-e
  have wf-A: (∀ T. statT = ArrayT T ⟶ invmode sm e ≠ SuperM)
    by (auto dest: invocationTypeExpr-noClassD)
  show ?thesis
  proof (cases invmode sm e = IntVir)
    case True

```

```

with invC-prop not-Null
have invC-prop': is-class G invC  $\wedge$ 
      (if ( $\exists T. \text{stat}T = \text{Array}T \ T$ ) then invC = Object
        else  $G \vdash \text{Class } invC \preceq \text{Ref}T \ \text{stat}T$ )
  by (auto simp add: DynT-prop-def)
from True
have  $\neg \text{is-static } sm$ 
  by (simp add: invmode-IntVir-eq member-is-static-simp)
with invC-prop' not-Null
have  $G, \text{stat}T \vdash invC \text{ valid-lookup-cls-for } (\text{is-static } sm)$ 
  by (cases statT) auto
with sm wf type-statT obtain dm where
  dm: dynlookup G statT invC sig = Some dm and
  resT-dm:  $G \vdash \text{resTy } (\text{methd } dm) \preceq \text{resTy } sm$  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:  $G \vdash invC \preceq_C \text{declC} \wedge \text{is-class } G \ \text{declC}$ 
  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:  $(\forall \text{stat}C. \text{stat}T = \text{Class}T \ \text{stat}C \longrightarrow G \vdash invC \preceq_C \text{stat}C)$ 
  by auto
from True dm' resT-dm wf-dm invC-prop' declC-prop statC-prop declC' static
  dm
show ?thesis by auto
next
case False
with type-statT wf invC not-Null wf-I wf-A
have invC-prop': is-class G invC  $\wedge$ 
      ( $(\exists \text{stat}C. \text{stat}T = \text{Class}T \ \text{stat}C \wedge invC = \text{stat}C) \vee$ 
        $(\forall \text{stat}C. \text{stat}T \neq \text{Class}T \ \text{stat}C \wedge invC = \text{Object})$ )
  by (case-tac statT) (auto simp add: invocation-class-def
    split: inv-mode.splits)

with not-Null wf
have dynlookup-static: dynlookup G statT invC sig = methd G invC sig
  by (case-tac statT) (auto simp add: dynlookup-def dynmethd-C-C
    dynimethd-def)
from sm wf wt-e not-Null False invC-prop' obtain dm where
  dm: methd G invC sig = Some dm and
  eq-declC-sm-dm: statDeclT = ClassT (declclass dm) and
  eq-mheads: sm = mhead (methd dm)
  by - (drule static-mheadsD, (force dest: accmethd-SomeD)+)
then have static: is-static dm = is-static sm by - (auto)
with declC invC dynlookup-static dm
have declC': declC = (declclass dm)
  by (auto simp add: invocation-declclass-def)
from invC-prop' wf declC' dm
have dm': methd G declC sig = Some dm
  by (auto intro: methd-declclass)
from dynlookup-static dm

```

```

have dm'': dynlookup G statT invC sig = Some dm
  by simp
from wf dm invC-prop' declC' type-statT
have declC-prop:  $G \vdash \text{invC} \preceq_C \text{declC} \wedge \text{is-class } G \text{ declC}$ 
  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:  $(\text{prg} = G, \text{cls} = C, \text{lcl} = L) \vdash e :: \text{RefT statT}$ 
  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:  $\bigwedge dm. \llbracket \text{methd } G \ \text{declC} \ \text{sig} = \text{Some } dm; \text{dynlookup } G \ \text{statT} \ \text{invC} \ \text{sig} = \text{Some } dm; \text{G} \vdash \text{resTy } (\text{methd } dm) \preceq \text{resTy } \text{sm}; \text{wf-mdecl } G \ \text{declC} \ (\text{sig}, \text{methd } dm); \text{declC} = \text{declclass } dm; \text{is-static } dm = \text{is-static } \text{sm}; \text{is-class } G \ \text{invC}; \text{is-class } G \ \text{declC}; G \vdash \text{invC} \preceq_C \text{declC}; \text{if invmode } \text{sm} \ e = \text{IntVir} \text{ then } (\forall \text{statC}. \text{statT} = \text{ClassT statC} \longrightarrow G \vdash \text{invC} \preceq_C \text{statC}) \text{ else } ( (\exists \text{statC}. \text{statT} = \text{ClassT statC} \wedge G \vdash \text{statC} \preceq_C \text{declC}) \vee (\forall \text{statC}. \text{statT} \neq \text{ClassT statC} \wedge \text{declC} = \text{Object}) ) \wedge \text{statDeclT} = \text{ClassT } (\text{declclass } dm) \rrbracket \implies P$ 

```

**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*:  $\llbracket G \vdash \text{invocation-class } \text{mode } s \ a' \ \text{statT} \preceq_C \text{declC}; \text{wf-prog } G; \text{isrtype } G \ (\text{statT});$

$G, s \vdash a' :: \preceq \text{RefT statT}; \text{mode} = \text{IntVir} \longrightarrow a' \neq \text{Null};$

$\text{mode} \neq \text{IntVir} \longrightarrow (\exists \text{statC}. \text{statT} = \text{ClassT statC} \wedge G \vdash \text{statC} \preceq_C \text{declC})$

```

    ∨ (∀ statC. statT ≠ ClassT statC ∧ declC = Object)
  ⟹ G, s ⊢ a' :: ≤ Class declC
apply (case-tac mode = IntVir)
apply (drule conf-RefTD)
apply (force intro!: conf-AddrI
    simp add: obj-class-def split add: obj-tag.split-asm)
apply clarsimp
apply safe
apply (erule (1) widen.subcls [THEN conf-widen])
apply (erule wf-ws-prog)

apply (frule widen-Object) apply (erule wf-ws-prog)
apply (erule (1) conf-widen) apply (erule wf-ws-prog)
done

```

**lemma** Ass-lemma:

```

  ⟦ G ⊢ Norm s0 -var=>(w, f) → Norm s1; G ⊢ Norm s1 -e->v → Norm s2;
    G, s2 ⊢ v :: ≤ eT; s1 ≤ s2 ⟹ assign f v (Norm s2) :: ≤ (G, L) ⟧
  ⟹ assign f v (Norm s2) :: ≤ (G, L) ∧
    (normal (assign f v (Norm s2)) ⟹ G, store (assign f v (Norm s2)) ⊢ v :: ≤ eT)
apply (drule-tac x = None and s = s2 and v = v in evar-geat-f)
apply (drule eval-geat', clarsimp)
apply (erule conf-geat)
apply simp
done

```

**lemma** Throw-lemma:  $\llbracket G \vdash tn \preceq_C SXcpt Throwable; wf\text{-}prog\ G; (x1, s1) :: \preceq (G, L);$   
 $x1 = None \longrightarrow G, s1 \vdash a' :: \preceq Class\ tn \rrbracket \Longrightarrow (throw\ a'\ x1, s1) :: \preceq (G, L)$

```

apply (auto split add: 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:  $\llbracket G \vdash obj\text{-}ty\ (the\ (globs\ s1'\ (Heap\ a))) \preceq Class\ tn;$   
 $(Some\ (Xcpt\ (Loc\ a)), s1') :: \preceq (G, L); wf\text{-}prog\ G \rrbracket$   
 $\Longrightarrow Norm\ (lupd\ (vn \mapsto Addr\ a)\ s1') :: \preceq (G, L\ (vn \mapsto Class\ tn))$

```

apply (rule conforms-allocL)
apply (erule conforms-NormI)
apply (drule conforms-XcptLocD [THEN conf-RefTD], rule HOL.refl)
apply (auto intro!: conf-AddrI)
done

```

**lemma** Fin-lemma:

```

  ⟦ G ⊢ Norm s1 -c2→ (x2, s2); wf-prog G; (Some a, s1) :: ≤ (G, L); (x2, s2) :: ≤ (G, L);
    dom (locals s1) ⊆ dom (locals s2) ⟧
  ⟹ (abrupt-if True (Some a) x2, s2) :: ≤ (G, L)
apply (auto elim: eval-geat' conforms-xgeat split add: split-abrupt-if)
done

```

**lemma** FVar-lemma1:

```

  ⟦ table-of (DeclConcepts.fields G statC) (fn, statDeclC) = Some f ;
    x2 = None ⟹ G, s2 ⊢ a :: ≤ Class statC; wf-prog G; G ⊢ statC ≤_C statDeclC;
    statDeclC ≠ Object;

```

```

class G statDeclC = Some y; (x2,s2)::≼(G, L); s1≤|s2;
initd statDeclC (globs s1);
(if static f then id else np a) x2 = None]
⇒
  ∃ obj. globs s2 (if static f then Inr statDeclC else Inl (the-Addr a))
    = Some obj ∧
  var-tys G (tag obj) (if static f then Inr statDeclC else Inl(the-Addr a))
    (Inl(fn,statDeclC)) = Some (type f)
apply (drule initdD)
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 add: obj-tag.split)
apply (rule fields-table-SomeI)
apply (auto elim!: fields-mono subcls-is-class2)
done

```

**lemma FVar-lemma2:** *error-free state*  
 $\Rightarrow$  *error-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 split-if [split del] split-if-asm [split del]
      option.split [split del] option.split-asm [split del]
declaration << K (Simplifier.map-ss (fn ss => ss delloop split-all-tac)) >>
declaration << K (Classical.map-cs (fn cs => cs delSWrapper split-all-tac)) >>

```

**lemma FVar-lemma:**  
 $\llbracket (v, f), \text{Norm } s2' \rrbracket = \text{fvar statDeclC (static field) fn a (x2, s2);}$   
 $G \vdash \text{statC} \preceq_C \text{statDeclC};$   
 $\text{table-of (DeclConcepts.fields G statC) (fn, statDeclC) = Some field};$   
 $\text{wf-prog G};$   
 $x2 = \text{None} \longrightarrow G, s2 \vdash a :: \preceq \text{Class statC};$   
 $\text{statDeclC} \neq \text{Object}; \text{class G statDeclC} = \text{Some y};$   
 $(x2, s2) :: \preceq (G, L); s1 \leq |s2; \text{initd statDeclC (globs s1)} \rrbracket \Rightarrow$   
 $G, s2 \uparrow v :: \preceq \text{type field} \wedge s2' \leq |f \preceq \text{type field} :: \preceq (G, L)$

```

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 split-if [split] split-if-asm [split]
      option.split [split] option.split-asm [split]
declaration  $\ll K \text{ (Classical.map-cs (fn cs => cs addSbefore (split-all-tac, split-all-tac)))} \gg$ 
declaration  $\ll K \text{ (Simplifier.map-ss (fn ss => ss addloop (split-all-tac, split-all-tac)))} \gg$ 

```

```

lemma AVar-lemma1:  $\ll$ globs  $s$  (Inl  $a$ ) = Some  $obj$ ; tag  $obj$  = Arr  $ty$   $i$ ;
  the-Intg  $i'$  in-bounds  $i$ ; wf-prog  $G$ ;  $G \vdash ty.[] \preceq Tb.[]$ ; Norm  $s :: \preceq (G, L)$ 
 $\gg \implies G, s \vdash the ((values\ obj) (Inr (the-Intg\ i')) :: \preceq Tb$ 
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:  $\exists t\ vs. obj = \langle tag=t, values=vs \rangle$ 
  by (cases obj) auto

```

```

lemma AVar-lemma2: error-free state
 $\implies$  error-free
  (assign
    ( $\lambda v\ (x, s').$ 
      ((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:  $\ll$ wf-prog  $G$ ;  $G \vdash (x1, s1) -e2-\triangleright i \rightarrow (x2, s2)$ ;
  (( $v, f$ ), Norm  $s2'$ ) = avar  $G\ i\ a\ (x2, s2)$ ;  $x1 = None \longrightarrow G, s1 \vdash a :: \preceq Ta.[]$ ;
  ( $x2, s2$ ) ::  $\preceq (G, L)$ ;  $s1 \leq |s2| \implies G, s2 \vdash v :: \preceq Ta \wedge s2' \leq |f \preceq Ta :: \preceq (G, L)$ 
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:  $\llbracket wf\text{-prog } G;$ 
   $wf\text{-mhead } G \ P \ sig \ mh;$ 
   $list\text{-all2 } (conf \ G \ s) \ pvs \ pTsa; \ G \vdash pTsa[\preceq](parTs \ sig) \rrbracket \implies$ 
   $G, s \vdash empty \ (pars \ mh[\mapsto] pvs)$ 
   $[\sim::\preceq] table\text{-of } lvars(pars \ mh[\mapsto] parTs \ sig)$ 
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]:
   $G, s \vdash (lname\text{-case } l1 \ l2)[::\preceq](lname\text{-case } L1 \ L2)$ 
  =
   $(G, s \vdash l1[::\preceq] L1 \wedge G, s \vdash (\lambda x::unit. l2)[::\preceq](\lambda x::unit. L2))$ 
apply (unfold lconf-def)
apply (auto split add: lname.splits)
done

```

```

lemma wlconf-map-lname [simp]:
   $G, s \vdash (lname\text{-case } l1 \ l2)[\sim::\preceq](lname\text{-case } L1 \ L2)$ 
  =
   $(G, s \vdash l1[\sim::\preceq] L1 \wedge G, s \vdash (\lambda x::unit. l2)[\sim::\preceq](\lambda x::unit. L2))$ 
apply (unfold wlconf-def)
apply (auto split add: lname.splits)
done

```

```

lemma lconf-map-ename [simp]:
   $G, s \vdash (ename\text{-case } l1 \ l2)[::\preceq](ename\text{-case } L1 \ L2)$ 
  =
   $(G, s \vdash l1[::\preceq] L1 \wedge G, s \vdash (\lambda x::unit. l2)[::\preceq](\lambda x::unit. L2))$ 
apply (unfold lconf-def)
apply (auto split add: ename.splits)
done

```

```

lemma wlconf-map-ename [simp]:

```



```

  G, s ⊢ (ename-case l1 l2)[~::≤](ename-case L1 L2)
  =
  (G, s ⊢ l1[~::≤]L1 ∧ G, s ⊢ (λx::unit. l2)[~::≤](λx::unit. L2))
apply (unfold wlconf-def)
apply (auto split add: 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 split-if [split del] split-if-asm [split del]
  option.split [split del] option.split-asm [split del]
declaration ⟨⟨ K (Simplifier.map-ss (fn ss => ss delloop split-all-tac)) ⟩⟩
declaration ⟨⟨ K (Classical.map-cs (fn cs => cs 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 add: split-if)
apply (drule (4) DynT-conf)
apply clarsimp

```

```

apply (drule (3) conforms-init-lvars-lemma
  [where ?lvars=(lcls (mbdy (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 split-if [split] split-if-asm [split]
  option.split [split] option.split-asm [split]
declaration  $\ll K \text{ (Classical.map-cs (fn cs => cs addSbefore (split-all-tac, split-all-tac)))} \gg$ 
declaration  $\ll K \text{ (Simplifier.map-ss (fn ss => ss addloop (split-all-tac, split-all-tac)))} \gg$ 

```

## 47 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: accfield G accC statC fn = Some f and
  dynC: if stat then dynC = declclass f
    else dynC = obj-class (lookup-obj (store s) a) and
  stat: if stat then (is-static f) else  $(\neg \text{is-static } f)$ 
shows table-of (DeclConcepts.fields G dynC) (fn, declclass f) = Some (fld f)  $\wedge$ 
   $G \vdash \text{Field fn f in dynC dyn-accessible-from accC}$ 

```

**proof** (cases stat)

```

case True
with stat have static: (is-static f) by simp
from True dynC
have dynC': dynC = declclass f by simp
with f
have table-of (DeclConcepts.fields G statC) (fn, declclass f) = Some (fld f)
  by (auto simp add: accfield-def Let-def intro!: table-of-remap-SomeD)
moreover
from wt-e wf have is-class G statC
  by (auto dest!: ty-expr-is-type)
moreover note wf dynC'
ultimately have
  table-of (DeclConcepts.fields G dynC) (fn, declclass f) = Some (fld f)
  by (auto dest: fields-declC)
with dynC' f static wf
show ?thesis
  by (auto dest: static-to-dynamic-accessible-from-static
    dest!: accfield-accessibleD )

```

**next**

```

case False
with wf conform-a not-Null conform-s dynC

```

```

obtain subclseq:  $G \vdash \text{dyn}C \preceq_C \text{stat}C$  and
  is-class  $G \text{ dyn}C$ 
  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  $G \text{ dyn}C$ ) (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  $G \vdash \text{Field fn f in dyn}C \text{ dyn-accessible-from acc}C$ 
  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  $G \text{ acc}C \text{ stat}C \text{ fn} = \text{Some} (\text{statDecl}C, f)$  and
  wt-e: ( $\text{prg} = G, \text{cls} = \text{acc}C, \text{lcl} = L$ ) $\vdash e :: \neg \text{Class stat}C$  and
  eval-init:  $G \vdash \text{Norm } s0 \text{ --Init statDecl}C \rightarrow s1$  and
  eval-e:  $G \vdash s1 \text{ --e--} \rightarrow a \rightarrow s2$  and
  conf-s2:  $s2 :: \preceq (G, L)$  and
  conf-a: normal  $s2 \implies G, \text{store } s2 \vdash a :: \preceq \text{Class stat}C$  and
  fvar: ( $v, s2'$ ) = fvar statDeclC (is-static f)  $\text{fn } a \text{ } s2$  and
  wf: wf-prog  $G$ 
shows check-field-access  $G \text{ acc}C \text{ statDecl}C \text{ fn} (\text{is-static } f) a \text{ } s2' = s2'$ 
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:  $\neg (\text{is-static } f) \longrightarrow a \neq \text{Null}$ 
      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':  $G, \text{store } s2 \vdash a :: \preceq \text{Class stat}C$ 
      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: invC = invocation-class (invmode statM e) s a statT
shows  $\exists \text{ dynM. } \text{dynlookup } G \text{ statT } \text{invC } \text{sig} = \text{Some dynM} \wedge$ 
   $G \vdash \text{Methd } \text{sig } \text{dynM} \text{ in } \text{invC } \text{dyn-accessible-from accC}$ 
proof –
  from wt-e wf have type-statT: is-type G (RefT statT)
    by (auto dest: ty-expr-is-type)
  from statM have not-Null: statT  $\neq \text{NullT}$  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  $\vdash \text{invC } \text{valid-lookup-cls-for is-static statM}$ 
      by (cases statT) (auto simp add: invmode-def)
    with statM type-statT wf
    show ?thesis
      by – (rule dynlookup-access, auto)
  next
  case False
  with type-statT wf invC not-Null wf-I wf-A
  have invC-prop': is-class G invC  $\wedge$ 
     $((\exists \text{ statC. } \text{statT} = \text{ClassT } \text{statC} \wedge \text{invC} = \text{statC}) \vee$ 
       $(\forall \text{ statC. } \text{statT} \neq \text{ClassT } \text{statC} \wedge \text{invC} = \text{Object}))$ 
    by (case-tac statT) (auto simp add: invocation-class-def
      split: inv-mode.splits)
  with not-Null wf
  have dynlookup-static: 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:  $G \vdash s1 \multimap \text{args} \Rightarrow vs \rightarrow s2$  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:  $\text{normal } s1 \Rightarrow G, \text{store } s1 \vdash a :: \preceq \text{RefT statT}$  and
  invProp:  $\text{normal } s3 \Rightarrow$ 
     $G \vdash \text{invmode statM } e \rightarrow \text{invC} \preceq \text{statT}$  and
    s3:  $s3 = \text{init-lvars } G \text{ invDeclC } (\text{name} = mn, \text{parTs} = pTs')$ 
      (invmode statM e) a vs s2 and
    invC:  $\text{invC} = \text{invocation-class } (\text{invmode statM } e) (\text{store } s2) \text{ a statT}$  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 (invmode statM e) (name=mn, parTs=pTs') a s3
  = s3
proof (cases normal s2)
case False
with s3
have abrupt s3 = abrupt s2
  by (auto simp add: init-lvars-def2)
with False
show ?thesis
  by (auto simp add: check-method-access-def Let-def)
next
case True
note normal-s2 = True
with eval-args
have normal-s1: normal s1
  by (cases normal s1) auto
with conf-a eval-args
have conf-a-s2:  $G, \text{store } s2 \vdash a :: \preceq \text{RefT statT}$ 
  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

```

```

    statM': (statDeclT, statM) ∈ mheads G accC statT (⟦name=mn, parTs=pTs'⟧)
  by (blast dest: max-spec2mheads)
from True normal-s2 s3
have normal s3
  by (auto simp add: init-lvars-def2)
then have  $G \vdash \text{invmode statM } e \rightarrow \text{invC} \preceq \text{statT}$ 
  by (rule invProp)
with wt-e statM' wf invC
obtain dynM where
  dynM: dynlookup G statT invC (⟦name=mn, parTs=pTs'⟧) = Some dynM and
  acc-dynM:  $G \vdash \text{Methd } (⟦name=mn, parTs=pTs'⟧) \text{ dynM}$ 
    in invC dyn-accessible-from accC
  by (force dest!: call-access-ok)
moreover
from s3 invC
have invC': invC = (invocation-class (invmode statM e) (store s3) a statT)
  by (cases s2, cases invmode statM e)
  (simp add: init-lvars-def2 del: invmode-Static-eq) +
ultimately
show ?thesis
  by (auto simp add: check-method-access-def Let-def)
qed
qed

```

**lemma** map-upds-eq-length-append-simp:

```

 $\bigwedge \text{tab } qs. \text{length } ps = \text{length } qs \implies \text{tab}(ps[\mapsto]qs@zs) = \text{tab}(ps[\mapsto]qs)$ 
proof (induct ps)
  case Nil thus ?case by simp
next
  case (Cons p ps tab qs)
  from ⟨length (p#ps) = length qs⟩
  obtain q qs' where qs:  $qs = q \# qs'$  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:  $\text{length } ps = \text{length } qs'$ 
  by (cases qs) auto
  from eq-length
  have  $(\text{tab}(p \mapsto q))(ps[\mapsto]qs')(x \mapsto y) = (\text{tab}(p \mapsto q))(ps@[x][\mapsto]qs'@[y])$ 
  by (rule Cons.hyps)
  with qs show ?case
  by simp
qed

```

**lemma** *map-upd-cong*:  $tab = tab' \implies tab(x \mapsto y) = tab'(x \mapsto y)$   
**by** *simp*

**lemma** *map-upd-cong-ext*:  $tab\ z = tab'\ z \implies (tab(x \mapsto y))\ z = (tab'(x \mapsto y))\ z$   
**by** (*simp add: fun-upd-def*)

**lemma** *map-upds-cong*:  $tab = tab' \implies tab(xs[\mapsto]ys) = tab'(xs[\mapsto]ys)$   
**by** (*cases xs*) *simp+*

**lemma** *map-upds-cong-ext*:  
 $\bigwedge\ tab\ tab'\ ys.\ tab\ z = tab'\ z \implies (tab(xs[\mapsto]ys))\ z = (tab'(xs[\mapsto]ys))\ z$   
**proof** (*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**  $(tab(x \mapsto y)(xs[\mapsto]ys'))\ z = (tab'(x \mapsto y)(xs[\mapsto]ys'))\ z$   
  **by** (*iprover intro: Hyps map-upd-cong-ext*)  
  **with Cons** **show** ?*thesis*  
  **by** *simp*  
**qed**  
**qed**

**lemma** *map-upd-override*:  $(tab(x \mapsto y))\ x = (tab'(x \mapsto y))\ x$   
**by** *simp*

**lemma** *map-upds-eq-length-suffix*:  $\bigwedge\ tab\ qs.\$   
 $length\ ps = length\ qs \implies tab(ps @ xs[\mapsto]qs) = tab(ps[\mapsto]qs)(xs[\mapsto][])$   
**proof** (*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**  $tab(p \mapsto q)(ps @ xs[\mapsto]qs') = tab(p \mapsto q)(ps[\mapsto]qs')(xs[\mapsto][])$   
  **by** (*rule Cons.hyps*)  
  **with qs** **show** ?*case*  
  **by** *simp*  
**qed**

**lemma** *map-upds-upds-eq-length-prefix-simp*:  
 $\bigwedge\ tab\ qs.\ length\ ps = length\ qs$   
 $\implies tab(ps[\mapsto]qs)(xs[\mapsto]ys) = tab(ps @ xs[\mapsto]qs @ ys)$

```

proof (induct ps)
  case Nil thus ?case by simp
next
  case (Cons p ps tab qs)
  then obtain q qs' where qs: qs=q#qs' and eq-length: length ps=length qs'
  by (cases qs) auto
  from eq-length
  have tab(p↦q)(ps[↦]qs')(xs[↦]ys) = tab(p↦q)(ps @ xs[↦](qs' @ ys))
  by (rule Cons.hyps)
  with qs
  show ?case by simp
qed

```

**lemma** map-upd-cut-irrelevant:  
 $\llbracket (tab(x \mapsto y)) \ vn = Some\ el; (tab'(x \mapsto y)) \ vn = None \rrbracket$   
 $\implies tab\ vn = Some\ el$   
**by** (cases tab' vn = None) (simp add: fun-upd-def)+

**lemma** map-upd-Some-expand:  
 $\llbracket tab\ vn = Some\ z \rrbracket$   
 $\implies \exists\ z. (tab(x \mapsto y)) \ vn = Some\ z$   
**by** (simp add: fun-upd-def)

**lemma** map-upds-Some-expand:  
 $\bigwedge\ tab\ ys\ z. \llbracket tab\ vn = Some\ z \rrbracket$   
 $\implies \exists\ z. (tab(xs[↦]ys)) \ vn = Some\ z$   
**proof** (induct xs)  
**case** Nil **thus** ?case **by** simp  
**next**  
**case** (Cons x xs tab ys z)  
**note** z =  $\langle tab\ vn = Some\ z \rangle$   
**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** (tab(x↦y)) vn = Some z'  
**by** (rule map-upd-Some-expand [of tab, elim-format]) blast  
**hence**  $\exists\ z. ((tab(x \mapsto y))(xs[↦]ys')) \ vn = Some\ z$   
**by** (rule Cons.hyps)  
**with** ys **show** ?thesis  
**by** simp  
**qed**  
**qed**

**lemma** map-upd-Some-swap:  
 $(tab(r \mapsto w)(u \mapsto v)) \ vn = Some\ z \implies \exists\ z. (tab(u \mapsto v)(r \mapsto w)) \ vn = Some\ z$   
**by** (simp add: fun-upd-def)

**lemma** map-upd-None-swap:  
 $(tab(r \mapsto w)(u \mapsto v)) \ vn = None \implies (tab(u \mapsto v)(r \mapsto w)) \ vn = None$



by (simp add: fun-upd-def)

**lemma** map-eq-upd-eq:  $tab\ vn = tab'\ vn \implies (tab(x \mapsto y))\ vn = (tab'(x \mapsto y))\ vn$   
 by (simp add: fun-upd-def)

**lemma** map-upd-in-expansion-map-swap:  

$$\llbracket (tab(x \mapsto y))\ vn = Some\ z; tab\ vn \neq Some\ z \rrbracket$$
  

$$\implies (tab'(x \mapsto y))\ vn = Some\ z$$
  
 by (simp add: fun-upd-def)

**lemma** map-upds-in-expansion-map-swap:  

$$\wedge tab\ tab'\ ys\ z. \llbracket (tab(xs[\mapsto]ys))\ vn = Some\ z; tab\ vn \neq Some\ z \rrbracket$$
  

$$\implies (tab'(xs[\mapsto]ys))\ vn = Some\ z$$

**proof** (induct xs)  
 case Nil **thus** ?case **by** simp  
**next**  
 case (Cons x xs tab tab' ys z)  
**note** some =  $\langle (tab(x \# xs[\mapsto]ys))\ vn = Some\ z \rangle$   
**note** tab-not-z =  $\langle tab\ vn \neq Some\ z \rangle$   
**show** ?case  
**proof** (cases ys)  
 case Nil **with** some tab-not-z **show** ?thesis **by** simp  
**next**  
 case (Cons y tl)  
**note** ys =  $\langle ys = y \# tl \rangle$   
**show** ?thesis  
**proof** (cases (tab(x  $\mapsto$  y)) vn  $\neq$  Some z)  
 case True  
**with** some ys **have** (tab'(x  $\mapsto$  y)(xs[\mapsto]tl)) vn = Some z  
**by** (fastsimp intro: Cons.hyps)  
**with** ys **show** ?thesis  
**by** simp  
**next**  
 case False  
**hence** tabx-z: (tab(x  $\mapsto$  y)) vn = Some z **by** blast  
**moreover**  
**from** tabx-z tab-not-z  
**have** (tab'(x  $\mapsto$  y)) vn = Some z  
**by** (rule map-upd-in-expansion-map-swap)  
**ultimately**  
**have** (tab(x  $\mapsto$  y)) vn = (tab'(x  $\mapsto$  y)) vn  
**by** simp  
**hence** (tab(x  $\mapsto$  y)(xs[\mapsto]tl)) vn = (tab'(x  $\mapsto$  y)(xs[\mapsto]tl)) vn  
**by** (rule map-upds-cong-ext)  
**with** some ys  
**show** ?thesis  
**by** simp  
**qed**  
**qed**  
**qed**

**lemma** map-upds-Some-swap:  
**assumes** r-u:  $(tab(r \mapsto w)(u \mapsto v)(xs[\mapsto]ys))\ vn = Some\ z$   
**shows**  $\exists z. (tab(u \mapsto v)(r \mapsto w)(xs[\mapsto]ys))\ vn = Some\ z$

**proof** (cases (tab( $r \mapsto w$ ))( $u \mapsto v$ ))  $vn = \text{Some } z$ )  
 case *True*  
 then obtain  $z'$  where (tab( $u \mapsto v$ ))( $r \mapsto w$ ))  $vn = \text{Some } z'$   
 by (rule map-upd-Some-swap [elim-format]) blast  
 thus  $\exists z. (\text{tab}(u \mapsto v)(r \mapsto w)(xs[\mapsto]ys)) \text{ } vn = \text{Some } z$   
 by (rule map-upds-Some-expand)  
 next  
 case *False*  
 with  $r \sim u$   
 have (tab( $u \mapsto v$ ))( $r \mapsto w$ ))( $xs[\mapsto]ys$ )  $vn = \text{Some } z$   
 by (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** (cases  $\exists z. \text{tab } vn = \text{Some } z$ )  
 case *True*  
 then obtain  $z'$  where  $\text{tab } vn = \text{Some } z'$  by blast  
 then obtain  $z''$  where (tab( $u \mapsto v$ ))  $vn = \text{Some } z''$   
 by (rule map-upd-Some-expand [elim-format]) blast  
 thus ?thesis  
 by (rule map-upds-Some-expand)  
 next  
 case *False*  
 hence  $\text{tab } vn \neq \text{Some } z$  by simp  
 with  $z$   
 have (tab( $u \mapsto v$ ))( $xs[\mapsto]ys$ )  $vn = \text{Some } z$   
 by (rule map-upds-in-expansion-map-swap)  
 thus ?thesis ..  
 qed

**lemma** map-upds-None-cut:  
 assumes expand-None: (tab( $xs[\mapsto]ys$ ))  $vn = \text{None}$   
 shows  $\text{tab } vn = \text{None}$   
**proof** (cases  $\text{tab } vn = \text{None}$ )  
 case *True* thus ?thesis by simp  
 next  
 case *False* then obtain  $z$  where  $\text{tab } vn = \text{Some } z$  by blast  
 then obtain  $z'$  where (tab( $xs[\mapsto]ys$ ))  $vn = \text{Some } z'$   
 by (rule map-upds-Some-expand [where ?tab=tab,elim-format]) blast  
 with expand-None show ?thesis  
 by simp  
 qed

**lemma** map-upds-cut-irrelevant:  
 $\bigwedge \text{tab } \text{tab}' \text{ } ys. \llbracket (\text{tab}(xs[\mapsto]ys)) \text{ } vn = \text{Some } el; (\text{tab}'(xs[\mapsto]ys)) \text{ } vn = \text{None} \rrbracket$   
 $\implies \text{tab } vn = \text{Some } el$   
**proof** (induct  $xs$ )  
 case *Nil* thus ?case by simp  
 next  
 case (Cons  $x \text{ } xs \text{ } \text{tab } \text{tab}' \text{ } ys$ )  
 note  $\text{tab} \cdot \text{tab}' = \langle \text{tab}(x \# xs[\mapsto]ys) \rangle \text{ } vn = \text{Some } el$

```

note  $tab'-vn = \langle (tab'(x \# xs[\mapsto]ys)) \ vn = None \rangle$ 
show  $?case$ 
proof ( $cases\ ys$ )
  case  $Nil$ 
    with  $tab-vn$  show  $?thesis$  by  $simp$ 
next
  case ( $Cons\ y\ tl$ )
    note  $ys = \langle ys=y\#tl \rangle$ 
    with  $tab-vn\ tab'-vn$ 
    have ( $tab(x \mapsto y)$ )  $vn = Some\ el$ 
      by  $-\ (rule\ Cons.hyps, auto)$ 
    moreover from  $tab'-vn\ ys$ 
    have ( $tab'(x \mapsto y)(xs[\mapsto]tl)$ )  $vn = None$ 
      by  $simp$ 
    hence ( $tab'(x \mapsto y)$ )  $vn = None$ 
      by ( $rule\ map-upds-None-cut$ )
    ultimately show  $tab\ vn = Some\ el$ 
      by ( $rule\ map-upd-cut-irrelevant$ )
qed
qed

```

**lemma**  $dom-vname-split$ :

```

 $dom\ (lname-case\ (ename-case\ (tab(x \mapsto y)(xs[\mapsto]ys))\ a)\ b)$ 
  =  $dom\ (lname-case\ (ename-case\ (tab(x \mapsto y))\ a)\ b) \cup$ 
     $dom\ (lname-case\ (ename-case\ (tab(xs[\mapsto]ys))\ a)\ b)$ 
  (is  $?List\ x\ xs\ y\ ys = ?Hd\ x\ y \cup ?Tl\ xs\ ys$ )
proof
  show  $?List\ x\ xs\ y\ ys \subseteq ?Hd\ x\ y \cup ?Tl\ xs\ ys$ 
  proof
    fix  $el$ 
    assume  $el-in-list: el \in ?List\ x\ xs\ y\ ys$ 
    show  $el \in ?Hd\ x\ y \cup ?Tl\ xs\ ys$ 
    proof ( $cases\ el$ )
      case  $This$ 
        with  $el-in-list$  show  $?thesis$  by ( $simp\ add: dom-def$ )
    next
      case ( $EName\ en$ )
        show  $?thesis$ 
        proof ( $cases\ en$ )
          case  $Res$ 
            with  $EName\ el-in-list$  show  $?thesis$  by ( $simp\ add: dom-def$ )
          next
            case ( $VNam\ vn$ )
              with  $EName\ el-in-list$  show  $?thesis$ 
              by ( $auto\ simp\ add: dom-def\ dest: map-upds-cut-irrelevant$ )
        qed
      qed
    qed
  next
    show  $?Hd\ x\ y \cup ?Tl\ xs\ ys \subseteq ?List\ x\ xs\ y\ ys$ 
    proof ( $rule\ subsetI$ )
      fix  $el$ 
      assume  $el-in-hd-tl: el \in ?Hd\ x\ y \cup ?Tl\ xs\ ys$ 
      show  $el \in ?List\ x\ xs\ y\ ys$ 
      proof ( $cases\ el$ )
        case  $This$ 
          with  $el-in-hd-tl$  show  $?thesis$  by ( $simp\ add: dom-def$ )
      qed
    qed
  qed

```

```

next
  case (EName en)
  show ?thesis
  proof (cases en)
    case Res
    with EName el-in-hd-tl show ?thesis by (simp add: dom-def)
  next
    case (VNam vn)
    with EName el-in-hd-tl show ?thesis
    by (auto simp add: dom-def intro: map-upds-Some-expand
                    map-upds-Some-insert)
  qed
qed
qed
qed

```

**lemma** *dom-map-upd*:  $\bigwedge \text{tab}. \text{dom } (\text{tab}(x \mapsto y)) = \text{dom } \text{tab} \cup \{x\}$   
**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)  
**with** len **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,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-ename-case-None-simp*:  
 $\text{dom } (\text{ename-case } \text{vname-tab } \text{None}) = \text{VNam } ' (\text{dom } \text{vname-tab})$   
**apply** (auto simp add: dom-def image-def )  
**apply** (case-tac x)  
**apply** auto  
**done**

**lemma** *dom-ename-case-Some-simp*:  
 $\text{dom } (\text{ename-case } \text{vname-tab } (\text{Some } a)) = \text{VNam } ' (\text{dom } \text{vname-tab}) \cup \{\text{Res}\}$   
**apply** (auto simp add: dom-def image-def )  
**apply** (case-tac x)  
**apply** auto

done

**lemma** *dom-lname-case-None-simp*:

*dom (lname-case ename-tab None) = EName ‘ (dom ename-tab)*  
**apply** (*auto simp add: dom-def image-def*)  
**apply** (*case-tac x*)  
**apply** *auto*  
**done**

**lemma** *dom-lname-case-Some-simp*:

*dom (lname-case ename-tab (Some a)) = EName ‘ (dom ename-tab)  $\cup$  {This}*  
**apply** (*auto simp add: dom-def image-def*)  
**apply** (*case-tac x*)  
**apply** *auto*  
**done**

**lemmas** *dom-lname-ename-case-simps* =

*dom-ename-case-None-simp dom-ename-case-Some-simp*  
*dom-lname-case-None-simp dom-lname-case-Some-simp*

**lemma** *image-comp*:

*f ‘ g ‘ A = (f  $\circ$  g) ‘ A*  
**by** (*auto simp add: image-def*)

**lemma** *dom-locals-init-lvars*:

**assumes** *m: m=(methd (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 (empty(pars m[ $\mapsto$ ]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-ename-case-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-ename-case-simps image-comp*)

**qed**

**qed**

**lemma** *da-e2-BinOp*:

**assumes** *da*:  $(\text{prg}=G, \text{cls}=\text{acc}C, \text{lcl}=L) \vdash \text{dom}(\text{locals}(\text{store } s0)) \gg \langle \text{BinOp binop } e1 \ e2 \rangle_e \gg A$   
**and** *wt-e1*:  $(\text{prg}=G, \text{cls}=\text{acc}C, \text{lcl}=L) \vdash e1 :: -e1T$   
**and** *wt-e2*:  $(\text{prg}=G, \text{cls}=\text{acc}C, \text{lcl}=L) \vdash e2 :: -e2T$   
**and** *wt-binop*:  $\text{wt-binop } G \text{ binop } e1T \ e2T$   
**and** *conf-s0*:  $s0 :: \preceq(G, L)$   
**and** *normal-s1*: *normal s1*  
**and** *eval-e1*:  $G \vdash s0 -e1 -\succ v1 \rightarrow s1$   
**and** *conf-v1*:  $G, \text{store } s1 \vdash v1 :: \preceq e1T$   
**and** *wf*: *wf-prog G*  
**shows**  $\exists E2. (\text{prg}=G, \text{cls}=\text{acc}C, \text{lcl}=L) \vdash \text{dom}(\text{locals}(\text{store } s1)) \gg (\text{if need-second-arg binop } v1 \text{ then } \langle e2 \rangle_e \text{ else } \langle \text{Skip} \rangle_s) \gg E2$

**proof** –

**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*)  
**{**  
**assume** *condAnd*: *binop=CondAnd*  
**have** *?thesis*  
**proof** –  
**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]*)  
*(insert wt-e1, simp-all add: e1T v1)*  
**with** *s0-s1* **show** *?thesis* **by** (*rule Un-least*)  
**qed**

```

ultimately
show ?thesis
  using that by (cases rule: da-weakenE) (simp add: True)
qed
}
moreover
{
  assume condOr: binop=CondOr
  have ?thesis

proof -
  from da obtain E2' where
    ( $\langle \text{prg}=G, \text{cls}=\text{acc}C, \text{lcl}=L \rangle$ )
     $\vdash \text{dom} (\text{locals} (\text{store } s0)) \cup \text{assigns-if False } e1 \gg \langle e2 \rangle_e \gg E2'$ 
  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:  $e1T=\text{Prim}T \text{ Boolean}$ 
  by simp
  with normal-s1 conf-v1 obtain b where  $v1=\text{Bool } b$ 
  by (auto dest: conf-Boolean)
  with True condOr
  have v1:  $v1=\text{Bool False}$ 
  by simp
  from eval-e1 normal-s1
  have  $\text{assigns-if False } e1 \subseteq \text{dom} (\text{locals} (\text{store } s1))$ 
  by (rule assigns-if-good-approx' [elim-format])
    (insert wt-e1, simp-all add: e1T v1)
  with s0-s1 show ?thesis by (rule Un-least)
qed
ultimately
show ?thesis
  using that by (rule da-weakenE) (simp add: True)
qed
}
moreover
{
  assume notAndOr:  $\text{binop} \neq \text{CondAnd } \text{binop} \neq \text{CondOr}$ 
  have ?thesis

proof -
  from da notAndOr obtain E1' where
    da-e1: ( $\langle \text{prg}=G, \text{cls}=\text{acc}C, \text{lcl}=L \rangle$ )
       $\vdash \text{dom} (\text{locals} (\text{store } s0)) \gg \langle e1 \rangle_e \gg E1'$ 
    and da-e2: ( $\langle \text{prg}=G, \text{cls}=\text{acc}C, \text{lcl}=L \rangle$ )  $\vdash \text{nrm } E1' \gg \text{In1l } e2 \gg A$ 
  by cases simp+
  from eval-e1 wt-e1 da-e1 wf normal-s1
  have  $\text{nrm } E1' \subseteq \text{dom} (\text{locals} (\text{store } s1))$ 
  by (cases rule: da-good-approxE') iprover
  with da-e2 show ?thesis
  using that by (rule da-weakenE) (simp add: True)
qed
}
ultimately show ?thesis
  by (cases binop) auto
qed
thus ?thesis ..
qed

```

## main proof of type safety

**lemma** *eval-type-sound*:

**assumes** *eval*:  $G \vdash s0 \multimap t \rightarrow (v, s1)$   
**and** *wt*:  $(\text{prg} = G, \text{cls} = \text{acc}C, \text{lcl} = L) \vdash t :: T$   
**and** *da*:  $(\text{prg} = G, \text{cls} = \text{acc}C, \text{lcl} = L) \vdash \text{dom } (\text{locals } (\text{store } s0)) \gg t \gg A$   
**and** *wf*: *wf-prog* *G*  
**and** *conf-s0*:  $s0 :: \preceq (G, L)$   
**shows**  $s1 :: \preceq (G, L) \wedge (\text{normal } s1 \rightarrow G, L, \text{store } s1 \vdash t \gg v :: \preceq T) \wedge$   
 $(\text{error-free } s0 = \text{error-free } s1)$

**proof** –

**note** *inj-term-simps* [*simp*]  
**let** *?TypeSafeObj* =  $\lambda s0 s1 t v.$   
 $\forall L \text{ acc}C T A. s0 :: \preceq (G, L) \rightarrow (\text{prg} = G, \text{cls} = \text{acc}C, \text{lcl} = L) \vdash t :: T$   
 $\rightarrow (\text{prg} = G, \text{cls} = \text{acc}C, \text{lcl} = L) \vdash \text{dom } (\text{locals } (\text{store } s0)) \gg t \gg A$   
 $\rightarrow s1 :: \preceq (G, L) \wedge (\text{normal } s1 \rightarrow G, L, \text{store } s1 \vdash t \gg v :: \preceq T)$   
 $\wedge (\text{error-free } s0 = \text{error-free } s1)$

**from** *eval*

**have**  $\bigwedge L \text{ acc}C T A. \llbracket s0 :: \preceq (G, L); (\text{prg} = G, \text{cls} = \text{acc}C, \text{lcl} = L) \vdash t :: T;$   
 $(\text{prg} = G, \text{cls} = \text{acc}C, \text{lcl} = L) \vdash \text{dom } (\text{locals } (\text{store } s0)) \gg t \gg A \rrbracket$   
 $\implies s1 :: \preceq (G, L) \wedge (\text{normal } s1 \rightarrow G, L, \text{store } s1 \vdash t \gg v :: \preceq T)$   
 $\wedge (\text{error-free } s0 = \text{error-free } s1)$   
**(is** *PROP ?TypeSafe s0 s1 t v*  
**is**  $\bigwedge L \text{ acc}C T A. ?\text{Conform } L s0 \implies ?\text{WellTyped } L \text{ acc}C T t$   
 $\implies ?\text{DefAss } L \text{ acc}C s0 t A$   
 $\implies ?\text{Conform } L s1 \wedge ?\text{ValueTyped } L T s1 t v \wedge$   
 $? \text{ErrorFree } s0 s1)$

**proof** (*induct*)

**case** (*Abrupt xc s t L accC T A*)  
**from**  $\langle (\text{Some } xc, s) :: \preceq (G, L) \rangle$   
**show**  $(\text{Some } xc, s) :: \preceq (G, L) \wedge$   
 $(\text{normal } (\text{Some } xc, s) \rightarrow G, L, \text{store } (\text{Some } xc, s) \vdash t \gg \text{arbitrary3 } t :: \preceq T) \wedge$   
 $(\text{error-free } (\text{Some } xc, s) = \text{error-free } (\text{Some } xc, s))$   
**by** *simp*

**next**

**case** (*Skip s L accC T A*)  
**from**  $\langle \text{Norm } s :: \preceq (G, L) \rangle$  **and**  
 $\langle (\text{prg} = G, \text{cls} = \text{acc}C, \text{lcl} = L) \vdash \text{In1r } \text{Skip} :: T \rangle$   
**show**  $\text{Norm } s :: \preceq (G, L) \wedge$   
 $(\text{normal } (\text{Norm } s) \rightarrow G, L, \text{store } (\text{Norm } s) \vdash \text{In1r } \text{Skip} \gg \Diamond :: \preceq T) \wedge$   
 $(\text{error-free } (\text{Norm } s) = \text{error-free } (\text{Norm } s))$   
**by** *simp*

**next**

**case** (*Expr s0 e v s1 L accC T A*)  
**note**  $\langle G \vdash \text{Norm } s0 \multimap e \multimap v \rightarrow s1 \rangle$   
**note** *hyp* =  $\langle \text{PROP ?TypeSafe } (\text{Norm } s0) s1 (\text{In1l } e) (\text{In1 } v) \rangle$   
**note** *conf-s0* =  $\langle \text{Norm } s0 :: \preceq (G, L) \rangle$   
**moreover**  
**note** *wt* =  $\langle (\text{prg} = G, \text{cls} = \text{acc}C, \text{lcl} = L) \vdash \text{In1r } (\text{Expr } e) :: T \rangle$   
**then obtain** *eT*  
**where**  $(\text{prg} = G, \text{cls} = \text{acc}C, \text{lcl} = L) \vdash \text{In1l } e :: eT$   
**by** (*rule wt-elim-cases*) *blast*  
**moreover**  
**from** *Expr.premis* **obtain** *E* **where**  
 $(\text{prg} = G, \text{cls} = \text{acc}C, \text{lcl} = L) \vdash \text{dom } (\text{locals } (\text{store } ((\text{Norm } s0) :: \text{state}))) \gg \text{In1l } e \gg E$   
**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  $s1::\preceq(G, L) \wedge$ 
  ( $\text{normal } s1 \longrightarrow G, L, \text{store } s1 \vdash \text{In1r } (\text{Expr } e) \succ \Diamond::\preceq T$ )  $\wedge$ 
  ( $\text{error-free } (\text{Norm } s0) = \text{error-free } s1$ )
  by (simp)
next
case (Lab s0 c s1 l L accC T A)
note hyp =  $\langle \text{PROP ?TypeSafe } (\text{Norm } s0) s1 (\text{In1r } c) \Diamond \rangle$ 
note conf-s0 =  $\langle \text{Norm } s0::\preceq(G, L) \rangle$ 
moreover
note wt =  $\langle \langle \text{prg} = G, \text{cls} = \text{accC}, \text{lcl} = L \rangle \vdash \text{In1r } (l \cdot c)::T \rangle$ 
then have  $\langle \text{prg} = G, \text{cls} = \text{accC}, \text{lcl} = L \rangle \vdash c::\checkmark$ 
  by (rule wt-elim-cases) blast
moreover from Lab.premis obtain C where
   $\langle \text{prg}=G, \text{cls}=\text{accC}, \text{lcl}=L \rangle \vdash \text{dom } (\text{locals } (\text{store } ((\text{Norm } s0)::\text{state}))) \gg \text{In1r } c \gg C$ 
  by (elim da-elim-cases) simp
ultimately
obtain conf-s1:  $s1::\preceq(G, L)$  and
  error-free-s1:  $\text{error-free } s1$ 
  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)  $s1::\preceq(G, L) \wedge$ 
  ( $\text{normal } (\text{abupd } (\text{absorb } l) s1) \longrightarrow G, L, \text{store } (\text{abupd } (\text{absorb } l) s1) \vdash \text{In1r } (l \cdot c) \succ \Diamond::\preceq T$ )  $\wedge$ 
  ( $\text{error-free } (\text{Norm } s0) = \text{error-free } (\text{abupd } (\text{absorb } l) s1)$ )
  by (simp)
next
case (Comp s0 c1 s1 c2 s2 L accC T A)
note eval-c1 =  $\langle G \vdash \text{Norm } s0 -c1 \rightarrow s1 \rangle$ 
note eval-c2 =  $\langle G \vdash s1 -c2 \rightarrow s2 \rangle$ 
note hyp-c1 =  $\langle \text{PROP ?TypeSafe } (\text{Norm } s0) s1 (\text{In1r } c1) \Diamond \rangle$ 
note hyp-c2 =  $\langle \text{PROP ?TypeSafe } s1 s2 (\text{In1r } c2) \Diamond \rangle$ 
note conf-s0 =  $\langle \text{Norm } s0::\preceq(G, L) \rangle$ 
note wt =  $\langle \langle \text{prg} = G, \text{cls} = \text{accC}, \text{lcl} = L \rangle \vdash \text{In1r } (c1;; c2)::T \rangle$ 
then 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 (rule wt-elim-cases) blast
from Comp.premis
obtain C1 C2
  where da-c1:  $\langle \text{prg}=G, \text{cls}=\text{accC}, \text{lcl}=L \rangle \vdash$ 
     $\text{dom } (\text{locals } (\text{store } ((\text{Norm } s0)::\text{state}))) \gg \text{In1r } c1 \gg C1$  and
    da-c2:  $\langle \text{prg}=G, \text{cls}=\text{accC}, \text{lcl}=L \rangle \vdash \text{nrm } C1 \gg \text{In1r } c2 \gg C2$ 
  by (elim da-elim-cases) simp
from conf-s0 wt-c1 da-c1
obtain conf-s1:  $s1::\preceq(G, L)$  and
  error-free-s1:  $\text{error-free } s1$ 
  by (rule hyp-c1 [elim-format]) simp
show  $s2::\preceq(G, L) \wedge$ 
  ( $\text{normal } s2 \longrightarrow G, L, \text{store } s2 \vdash \text{In1r } (c1;; c2) \succ \Diamond::\preceq T$ )  $\wedge$ 
  ( $\text{error-free } (\text{Norm } s0) = \text{error-free } s2$ )
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
  ( $\text{prg}=G, \text{cls}=\text{acc}C, \text{lcl}=L$ ) $\vdash \text{dom}(\text{locals}(\text{store } s1)) \gg \text{In1r } c2 \gg C2'$ 
proof –
  from eval-c1 wt-c1 da-c1 wf True
  have  $\text{nrm } C1 \subseteq \text{dom}(\text{locals}(\text{store } s1))$ 
    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  $s2::\preceq(G, L)$  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 =  $\langle G \vdash \text{Norm } s0 -e-\triangleright b \rightarrow s1 \rangle$ 
note eval-then-else =  $\langle G \vdash s1 -(\text{if the-Bool } b \text{ then } c1 \text{ else } c2) \rightarrow s2 \rangle$ 
note hyp-e =  $\langle \text{PROP } ?\text{TypeSafe}(\text{Norm } s0) s1 (\text{In1l } e) (\text{In1 } b) \rangle$ 
note hyp-then-else =
   $\langle \text{PROP } ?\text{TypeSafe } s1 s2 (\text{In1r } (\text{if the-Bool } b \text{ then } c1 \text{ else } c2)) \Diamond \rangle$ 
note conf-s0 =  $\langle \text{Norm } s0::\preceq(G, L) \rangle$ 
note wt =  $\langle (\text{prg} = G, \text{cls} = \text{acc}C, \text{lcl} = L) \vdash \text{In1r } (\text{If}(e) c1 \text{ Else } c2)::T \rangle$ 
then obtain
  wt-e:  $(\text{prg}=G, \text{cls}=\text{acc}C, \text{lcl}=L) \vdash e::\neg \text{Prim}T \text{ Boolean}$  and
  wt-then-else:  $(\text{prg}=G, \text{cls}=\text{acc}C, \text{lcl}=L) \vdash (\text{if the-Bool } b \text{ then } c1 \text{ else } c2)::\checkmark$ 

  by (rule wt-elim-cases) (auto split add: split-if)
from If.premis obtain  $E C$  where
  da-e:  $(\text{prg}=G, \text{cls}=\text{acc}C, \text{lcl}=L) \vdash \text{dom}(\text{locals}(\text{store } ((\text{Norm } s0)::\text{state})))$ 
     $\gg \text{In1l } e \gg E$  and
  da-then-else:
   $(\text{prg}=G, \text{cls}=\text{acc}C, \text{lcl}=L) \vdash$ 
   $(\text{dom}(\text{locals}(\text{store } ((\text{Norm } s0)::\text{state})))) \cup \text{assigns-if } (\text{the-Bool } b) e$ 
   $\gg \text{In1r } (\text{if the-Bool } b \text{ then } c1 \text{ else } c2) \gg C$ 

  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  $s2::\preceq(G, L) \wedge$ 
   $(\text{normal } s2 \longrightarrow G, L, \text{store } s2 \vdash \text{In1r } (\text{If}(e) c1 \text{ Else } c2) \triangleright \Diamond::\preceq T) \wedge$ 
   $(\text{error-free } (\text{Norm } s0) = \text{error-free } s2)$ 
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
   $(\text{prg}=G, \text{cls}=\text{acc}C, \text{lcl}=L) \vdash$ 
   $(\text{dom}(\text{locals}(\text{store } s1))) \gg \text{In1r } (\text{if the-Bool } b \text{ then } c1 \text{ else } c2) \gg C'$ 
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 assigns-if (the-Bool b) e ⊆ dom (locals (store s1))
  by (rule assigns-if-good-approx')
ultimately
have dom (locals (store ((Norm s0)::state)))
   $\cup$  assigns-if (the-Bool b) e ⊆ dom (locals (store s1))
  by (rule Un-least)
with da-then-else show thesis
  by (rule da-weakenE) (rule that)
qed
with conf-s1 wt-then-else
obtain s2::⊢(G, L) and error-free s2
  by (rule hyp-then-else [elim-format]) (simp add: error-free-s1)
with wt show ?thesis
  by simp
qed

```

— Note that we don't have to show that  $b$  really is a boolean value. With *the-Bool* we enforce to get a value of boolean type. So execution will be type safe, even if  $b$  would be a string, for example. We might not expect such a behaviour to be called type safe. To remedy the situation we would have to change the 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 = ⟨G ⊢ Norm s0 -e-> b → s1⟩
note hyp-e = ⟨PROP ?TypeSafe (Norm s0) s1 (In1l e) (In1 b)⟩
note conf-s0 = ⟨Norm s0::⊢(G, L)⟩
note wt = ⟨⟦prg = G, cls = accC, lcl = L⟧ ⊢ In1r (l. While(e) c)::T⟩
then obtain wt-e: ⟨⟦prg = G, cls = accC, lcl = L⟧ ⊢ e::-PrimT Boolean and
  wt-c: ⟨⟦prg = G, cls = accC, lcl = L⟧ ⊢ c::√
  by (rule wt-elim-cases) blast
note da = ⟨⟦prg=G, cls=accC, lcl=L⟧
   $\vdash$  dom (locals(store ((Norm s0)::state))) » In1r (l. While(e) c) » A
then
obtain E C where
  da-e: ⟨⟦prg=G, cls=accC, lcl=L⟧
   $\vdash$  dom (locals (store ((Norm s0)::state))) » In1l e » E and
  da-c: ⟨⟦prg=G, cls=accC, lcl=L⟧
   $\vdash$  (dom (locals (store ((Norm s0)::state)))
   $\cup$  assigns-if True e) » In1r c » C
  by (rule da-elim-cases) simp
from conf-s0 wt-e da-e
obtain conf-s1: s1::⊢(G, L) and error-free-s1: error-free s1
  by (rule hyp-e [elim-format]) simp
show s3::⊢(G, L) ∧
  (normal s3 → G, L, store s3 ⊢ In1r (l. While(e) c) >◇::⊢ T)  $\wedge$ 
  (error-free (Norm s0) = error-free s3)
proof (cases normal s1)
case True
note normal-s1 = this
show ?thesis
proof (cases the-Bool b)
case True
with Loop.hyps obtain
  eval-c: G ⊢ s1 -c→ s2 and
  eval-while: G ⊢ abupd (absorb (Cont l)) s2 -l. While(e) c→ s3
  by simp
have ?TypeSafeObj s1 s2 (In1r c) ◇
  using Loop.hyps True by simp

```

```

note hyp-c = this [rule-format]
have ?TypeSafeObj (abupd (absorb (Cont l)) s2)
  s3 (In1r (l. While(e) c)) ◇
  using Loop.hyps True by simp
note hyp-w = this [rule-format]
from eval-e have
  s0-s1: dom (locals (store ((Norm s0)::state)))
    ⊆ dom (locals (store s1))
  by (rule dom-locals-eval-mono-elim)
obtain C' where
  (⟦prg=G, cls=accC, lcl=L⟧ ⊢ (dom (locals (store s1))) ⟦In1r c⟧ C')
proof –
  note s0-s1
  moreover
  from eval-e normal-s1 wt-e
  have assigns-if True e ⊆ dom (locals (store s1))
    by (rule assigns-if-good-approx' [elim-format]) (simp add: True)
  ultimately
  have dom (locals (store ((Norm s0)::state)))
    ∪ assigns-if True e ⊆ dom (locals (store s1))
    by (rule Un-least)
  with da-c show thesis
    by (rule da-weakenE) (rule that)
qed
with conf-s1 wt-c
obtain conf-s2: s2::≼(G, L) and error-free-s2: error-free s2
  by (rule hyp-c [elim-format]) (simp add: error-free-s1)
from error-free-s2
have error-free-ab-s2: error-free (abupd (absorb (Cont l)) s2)
  by simp
from conf-s2 have abupd (absorb (Cont l)) s2 ::≼(G, L)
  by (cases s2) (auto intro: conforms-absorb)
moreover note wt
moreover
obtain A' where
  (⟦prg=G, cls=accC, lcl=L⟧ ⊢
    dom (locals (store (abupd (absorb (Cont l)) s2)))
    ⟦In1r (l. While(e) c)⟧ A')
proof –
  note s0-s1
  also from eval-c
  have dom (locals (store s1)) ⊆ dom (locals (store s2))
    by (rule dom-locals-eval-mono-elim)
  also have ... ⊆ dom (locals (store (abupd (absorb (Cont l)) s2)))
    by simp
  finally
  have dom (locals (store ((Norm s0)::state))) ⊆ ...
  with da show thesis
    by (rule da-weakenE) (rule that)
qed
ultimately obtain s3::≼(G, L) and error-free s3
  by (rule hyp-w [elim-format]) (simp add: error-free-ab-s2)
with wt show ?thesis
  by simp
next
case False
with Loop.hyps have s3=s1 by simp
with conf-s1 error-free-s1 wt
show ?thesis

```

```

    by simp
  qed
next
  case False
  have s3=s1
  proof -
    from False obtain abr where abr: abrupt s1 = Some abr
    by (cases s1) auto
    from eval-e - wt-e have no-jmp:  $\bigwedge j. \text{abrupt } s1 \neq \text{Some } (\text{Jump } j)$ 
    by (rule eval-expression-no-jump
      [where ?Env=(|prg=G,cls=accC,lcl=L|),simplified])
      (simp-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:  $G \vdash \text{abupd } (\text{absorb } (\text{Cont } l)) \ s2 \rightarrow l \cdot \text{While}(e) \ c \rightarrow s3$ 
    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
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):: $\preceq(G, L)$ 
  moreover
  from Jmp.prem
  have j=Ret  $\longrightarrow \text{Result} \in \text{dom } (\text{locals } (\text{store } ((\text{Norm } s)::\text{state})))$ 
  by (elim da-elim-cases)
  ultimately have (Some (Jump j), s):: $\preceq(G, L)$  by auto
  then
  show (Some (Jump j), s):: $\preceq(G, L) \wedge$ 
    (normal (Some (Jump j), s)
       $\longrightarrow G, L, \text{store } (\text{Some } (\text{Jump } j), s) \vdash \text{In1r } (\text{Jmp } j) \succ \Diamond::\preceq T) \wedge$ 
      (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 $\vdash$ Norm s0  $\rightarrow e \rightarrow a \rightarrow s1$ )
    note hyp = (PROP ?TypeSafe (Norm s0) s1 (In1l e) (In1 a))
    note conf-s0 = (Norm s0):: $\preceq(G, L)$ 
    note wt = (|prg = G, cls = accC, lcl = L| $\vdash$ In1r (Throw e)::T)
    then obtain tn
      where wt-e: (|prg = G, cls = accC, lcl = L| $\vdash$ e::-Class tn and
        throwable:  $G \vdash tn \preceq_C \text{SXcpt Throwable}$ 
      by (rule wt-elim-cases) (auto)

```

```

from Throw.prems obtain E where
  da-e: ( $\Downarrow \text{prg} = G, \text{cls} = \text{acc}C, \text{lcl} = L$ )
     $\vdash \text{dom} (\text{locals} (\text{store} ((\text{Norm } s0)::\text{state}))) \gg \text{In1l } e \gg E$ 
  by (elim da-elim-cases) simp
from conf-s0 wt-e da-e obtain
  s1:: $\preceq(G, L)$  and
  (normal s1  $\longrightarrow G, \text{store } s1 \vdash a::\preceq \text{Class } tn$ ) and
  error-free-s1: error-free s1
  by (rule hyp [elim-format]) simp
with wf throwable
have abupd (throw a) s1:: $\preceq(G, L)$ 
  by (cases s1) (auto dest: Throw-lemma)
with wt error-free-s1
show abupd (throw a) s1:: $\preceq(G, L) \wedge$ 
  (normal (abupd (throw a) s1)  $\longrightarrow$ 
     $G, L, \text{store} (\text{abupd} (\text{throw } a) s1) \vdash \text{In1r} (\text{Throw } e) \succ \Diamond::\preceq T$ )  $\wedge$ 
    (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 =  $\langle G \vdash \text{Norm } s0 -c1 \rightarrow s1 \rangle$ 
note sx-alloc =  $\langle G \vdash s1 -\text{salloc} \rightarrow s2 \rangle$ 
note hyp-c1 =  $\langle \text{PROP } ?\text{TypeSafe} (\text{Norm } s0) s1 (\text{In1r } c1) \Diamond \rangle$ 
note conf-s0 =  $\langle \text{Norm } s0::\preceq(G, L) \rangle$ 
note wt =  $\langle \Downarrow \text{prg} = G, \text{cls} = \text{acc}C, \text{lcl} = L \rangle \vdash \text{In1r} (\text{Try } c1 \text{ Catch}(\text{catchC } vn) c2)::T$ 
then obtain
  wt-c1: ( $\Downarrow \text{prg} = G, \text{cls} = \text{acc}C, \text{lcl} = L$ )  $\vdash c1::\checkmark$  and
  wt-c2: ( $\Downarrow \text{prg} = G, \text{cls} = \text{acc}C, \text{lcl} = L(V\text{Name } vn \mapsto \text{Class } \text{catchC})$ )  $\vdash c2::\checkmark$  and
  fresh-vn:  $L(V\text{Name } vn) = \text{None}$ 
  by (rule wt-elim-cases) simp
from Try.prems obtain C1 C2 where
  da-c1: ( $\Downarrow \text{prg} = G, \text{cls} = \text{acc}C, \text{lcl} = L$ )
     $\vdash \text{dom} (\text{locals} (\text{store} ((\text{Norm } s0)::\text{state}))) \gg \text{In1r } c1 \gg C1$  and
  da-c2:
    ( $\Downarrow \text{prg} = G, \text{cls} = \text{acc}C, \text{lcl} = L(V\text{Name } vn \mapsto \text{Class } \text{catchC})$ )
     $\vdash (\text{dom} (\text{locals} (\text{store} ((\text{Norm } s0)::\text{state}))) \cup \{V\text{Name } vn\}) \gg \text{In1r } c2 \gg C2$ 
  by (elim da-elim-cases) simp
from conf-s0 wt-c1 da-c1
obtain conf-s1: s1:: $\preceq(G, L)$  and error-free-s1: error-free s1
  by (rule hyp-c1 [elim-format]) simp
from conf-s1 sx-alloc wf
have conf-s2: s2:: $\preceq(G, L)$ 
  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:: $\preceq(G, L) \wedge$ 
  (normal s3  $\longrightarrow G, L, \text{store } s3 \vdash \text{In1r} (\text{Try } c1 \text{ Catch}(\text{catchC } vn) c2) \succ \Diamond::\preceq T$ )  $\wedge$ 
  (error-free (Norm s0) = error-free s3)
proof (cases  $\exists x. \text{abrupt } s1 = \text{Some } (X\text{cpt } x)$ )
  case False
  from sx-alloc wf
  have eq-s2-s1: s2=s1
    by (rule sxalloc-type-sound [elim-format])
    (insert False, auto split: option.splits abrupt.splits)
  with False
  have  $\neg G, s2 \vdash \text{catch } \text{catchC}$ 
    by (simp add: catch-def)
  with Try

```

```

have  $s3=s2$ 
  by simp
with  $wt\ conf\text{-}s1\ error\text{-}free\text{-}s1\ eq\text{-}s2\text{-}s1$ 
show  $?thesis$ 
  by simp
next
  case True
  note  $exception\text{-}s1 = this$ 
  show  $?thesis$ 
  proof ( $cases\ G, s2 \vdash catch\ catchC$ )
    case False
    with Try
    have  $s3=s2$ 
      by simp
    with  $wt\ conf\text{-}s2\ error\text{-}free\text{-}s2$ 
    show  $?thesis$ 
      by simp
  next
  case True
  with Try have  $G \vdash new\text{-}xcpt\text{-}var\ vn\ s2 \text{---} c2 \rightarrow s3$  by simp
  from True Try.hyps
  have  $?TypeSafeObj\ (new\text{-}xcpt\text{-}var\ vn\ s2)\ s3\ (In1r\ c2) \Diamond$ 
    by simp
  note  $hyp\text{-}c2 = this$  [rule-format]
  from  $exception\text{-}s1\ sx\text{-}alloc\ wf$ 
  obtain  $a$ 
    where  $xcpt\text{-}s2: abrupt\ s2 = Some\ (Xcpt\ (Loc\ a))$ 
    by ( $auto\ dest!; sxalloc\text{-}type\text{-}sound\ split: option.splits\ abrupt.splits$ )
  with True
  have  $G \vdash obj\text{-}ty\ (the\ (globs\ (store\ s2)\ (Heap\ a))) \preceq Class\ catchC$ 
    by ( $cases\ s2$ ) simp
  with  $xcpt\text{-}s2\ conf\text{-}s2\ wf$ 
  have  $new\text{-}xcpt\text{-}var\ vn\ s2 :: \preceq (G, L(VName\ vn \mapsto Class\ catchC))$ 
    by ( $auto\ dest: Try\text{-}lemma$ )
  moreover note  $wt\text{-}c2$ 
  moreover
  obtain  $C2'$  where
    ( $\langle prg=G, cls=accC, lcl=L(VName\ vn \mapsto Class\ catchC) \rangle$ )
     $\vdash (dom\ (locals\ (store\ (new\text{-}xcpt\text{-}var\ vn\ s2)))) \gg In1r\ c2 \gg C2'$ 
  proof –
    have  $(dom\ (locals\ (store\ ((Norm\ s0)::state))) \cup \{VName\ vn\})$ 
       $\subseteq dom\ (locals\ (store\ (new\text{-}xcpt\text{-}var\ vn\ s2)))$ 
    proof –
      from  $\langle G \vdash Norm\ s0 \text{---} c1 \rightarrow s1 \rangle$ 
      have  $dom\ (locals\ (store\ ((Norm\ s0)::state)))$ 
         $\subseteq dom\ (locals\ (store\ s1))$ 
      by (rule dom-locals-eval-mono-elim)
      also
      from  $sx\text{-}alloc$ 
      have  $\dots \subseteq dom\ (locals\ (store\ s2))$ 
        by (rule dom-locals-sxalloc-mono)
      also
      have  $\dots \subseteq dom\ (locals\ (store\ (new\text{-}xcpt\text{-}var\ vn\ s2)))$ 
        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 show thesis
    by (rule da-weakenE) (rule that)
  qed
  ultimately
  obtain conf-s3:  $s3 :: \preceq(G, L(VName\ vn \mapsto Class\ catchC))$  and
    error-free-s3: error-free s3
    by (rule hyp-c2 [elim-format])
    (cases s2, simp add: xcpt-s2 error-free-s2)
  from conf-s3 fresh-vn
  have  $s3 :: \preceq(G, L)$ 
    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 eval-c1 =  $\langle G \vdash Norm\ s0 \rightarrow c1 \rightarrow (x1, s1) \rangle$ 
  note eval-c2 =  $\langle G \vdash Norm\ s1 \rightarrow c2 \rightarrow s2 \rangle$ 
  note s3 =  $\langle s3 = (if\ \exists\ err.\ x1 = Some\ (Error\ err)\ then\ (x1, s1)\ else\ abrupt\ (abrupt-if\ (x1 \neq None)\ x1)\ s2) \rangle$ 
  note hyp-c1 =  $\langle PROP\ ?TypeSafe\ (Norm\ s0)\ (x1, s1)\ (In1r\ c1)\ \Diamond \rangle$ 
  note hyp-c2 =  $\langle PROP\ ?TypeSafe\ (Norm\ s1)\ s2\ (In1r\ c2)\ \Diamond \rangle$ 
  note conf-s0 =  $\langle Norm\ s0 :: \preceq(G, L) \rangle$ 
  note wt =  $\langle \langle prg = G, cls = accC, lcl = L \rangle \vdash In1r\ (c1\ Finally\ c2) :: T \rangle$ 
  then obtain
    wt-c1:  $\langle \langle prg = G, cls = accC, lcl = L \rangle \vdash c1 :: \checkmark \rangle$  and
    wt-c2:  $\langle \langle prg = G, cls = accC, lcl = L \rangle \vdash c2 :: \checkmark \rangle$ 
    by (rule wt-elim-cases) blast
  from Fin.prems obtain C1 C2 where
    da-c1:  $\langle \langle prg = G, cls = accC, lcl = L \rangle \vdash dom\ (locals\ (store\ ((Norm\ s0) :: state))) \rangle \gg In1r\ c1 \gg C1$  and
    da-c2:  $\langle \langle prg = G, cls = accC, lcl = L \rangle \vdash dom\ (locals\ (store\ ((Norm\ s0) :: state))) \rangle \gg In1r\ c2 \gg C2$ 
    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  $\langle \langle prg = G, cls = accC, lcl = L \rangle \vdash dom\ (locals\ (store\ ((Norm\ s1) :: state))) \rangle \gg In1r\ c2 \gg C2'$ 
  proof -
    from eval-c1
    have  $dom\ (locals\ (store\ ((Norm\ s0) :: state))) \subseteq dom\ (locals\ (store\ (x1, s1)))$ 
    by (rule dom-locals-eval-mono-elim)
    hence  $dom\ (locals\ (store\ ((Norm\ s0) :: state))) \subseteq dom\ (locals\ (store\ ((Norm\ s1) :: state)))$ 
    by simp
    with da-c2 show thesis
    by (rule da-weakenE) (rule that)
  qed
  ultimately

```



```

obtain conf-s2:  $s2 :: \preceq(G, L)$  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  $s3 :: \preceq(G, L) \wedge$ 
  ( $\text{normal } s3 \longrightarrow G, L, \text{store } s3 \vdash \text{In1r } (c1 \text{ Finally } c2) \triangleright \Diamond :: \preceq T$ )  $\wedge$ 
  ( $\text{error-free } (\text{Norm } s0) = \text{error-free } s3$ )
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  $\text{error-free } (\text{abrupt-if } \text{True } (\text{Some } x) (\text{abrupt } s2), \text{store } s2)$ 
  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 =  $\langle \text{the } (\text{class } G C) = c \rangle$ 
  note conf-s0 =  $\langle \text{Norm } s0 :: \preceq(G, L) \rangle$ 
  note wt =  $\langle \langle \text{prg} = G, \text{cls} = \text{accC}, \text{lcl} = L \rangle \vdash \text{In1r } (\text{Init } C) :: T \rangle$ 
  with cls
  have cls-C:  $\text{class } G C = \text{Some } c$ 
  by - (erule wt-elim-cases, auto)
  show  $s3 :: \preceq(G, L) \wedge (\text{normal } s3 \longrightarrow G, L, \text{store } s3 \vdash \text{In1r } (\text{Init } C) \triangleright \Diamond :: \preceq T) \wedge$ 
  ( $\text{error-free } (\text{Norm } s0) = \text{error-free } s3$ )
  proof (cases inited C (globs s0))
  case True
  with Init.hyps have  $s3 = \text{Norm } s0$ 
  by simp
  with conf-s0 wt show ?thesis
  by simp
next
  case False
  with Init.hyps obtain
    eval-init-super:
       $G \vdash \text{Norm } ((\text{init-class-obj } G C) s0)$ 
       $-(\text{if } C = \text{Object then Skip else Init } (\text{super } c)) \rightarrow s1$  and
    eval-init:  $G \vdash (\text{set-lvars empty}) s1 \text{ --init } c \rightarrow s2$  and
     $s3 : s3 = (\text{set-lvars } (\text{locals } (\text{store } s1))) s2$ 
  by simp
  have ?TypeSafeObj ( $\text{Norm } ((\text{init-class-obj } G C) s0)$ ) s1
    ( $\text{In1r } (\text{if } C = \text{Object then Skip else Init } (\text{super } c))) \Diamond$ 
  using False Init.hyps by simp
  note hyp-init-super = this [rule-format]
  have ?TypeSafeObj  $((\text{set-lvars empty}) s1) s2 (\text{In1r } (\text{init } c)) \Diamond$ 
  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)):: $\preceq$ (G, L)
  by (auto dest: conforms-init-class-obj)
moreover from wf cls-C have
  wt-init-super: ( $\lfloor \text{prg} = G, \text{cls} = \text{acc}C, \text{lcl} = L \rfloor$ )
     $\vdash$  (if C = Object then Skip else Init (super c)):: $\checkmark$ 
  by (cases C=Object)
    (auto dest: wf-prog-cdecl wf-cdecl-supD is-acc-classD)
moreover
obtain S where
  da-init-super:
    ( $\lfloor \text{prg} = G, \text{cls} = \text{acc}C, \text{lcl} = L \rfloor$ )
     $\vdash$  dom (locals (store ((Norm ((init-class-obj G C) s0))::state)))
      »In1r (if C = Object then Skip else Init (super c))» S
proof (cases C=Object)
  case True
    with da-Skip show ?thesis
      using that by (auto intro: assigned.select-convs)
  next
    case False
      with da-Init show ?thesis
        by – (rule that, auto intro: assigned.select-convs)
qed
ultimately
obtain conf-s1: s1:: $\preceq$ (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:  $\bigwedge j. \text{abrupt } s1 \neq \text{Some } (\text{Jump } j)$ 
  by – (rule eval-statement-no-jump [where ?Env= $\lfloor \text{prg} = G, \text{cls} = \text{acc}C, \text{lcl} = L \rfloor$ , auto])
with conf-s1
have (set-lvars empty) s1:: $\preceq$ (G, empty)
  by (cases s1) (auto intro: conforms-set-locals)
moreover
from error-free-s1
have error-free-empty: error-free ((set-lvars empty) s1)
  by simp
from cls-C wf have wt-init-c: ( $\lfloor \text{prg} = G, \text{cls} = C, \text{lcl} = \text{empty} \rfloor$ ) $\vdash$ (init c):: $\checkmark$ 
  by (rule wf-prog-cdecl [THEN wf-cdecl-wt-init])
moreover from cls-C wf obtain I
  where ( $\lfloor \text{prg} = G, \text{cls} = C, \text{lcl} = \text{empty} \rfloor$ ) $\vdash$  { } »In1r (init c)» I
  by (rule wf-prog-cdecl [THEN wf-cdeclE,simplified]) blast

then obtain I' where
  ( $\lfloor \text{prg} = G, \text{cls} = C, \text{lcl} = \text{empty} \rfloor$ ) $\vdash$  dom (locals (store ((set-lvars empty) s1)))
    »In1r (init c)» I'
  by (rule da-weakenE) simp
ultimately
obtain conf-s2: s2:: $\preceq$ (G, empty) and error-free-s2: error-free s2
  by (rule hyp-init-c [elim-format]) (simp add: error-free-empty)
have abrupt s2  $\neq$  Some (Jump Ret)
proof –
  from s1-no-ret
  have  $\bigwedge j. \text{abrupt } ((\text{set-lvars empty}) s1) \neq \text{Some } (\text{Jump } j)$ 
    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=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
  ⟨prg=G,cls=accC,lcl=L⟩
  ⊢ dom (locals (store ((Norm s0)::state))) »In1r (Init C)» I
  by (auto intro: da-Init [simplified] assigned.select-convs)

ultimately
obtain conf-s1: s1::≲(G, L) and error-free-s1: error-free s1
  by (rule hyp [elim-format]) simp
from conf-s1 halloc wf is-cls-C
obtain halloc-type-safe: s2::≲(G, L)
  (normal s2 → G,store s2⊢Addr a::≲Class C)
  by (cases s2) (auto dest!: halloc-type-sound)
from halloc error-free-s1
have error-free s2
  by (rule error-free-halloc)
with halloc-type-safe T
show s2::≲(G, L) ∧
  (normal s2 → G,L,store s2⊢In1l (NewC C)>In1 (Addr a)::≲T) ∧
  (error-free (Norm s0) = error-free s2)
  by auto
next
case (NewA s0 elT s1 e i s2 a s3 L accC T A)
note eval-init = ⟨G⊢Norm s0 -init-comp-ty elT→ s1⟩
note eval-e = ⟨G⊢s1 -e->i→ s2⟩
note halloc = ⟨G⊢abupd (check-neg i) s2-halloc Arr elT (the-Intg i)>a→ s3⟩
note hyp-init = ⟨PROP ?TypeSafe (Norm s0) s1 (In1r (init-comp-ty elT))⟩ ◇
note hyp-size = ⟨PROP ?TypeSafe s1 s2 (In1l e) (In1 i)⟩
note conf-s0 = ⟨Norm s0::≲(G, L)⟩
note wt = ⟨⟦prg = G, cls = accC, lcl = L⟧⊢In1l (New elT[e])::T⟩
then obtain
  wt-init: ⟨prg = G, cls = accC, lcl = L⟩⊢init-comp-ty elT::√ and

```

```

    wt-size: ( $\text{prg} = G, \text{cls} = \text{acc}C, \text{lcl} = L$ )  $\vdash e :: \text{--PrimT Integer}$  and
      elT: is-type  $G$  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
have da-e: ( $\text{prg} = G, \text{cls} = \text{acc}C, \text{lcl} = L$ )
   $\vdash \text{dom } (\text{locals } (\text{store } ((\text{Norm } s0)::\text{state}))) \gg \text{In1l } e \gg A$ 
  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
    ( $\text{prg} = G, \text{cls} = \text{acc}C, \text{lcl} = L$ )
     $\vdash \text{dom } (\text{locals } (\text{store } ((\text{Norm } s0)::\text{state}))) \gg \text{In1r } (\text{init-comp-ty } \text{elT}) \gg I$ 
proof (cases  $\exists C. \text{elT} = \text{Class } C$ )
  case True
  thus ?thesis
    by – (rule that, (auto intro: da-Init [simplified]
      assigned.select-convs
      simp add: init-comp-ty-def))

  next
  case False
  thus ?thesis
    by – (rule that, (auto intro: da-Skip [simplified]
      assigned.select-convs
      simp add: init-comp-ty-def))

  qed
  ultimately show thesis
    by (rule hyp-init [elim-format]) (auto intro: that)
qed
obtain conf-s2:  $s2 :: \preceq(G, L)$  and error-free-s2: error-free  $s2$ 
proof –
  from eval-init
  have dom (locals (store ((Norm  $s0$ )::state)))  $\subseteq \text{dom } (\text{locals } (\text{store } s1))$ 
    by (rule dom-locals-eval-mono-elim)
  with da-e
  obtain A' where
    ( $\text{prg} = G, \text{cls} = \text{acc}C, \text{lcl} = L$ )
     $\vdash \text{dom } (\text{locals } (\text{store } s1)) \gg \text{In1l } e \gg A'$ 
    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  $i$ )  $s2 :: \preceq(G, L)$ 
  by (cases  $s2$ ) auto
with halloc wf elT
have halloc-type-safe:
   $s3 :: \preceq(G, L) \wedge (\text{normal } s3 \longrightarrow G, \text{store } s3 \vdash \text{Addr } a :: \preceq \text{elT}.)$ 
  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  $s3 :: \preceq(G, L) \wedge$ 
   $(\text{normal } s3 \longrightarrow G, L, \text{store } s3 \vdash \text{In1l } (\text{New } \text{elT}[e]) \gg \text{In1 } (\text{Addr } a) :: \preceq T) \wedge$ 
   $(\text{error-free } (\text{Norm } s0) = \text{error-free } s3)$ 

```

```

  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
{
  assume norm-s2: normal s2
  have G,L,store s2 ⊢ In1l (Cast castT e) > In1 v :: ≤ T
  proof -
    from s2 norm-s2 have normal s1
    by (cases s1) simp
    with v-ok
    have G,store s1 ⊢ v :: ≤ eT
    by simp
    with eT wf s2 T norm-s2
    show ?thesis
    by (cases s1) (auto dest: fits-conf)
  qed
}
with conf-s2 error-free-s2
show s2 :: ≤ (G, L) ∧
  (normal s2 → G,L,store s2 ⊢ In1l (Cast castT e) > In1 v :: ≤ T) ∧
  (error-free (Norm s0) = error-free s2)
  by blast
next
case (Inst s0 e v s1 b instT L accC T A)
note hyp = ⟨PROP ?TypeSafe (Norm s0) s1 (In1l e) (In1 v)⟩
note conf-s0 = ⟨Norm s0 :: ≤ (G, L)⟩
from Inst.prem obtain eT
where wt-e: (prg = G, cls = accC, lcl = L) ⊢ e :: -RefT eT and
  T: T = Inl (PrimT Boolean)
  by (elim wt-elim-cases) simp
from Inst.prem
have da-e: (prg = G, cls = accC, lcl = L)
  ⊢ dom (locals (store ((Norm s0) :: state))) » In1l e » A
  by (elim da-elim-cases) simp

```

```

from conf-s0 wt-e da-e
obtain conf-s1: s1::≤(G, L) and
      v-ok: normal s1 → G,store s1⊢v::≤RefT eT and
      error-free-s1: error-free s1
by (rule hyp [elim-format]) simp
with T show ?case
by simp
next
case (Lit s v L accC T A)
then show ?case
by (auto elim!: wt-elim-cases
      intro: conf-litval simp add: empty-dt-def)
next
case (UnOp s0 e v s1 unop L accC T A)
note hyp = ⟨PROP ?TypeSafe (Norm s0) s1 (In1l e) (In1 v)⟩
note conf-s0 = ⟨Norm s0::≤(G, L)⟩
note wt = ⟨(prg = G, cls = accC, lcl = L)⊢In1l (UnOp unop e)::T⟩
then obtain eT
where wt-e: (prg = G, cls = accC, lcl = L)⊢e::-eT and
      wt-unop: wt-unop unop eT and
      T: T=Inl (PrimT (unop-type unop))
by (auto elim!: wt-elim-cases)
from UnOp.prems obtain A where
      da-e: (prg=G,cls=accC,lcl=L)
      ⊢ dom (locals (store ((Norm s0)::state))) »In1l e» A
by (elim da-elim-cases) simp
from conf-s0 wt-e da-e
obtain conf-s1: s1::≤(G, L) and
      wt-v: normal s1 → G,store s1⊢v::≤eT and
      error-free-s1: error-free s1
by (rule hyp [elim-format]) simp
from wt-v T wt-unop
have normal s1 → G,L,snd s1⊢In1l (UnOp unop e)⋗In1 (eval-unop unop v)::≤T
by (cases unop) auto
with conf-s1 error-free-s1
show s1::≤(G, L) ∧
      (normal s1 → G,L,snd s1⊢In1l (UnOp unop e)⋗In1 (eval-unop unop v)::≤T) ∧
      error-free (Norm s0) = error-free s1
by simp
next
case (BinOp s0 e1 v1 s1 binop e2 v2 s2 L accC T A)
note eval-e1 = ⟨G⊢Norm s0 -e1-⋗v1→ s1⟩
note eval-e2 = ⟨G⊢s1 -(if need-second-arg binop v1 then In1l e2
      else In1r Skip)⋗→ (In1 v2, s2)⟩
note hyp-e1 = ⟨PROP ?TypeSafe (Norm s0) s1 (In1l e1) (In1 v1)⟩
note hyp-e2 = ⟨PROP ?TypeSafe s1 s2
      (if need-second-arg binop v1 then In1l e2 else In1r Skip)
      (In1 v2)⟩
note conf-s0 = ⟨Norm s0::≤(G, L)⟩
note wt = ⟨(prg = G, cls = accC, lcl = L)⊢In1l (BinOp binop e1 e2)::T⟩
then obtain e1T e2T where
      wt-e1: (prg = G, cls = accC, lcl = L)⊢e1::-e1T and
      wt-e2: (prg = G, cls = accC, lcl = L)⊢e2::-e2T and
      wt-binop: wt-binop G binop e1T e2T and
      T: T=Inl (PrimT (binop-type binop))
by (elim wt-elim-cases) simp
have wt-Skip: (prg = G, cls = accC, lcl = L)⊢Skip::✓
by simp
obtain S where

```

```

daSkip: (|prg=G,cls=accC,lcl=L|)
  ⊢ dom (locals (store s1)) »In1r Skip» S
by (auto intro: da-Skip [simplified] assigned.select-convs)
note da = (|prg=G,cls=accC,lcl=L|) ⊢ dom (locals (store ((Norm s0)::state)))
  »(BinOp binop e1 e2)e» A)
then obtain E1 where
  da-e1: (|prg=G,cls=accC,lcl=L|)
    ⊢ dom (locals (store ((Norm s0)::state))) »In1l e1» E1
  by (elim da-elim-cases) simp+
from conf-s0 wt-e1 da-e1
obtain conf-s1: s1::≼(G, L) and
  wt-v1: normal s1 ⟶ G,store s1⊢v1::≼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⊢In1l (BinOp binop e1 e2)⊢In1 (eval-binop binop v1 v2)::≼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 an 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))) ⊆ dom (locals (store s1))
  by (rule dom-locals-eval-mono-elim)
show s2::≼(G, L) ∧
  (normal s2 ⟶
    G,L,snd s2⊢In1l (BinOp binop e1 e2)⊢In1 (eval-binop binop v1 v2)::≼T) ∧
  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  $s2::\preceq(G, L)$  and error-free  $s2$ 
      by (rule hyp-e2' [elim-format]) (simp add: error-free-s1)
    with conf-v show ?thesis by simp
  qed
next
case (Super  $s$   $L$  accC  $T$   $A$ )
note conf-s =  $\langle \text{Norm } s::\preceq(G, L) \rangle$ 
note  $wt = \langle \langle \text{prg} = G, \text{cls} = \text{accC}, \text{lcl} = L \rangle \vdash \text{In1l Super}::T \rangle$ 
then obtain  $C$   $c$  where
   $C: L \text{ This} = \text{Some } (\text{Class } C)$  and
  neg-Obj:  $C \neq \text{Object}$  and
  cls-C:  $\text{class } G \ C = \text{Some } c$  and
   $T: T = \text{Inl } (\text{Class } (\text{super } c))$ 
  by (rule wt-elim-cases) auto
from Super.prems
obtain  $\text{This} \in \text{dom } (\text{locals } s)$ 
  by (elim da-elim-cases) simp
with conf-s  $C$  have  $G, s \vdash \text{val-this } s::\preceq \text{Class } C$ 
  by (auto dest: conforms-localD [THEN wlconfD])
with neg-Obj cls-C wf
have  $G, s \vdash \text{val-this } s::\preceq \text{Class } (\text{super } c)$ 
  by (auto intro: conf-widen
      dest: subcls-direct[THEN widen.subcls])
with  $T$  conf-s
show  $\text{Norm } s::\preceq(G, L) \wedge$ 
  (normal ( $\text{Norm } s$ )  $\longrightarrow$ 
     $G, L, \text{store } (\text{Norm } s) \vdash \text{In1l Super} \succ \text{In1 } (\text{val-this } s)::\preceq T) \wedge$ 
  (error-free ( $\text{Norm } s$ ) = error-free ( $\text{Norm } s$ ))
  by simp
next
case (Acc  $s0$   $v$   $w$  upd  $s1$   $L$  accC  $T$   $A$ )
note hyp =  $\langle \text{PROP } ?\text{TypeSafe } (\text{Norm } s0) \ s1 \ (\text{In2 } v) \ (\text{In2 } (w, \text{upd})) \rangle$ 
note conf-s0 =  $\langle \text{Norm } s0::\preceq(G, L) \rangle$ 
from Acc.prems obtain  $vT$  where
   $wt\text{-}v: \langle \langle \text{prg} = G, \text{cls} = \text{accC}, \text{lcl} = L \rangle \vdash v::=vT \text{ and}$ 
   $T: T = \text{Inl } vT$ 
  by (elim wt-elim-cases) simp
from Acc.prems obtain  $V$  where
   $da\text{-}v: \langle \langle \text{prg} = G, \text{cls} = \text{accC}, \text{lcl} = L \rangle$ 
     $\vdash \text{dom } (\text{locals } (\text{store } ((\text{Norm } s0)::\text{state}))) \gg \text{In2 } v \gg V$ 
  by (cases  $\exists n. v = \text{LVar } n$ ) (insert da.LVar, auto elim!: da-elim-cases)
{
  fix  $n$  assume lvar:  $v = \text{LVar } n$ 
  have  $\text{locals } (\text{store } s1) \ n \neq \text{None}$ 
  proof -
    from Acc.prems lvar have
       $n \in \text{dom } (\text{locals } s0)$ 
      by (cases  $\exists n. v = \text{LVar } n$ ) (auto elim!: da-elim-cases)
    also
    have  $\text{dom } (\text{locals } s0) \subseteq \text{dom } (\text{locals } (\text{store } s1))$ 
    proof -
      from  $\langle G \vdash \text{Norm } s0 \text{ --}v\text{--}\succ(w, \text{upd}) \rightarrow s1 \rangle$ 
      show ?thesis
        by (rule dom-locals-eval-mono-elim) simp
    qed
    finally show ?thesis
      by blast
  qed
}

```



```

} note lvar-in-locals = this
from conf-s0 wt-v da-v
obtain conf-s1: s1::≤(G, L)
  and conf-var: (normal s1 → G,L,store s1⊢In2 v>In2 (w, upd)::≤Inl vT)
  and error-free-s1: error-free s1
  by (rule hyp [elim-format]) simp
from lvar-in-locals conf-var T
have (normal s1 → G,L,store s1⊢In1l (Acc v)>In1 w::≤T)
  by (cases ∃ n. v=LVar n) auto
with conf-s1 error-free-s1 show ?case
  by simp
next
case (Ass s0 var w upd s1 e v s2 L accC T A)
note eval-var = ⟨G⊢Norm s0 -var=>(w, upd)→ s1⟩
note eval-e = ⟨G⊢s1 -e->v→ s2⟩
note hyp-var = ⟨PROP ?TypeSafe (Norm s0) s1 (In2 var) (In2 (w,upd))⟩
note hyp-e = ⟨PROP ?TypeSafe s1 s2 (In1l e) (In1 v)⟩
note conf-s0 = ⟨Norm s0::≤(G, L)⟩
note wt = ⟨(prg = G, cls = accC, lcl = L)⊢In1l (var:=e)::T⟩
then obtain varT eT where
  wt-var: (prg = G, cls = accC, lcl = L)⊢var::=varT and
  wt-e: (prg = G, cls = accC, lcl = L)⊢e::=eT and
  widen: G⊢eT≤varT and
  T: T=Inl eT
  by (rule wt-elim-cases) auto
show assign upd v s2::≤(G, L) ∧
  (normal (assign upd v s2) →
    G,L,store (assign upd v s2)⊢In1l (var:=e)>In1 v::≤T) ∧
  (error-free (Norm s0) = error-free (assign upd v s2))
proof (cases ∃ vn. var=LVar vn)
case False
with Ass.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)::≤Inl varT
  and error-free-s1: error-free s1
  by (rule hyp-var [elim-format]) simp
show ?thesis
proof (cases normal s1)
case False
with eval-e have s2=s1 by auto
with False have assign upd v s2=s1
  by simp
with conf-s1 error-free-s1 False show ?thesis
  by auto
next
case True
note normal-s1=this
obtain A' where (prg=G,cls=accC,lcl=L)
  ⊢ dom (locals (store s1)) »In1l e» A'
proof -
  from eval-var wt-var da-var wf normal-s1
  have nrm V ⊆ dom (locals (store s1))

```

```

    by (cases rule: da-good-approxE') iprover
  with da-e show thesis
    by (rule da-weakenE) (rule that)
qed
with conf-s1 wt-e
obtain conf-s2: s2::≤(G, L) and
  conf-v: normal s2 → G,store s2⊢v::≤eT and
  error-free-s2: error-free s2
  by (rule hyp-e [elim-format]) (simp add: error-free-s1)
show ?thesis
proof (cases normal s2)
  case False
  with conf-s2 error-free-s2
  show ?thesis
    by auto
next
  case True
  from True conf-v
  have conf-v-eT: G,store s2⊢v::≤eT
    by simp
  with widen wf
  have conf-v-varT: G,store s2⊢v::≤varT
    by (auto intro: conf-widen)
  from normal-s1 conf-var
  have G,L,store s1⊢In2 var>In2 (w, upd)::≤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: (|prg=G,cls=accC,lcl=L|)
    ⊢ dom (locals (store ((Norm s0)::state))) »In1l e» E
  by (elim da-elim-cases) simp+
from da.LVar vn obtain V where
  da-var: (|prg=G,cls=accC,lcl=L|)
    ⊢ dom (locals (store ((Norm s0)::state))) »In2 var» V
  by auto
obtain E' where
  da-e': (|prg=G,cls=accC,lcl=L|)
    ⊢ dom (locals (store s1)) »In1l e» E'
proof -
  have dom (locals (store ((Norm s0)::state)))
    ⊆ dom (locals (store s1))
    by (rule dom-locals-eval-mono-elim) (rule Ass.hyps)
  with da-e show thesis
    by (rule da-weakenE) (rule that)
qed

```

```

from conf-s0 wt-var da-var
obtain conf-s1: s1::⊆(G, L)
  and conf-var: normal s1
     $\longrightarrow G, L, \text{store } s1 \vdash \text{In2 } \text{var} \succ \text{In2 } (w, \text{upd}) :: \subseteq \text{In1 } \text{var} T$ 
  and error-free-s1: error-free s1
  by (rule hyp-var [elim-format]) simp
show ?thesis
proof (cases normal s1)
  case False
    with eval-e have s2=s1 by auto
    with False have assign upd v s2=s1
      by simp
    with conf-s1 error-free-s1 False show ?thesis
      by auto
next
  case True
    note normal-s1 = this
    from conf-s1 wt-e da-e'
    obtain conf-s2: s2::⊆(G, L) and
      conf-v: normal s2  $\longrightarrow G, \text{store } s2 \vdash v :: \subseteq e T$  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 :: ⊆ e T
            by simp
          with widen wf
          have conf-v-varT: G, store s2 ⊢ v :: ⊆ var T
            by (auto intro: conf-widen)
          from normal-s1 conf-var
          have  $G, L, \text{store } s1 \vdash \text{In2 } \text{var} \succ \text{In2 } (w, \text{upd}) :: \subseteq \text{In1 } \text{var} T$ 
            by simp
          then
          have conf-assign: store s1 ≤ |upd| var T :: ⊆(G, L)
            by (simp add: rconf-def)
          from conf-v-eT conf-v-varT conf-assign normal-s1 True wf eval-var
            eval-e T conf-s2 error-free-s2
          show ?thesis
            by (cases s1, cases s2)
              (auto dest!: Ass-lemma simp add: assign-conforms-def)
        qed
      qed
    qed
  next
  case (Cond s0 e0 b s1 e1 e2 v s2 L accC T A)
    note eval-e0 = ⟨G ⊢ Norm s0 -e0- > b → s1⟩
    note eval-e1-e2 = ⟨G ⊢ s1 -(if the-Bool b then e1 else e2)- > v → s2⟩
    note hyp-e0 = ⟨PROP ?TypeSafe (Norm s0) s1 (In1l e0) (In1 b)⟩
    note hyp-if = ⟨PROP ?TypeSafe s1 s2
      (In1l (if the-Bool b then e1 else e2)⟩) (In1 v⟩)
    note conf-s0 = ⟨Norm s0 :: ⊆(G, L)⟩
    note wt = ⟨(|prg = G, cls = accC, lcl = L|) ⊢ In1l (e0 ? e1 : e2) :: T⟩

```

```

then obtain  $T1\ T2\ statT$  where
   $wt-e0: (\text{prg} = G, \text{cls} = accC, \text{lcl} = L) \vdash e0 :: -PrimT\ Boolean$  and
   $wt-e1: (\text{prg} = G, \text{cls} = accC, \text{lcl} = L) \vdash e1 :: -T1$  and
   $wt-e2: (\text{prg} = G, \text{cls} = accC, \text{lcl} = L) \vdash e2 :: -T2$  and
   $statT: G \vdash T1 \preceq T2 \wedge statT = T2 \vee G \vdash T2 \preceq T1 \wedge statT = T1$  and
   $T : T = Inl\ statT$ 
by (rule wt-elim-cases) auto
with Cond.premis obtain  $E0\ E1\ E2$  where
   $da-e0: (\text{prg} = G, \text{cls} = accC, \text{lcl} = L)$ 
     $\vdash \text{dom} (\text{locals} (\text{store} ((\text{Norm } s0)::state)))$ 
     $\gg In1l\ e0 \gg E0$  and
   $da-e1: (\text{prg} = G, \text{cls} = accC, \text{lcl} = L)$ 
     $\vdash (\text{dom} (\text{locals} (\text{store} ((\text{Norm } s0)::state)))$ 
     $\cup \text{assigns-if } True\ e0) \gg In1l\ e1 \gg E1$  and
   $da-e2: (\text{prg} = G, \text{cls} = accC, \text{lcl} = L)$ 
     $\vdash (\text{dom} (\text{locals} (\text{store} ((\text{Norm } s0)::state)))$ 
     $\cup \text{assigns-if } False\ e0) \gg In1l\ e2 \gg E2$ 
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  $s2 :: \preceq (G, L) \wedge$ 
   $(normal\ s2 \longrightarrow G, L, \text{store } s2 \vdash In1l\ (e0\ ?\ e1 : e2) \gg In1\ v :: \preceq T) \wedge$ 
   $(error-free\ (\text{Norm } s0) = error-free\ s2)$ 
proof (cases normal s1)
  case False
    with eval-e1-e2 have  $s2 = s1$  by auto
    with conf-s1 error-free-s1 False show ?thesis
    by auto
  next
    case True
    have  $s0-s1: \text{dom} (\text{locals} (\text{store} ((\text{Norm } s0)::state)))$ 
       $\cup \text{assigns-if } (the-Bool\ b)\ e0 \subseteq \text{dom} (\text{locals} (\text{store } s1))$ 
    proof –
      from eval-e0 have
         $\text{dom} (\text{locals} (\text{store} ((\text{Norm } s0)::state))) \subseteq \text{dom} (\text{locals} (\text{store } s1))$ 
        by (rule dom-locals-eval-mono-elim)
      moreover
        from eval-e0 True wt-e0
        have  $\text{assigns-if } (the-Bool\ b)\ e0 \subseteq \text{dom} (\text{locals} (\text{store } s1))$ 
        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
         $(\text{prg} = G, \text{cls} = accC, \text{lcl} = L) \vdash (\text{dom} (\text{locals} (\text{store } s1))) \gg In1l\ e1 \gg E1'$ 
        by – (rule da-weakenE, auto iff del: Un-subset-iff)
      with conf-s1 wt-e1
      obtain
         $s2 :: \preceq (G, L)$ 
         $(normal\ s2 \longrightarrow G, L, \text{store } s2 \vdash In1l\ e1 \gg In1\ v :: \preceq Inl\ T1)$ 
         $error-free\ s2$ 
        by (rule hyp-e1 [elim-format]) (simp add: error-free-s1)
      moreover
      from statT

```

```

have  $G \vdash T1 \preceq \text{stat}T$ 
  by auto
ultimately show ?thesis
  using  $T$  wf by auto
next
case False
with hyp-if have hyp-e2:  $\text{PROP } ?\text{TypeSafe } s1 \ s2 \ (\text{In1l } e2) \ (\text{In1 } v)$ 
  by simp
from da-e2 s0-s1 False obtain  $E2'$  where
   $(\llbracket \text{prg} = G, \text{cls} = \text{acc}C, \text{lcl} = L \rrbracket \vdash (\text{dom } (\text{locals } (\text{store } s1))) \gg \text{In1l } e2 \gg E2')$ 
  by  $-\text{ (rule da-weakenE, auto iff del: Un-subset-iff)}$ 
with conf-s1 wt-e2
obtain
   $s2 :: \preceq (G, L)$ 
   $(\text{normal } s2 \longrightarrow G, L, \text{store } s2 \vdash \text{In1l } e2 \succ \text{In1 } v :: \preceq \text{Inl } T2)$ 
  error-free s2
  by  $(\text{rule hyp-e2 [elim-format]}) \text{ (simp add: error-free-s1)}$ 
moreover
from statT
have  $G \vdash T2 \preceq \text{stat}T$ 
  by auto
ultimately show ?thesis
  using  $T$  wf by auto
qed
qed
next
case  $(\text{Call } s0 \ e \ a \ s1 \ \text{args } vs \ s2 \ \text{invDecl}C \ \text{mode } \text{stat}T \ mn \ pTs' \ s3 \ s3' \ \text{acc}C' \ v \ s4 \ L \ \text{acc}C \ T \ A)$ 
note  $\text{eval-e} = \langle G \vdash \text{Norm } s0 \ -e- \succ a \rightarrow s1 \rangle$ 
note  $\text{eval-args} = \langle G \vdash s1 \ -\text{args} \dot{=} \succ vs \rightarrow s2 \rangle$ 
note  $\text{invDecl}C = \langle \text{invDecl}C$ 
   $= \text{invocation-declclass } G \ \text{mode } (\text{store } s2) \ a \ \text{stat}T$ 
   $(\llbracket \text{name} = mn, \text{parTs} = pTs' \rrbracket) \rangle$ 
note  $\text{init-lvars} =$ 
   $\langle s3 = \text{init-lvars } G \ \text{invDecl}C \ (\llbracket \text{name} = mn, \text{parTs} = pTs' \rrbracket) \ \text{mode } a \ vs \ s2 \rangle$ 
note  $\text{check} = \langle s3' =$ 
   $\text{check-method-access } G \ \text{acc}C' \ \text{stat}T \ \text{mode } (\llbracket \text{name} = mn, \text{parTs} = pTs' \rrbracket) \ a \ s3 \rangle$ 
note  $\text{eval-methd} =$ 
   $\langle G \vdash s3' \ -\text{Methd } \text{invDecl}C \ (\llbracket \text{name} = mn, \text{parTs} = pTs' \rrbracket) \ -\succ v \rightarrow s4 \rangle$ 
note  $\text{hyp-e} = \langle \text{PROP } ?\text{TypeSafe } (\text{Norm } s0) \ s1 \ (\text{In1l } e) \ (\text{In1 } a) \rangle$ 
note  $\text{hyp-args} = \langle \text{PROP } ?\text{TypeSafe } s1 \ s2 \ (\text{In3 } \text{args}) \ (\text{In3 } vs) \rangle$ 
note  $\text{hyp-methd} = \langle \text{PROP } ?\text{TypeSafe } s3' \ s4$ 
   $(\text{In1l } (\text{Methd } \text{invDecl}C \ (\llbracket \text{name} = mn, \text{parTs} = pTs' \rrbracket))) \ (\text{In1 } v) \rangle$ 
note  $\text{conf-s0} = \langle \text{Norm } s0 :: \preceq (G, L) \rangle$ 
note  $\text{wt} = \langle \llbracket \text{prg} = G, \text{cls} = \text{acc}C, \text{lcl} = L \rrbracket$ 
   $\vdash \text{In1l } (\{ \text{acc}C', \text{stat}T, \text{mode} \} e \cdot mn \{ pTs' \} \text{args}) :: T \rangle$ 
from  $\text{wt}$  obtain  $pTs \ \text{statDecl}T \ \text{stat}M$  where
   $\text{wt-e}: \llbracket \text{prg} = G, \text{cls} = \text{acc}C, \text{lcl} = L \rrbracket \vdash e :: -\text{Ref}T \ \text{stat}T$  and
   $\text{wt-args}: \llbracket \text{prg} = G, \text{cls} = \text{acc}C, \text{lcl} = L \rrbracket \vdash \text{args} :: \dot{=} pTs$  and
   $\text{stat}M: \text{max-spec } G \ \text{acc}C \ \text{stat}T \ (\llbracket \text{name} = mn, \text{parTs} = pTs \rrbracket)$ 
   $= \{ (\text{statDecl}T, \text{stat}M), pTs' \}$  and
   $\text{mode}: \text{mode} = \text{invmode } \text{stat}M \ e$  and
   $T: T = \text{Inl } (\text{resTy } \text{stat}M)$  and
   $\text{eq-acc}C\text{-acc}C': \text{acc}C = \text{acc}C'$ 
by  $(\text{rule wt-elim-cases}) \text{ fastsimp+}$ 
from  $\text{Call.premis}$  obtain  $E$  where
   $\text{da-e}: \llbracket \text{prg} = G, \text{cls} = \text{acc}C, \text{lcl} = L \rrbracket$ 
   $\vdash (\text{dom } (\text{locals } (\text{store } ((\text{Norm } s0) :: \text{state})))) \gg \text{In1l } e \gg E$  and
   $\text{da-args}: \llbracket \text{prg} = G, \text{cls} = \text{acc}C, \text{lcl} = L \rrbracket \vdash \text{nrm } E \gg \text{In3 } \text{args} \gg A$ 

```

```

  by (elim da-elim-cases) simp
from conf-s0 wt-e da-e
obtain conf-s1: s1::≤(G, L) and
  conf-a: normal s1 ⇒ G, store s1⊢a::≤RefT statT and
  error-free-s1: error-free s1
by (rule hyp-e [elim-format]) simp
{
  assume abnormal-s2: ¬ normal s2
  have set-lvars (locals (store s2)) s4 = s2
  proof -
    from abnormal-s2 init-lvars
    obtain keep-abrupt: abrupt s3 = abrupt s2 and
      store s3 = store (init-lvars G invDeclC (λname = mn, parTs = pTs'))
        mode a vs s2)
    by (auto simp add: init-lvars-def2)
    moreover
    from keep-abrupt abnormal-s2 check
    have eq-s3'-s3: s3'=s3
    by (auto simp add: check-method-access-def Let-def)
    moreover
    from eq-s3'-s3 abnormal-s2 keep-abrupt eval-methd
    have s4=s3'
    by auto
    ultimately show
      set-lvars (locals (store s2)) s4 = s2
    by (cases s2,cases s3) (simp add: init-lvars-def2)
  qed
} note propagate-abnormal-s2 = this
show (set-lvars (locals (store s2))) s4::≤(G, L) ∧
  (normal ((set-lvars (locals (store s2))) s4) →
    G,L,store ((set-lvars (locals (store s2))) s4)
    ⊢In1l ({accC',statT,mode}e.mn( {pTs'}args))>In1 v::≤T) ∧
  (error-free (Norm s0) =
    error-free ((set-lvars (locals (store s2))) s4))
proof (cases normal s1)
  case False
  with eval-args have s2=s1 by auto
  with False propagate-abnormal-s2 conf-s1 error-free-s1
  show ?thesis
  by auto
next
  case True
  note normal-s1 = this
  obtain A' where
    (λprg=G,cls=accC,lcl=L)⊢ dom (locals (store s1)) »In3 args» A'
  proof -
    from eval-e wt-e da-e wf normal-s1
    have nrm E ⊆ dom (locals (store s1))
    by (cases rule: da-good-approxE') iprover
    with da-args show thesis
    by (rule da-weakenE) (rule that)
  qed
with conf-s1 wt-args
obtain conf-s2: s2::≤(G, L) and
  conf-args: normal s2
  ⇒ list-all2 (conf G (store s2)) vs pTs and
  error-free-s2: error-free s2
by (rule hyp-args [elim-format]) (simp add: error-free-s1)
from error-free-s2 init-lvars

```

```

have error-free-s3: error-free s3
  by (auto simp add: init-lvars-def2)
from statM
obtain
  statM': (statDeclT, statM) ∈ mheads G accC statT (⟦name = mn, parTs = pTs'⟧) and
  pTs-widen:  $G \vdash pTs \sqsubseteq pTs'$ 
  by (blast dest: max-spec2mheads)
from check
have eq-store-s3'-s3: store s3' = store s3
  by (cases s3) (simp add: check-method-access-def Let-def)
obtain invC
  where invC: invC = invocation-class mode (store s2) a statT
  by simp
with init-lvars
have invC': invC = (invocation-class mode (store s3) a statT)
  by (cases s2, cases mode) (auto simp add: init-lvars-def2)
show ?thesis
proof (cases normal s2)
  case False
  with propagate-abnormal-s2 conf-s2 error-free-s2
  show ?thesis
  by auto
next
  case True
  note normal-s2 = True
  with normal-s1 conf-a eval-args
  have conf-a-s2:  $G, \text{store } s2 \vdash a :: \preceq \text{RefT } \text{statT}$ 
  by (auto dest: eval-geat intro: conf-geat)
  show ?thesis
  proof (cases a = Null  $\longrightarrow$  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  $\longrightarrow$  a  $\neq$  Null
  by (auto dest!: Null-staticD)
  with conf-s2 conf-a-s2 wf invC
  have dynT-prop:  $G \vdash \text{mode} \rightarrow \text{invC} \preceq \text{statT}$ 

```

```

  by (cases s2) (auto intro: DynT-propI)
with wt-e statM' invC mode wf
obtain dynM where
  dynM: dynlookup G statT invC ( $\langle \text{name}=\text{mn}, \text{parTs}=\text{pTs}' \rangle$ ) = Some dynM and
  acc-dynM:  $G \vdash \text{Methd } (\langle \text{name}=\text{mn}, \text{parTs}=\text{pTs}' \rangle) \text{ dynM}$ 
    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 ( $\langle \text{name}=\text{mn}, \text{parTs}=\text{pTs}' \rangle, \text{mthd dynM}$ ) and
  dynM': methd G invDeclC ( $\langle \text{name}=\text{mn}, \text{parTs}=\text{pTs}' \rangle$ ) = Some dynM and
  iscls-invDeclC: is-class G invDeclC and
  invDeclC': invDeclC = declclass dynM and
  invC-widen:  $G \vdash \text{invC} \preceq_C \text{invDeclC}$  and
  resTy-widen:  $G \vdash \text{resTy dynM} \preceq_{\text{resTy}} \text{statM}$  and
  is-static-eq: is-static dynM = is-static statM and
  involved-classes-prop:
    (if invmode statM e = IntVir
     then  $\forall \text{statC}. \text{statT} = \text{ClassT statC} \longrightarrow G \vdash \text{invC} \preceq_C \text{statC}$ 
     else  $((\exists \text{statC}. \text{statT} = \text{ClassT statC} \wedge G \vdash \text{statC} \preceq_C \text{invDeclC}) \vee$ 
        $(\forall \text{statC}. \text{statT} \neq \text{ClassT statC} \wedge \text{invDeclC} = \text{Object})) \wedge$ 
        $\text{statDeclT} = \text{ClassT invDeclC}$ )
  by (cases rule: DynT-mheadsE) simp
obtain L' where
  L':L'=( $\lambda k.$ 
    (case k of
      EName e
       $\Rightarrow$  (case e of
        VNam v
         $\Rightarrow$  (table-of (lcls (mbody (mthd dynM)))
          (pars (mthd dynM)[ $\mapsto$ ]pTs')) v
        | Res  $\Rightarrow$  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
  ( $\langle \text{name}=\text{mn}, \text{parTs}=\text{pTs}' \rangle$ ) dynM store s2 vs pTs abrupt s2
  L statT invC a (statDeclT, statM) e])
apply (rule wf)
apply (rule conf-args, assumption)
apply (simp add: pTs-widen)
apply (cases s2, simp)
apply (rule dynM')
apply (force dest: ty-expr-is-type)
apply (rule invC-widen)
apply (force intro: conf-gext dest: eval-gext)
apply simp
apply simp
apply (simp add: invC)
apply (simp add: invDeclC)
apply (simp add: normal-s2)

```



```

apply (cases s2, simp add: L' init-lvars
      cong add: lname.case-cong ename.case-cong)
done
with eq-s3'-s3
have conf-s3': s3'::≤(G,L') by simp
moreover
from is-static-eq wf-dynM L'
obtain mthdT where
  (⟦prg=G,cls=invDeclC,lcl=L'⟧
   ⊢ Body invDeclC (stmt (mbody (mthd dynM))))::-mthdT and
  mthdT-widen: G⊢mthdT≤resTy dynM
by - (drule wf-mdecl-bodyD,
      auto simp add: callee-lcl-def
      cong add: lname.case-cong ename.case-cong)
with dynM' iscls-invDeclC invDeclC'
have
  (⟦prg=G,cls=invDeclC,lcl=L'⟧
   ⊢ (Methd invDeclC (⟦name = mn, parTs = pTs'⟧))::-mthdT
  by (auto intro: wt.Methd)
moreover
obtain M where
  (⟦prg=G,cls=invDeclC,lcl=L'⟧
   ⊢ dom (locals (store s3'))
   » In1l (Methd invDeclC (⟦name = mn, parTs = pTs'⟧))» M
proof -
from wf-dynM
obtain M' where
  da-body:
  (⟦prg=G, cls=invDeclC
   ,lcl=callee-lcl invDeclC (⟦name = mn, parTs = pTs'⟧) (mthd dynM)
   ⟧ ⊢ parameters (mthd dynM) » (stmt (mbody (mthd dynM)))» M' and
  res: Result ∈ nrm M'
by (rule wf-mdeclE) iprover
from da-body is-static-eq L' have
  (⟦prg=G, cls=invDeclC,lcl=L'⟧
   ⊢ parameters (mthd dynM) » (stmt (mbody (mthd dynM)))» M'
by (simp add: callee-lcl-def
      cong add: lname.case-cong ename.case-cong)
moreover have parameters (mthd dynM) ⊆ dom (locals (store s3'))
proof -
from is-static-eq
have (invmode (mthd dynM) e) = (invmode statM e)
by (simp add: invmode-def)
moreover
have length (pars (mthd dynM)) = length vs
proof -
from normal-s2 conf-args
have length vs = length pTs
by (simp add: list-all2-def)
also from pTs-widen
have ... = length pTs'
by (simp add: widens-def list-all2-def)
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 ( $\langle \text{name}=\text{mn}, \text{parTs}=\text{pTs}' \rangle$ ) vs e a s2]
by simp
also from check
have dom (locals (store s3))  $\subseteq$  dom (locals (store s3'))
by (simp add: eq-s3'-s3)
finally show ?thesis .
qed
ultimately obtain M2 where
  da:
  ( $\langle \text{prg}=G, \text{cls}=\text{invDeclC}, \text{lcl}=L' \rangle$ )
   $\vdash$  dom (locals (store s3'))  $\gg$  (stmt (mbody (mthd dynM)))  $\gg$  M2 and
  M2: nrm M'  $\subseteq$  nrm M2
by (rule da-weakenE)
from res M2 have Result  $\in$  nrm M2
by blast
moreover from wf-dynM
have jumpNestingOkS {Ret} (stmt (mbody (mthd dynM)))
by (rule wf-mdeclE)
ultimately
obtain M3 where
  ( $\langle \text{prg}=G, \text{cls}=\text{invDeclC}, \text{lcl}=L' \rangle$ )  $\vdash$  dom (locals (store s3'))
   $\gg$  (Body (declclass dynM) (stmt (mbody (mthd dynM))))  $\gg$  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 ::  $\preceq(G, L')$  and
  conf-Res: normal s4  $\longrightarrow G, \text{store } s4 \vdash v :: \preceq \text{mthd} T$  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  $\leq$  store s4
by (cases s2) (auto dest!: eval-gext simp add: init-lvars-def2)
moreover
have abrupt s4  $\neq$  Some (Jump Ret)
proof –
from normal-s2 init-lvars
have abrupt s3  $\neq$  Some (Jump Ret)
by (cases s2) (simp add: init-lvars-def2 abrupt-if-def)
with check
have abrupt s3'  $\neq$  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 ::  $\preceq(G, L)$ 
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 = ⟨G ⊢ Norm s0 -Init D → s1⟩
note eval-c = ⟨G ⊢ s1 -c → s2⟩
note hyp-init = ⟨PROP ?TypeSafe (Norm s0) s1 (In1r (Init D)) ⋄⟩
note hyp-c = ⟨PROP ?TypeSafe s1 s2 (In1r c) ⋄⟩
note conf-s0 = ⟨Norm s0 :: ⋚(G, L)⟩
note wt = ⟨(prg = G, cls = accC, lcl = L) ⊢ In1l (Body D c) :: T⟩
then obtain bodyT where
  iscls-D: is-class G D and
  wt-c: (prg = G, cls = accC, lcl = L) ⊢ c :: √ and
  resultT: L Result = Some bodyT and
  isty-bodyT: is-type G bodyT and
  T: T = Inl bodyT
  by (rule wt-elim-cases) auto
from Body.premis obtain C where
  da-c: (prg = G, cls = accC, lcl = L)
    ⊢ (dom (locals (store ((Norm s0)::state)))) » In1r c » C and
  jmpOk: jumpNestingOkS {Ret} c and
  res: Result ∈ nrm C
  by (elim da-elim-cases) simp

```

```

note conf-s0
moreover from iscls-D
have ( $\text{prg}=G, \text{cls}=\text{acc}C, \text{lcl}=L$ ) $\vdash$ Init D:: $\surd$  by auto
moreover obtain I where
  ( $\text{prg}=G, \text{cls}=\text{acc}C, \text{lcl}=L$ )
   $\vdash \text{dom}(\text{locals}(\text{store}((\text{Norm } s0)::\text{state}))) \gg \text{In1r}(\text{Init } D) \gg I$ 
  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': ( $\text{prg}=G, \text{cls}=\text{acc}C, \text{lcl}=L$ )
   $\vdash (\text{dom}(\text{locals}(\text{store } s1))) \gg \text{In1r } c \gg C'$ 
  and nrm-C':  $\text{nrm } C \subseteq \text{nrm } C'$ 
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)  $s3$ )  $\neq \text{Some}(\text{Jump Ret})$ 
  by (cases s3) (simp add: absorb-def)
moreover have  $s3=s2$ 
proof –
  from iscls-D
  have wt-init: ( $\text{prg}=G, \text{cls}=\text{acc}C, \text{lcl}=L$ ) $\vdash$ (Init D):: $\surd$ 
  by auto
from eval-init wf
  have s1-no-jmp:  $\bigwedge j. \text{abrupt } s1 \neq \text{Some}(\text{Jump } j)$ 
  by – (rule eval-statement-no-jump [OF - - wt-init], auto)
from eval-c - wt-c wf
  have  $\bigwedge j. \text{abrupt } s2 = \text{Some}(\text{Jump } j) \implies j=\text{Ret}$ 
  by (rule jumpNestingOk-evalE) (auto intro: jmpOk simp add: s1-no-jmp)
moreover
note  $s3 =$ 
  (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 abupd ( $\lambda x. \text{Some}(\text{Error CrossMethodJump}) s2$  else  $s2$ )
  ultimately show ?thesis
  by force
qed
moreover
{
  assume normal-upd-s2: normal (abupd (absorb Ret)  $s2$ )
  have Result  $\in \text{dom}(\text{locals}(\text{store } s2))$ 
  proof –
    from normal-upd-s2
    have 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 abrupt s2 = Some (Jump Ret)
  with conf-s2 show ?thesis
    by (cases s2) (auto dest: conforms-RetD simp add: dom-def)
qed
qed
}
moreover note T resultT
ultimately
show abupd (absorb Ret) s3::≤(G, L) ∧
  (normal (abupd (absorb Ret) s3) →
    G,L,store (abupd (absorb Ret) s3)
    ⊢In1l (Body D c)⋗In1 (the (locals (store s2) Result))::≤T) ∧
  (error-free (Norm s0) = error-free (abupd (absorb Ret) s3))
  by (cases s2) (auto intro: conforms-locals)
next
case (LVar s vn L accC T)
note conf-s = ⟨Norm s::≤(G, L)⟩ and
  wt = ⟨(prg = G, cls = accC, lcl = L)⋗In2 (LVar vn)::T⟩
then obtain vnT where
  vnT: L vn = Some vnT and
  T: T=Inl vnT
  by (auto elim!: wt-elim-cases)
from conf-s vnT
have conf-fst: locals s vn ≠ None → G,s⋗fst (lvar vn s)::≤vnT
  by (auto elim: conforms-localD [THEN wlconfD]
    simp add: lvar-def)
moreover
from conf-s conf-fst vnT
have s≤|snd (lvar vn s)≤vnT::≤(G, L)
  by (auto elim: conforms-lupd simp add: assign-conforms-def lvar-def)
moreover note conf-s T
ultimately
show Norm s::≤(G, L) ∧
  (normal (Norm s) →
    G,L,store (Norm s)⋗In2 (LVar vn)⋗In2 (lvar vn s)::≤T) ∧
  (error-free (Norm s) = error-free (Norm s))
  by (simp add: lvar-def)
next
case (FVar s0 statDeclC s1 e a s2 v s2' stat fn s3 accC L accC' T A)
note eval-init = ⟨G⋗Norm s0 -Init statDeclC→ s1⟩
note eval-e = ⟨G⋗s1 -e-⋗a→ s2⟩
note fvar = ⟨(v, s2') = fvar statDeclC stat fn a s2⟩
note check = ⟨s3 = check-field-access G accC statDeclC fn stat a s2'⟩
note hyp-init = ⟨PROP ?TypeSafe (Norm s0) s1 (In1r (Init statDeclC)) ⋄⟩
note hyp-e = ⟨PROP ?TypeSafe s1 s2 (In1l e) (In1 a)⟩
note conf-s0 = ⟨Norm s0::≤(G, L)⟩
note wt = ⟨(prg=G, cls=accC', lcl=L)⋗In2 ({accC,statDeclC,stat}e..fn)::T⟩
then obtain statC f where
  wt-e: (prg=G, cls=accC, lcl=L)⋗e::-Class statC and
  accfield: accfield G accC statC fn = Some (statDeclC,f) and
  eq-accC-accC': accC=accC' and
  stat: stat=is-static f and

```

```

      T: T=(Inl (type f))
    by (rule wt-elim-cases) (auto simp add: member-is-static-simp)
  from FVar.premis eq-accC-accC'
  have da-e: (prg=G, cls=accC, lcl=L)
    ⊢ (dom (locals (store ((Norm s0)::state)))) » In1l e » A
    by (elim da-elim-cases) simp
  note conf-s0
  moreover
  from wf wt-e
  have iscls-statC: is-class G statC
    by (auto dest: ty-expr-is-type type-is-class)
  with wf accfield
  have iscls-statDeclC: is-class G statDeclC
    by (auto dest!: accfield-fields dest: fields-declC)
  hence (prg=G, cls=accC, lcl=L) ⊢ (Init statDeclC)::√
    by simp
  moreover obtain I where
    (prg=G, cls=accC, lcl=L)
    ⊢ dom (locals (store ((Norm s0)::state))) » In1r (Init statDeclC) » I
    by (auto intro: da-Init [simplified] assigned.select-convs)
  ultimately
  obtain conf-s1: s1::≤(G, L) and error-free-s1: error-free s1
    by (rule hyp-init [elim-format]) simp
  obtain A' where
    (prg=G, cls=accC, lcl=L) ⊢ (dom (locals (store s1))) » In1l e » A'
  proof -
    from eval-init
    have (dom (locals (store ((Norm s0)::state))))
      ⊆ (dom (locals (store s1)))
      by (rule dom-locals-eval-mono-elim)
    with da-e show thesis
      by (rule da-weakenE) (rule that)
  qed
  with conf-s1 wt-e
  obtain
    conf-s2: s2::≤(G, L) and
    conf-a: normal s2 ⟶ G, store s2 ⊢ a::≤Class statC and
    error-free-s2: error-free s2
    by (rule hyp-e [elim-format]) (simp add: error-free-s1)
  from fvar
  have store-s2': store s2' = store s2
    by (cases s2) (simp add: fvar-def2)
  with fvar conf-s2
  have conf-s2': s2'::≤(G, L)
    by (cases s2, cases stat) (auto simp add: fvar-def2)
  from eval-init
  have initd-statDeclC-s1: initd statDeclC s1
    by (rule init-yields-initd)
  from accfield wt-e eval-init eval-e conf-s2 conf-a fvar stat check wf
  have eq-s3-s2': s3 = s2'
    by (auto dest!: error-free-field-access)
  have conf-v: normal s2' ⟹
    G, store s2' ⊢ fst v::≤type f ∧ store s2' ≤ |snd v| ≤ type f::≤(G, L)
  proof -
    assume normal: normal s2'
    obtain vv vf x2 store2 store2'
      where v: v=(vv, vf) and
            s2: s2=(x2, store2) and
            store2': store s2' = store2'
      by (cases v, cases s2, cases s2') blast
  
```

```

from iscls-statDeclC obtain c
  where c: class G statDeclC = Some c
  by auto
have G,store2⊢vv::≤type f ∧ store2'≤|vf≤type f::≤(G, L)
proof (rule FVar-lemma [of vv vf store2' statDeclC f fn a x2 store2
  statC G c L store s1])
  from v normal s2 fvar stat store2'
  show ((vv, vf), Norm store2') =
    fvar statDeclC (static f) fn a (x2, store2)
    by (auto simp add: member-is-static-simp)
  from accfield iscls-statC wf
  show G⊢statC≤C statDeclC
    by (auto dest!: accfield-fields dest: fields-declC)
  from accfield
  show fld: table-of (DeclConcepts.fields G statC) (fn, statDeclC) = Some f
    by (auto dest!: accfield-fields)
  from wf show wf-prog G .
  from conf-a s2 show x2 = None → G,store2⊢a::≤Class statC
    by auto
  from fld wf iscls-statC
  show statDeclC ≠ Object
    by (cases statDeclC=Object) (drule fields-declC,simp+)+
  from c show class G statDeclC = Some c .
  from conf-s2 s2 show (x2, store2)::≤(G, L) by simp
  from eval-e s2 show snd s1≤|store2 by (auto dest: eval-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::≤(G, L) ∧
  (normal s3
    → G,L,store s3⊢In2 ( {accC,statDeclC,stat} e..fn )>In2 v::≤T) ∧
  (error-free (Norm s0) = error-free s3)
  by auto
next
case (AVar s0 e1 a s1 e2 i s2 v s2' L accC T A)
note eval-e1 = ⟨G⊢Norm s0 -e1->a→ s1⟩
note eval-e2 = ⟨G⊢s1 -e2->i→ s2⟩
note hyp-e1 = ⟨PROP ?TypeSafe (Norm s0) s1 (In1l e1) (In1 a)⟩
note hyp-e2 = ⟨PROP ?TypeSafe s1 s2 (In1l e2) (In1 i)⟩
note avar = ⟨(v, s2') = avar G i a s2⟩
note conf-s0 = ⟨Norm s0::≤(G, L)⟩
note wt = ⟨(prg = G, cls = accC, lcl = L)⊢In2 (e1.[e2])::T⟩
then obtain elemT
  where wt-e1: (prg=G,cls=accC,lcl=L)⊢e1::-elemT.[] and
    wt-e2: (prg=G,cls=accC,lcl=L)⊢e2::-PrimT Integer and
    T: T= Inl elemT
  by (rule wt-elim-cases) auto
from AVar.premis obtain E1 where
  da-e1: (prg=G,cls=accC,lcl=L)
    ⊢ (dom (locals (store ((Norm s0)::state))))»In1l e1» E1 and

```

```

  da-e2: ( $\text{prg}=G, \text{cls}=\text{acc}C, \text{lcl}=L$ ) $\vdash$  nrm  $E1 \gg \text{In}1l\ e2 \gg A$ 
  by (elim da-elim-cases) simp
from conf-s0 wt-e1 da-e1
obtain conf-s1:  $s1::\preceq(G, L)$  and
  conf-a: ( $\text{normal } s1 \longrightarrow G, \text{store } s1 \vdash a::\preceq \text{elem}T.[]$ ) and
  error-free-s1: error-free  $s1$ 
  by (rule hyp-e1 [elim-format]) simp
show  $s2'::\preceq(G, L) \wedge$ 
  ( $\text{normal } s2' \longrightarrow G, L, \text{store } s2 \vdash \text{In}2\ (e1.[e2]) \succ \text{In}2\ v::\preceq T$ )  $\wedge$ 
  (error-free ( $\text{Norm } s0$ ) = 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  $\neg \text{normal } s2$  by simp
  then have snd ( $\text{avar } G\ i\ a\ s2$ ) =  $s2$ 
    by (cases  $s2$ ) (simp add: avar-def2)
  with avar have  $s2'=s2$ 
    by (cases ( $\text{avar } G\ i\ a\ s2$ )) simp
  ultimately show ?thesis
    using conf-s1 error-free-s1
    by auto
next
  case True
  obtain A' where
    ( $\text{prg}=G, \text{cls}=\text{acc}C, \text{lcl}=L$ ) $\vdash$  dom ( $\text{locals } (\text{store } s1)$ )  $\gg \text{In}1l\ e2 \gg A'$ 
  proof -
    from eval-e1 wt-e1 da-e1 wf True
    have nrm  $E1 \subseteq \text{dom } (\text{locals } (\text{store } s1))$ 
      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:  $s2::\preceq(G, L)$  and error-free-s2: error-free  $s2$ 
    by (rule hyp-e2 [elim-format]) (simp add: error-free-s1)
  from avar
  have store  $s2'=\text{store } s2$ 
    by (cases  $s2$ ) (simp add: avar-def2)
  with avar conf-s2
  have conf-s2':  $s2'::\preceq(G, L)$ 
    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  $s2' \implies$ 
     $G, \text{store } s2 \vdash \text{fst } v::\preceq \text{elem}T \wedge \text{store } s2' \leq |\text{snd } v| \preceq \text{elem}T::\preceq(G, L)$ 
  proof -
    assume normal: normal  $s2'$ 
    show ?thesis
    proof -
      obtain vv vf x1 store1 x2 store2 store2'
        where v:  $v=(vv, vf)$  and
          s1:  $s1=(x1, \text{store}1)$  and
          s2:  $s2=(x2, \text{store}2)$  and
          store2':  $\text{store}2'=\text{store } s2'$ 
        by (cases v, cases s1, cases s2, cases s2') blast
      have  $G, \text{store}2 \vdash vv::\preceq \text{elem}T \wedge \text{store}2' \leq |vf| \preceq \text{elem}T::\preceq(G, L)$ 

```



```

proof (rule AVar-lemma [of G x1 store1 e2 i x2 store2 vv vf store2' a,
      OF wf])
  from s1 s2 eval-e2 show  $G \vdash (x1, store1) -e2 \multimap i \rightarrow (x2, store2)$ 
    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  $(x2, store2) :: \preceq (G, L)$  by simp
  from s1 conf-a show  $x1 = None \longrightarrow G, store1 \vdash a :: \preceq elemT.[]$  by simp
  from eval-e2 s1 s2 show  $store1 \leq store2$  by (auto dest: eval-ge)
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 =  $\langle G \vdash Norm\ s0 -e \multimap v \rightarrow s1 \rangle$ 
  note eval-es =  $\langle G \vdash s1 -es \multimap vs \rightarrow s2 \rangle$ 
  note hyp-e =  $\langle PROP\ ?TypeSafe\ (Norm\ s0)\ s1\ (In1\ e)\ (In1\ v) \rangle$ 
  note hyp-es =  $\langle PROP\ ?TypeSafe\ s1\ s2\ (In3\ es)\ (In3\ vs) \rangle$ 
  note conf-s0 =  $\langle Norm\ s0 :: \preceq (G, L) \rangle$ 
  note wt =  $\langle \langle prg = G, cls = accC, lcl = L \rangle \vdash In3\ (e \# es) :: T \rangle$ 
  then obtain eT esT where
    wt-e:  $\langle prg = G, cls = accC, lcl = L \rangle \vdash e :: -eT$  and
    wt-es:  $\langle prg = G, cls = accC, lcl = L \rangle \vdash es :: -esT$  and
    T:  $T = Inr\ (eT \# esT)$ 
  by (rule wt-elim-cases) blast
  from Cons.premis obtain E where
    da-e:  $\langle prg = G, cls = accC, lcl = L \rangle \vdash$ 
       $\vdash (dom\ (locals\ (store\ ((Norm\ s0) :: state)))) \gg In1\ e \gg E$  and
    da-es:  $\langle prg = G, cls = accC, lcl = L \rangle \vdash nrm\ E \gg In3\ es \gg A$ 
  by (elim 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 and
    conf-v:  $normal\ s1 \longrightarrow G, store\ s1 \vdash v :: \preceq eT$ 
  by (rule hyp-e [elim-format]) simp
  show
     $s2 :: \preceq (G, L) \wedge$ 
     $(normal\ s2 \longrightarrow G, L, store\ s2 \vdash In3\ (e \# es) \gg In3\ (v \# vs) :: \preceq T) \wedge$ 
     $(error-free\ (Norm\ s0) = error-free\ s2)$ 
  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

```

```

    ( $\text{prg} = G, \text{cls} = \text{acc}C, \text{lcl} = L$ )  $\vdash \text{dom} (\text{locals} (\text{store } s1)) \gg \text{In3 } es \gg A'$ 
  proof -
    from eval-e wt-e da-e wf True
    have  $\text{nrm } E \subseteq \text{dom} (\text{locals} (\text{store } s1))$ 
      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:  $s2 :: \preceq (G, L)$  and
    error-free-s2: error-free s2 and
    conf-vs:  $\text{normal } s2 \longrightarrow \text{list-all2} (\text{conf } G (\text{store } s2)) \text{ vs } esT$ 
    by (rule hyp-es [elim-format]) (simp add: error-free-s1)
  moreover
  from True eval-es conf-v
  have conf-v':  $G, \text{store } s2 \vdash v :: \preceq eT$ 
    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:  $G \vdash s0 \dashv\rightarrow (v, s1)$ 
  and conf:  $s0 :: \preceq (G, L)$ 
  and wt:  $(\text{prg} = G, \text{cls} = \text{acc}C, \text{lcl} = L) \vdash t :: T$ 
  and da:  $(\text{prg} = G, \text{cls} = \text{acc}C, \text{lcl} = L) \vdash \text{dom} (\text{locals} (\text{snd } s0)) \gg t \gg A$ 
  and wf: wf-prog G
  and elim:  $\llbracket s1 :: \preceq (G, L); \text{normal } s1 \Longrightarrow G, L, \text{snd } s1 \vdash t \succ v :: \preceq T; \text{error-free } s0 = \text{error-free } s1 \rrbracket \Longrightarrow P$ 
  shows P
  using eval wt da wf conf
  by (rule eval-type-sound [elim-format]) (iprover intro: elim)

corollary eval-ts:
   $\llbracket G \vdash s \dashv\rightarrow v \rightarrow s'; \text{wf-prog } G; s :: \preceq (G, L); (\text{prg} = G, \text{cls} = C, \text{lcl} = L) \vdash e :: \dashv T; \text{prg} = G, \text{cls} = C, \text{lcl} = L \vdash \text{dom} (\text{locals} (\text{store } s)) \gg \text{In11 } e \gg A \rrbracket$ 
 $\Longrightarrow s' :: \preceq (G, L) \wedge (\text{normal } s' \longrightarrow G, \text{store } s' \vdash v :: \preceq T) \wedge (\text{error-free } s = \text{error-free } s')$ 
  apply (drule (4) eval-type-sound)
  apply clarsimp
  done

corollary evals-ts:
   $\llbracket G \vdash s \dashv\rightarrow es \dashv\rightarrow vs \rightarrow s'; \text{wf-prog } G; s :: \preceq (G, L); (\text{prg} = G, \text{cls} = C, \text{lcl} = L) \vdash es :: \dashv Ts; \text{prg} = G, \text{cls} = C, \text{lcl} = L \vdash \text{dom} (\text{locals} (\text{store } s)) \gg \text{In3 } es \gg A \rrbracket$ 
 $\Longrightarrow s' :: \preceq (G, L) \wedge (\text{normal } s' \longrightarrow \text{list-all2} (\text{conf } G (\text{store } s')) \text{ vs } Ts) \wedge (\text{error-free } s = \text{error-free } s')$ 
  apply (drule (4) eval-type-sound)
  apply clarsimp
  done

corollary evar-ts:
   $\llbracket G \vdash s \dashv\rightarrow v \dashv\rightarrow vf \rightarrow s'; \text{wf-prog } G; s :: \preceq (G, L); (\text{prg} = G, \text{cls} = C, \text{lcl} = L) \vdash v :: T; \text{prg} = G, \text{cls} = C, \text{lcl} = L \vdash \text{dom} (\text{locals} (\text{store } s)) \gg \text{In3 } v \gg A \rrbracket$ 

```

```

( $\llbracket \text{prg} = G, \text{cls} = C, \text{lcl} = L \rrbracket \vdash_{\text{dom}} (\text{locals } (\text{store } s)) \gg \text{In2 } v \gg A \rrbracket \implies$ 
 $s' :: \preceq (G, L) \wedge (\text{normal } s' \longrightarrow G, L, (\text{store } s') \vdash_{\text{In2}} v \gg \text{In2 } v f :: \preceq \text{Inl } T) \wedge$ 
 $(\text{error-free } s = \text{error-free } s')$ 
apply (drule (4) eval-type-sound)
apply clarsimp
done

```

```

theorem exec-ts:
 $\llbracket G \vdash s - c \rightarrow s'; \text{wf-prog } G; s :: \preceq (G, L); (\llbracket \text{prg} = G, \text{cls} = C, \text{lcl} = L \rrbracket \vdash c :: \sqrt{\phantom{x}};$ 
 $\llbracket \text{prg} = G, \text{cls} = C, \text{lcl} = L \rrbracket \vdash_{\text{dom}} (\text{locals } (\text{store } s)) \gg \text{In1r } c \gg A \rrbracket$ 
 $\implies s' :: \preceq (G, L) \wedge (\text{error-free } s \longrightarrow \text{error-free } s')$ 
apply (drule (4) eval-type-sound)
apply clarsimp
done

```

**lemma** wf-eval-Fin:

```

assumes wf: wf-prog G
and wt-c1: ( $\llbracket \text{prg} = G, \text{cls} = C, \text{lcl} = L \rrbracket \vdash_{\text{In1r}} c1 :: \text{Inl } (\text{PrimT } \text{Void})$ )
and da-c1: ( $\llbracket \text{prg} = G, \text{cls} = C, \text{lcl} = L \rrbracket \vdash_{\text{dom}} (\text{locals } (\text{store } (\text{Norm } s0))) \gg \text{In1r } c1 \gg A$ )
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

```

## 48 Ideas for the future

In the type soundness proof and the correctness proof of definite assignment we perform induction on the evaluation relation with the further preconditions that the term is welltyped and definitely assigned. During the proofs we have to establish the welltypedness and definite assignment of the subterms to be able to apply the induction hypothesis. So large parts of both proofs are the same work in propagating welltypedness and definite assignment. So we can derive a new induction rule for induction on the evaluation of a wellformed term, were these propagations is already done, once and forever. Then we can do the proofs with this rule and can enjoy the time we have saved. Here is a first and incomplete sketch of such a rule.

```

theorem wellformed-eval-induct [consumes 4, case-names Abrupt Skip Expr Lab
Comp If]:
assumes eval:  $G \vdash s0 - t \rightarrow (v, s1)$ 
and wt: ( $\llbracket \text{prg} = G, \text{cls} = \text{acc } C, \text{lcl} = L \rrbracket \vdash t :: T$ )
and da: ( $\llbracket \text{prg} = G, \text{cls} = \text{acc } C, \text{lcl} = L \rrbracket \vdash_{\text{dom}} (\text{locals } (\text{store } s0)) \gg t \gg A$ )
and wf: wf-prog G
and abrupt:  $\bigwedge s \ t \ \text{abr } L \ \text{acc } C \ T \ A.$ 
 $\llbracket \llbracket \text{prg} = G, \text{cls} = \text{acc } C, \text{lcl} = L \rrbracket \vdash t :: T;$ 
 $\llbracket \llbracket \text{prg} = G, \text{cls} = \text{acc } C, \text{lcl} = L \rrbracket \vdash_{\text{dom}} (\text{locals } (\text{store } (\text{Some } \text{abr}, s))) \gg t \gg A$ 
 $\rrbracket \implies P \ L \ \text{acc } C \ (\text{Some } \text{abr}, s) \ t \ (\text{arbitrary3 } t) \ (\text{Some } \text{abr}, s)$ 
and skip:  $\bigwedge s \ L \ \text{acc } C. P \ L \ \text{acc } C \ (\text{Norm } s) \ \langle \text{Skip} \rangle_s \diamond (\text{Norm } s)$ 
and expr:  $\bigwedge e \ s0 \ s1 \ v \ L \ \text{acc } C \ eT \ E.$ 
 $\llbracket \llbracket \text{prg} = G, \text{cls} = \text{acc } C, \text{lcl} = L \rrbracket \vdash e :: - eT;$ 

```

$$\begin{aligned}
& (\text{prg} = G, \text{cls} = \text{acc}C, \text{lcl} = L) \\
& \vdash \text{dom} (\text{locals} (\text{store} ((\text{Norm } s0)::\text{state}))) \gg \langle e \rangle_e \gg E; \\
& P \ L \ \text{acc}C \ (\text{Norm } s0) \ \langle e \rangle_e \ [v]_e \ s1 \\
& \implies P \ L \ \text{acc}C \ (\text{Norm } s0) \ \langle \text{Expr } e \rangle_s \ \Diamond \ s1
\end{aligned}$$

**and**  $\text{lab}: \bigwedge c \ l \ s0 \ s1 \ L \ \text{acc}C \ C.$

$$\begin{aligned}
& \llbracket (\text{prg} = G, \text{cls} = \text{acc}C, \text{lcl} = L) \vdash c::\checkmark; \\
& (\text{prg} = G, \text{cls} = \text{acc}C, \text{lcl} = L) \\
& \vdash \text{dom} (\text{locals} (\text{store} ((\text{Norm } s0)::\text{state}))) \gg \langle c \rangle_s \gg C; \\
& P \ L \ \text{acc}C \ (\text{Norm } s0) \ \langle c \rangle_s \ \Diamond \ s1 \\
& \implies P \ L \ \text{acc}C \ (\text{Norm } s0) \ \langle l \cdot c \rangle_s \ \Diamond \ (\text{abupd} (\text{absorb } l) \ s1)
\end{aligned}$$

**and**  $\text{comp}: \bigwedge c1 \ c2 \ s0 \ s1 \ s2 \ L \ \text{acc}C \ C1.$

$$\begin{aligned}
& \llbracket G \vdash \text{Norm } s0 \ -c1 \rightarrow s1; G \vdash s1 \ -c2 \rightarrow s2; \\
& (\text{prg} = G, \text{cls} = \text{acc}C, \text{lcl} = L) \vdash c1::\checkmark; \\
& (\text{prg} = G, \text{cls} = \text{acc}C, \text{lcl} = L) \vdash c2::\checkmark; \\
& (\text{prg} = G, \text{cls} = \text{acc}C, \text{lcl} = L) \vdash \\
& \text{dom} (\text{locals} (\text{store} ((\text{Norm } s0)::\text{state}))) \gg \langle c1 \rangle_s \gg C1; \\
& P \ L \ \text{acc}C \ (\text{Norm } s0) \ \langle c1 \rangle_s \ \Diamond \ s1; \\
& \bigwedge Q. \llbracket \text{normal } s1; \\
& \quad \bigwedge C2. \llbracket (\text{prg} = G, \text{cls} = \text{acc}C, \text{lcl} = L) \\
& \quad \vdash \text{dom} (\text{locals} (\text{store } s1)) \gg \langle c2 \rangle_s \gg C2; \\
& \quad P \ L \ \text{acc}C \ s1 \ \langle c2 \rangle_s \ \Diamond \ s2 \rrbracket \implies Q \\
& \rrbracket \implies Q \\
& \rrbracket \implies P \ L \ \text{acc}C \ (\text{Norm } s0) \ \langle c1;; c2 \rangle_s \ \Diamond \ s2
\end{aligned}$$

**and**  $\text{if}: \bigwedge b \ c1 \ c2 \ e \ s0 \ s1 \ s2 \ L \ \text{acc}C \ E.$

$$\begin{aligned}
& \llbracket G \vdash \text{Norm } s0 \ -e \multimap b \rightarrow s1; \\
& G \vdash s1 \ -(\text{if the-Bool } b \text{ then } c1 \text{ else } c2) \rightarrow s2; \\
& (\text{prg} = G, \text{cls} = \text{acc}C, \text{lcl} = L) \vdash e::\neg \text{PrimT Boolean}; \\
& (\text{prg} = G, \text{cls} = \text{acc}C, \text{lcl} = L) \vdash (\text{if the-Bool } b \text{ then } c1 \text{ else } c2)::\checkmark; \\
& (\text{prg} = G, \text{cls} = \text{acc}C, \text{lcl} = L) \vdash \\
& \text{dom} (\text{locals} (\text{store} ((\text{Norm } s0)::\text{state}))) \gg \langle e \rangle_e \gg E; \\
& P \ L \ \text{acc}C \ (\text{Norm } s0) \ \langle e \rangle_e \ [b]_e \ s1; \\
& \bigwedge Q. \llbracket \text{normal } s1; \\
& \quad \bigwedge C. \llbracket (\text{prg} = G, \text{cls} = \text{acc}C, \text{lcl} = L) \vdash (\text{dom} (\text{locals} (\text{store } s1))) \\
& \quad \gg \langle \text{if the-Bool } b \text{ then } c1 \text{ else } c2 \rangle_s \gg C; \\
& \quad P \ L \ \text{acc}C \ s1 \ \langle \text{if the-Bool } b \text{ then } c1 \text{ else } c2 \rangle_s \ \Diamond \ s2 \\
& \rrbracket \implies Q \\
& \rrbracket \implies Q \\
& \rrbracket \implies P \ L \ \text{acc}C \ (\text{Norm } s0) \ \langle \text{If}(e) \ c1 \ \text{Else } c2 \rangle_s \ \Diamond \ s2
\end{aligned}$$

**shows**  $P \ L \ \text{acc}C \ s0 \ t \ v \ s1$

**proof** –

**note** *inj-term-simps* [*simp*]

**from** *eval*

**show**  $\bigwedge L \ \text{acc}C \ T \ A. \llbracket (\text{prg} = G, \text{cls} = \text{acc}C, \text{lcl} = L) \vdash t::T; \\ (\text{prg} = G, \text{cls} = \text{acc}C, \text{lcl} = L) \vdash \text{dom} (\text{locals} (\text{store } s0)) \gg t \gg A \rrbracket$

$\implies P \ L \ \text{acc}C \ s0 \ t \ v \ s1 \ (\text{is PROP ?Hyp } s0 \ t \ v \ s1)$

**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* *accC* *T* *A*)

**from** *Expr.prem*s **obtain** *eT* **where**

$(\text{prg} = G, \text{cls} = \text{acc}C, \text{lcl} = L) \vdash e::\neg eT$

**by** (*elim wt-elim-cases*)

**moreover**

**from** *Expr.prem*s **obtain** *E* **where**

$(\text{prg} = G, \text{cls} = \text{acc}C, \text{lcl} = L) \vdash \text{dom} (\text{locals} (\text{store} ((\text{Norm } s0)::\text{state}))) \gg \langle e \rangle_e \gg E$

**by** (*elim da-elim-cases*) *simp*

**moreover from** *calculation*

```

have  $P\ L\ accC\ (Norm\ s0)\ \langle e \rangle_e\ [v]_e\ s1$ 
  by (rule Expr.hyps)
ultimately show ?case
  by (rule expr)
next
case ( $Lab\ s0\ c\ s1\ l\ L\ accC\ T\ A$ )
from Lab.prems
have ( $\langle prg = G, cls = accC, lcl = L \rangle \vdash c :: \checkmark$ )
  by (elim wt-elim-cases)
moreover
from Lab.prems obtain  $C$  where
  ( $\langle prg = G, cls = accC, lcl = L \rangle \vdash dom\ (locals\ (store\ ((Norm\ s0)::state))) \gg \langle c \rangle_s \gg C$ )
  by (elim da-elim-cases) simp
moreover from calculation
have  $P\ L\ accC\ (Norm\ s0)\ \langle c \rangle_s\ \Diamond\ s1$ 
  by (rule Lab.hyps)
ultimately show ?case
  by (rule lab)
next
case ( $Comp\ s0\ c1\ s1\ c2\ s2\ L\ accC\ T\ A$ )
note eval-c1 = ( $G \vdash Norm\ s0\ -c1 \rightarrow s1$ )
note eval-c2 = ( $G \vdash s1\ -c2 \rightarrow s2$ )
from Comp.prems obtain
   $wt-c1: \langle prg = G, cls = accC, lcl = L \rangle \vdash c1 :: \checkmark$  and
   $wt-c2: \langle prg = G, cls = accC, lcl = L \rangle \vdash c2 :: \checkmark$ 
  by (elim wt-elim-cases)
from Comp.prems
obtain  $C1\ C2$ 
  where  $da-c1: \langle prg = G, cls = accC, lcl = L \rangle \vdash$ 
     $dom\ (locals\ (store\ ((Norm\ s0)::state))) \gg \langle c1 \rangle_s \gg C1$  and
     $da-c2: \langle prg = G, cls = accC, lcl = L \rangle \vdash nrm\ C1 \gg \langle c2 \rangle_s \gg C2$ 
  by (elim da-elim-cases) simp
from wt-c1 da-c1
have  $P-c1: P\ L\ accC\ (Norm\ s0)\ \langle c1 \rangle_s\ \Diamond\ s1$ 
  by (rule Comp.hyps)
{
  fix  $Q$ 
  assume normal-s1: normal s1
  assume elim:  $\bigwedge C2'. \llbracket \langle prg = G, cls = accC, lcl = L \rangle \vdash dom\ (locals\ (store\ s1)) \gg \langle c2 \rangle_s \gg C2';$ 
     $P\ L\ accC\ s1\ \langle c2 \rangle_s\ \Diamond\ s2 \rrbracket \implies Q$ 

  have  $Q$ 
  proof –
    obtain  $C2'$  where
       $da: \langle prg = G, cls = accC, lcl = L \rangle \vdash dom\ (locals\ (store\ s1)) \gg \langle c2 \rangle_s \gg C2'$ 
    proof –
      from eval-c1 wt-c1 da-c1 wf normal-s1
      have  $nrm\ C1 \subseteq dom\ (locals\ (store\ s1))$ 
      by (cases rule: da-good-approxE') iprover
      with da-c2 show thesis
      by (rule da-weakenE) (rule that)
    qed
    with wt-c2 have  $P\ L\ accC\ s1\ \langle c2 \rangle_s\ \Diamond\ s2$ 
    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 =  $\langle G \vdash \text{Norm } s0 \multimap e \multimap b \rightarrow s1 \rangle$ 
  note eval-then-else =  $\langle G \vdash s1 \multimap (\text{if the-Bool } b \text{ then } c1 \text{ else } c2) \rightarrow s2 \rangle$ 
  from If.premis
  obtain
    wt-e:  $\langle \text{prg} = G, \text{cls} = \text{accC}, \text{lcl} = L \rangle \vdash e :: \text{PrimT Boolean}$  and
    wt-then-else:  $\langle \text{prg} = G, \text{cls} = \text{accC}, \text{lcl} = L \rangle \vdash (\text{if the-Bool } b \text{ then } c1 \text{ else } c2) :: \checkmark$ 
    by (elim wt-elim-cases) (auto split add: split-if)
  from If.premis obtain E C where
    da-e:  $\langle \text{prg} = G, \text{cls} = \text{accC}, \text{lcl} = L \rangle \vdash \text{dom } (\text{locals } (\text{store } ((\text{Norm } s0) :: \text{state})))$ 
       $\gg \langle e \rangle_e \gg E$  and
    da-then-else:
       $\langle \text{prg} = G, \text{cls} = \text{accC}, \text{lcl} = L \rangle \vdash$ 
       $(\text{dom } (\text{locals } (\text{store } ((\text{Norm } s0) :: \text{state})))) \cup \text{assigns-if } (\text{the-Bool } b) e$ 
       $\gg \langle \text{if the-Bool } b \text{ then } c1 \text{ else } c2 \rangle_s \gg C$ 
    by (elim da-elim-cases) (cases the-Bool b, auto)
  from wt-e da-e
  have P-e:  $P \ L \ \text{accC} \ (\text{Norm } s0) \ \langle e \rangle_e \ [b]_e \ s1$ 
    by (rule If.hyps)
  {
    fix Q
    assume normal-s1: normal s1
    assume elim:  $\bigwedge C. [\langle \text{prg} = G, \text{cls} = \text{accC}, \text{lcl} = L \rangle \vdash (\text{dom } (\text{locals } (\text{store } s1)))$ 
       $\gg \langle \text{if the-Bool } b \text{ then } c1 \text{ else } c2 \rangle_s \gg C;$ 
       $P \ L \ \text{accC} \ s1 \ \langle \text{if the-Bool } b \text{ then } c1 \text{ else } c2 \rangle_s \diamond s2$ 
       $\implies Q$ 
    have Q
    proof –
      obtain C' where
        da:  $\langle \text{prg} = G, \text{cls} = \text{accC}, \text{lcl} = L \rangle \vdash$ 
         $(\text{dom } (\text{locals } (\text{store } s1))) \gg \langle \text{if the-Bool } b \text{ then } c1 \text{ else } c2 \rangle_s \gg C'$ 
      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 normal-s1 wt-e
          have assigns-if (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 wt-then-else
      have  $P \ L \ \text{accC} \ s1 \ \langle \text{if the-Bool } b \text{ then } c1 \text{ else } c2 \rangle_s \diamond s2$ 
        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

## 49 Operational evaluation (big-step) semantics of Java expressions and statements

theory *Evaln* imports *TypeSafe* begin

Variant of *eval* relation with counter for bounded recursive depth. In principal *evaln* could replace *eval*.

Validity of the axiomatic semantics builds on *evaln*. For recursive method calls the axiomatic semantics rule assumes the method ok to derive a proof for the body. To prove the method rule sound we need to perform induction on the recursion depth. For the completeness proof of the axiomatic semantics the notion of the most general formula is used. The most general formula right now builds on the ordinary evaluation relation *eval*. So sometimes we have to switch between *evaln* and *eval* and vice versa. To make this switch easy *evaln* also does all the technical accessibility tests *check-field-access* and *check-method-access* like *eval*. If it would omit them *evaln* and *eval* would only be equivalent for welltyped, and definitely assigned terms.

inductive

```

evaln :: [prog, state, term, nat, vals, state] ⇒ bool
  (+- -->--> '(-, -') [61,61,80,61,0,0] 60)
and evaln :: [prog, state, var, vvar, nat, state] ⇒ bool
  (+- --=>--> - [61,61,90,61,61,61] 60)
and evaln :: [prog, state, expr, val, nat, state] ⇒ bool
  (+- --->--> - [61,61,80,61,61,61] 60)
and evaln :: [prog, state, expr list, val list, nat, state] ⇒ bool
  (+- --≡>--> - [61,61,61,61,61,61] 60)
and execn :: [prog, state, stmt, nat, state] ⇒ bool
  (+- ----> - [61,61,65, 61,61] 60)
for G :: prog

```

where

```

G ⊢ s -c -n→ s' ≡ G ⊢ s -In1r c>-n→ (◇ , s')
| G ⊢ s -e>-v -n→ s' ≡ G ⊢ s -In1l e>-n→ (In1 v , s')
| G ⊢ s -e>-vf -n→ s' ≡ G ⊢ s -In2 e>-n→ (In2 vf , s')
| G ⊢ s -e≡>-v -n→ s' ≡ G ⊢ s -In3 e>-n→ (In3 v , s')

```

— propagation of abrupt completion

```

| Abrupt: G ⊢ (Some xc,s) -t>-n→ (arbitrary3 t,(Some xc,s))

```

— evaluation of variables

```

| LVar: G ⊢ Norm s -LVar vn=>lvar vn s-n→ Norm s

| FVar: [G ⊢ Norm s0 -Init statDeclC-n→ s1; G ⊢ s1 -e>-a-n→ s2;
  (v,s2') = fvar statDeclC stat fn a s2;
  s3 = check-field-access G accC statDeclC fn stat a s2'] ⇒
  G ⊢ Norm s0 -{accC,statDeclC,stat}e..fn=>v-n→ s3

| AVar: [G ⊢ Norm s0 -e1>-a-n→ s1 ; G ⊢ s1 -e2>-i-n→ s2;
  (v,s2') = avar G i a s2'] ⇒
  G ⊢ Norm s0 -e1.[e2]=>v-n→ s2'

```

— evaluation of expressions

```

| NewC: [G ⊢ Norm s0 -Init C-n→ s1;

```

- $$\begin{array}{l}
G \vdash \quad s1 \text{ --halloc } (CInst \ C) \succ a \rightarrow s2 \parallel \implies \\
\quad G \vdash Norm \ s0 \text{ --NewC } C \text{ --}\succ Addr \ a \text{ --}n \rightarrow s2 \\
| \text{ NewA: } \parallel G \vdash Norm \ s0 \text{ --init-comp-ty } T \text{ --}n \rightarrow s1; \ G \vdash s1 \text{ --e--}\succ i' \text{ --}n \rightarrow s2; \\
\quad G \vdash abupd \ (check\text{-neg } i') \ s2 \text{ --halloc } (Arr \ T \ (the\text{-Intg } i')) \succ a \rightarrow s3 \parallel \implies \\
\quad G \vdash Norm \ s0 \text{ --New } T[e] \text{ --}\succ Addr \ a \text{ --}n \rightarrow s3 \\
| \text{ Cast: } \parallel G \vdash Norm \ s0 \text{ --e--}\succ v \text{ --}n \rightarrow s1; \\
\quad s2 = abupd \ (raise\text{-if } (\neg G, snd \ s1 \vdash v \text{ fits } T) \ ClassCast) \ s1 \parallel \implies \\
\quad G \vdash Norm \ s0 \text{ --Cast } T \ e \text{ --}\succ v \text{ --}n \rightarrow s2 \\
| \text{ Inst: } \parallel G \vdash Norm \ s0 \text{ --e--}\succ v \text{ --}n \rightarrow s1; \\
\quad b = (v \neq Null \wedge G, store \ s1 \vdash v \text{ fits } RefT \ T) \parallel \implies \\
\quad G \vdash Norm \ s0 \text{ --e } InstOf \ T \text{ --}\succ Bool \ b \text{ --}n \rightarrow s1 \\
| \text{ Lit:} \quad G \vdash Norm \ s \text{ --Lit } v \text{ --}\succ v \text{ --}n \rightarrow Norm \ s \\
| \text{ UnOp: } \parallel G \vdash Norm \ s0 \text{ --e--}\succ v \text{ --}n \rightarrow s1 \parallel \\
\implies G \vdash Norm \ s0 \text{ --UnOp } unop \ e \text{ --}\succ (eval\text{-unop } unop \ v) \text{ --}n \rightarrow s1 \\
| \text{ BinOp: } \parallel G \vdash Norm \ s0 \text{ --e1--}\succ v1 \text{ --}n \rightarrow s1; \\
\quad G \vdash s1 \text{ --(if need-second-arg binop v1 then (In1l e2) else (In1r Skip))} \\
\quad \text{ --}\succ \text{ --}n \rightarrow (In1 \ v2, s2) \parallel \\
\implies G \vdash Norm \ s0 \text{ --BinOp } binop \ e1 \ e2 \text{ --}\succ (eval\text{-binop } binop \ v1 \ v2) \text{ --}n \rightarrow s2 \\
| \text{ Super:} \quad G \vdash Norm \ s \text{ --Super--}\succ val\text{-this } s \text{ --}n \rightarrow Norm \ s \\
| \text{ Acc: } \parallel G \vdash Norm \ s0 \text{ --va=}\succ (v, f) \text{ --}n \rightarrow s1 \parallel \implies \\
\quad G \vdash Norm \ s0 \text{ --Acc } va \text{ --}\succ v \text{ --}n \rightarrow s1 \\
| \text{ Ass: } \parallel G \vdash Norm \ s0 \text{ --va=}\succ (w, f) \text{ --}n \rightarrow s1; \\
\quad G \vdash \quad s1 \text{ --e--}\succ v \quad \text{ --}n \rightarrow s2 \parallel \implies \\
\quad G \vdash Norm \ s0 \text{ --va:=e--}\succ v \text{ --}n \rightarrow assign \ f \ v \ s2 \\
| \text{ Cond: } \parallel G \vdash Norm \ s0 \text{ --e0--}\succ b \text{ --}n \rightarrow s1; \\
\quad G \vdash \quad s1 \text{ --(if the-Bool b then e1 else e2)--}\succ v \text{ --}n \rightarrow s2 \parallel \implies \\
\quad G \vdash Norm \ s0 \text{ --e0 ? e1 : e2--}\succ v \text{ --}n \rightarrow s2 \\
| \text{ Call:} \\
\parallel G \vdash Norm \ s0 \text{ --e--}\succ a' \text{ --}n \rightarrow s1; \ G \vdash s1 \text{ --args=}\succ vs \text{ --}n \rightarrow s2; \\
\quad D = invocation\text{-declclass } G \ mode \ (store \ s2) \ a' \ statT \ (\llbracket name=mn, parTs=pTs \rrbracket); \\
\quad s3 = init\text{-lvars } G \ D \ (\llbracket name=mn, parTs=pTs \rrbracket) \ mode \ a' \ vs \ s2; \\
\quad s3' = check\text{-method-access } G \ accC \ statT \ mode \ (\llbracket name=mn, parTs=pTs \rrbracket) \ a' \ s3; \\
\quad G \vdash s3' \text{ --Methd } D \ (\llbracket name=mn, parTs=pTs \rrbracket) \text{ --}\succ v \text{ --}n \rightarrow s4 \\
\parallel \\
\implies \\
\quad G \vdash Norm \ s0 \text{ --}\{accC, statT, mode\}e.mn(\{pTs\}args)\text{ --}\succ v \text{ --}n \rightarrow (restore\text{-lvars } s2 \ s4) \\
| \text{ Methd: } \parallel G \vdash Norm \ s0 \text{ --body } G \ D \ sig \text{ --}\succ v \text{ --}n \rightarrow s1 \parallel \implies \\
\quad G \vdash Norm \ s0 \text{ --Methd } D \ sig \text{ --}\succ v \text{ --}Suc \ n \rightarrow s1 \\
| \text{ Body: } \parallel G \vdash Norm \ s0 \text{ --Init } D \text{ --}n \rightarrow s1; \ G \vdash s1 \text{ --c--}n \rightarrow s2; \\
\quad s3 = (if \ (\exists \ l. abrupt \ s2 = Some \ (Jump \ (Break \ l))) \vee \\
\quad \quad \quad abrupt \ s2 = Some \ (Jump \ (Cont \ l))) \\
\quad \quad \quad then \ abupd \ (\lambda \ x. Some \ (Error \ CrossMethodJump)) \ s2 \\
\quad \quad \quad else \ s2) \parallel \implies \\
\quad G \vdash Norm \ s0 \text{ --Body } D \ c \\
\quad \text{ --}\succ the \ (locals \ (store \ s2) \ Result) \text{ --}n \rightarrow abupd \ (absorb \ Ret) \ s3
\end{array}$$

— evaluation of expression lists

| *Nil*:

$$G \vdash \text{Norm } s0 \text{ --} [\dot{=} \dot{\succ}] \text{ --} n \rightarrow \text{Norm } s0$$

| *Cons*:  $\llbracket G \vdash \text{Norm } s0 \text{ --} e \dot{\succ} v \text{ --} n \rightarrow s1;$

$$G \vdash s1 \text{ --} es \dot{=} \dot{\succ} vs \text{ --} n \rightarrow s2 \rrbracket \implies \\ G \vdash \text{Norm } s0 \text{ --} e \# es \dot{=} \dot{\succ} v \# vs \text{ --} n \rightarrow s2$$

— execution of statements

| *Skip*:

$$G \vdash \text{Norm } s \text{ --} \text{Skip} \text{ --} n \rightarrow \text{Norm } s$$

| *Expr*:  $\llbracket G \vdash \text{Norm } s0 \text{ --} e \dot{\succ} v \text{ --} n \rightarrow s1 \rrbracket \implies$

$$G \vdash \text{Norm } s0 \text{ --} \text{Expr } e \text{ --} n \rightarrow s1$$

| *Lab*:  $\llbracket G \vdash \text{Norm } s0 \text{ --} c \text{ --} n \rightarrow s1 \rrbracket \implies$

$$G \vdash \text{Norm } s0 \text{ --} l \cdot c \text{ --} n \rightarrow \text{abupd } (\text{absorb } l) s1$$

| *Comp*:  $\llbracket G \vdash \text{Norm } s0 \text{ --} c1 \text{ --} n \rightarrow s1;$

$$G \vdash s1 \text{ --} c2 \text{ --} n \rightarrow s2 \rrbracket \implies \\ G \vdash \text{Norm } s0 \text{ --} c1;; c2 \text{ --} n \rightarrow s2$$

| *If*:  $\llbracket G \vdash \text{Norm } s0 \text{ --} e \dot{\succ} b \text{ --} n \rightarrow s1;$

$$G \vdash s1 \text{ --} (\text{if the-Bool } b \text{ then } c1 \text{ else } c2) \text{ --} n \rightarrow s2 \rrbracket \implies \\ G \vdash \text{Norm } s0 \text{ --} \text{If}(e) c1 \text{ Else } c2 \text{ --} n \rightarrow s2$$

| *Loop*:  $\llbracket G \vdash \text{Norm } s0 \text{ --} e \dot{\succ} b \text{ --} n \rightarrow s1;$

$$\text{if the-Bool } b \\ \text{then } (G \vdash s1 \text{ --} c \text{ --} n \rightarrow s2 \wedge \\ G \vdash (\text{abupd } (\text{absorb } (\text{Cont } l)) s2) \text{ --} l \cdot \text{While}(e) c \text{ --} n \rightarrow s3) \\ \text{else } s3 = s1 \rrbracket \implies \\ G \vdash \text{Norm } s0 \text{ --} l \cdot \text{While}(e) c \text{ --} n \rightarrow s3$$

| *Jmp*:  $G \vdash \text{Norm } s \text{ --} \text{Jmp } j \text{ --} n \rightarrow (\text{Some } (\text{Jump } j), s)$

| *Throw*:  $\llbracket G \vdash \text{Norm } s0 \text{ --} e \dot{\succ} a' \text{ --} n \rightarrow s1 \rrbracket \implies$

$$G \vdash \text{Norm } s0 \text{ --} \text{Throw } e \text{ --} n \rightarrow \text{abupd } (\text{throw } a') s1$$

| *Try*:  $\llbracket G \vdash \text{Norm } s0 \text{ --} c1 \text{ --} n \rightarrow s1; G \vdash s1 \text{ --} \text{salloc} \rightarrow s2;$

$$\text{if } G, s2 \vdash \text{catch } tn \text{ then } G \vdash \text{new-xcpt-var } vn \text{ } s2 \text{ --} c2 \text{ --} n \rightarrow s3 \text{ else } s3 = s2 \rrbracket \\ \implies \\ G \vdash \text{Norm } s0 \text{ --} \text{Try } c1 \text{ Catch}(tn \text{ } vn) \text{ } c2 \text{ --} n \rightarrow s3$$

| *Fin*:  $\llbracket G \vdash \text{Norm } s0 \text{ --} c1 \text{ --} n \rightarrow (x1, s1);$

$$G \vdash \text{Norm } s1 \text{ --} c2 \text{ --} n \rightarrow s2; \\ s3 = (\text{if } (\exists \text{ err. } x1 = \text{Some } (\text{Error } \text{err})) \\ \text{then } (x1, s1) \\ \text{else } \text{abupd } (\text{abrupt-if } (x1 \neq \text{None}) x1) s2) \rrbracket \implies \\ G \vdash \text{Norm } s0 \text{ --} c1 \text{ Finally } c2 \text{ --} n \rightarrow s3$$

| *Init*:  $\llbracket \text{the } (\text{class } G \text{ } C) = c;$

$$\text{if inited } C \text{ (globs } s0) \text{ then } s3 = \text{Norm } s0 \\ \text{else } (G \vdash \text{Norm } (\text{init-class-obj } G \text{ } C \text{ } s0) \\ \text{--} (\text{if } C = \text{Object then Skip else Init } (\text{super } c)) \text{ --} n \rightarrow s1 \wedge \\ G \vdash \text{set-lvars empty } s1 \text{ --} \text{init } c \text{ --} n \rightarrow s2 \wedge \\ s3 = \text{restore-lvars } s1 \text{ } s2) \rrbracket \\ \implies$$

$$G \vdash \text{Norm } s0 \text{ --Init } C \text{ --}n \rightarrow s3$$
**monos***if-bool-eq-conj*

**declare** *split-if* [*split del*] *split-if-asm* [*split del*]  
*option.split* [*split del*] *option.split-asm* [*split del*]  
*not-None-eq* [*simp del*]  
*split-paired-All* [*simp del*] *split-paired-Ex* [*simp del*]  
**declaration**  $\ll K \text{ (Simplifier.map-ss (fn ss => ss delloop split-all-tac))} \gg$

**inductive-cases** *evaln-cases*:  $G \vdash s \text{ --}t \succ \text{--}n \rightarrow (v, s')$ **inductive-cases** *evaln-elim-cases*:

$G \vdash (\text{Some } xc, s) \text{ --}t$	$\succ \text{--}n \rightarrow (v, s')$
$G \vdash \text{Norm } s \text{ --In1r Skip}$	$\succ \text{--}n \rightarrow (x, s')$
$G \vdash \text{Norm } s \text{ --In1r (Jmp } j)$	$\succ \text{--}n \rightarrow (x, s')$
$G \vdash \text{Norm } s \text{ --In1r (l. } c)$	$\succ \text{--}n \rightarrow (x, s')$
$G \vdash \text{Norm } s \text{ --In3 } ([\ ])$	$\succ \text{--}n \rightarrow (v, s')$
$G \vdash \text{Norm } s \text{ --In3 (e\#es)}$	$\succ \text{--}n \rightarrow (v, s')$
$G \vdash \text{Norm } s \text{ --In1l (Lit } w)$	$\succ \text{--}n \rightarrow (v, s')$
$G \vdash \text{Norm } s \text{ --In1l (UnOp unop } e)$	$\succ \text{--}n \rightarrow (v, s')$
$G \vdash \text{Norm } s \text{ --In1l (BinOp binop } e1 \ e2)$	$\succ \text{--}n \rightarrow (v, s')$
$G \vdash \text{Norm } s \text{ --In2 (LVar } vn)$	$\succ \text{--}n \rightarrow (v, s')$
$G \vdash \text{Norm } s \text{ --In1l (Cast } T \ e)$	$\succ \text{--}n \rightarrow (v, s')$
$G \vdash \text{Norm } s \text{ --In1l (e InstOf } T)$	$\succ \text{--}n \rightarrow (v, s')$
$G \vdash \text{Norm } s \text{ --In1l (Super)}$	$\succ \text{--}n \rightarrow (v, s')$
$G \vdash \text{Norm } s \text{ --In1l (Acc } va)$	$\succ \text{--}n \rightarrow (v, s')$
$G \vdash \text{Norm } s \text{ --In1r (Expr } e)$	$\succ \text{--}n \rightarrow (x, s')$
$G \vdash \text{Norm } s \text{ --In1r (c1;; c2)}$	$\succ \text{--}n \rightarrow (x, s')$
$G \vdash \text{Norm } s \text{ --In1l (Methd } C \text{ sig)}$	$\succ \text{--}n \rightarrow (x, s')$
$G \vdash \text{Norm } s \text{ --In1l (Body } D \ c)$	$\succ \text{--}n \rightarrow (x, s')$
$G \vdash \text{Norm } s \text{ --In1l (e0 ? e1 : e2)}$	$\succ \text{--}n \rightarrow (v, s')$
$G \vdash \text{Norm } s \text{ --In1r (If(e) c1 Else c2)}$	$\succ \text{--}n \rightarrow (x, s')$
$G \vdash \text{Norm } s \text{ --In1r (l. While(e) c)}$	$\succ \text{--}n \rightarrow (x, s')$
$G \vdash \text{Norm } s \text{ --In1r (c1 Finally c2)}$	$\succ \text{--}n \rightarrow (x, s')$
$G \vdash \text{Norm } s \text{ --In1r (Throw } e)$	$\succ \text{--}n \rightarrow (x, s')$
$G \vdash \text{Norm } s \text{ --In1l (NewC } C)$	$\succ \text{--}n \rightarrow (v, s')$
$G \vdash \text{Norm } s \text{ --In1l (New } T[e])$	$\succ \text{--}n \rightarrow (v, s')$
$G \vdash \text{Norm } s \text{ --In1l (Ass } va \ e)$	$\succ \text{--}n \rightarrow (v, s')$
$G \vdash \text{Norm } s \text{ --In1r (Try c1 Catch(tn vn) c2)}$	$\succ \text{--}n \rightarrow (x, s')$
$G \vdash \text{Norm } s \text{ --In2 } (\{accC, statDeclC, stat\}e..fn)$	$\succ \text{--}n \rightarrow (v, s')$
$G \vdash \text{Norm } s \text{ --In2 } (e1.[e2])$	$\succ \text{--}n \rightarrow (v, s')$
$G \vdash \text{Norm } s \text{ --In1l } (\{accC, statT, mode\}e.mn(\{pT\}p))$	$\succ \text{--}n \rightarrow (v, s')$
$G \vdash \text{Norm } s \text{ --In1r (Init } C)$	$\succ \text{--}n \rightarrow (x, s')$
$G \vdash \text{Norm } s \text{ --In1r (Init } C)$	$\succ \text{--}n \rightarrow (x, s')$

**declare** *split-if* [*split*] *split-if-asm* [*split*]  
*option.split* [*split*] *option.split-asm* [*split*]  
*not-None-eq* [*simp*]  
*split-paired-All* [*simp*] *split-paired-Ex* [*simp*]  
**declaration**  $\ll K \text{ (Simplifier.map-ss (fn ss => ss addloop (split-all-tac, split-all-tac)))} \gg$

**lemma** *evaln-Inj-elim*:  $G \vdash s \text{ --}t \succ \text{--}n \rightarrow (w, s') \implies \text{case } t \text{ of In1 } ec \Rightarrow$ 

(*case ec of In1 e*  $\Rightarrow (\exists v. w = \text{In1 } v) \mid \text{Inr } c \Rightarrow w = \Diamond$ )  
 $\mid \text{In2 } e \Rightarrow (\exists v. w = \text{In2 } v) \mid \text{In3 } e \Rightarrow (\exists v. w = \text{In3 } v)$

**apply** (*erule evaln-cases* , *auto*)**apply** (*induct-tac t*)

```

apply (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:  $G \vdash s - In1l \ t \succ - n \rightarrow (w, s') = (\exists v. w = In1 \ v \wedge G \vdash s - t \succ v - n \rightarrow s')$ 
by (auto, frule evaln-Inj-elim, auto)

```

```

lemma evaln-var-eq:  $G \vdash s - In2 \ t \succ - n \rightarrow (w, s') = (\exists vf. w = In2 \ vf \wedge G \vdash s - t = \succ vf - n \rightarrow s')$ 
by (auto, frule evaln-Inj-elim, auto)

```

```

lemma evaln-exprs-eq:  $G \vdash s - In3 \ t \succ - n \rightarrow (w, s') = (\exists vs. w = In3 \ vs \wedge G \vdash s - t \dot{=} \succ vs - n \rightarrow s')$ 
by (auto, frule evaln-Inj-elim, auto)

```

```

lemma evaln-stmt-eq:  $G \vdash s - In1r \ t \succ - n \rightarrow (w, s') = (w = \Diamond \wedge G \vdash s - t - n \rightarrow s')$ 
by (auto, frule evaln-Inj-elim, auto, frule evaln-Inj-elim, auto)

```

```

simproc-setup evaln-expr (G ⊢ s - In1l t ≻ - n → (w, s')) = ⟨⟨
  fn - => fn - => fn ct =>
    (case Thm.term-of ct of
      (- $ - $ - $ - $ (Const - $ -) $ -) => NONE
      | - => SOME (mk-meta-eq @ {thm evaln-expr-eq})) ⟩⟩

```

```

simproc-setup evaln-var (G ⊢ s - In2 t ≻ - n → (w, s')) = ⟨⟨
  fn - => fn - => fn ct =>
    (case Thm.term-of ct of
      (- $ - $ - $ - $ (Const - $ -) $ -) => NONE
      | - => SOME (mk-meta-eq @ {thm evaln-var-eq})) ⟩⟩

```

```

simproc-setup evaln-exprs (G ⊢ s - In3 t ≻ - n → (w, s')) = ⟨⟨
  fn - => fn - => fn ct =>
    (case Thm.term-of ct of
      (- $ - $ - $ - $ (Const - $ -) $ -) => NONE
      | - => SOME (mk-meta-eq @ {thm evaln-exprs-eq})) ⟩⟩

```

```

simproc-setup evaln-stmt (G ⊢ s - In1r t ≻ - n → (w, s')) = ⟨⟨
  fn - => fn - => fn ct =>
    (case Thm.term-of ct of
      (- $ - $ - $ - $ (Const - $ -) $ -) => NONE
      | - => SOME (mk-meta-eq @ {thm evaln-stmt-eq})) ⟩⟩

```

```

ML ⟨⟨ bind-thms (evaln-AbruptIs, sum3-instantiate @ {thm evaln.Abrupt}) ⟩⟩
declare evaln-AbruptIs [intro!]

```

```

lemma evaln-Callee:  $G \vdash Norm \ s - In1l \ (Callee \ l \ e) \succ - n \rightarrow (v, s') = False$ 
proof -
  { fix s t v s'
    assume eval:  $G \vdash s - t \succ - n \rightarrow (v, s')$  and
      normal: normal s and
      callee:  $t = In1l \ (Callee \ l \ e)$ 
    then have False by induct auto
  }

```

```

then show ?thesis
  by (cases s') fastsimp
qed

```

```

lemma evaln-InsInitE:  $G \vdash \text{Norm } s - \text{In1l } (\text{InsInitE } c \ e) \succ -n \rightarrow (v, s') = \text{False}$ 
proof -
  { fix s t v s'
    assume eval:  $G \vdash s - t \succ -n \rightarrow (v, s')$  and
      normal: normal s and
      callee:  $t = \text{In1l } (\text{InsInitE } c \ e)$ 
    then have False by induct auto
  }
then show ?thesis
  by (cases s') fastsimp
qed

```

```

lemma evaln-InsInitV:  $G \vdash \text{Norm } s - \text{In2 } (\text{InsInitV } c \ w) \succ -n \rightarrow (v, s') = \text{False}$ 
proof -
  { fix s t v s'
    assume eval:  $G \vdash s - t \succ -n \rightarrow (v, s')$  and
      normal: normal s and
      callee:  $t = \text{In2 } (\text{InsInitV } c \ w)$ 
    then have False by induct auto
  }
then show ?thesis
  by (cases s') fastsimp
qed

```

```

lemma evaln-FinA:  $G \vdash \text{Norm } s - \text{In1r } (\text{FinA } a \ c) \succ -n \rightarrow (v, s') = \text{False}$ 
proof -
  { fix s t v s'
    assume eval:  $G \vdash s - t \succ -n \rightarrow (v, s')$  and
      normal: normal s and
      callee:  $t = \text{In1r } (\text{FinA } a \ c)$ 
    then have False by induct auto
  }
then show ?thesis
  by (cases s') fastsimp
qed

```

```

lemma evaln-abrupt-lemma:  $G \vdash s - e \succ -n \rightarrow (v, s') \implies$ 
   $\text{fst } s = \text{Some } xc \longrightarrow s' = s \wedge v = \text{arbitrary3 } e$ 
apply (erule evaln-cases , auto)
done

```

```

lemma evaln-abrupt:
   $\wedge s'. G \vdash (\text{Some } xc, s) - e \succ -n \rightarrow (w, s') = (s' = (\text{Some } xc, s) \wedge$ 
   $w = \text{arbitrary3 } e \wedge G \vdash (\text{Some } xc, s) - e \succ -n \rightarrow (\text{arbitrary3 } e, (\text{Some } xc, s)))$ 
apply auto
apply (frule evaln-abrupt-lemma, auto)+
done

```

```

simproc-setup evaln-abrupt ( $G \vdash (\text{Some } xc, s) - e \succ -n \rightarrow (w, s') = \ll$ 
   $\text{fn } - \implies \text{fn } - \implies \text{fn } ct \implies$ 

```

```

(case Thm.term-of ct of
  (- $ - $ - $ - $ - $ (Const (@{const-name Pair}, -) $ (Const (@{const-name Some},-) $ -)$ -))
  => NONE
| - => SOME (mk-meta-eq @{thm evaln-abrupt}))
>>

```

**lemma** *evaln-LitI*:  $G \vdash s \text{ --Lit } v \text{ --} \succ (\text{if normal } s \text{ then } v \text{ else arbitrary}) \text{ --} n \rightarrow s$   
**apply** (case-tac s, case-tac a = None)  
**by** (auto intro!: evaln.Lit)

**lemma** *CondI*:  
 $\bigwedge s1. \llbracket G \vdash s \text{ --} e \text{ --} \succ b \text{ --} n \rightarrow s1; G \vdash s1 \text{ --} (\text{if the-Bool } b \text{ then } e1 \text{ else } e2) \text{ --} \succ v \text{ --} n \rightarrow s2 \rrbracket \implies$   
 $G \vdash s \text{ --} e \text{ ? } e1 : e2 \text{ --} \succ (\text{if normal } s1 \text{ then } v \text{ else arbitrary}) \text{ --} n \rightarrow s2$   
**apply** (case-tac s, case-tac a = None)  
**by** (auto intro!: evaln.Cond)

**lemma** *evaln-SkipI* [intro!]:  $G \vdash s \text{ --Skip--} n \rightarrow s$   
**apply** (case-tac s, case-tac a = None)  
**by** (auto intro!: evaln.Skip)

**lemma** *evaln-ExprI*:  $G \vdash s \text{ --} e \text{ --} \succ v \text{ --} n \rightarrow s' \implies G \vdash s \text{ --Expr } e \text{ --} n \rightarrow s'$   
**apply** (case-tac s, case-tac a = None)  
**by** (auto intro!: evaln.Expr)

**lemma** *evaln-CompI*:  $\llbracket G \vdash s \text{ --} c1 \text{ --} n \rightarrow s1; G \vdash s1 \text{ --} c2 \text{ --} n \rightarrow s2 \rrbracket \implies G \vdash s \text{ --} c1;; c2 \text{ --} n \rightarrow s2$   
**apply** (case-tac s, case-tac a = None)  
**by** (auto intro!: evaln.Comp)

**lemma** *evaln-IfI*:  
 $\llbracket G \vdash s \text{ --} e \text{ --} \succ v \text{ --} n \rightarrow s1; G \vdash s1 \text{ --} (\text{if the-Bool } v \text{ then } c1 \text{ else } c2) \text{ --} n \rightarrow s2 \rrbracket \implies$   
 $G \vdash s \text{ --If}(e) \text{ } c1 \text{ Else } c2 \text{ --} n \rightarrow s2$   
**apply** (case-tac s, case-tac a = None)  
**by** (auto intro!: evaln.If)

**lemma** *evaln-SkipD* [dest!]:  $G \vdash s \text{ --Skip--} n \rightarrow s' \implies s' = s$   
**by** (erule evaln-cases, auto)

**lemma** *evaln-Skip-eq* [simp]:  $G \vdash s \text{ --Skip--} n \rightarrow s' = (s = s')$   
**apply** auto  
**done**

## evaln implies eval

**lemma** *evaln-eval*:  
**assumes** *evaln*:  $G \vdash s0 \text{ --} t \text{ --} \succ \text{--} n \rightarrow (v, s1)$   
**shows**  $G \vdash s0 \text{ --} t \text{ --} \succ \text{--} n \rightarrow (v, s1)$   
**using** *evaln*  
**proof** (induct)  
**case** (Loop s0 e b n s1 c s2 l s3)  
**note**  $\langle G \vdash \text{Norm } s0 \text{ --} e \text{ --} \succ b \text{ --} n \rightarrow s1 \rangle$   
**moreover**



```

have if the-Bool b
  then ( $G \vdash s1 \multimap c \rightarrow s2$ )  $\wedge$ 
     $G \vdash \text{abupd } (\text{absorb } (\text{Cont } l)) \ s2 \multimap l \cdot \text{While}(e) \ c \rightarrow s3$ 
  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 ( $G \vdash \text{Norm } s0 \multimap c1 \rightarrow s1$ )
moreover
note ( $G \vdash s1 \multimap \text{salloc} \rightarrow s2$ )
moreover
have if  $G, s2 \vdash \text{catch } C \text{ then } G \vdash \text{new-xcpt-var } vn \ s2 \multimap c2 \rightarrow s3$  else  $s3 = s2$ 
  using Try.hyps by simp
ultimately show ?case by (rule eval.Try)
next
case (Init C c s0 s3 n s1 s2)
note (the (class G C) = c)
moreover
have if inited C (globs s0)
  then  $s3 = \text{Norm } s0$ 
  else  $G \vdash \text{Norm } ((\text{init-class-obj } G \ C) \ s0)$ 
     $\multimap$  (if C = Object then Skip else Init (super c))  $\rightarrow s1 \wedge$ 
     $G \vdash (\text{set-lvars empty}) \ s1 \multimap \text{init } c \rightarrow s2 \wedge$ 
     $s3 = (\text{set-lvars } (\text{locals } (\text{store } s1))) \ s2$ 
  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]:
   $G \vdash s \multimap t \succ \multimap n \rightarrow (w, s') \implies \forall m. n \leq m \longrightarrow G \vdash s \multimap t \succ \multimap m \rightarrow (w, s')$ 
apply (erule evaln.induct)
apply (tactic  $\llbracket \text{ALLGOALS } (\text{EVERY}' [\text{strip-tac}, \text{TRY } o \text{ etac } (\text{thm } \text{Suc-le-D-lemma}),$ 
  REPEAT o smp-tac 1,
  resolve-tac (thms evaln.intros) THEN-ALL-NEW TRY o atac] ) \rrbracket)
apply (auto split del: split-if)
done

```

```

lemmas evaln-nonstrict-Suc = evaln-nonstrict [OF - le-refl [THEN le-SucI]]

```

```

lemma evaln-max2:  $\llbracket G \vdash s1 \multimap t1 \succ \multimap n1 \rightarrow (w1, s1'); G \vdash s2 \multimap t2 \succ \multimap n2 \rightarrow (w2, s2') \rrbracket \implies$ 
   $G \vdash s1 \multimap t1 \succ \multimap \text{max } n1 \ n2 \rightarrow (w1, s1') \wedge G \vdash s2 \multimap t2 \succ \multimap \text{max } n1 \ n2 \rightarrow (w2, s2')$ 
by (fast intro: le-maxI1 le-maxI2)

```

```

corollary evaln-max2E [consumes 2]:
   $\llbracket G \vdash s1 \multimap t1 \succ \multimap n1 \rightarrow (w1, s1'); G \vdash s2 \multimap t2 \succ \multimap n2 \rightarrow (w2, s2') \rrbracket$ 
   $\llbracket G \vdash s1 \multimap t1 \succ \multimap \text{max } n1 \ n2 \rightarrow (w1, s1'); G \vdash s2 \multimap t2 \succ \multimap \text{max } n1 \ n2 \rightarrow (w2, s2') \rrbracket \implies P \rrbracket \implies P$ 
by (drule (1) evaln-max2) simp

```

**lemma** *evaln-max3*:

$\llbracket G \vdash s1 -t1 \succ -n1 \rightarrow (w1, s1'); G \vdash s2 -t2 \succ -n2 \rightarrow (w2, s2'); G \vdash s3 -t3 \succ -n3 \rightarrow (w3, s3') \rrbracket \Rightarrow$   
 $G \vdash s1 -t1 \succ -\max (\max n1 n2) n3 \rightarrow (w1, s1') \wedge$   
 $G \vdash s2 -t2 \succ -\max (\max n1 n2) n3 \rightarrow (w2, s2') \wedge$   
 $G \vdash s3 -t3 \succ -\max (\max n1 n2) n3 \rightarrow (w3, s3')$   
**apply** (*drule* (1) *evaln-max2*, *erule thin-rl*)  
**apply** (*fast intro!*: *le-maxI1* *le-maxI2*)  
**done**

**corollary** *evaln-max3E*:

$\llbracket G \vdash s1 -t1 \succ -n1 \rightarrow (w1, s1'); G \vdash s2 -t2 \succ -n2 \rightarrow (w2, s2'); G \vdash s3 -t3 \succ -n3 \rightarrow (w3, s3') \rrbracket$   
 $\llbracket G \vdash s1 -t1 \succ -\max (\max n1 n2) n3 \rightarrow (w1, s1');$   
 $G \vdash s2 -t2 \succ -\max (\max n1 n2) n3 \rightarrow (w2, s2');$   
 $G \vdash s3 -t3 \succ -\max (\max n1 n2) n3 \rightarrow (w3, s3') \rrbracket \Rightarrow P$   
 $\rrbracket \Rightarrow P$   
**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 le-maxI1*)

**also**

**have**  $\max n2 n3 \leq \max n1 (\max n2 n3)$

**by** (*rule le-maxI2*)

**finally**

**show** *?thesis* .

**qed**

**lemma** *le-max3I2*:  $(n3::nat) \leq \max n1 (\max n2 n3)$

**proof** -

**have**  $n3 \leq \max n2 n3$

**by** (*rule le-maxI2*)

**also**

**have**  $\max n2 n3 \leq \max n1 (\max n2 n3)$

**by** (*rule le-maxI2*)

**finally**

**show** *?thesis* .

**qed**

**declare**  $\llbracket \text{simproc del: wt-expr wt-var wt-exprs wt-stmt} \rrbracket$

**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**

$G \vdash (\text{Some } xc, s) -t \succ -n \rightarrow (\text{arbitrary3 } t, (\text{Some } xc, s))$

**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
   $G \vdash \text{Norm } s0 - e - \succ v - n \rightarrow s1$ 
by (iprover)
then have  $G \vdash \text{Norm } s0 - \text{Expr } e - n \rightarrow s1$ 
by (rule evaln.Expr)
then show ?case ..
next
case (Lab s0 c s1 l)
then obtain n where
   $G \vdash \text{Norm } s0 - c - n \rightarrow s1$ 
by (iprover)
then have  $G \vdash \text{Norm } s0 - l \cdot c - n \rightarrow \text{abupd } (\text{absorb } l) s1$ 
by (rule evaln.Lab)
then show ?case ..
next
case (Comp s0 c1 s1 c2 s2)
then obtain n1 n2 where
   $G \vdash \text{Norm } s0 - c1 - n1 \rightarrow s1$ 
   $G \vdash s1 - c2 - n2 \rightarrow s2$ 
by (iprover)
then have  $G \vdash \text{Norm } s0 - c1 ;; c2 - \max n1 n2 \rightarrow s2$ 
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
   $G \vdash \text{Norm } s0 - e - \succ b - n1 \rightarrow s1$ 
   $G \vdash s1 - (\text{if the-Bool } b \text{ then } c1 \text{ else } c2) - n2 \rightarrow s2$ 
by (iprover)
then have  $G \vdash \text{Norm } s0 - \text{If}(e) c1 \text{ Else } c2 - \max n1 n2 \rightarrow s2$ 
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
   $G \vdash \text{Norm } s0 - e - \succ b - n1 \rightarrow s1$ 
by (iprover)
moreover from Loop.hyps obtain n2 where
  if the-Bool b
    then ( $G \vdash s1 - c - n2 \rightarrow s2 \wedge$ 
       $G \vdash (\text{abupd } (\text{absorb } (\text{Cont } l)) s2) - l \cdot \text{While}(e) c - n2 \rightarrow s3$ )
    else  $s3 = s1$ 
by simp (iprover intro: evaln-nonstrict le-maxI1 le-maxI2)
ultimately
have  $G \vdash \text{Norm } s0 - l \cdot \text{While}(e) c - \max n1 n2 \rightarrow s3$ 
apply -
apply (rule evaln.Loop)
apply (iprover intro: evaln-nonstrict intro: le-maxI1)

apply (auto intro: evaln-nonstrict intro: le-maxI2)
done
then show ?case ..
next
case (Jmp s j)
fix n have  $G \vdash \text{Norm } s - \text{Jmp } j - n \rightarrow (\text{Some } (\text{Jump } j), s)$ 

```

```

    by (rule evaln.Jmp)
  then show ?case ..
next
  case (Throw s0 e a s1)
  then obtain n where
     $G \vdash \text{Norm } s0 \text{ } -e \rightarrow a -n \rightarrow s1$ 
    by (iprover)
  then have  $G \vdash \text{Norm } s0 \text{ } -\text{Throw } e -n \rightarrow \text{abupd } (\text{throw } a) s1$ 
    by (rule evaln.Throw)
  then show ?case ..
next
  case (Try s0 c1 s1 s2 catchC vn c2 s3)
  from Try.hyps obtain n1 where
     $G \vdash \text{Norm } s0 \text{ } -c1 -n1 \rightarrow s1$ 
    by (iprover)
  moreover
  note sxalloc =  $\langle G \vdash s1 \text{ } -\text{sxalloc} \rightarrow s2 \rangle$ 
  moreover
  from Try.hyps obtain n2 where
    if  $G, s2 \vdash \text{catch } \text{catchC} \text{ then } G \vdash \text{new-xcpt-var } vn \text{ } s2 \text{ } -c2 -n2 \rightarrow s3 \text{ else } s3 = s2$ 
    by fastsimp
  ultimately
  have  $G \vdash \text{Norm } s0 \text{ } -\text{Try } c1 \text{ } \text{Catch}(\text{catchC } vn) \text{ } c2 -\max n1 \text{ } n2 \rightarrow s3$ 
    by (auto intro!: evaln.Try le-maxI1 le-maxI2)
  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  $\text{abupd } (\text{abrupt-if } (x1 \neq \text{None}) x1) s2) \rangle$ 
  ultimately
  have
     $G \vdash \text{Norm } s0 \text{ } -c1 \text{ } \text{Finally } c2 -\max n1 \text{ } n2 \rightarrow s3$ 
    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  $\text{inited } C \text{ } (\text{globs } s0) \text{ then } s3 = \text{Norm } s0$ 
    else  $(G \vdash \text{Norm } (\text{init-class-obj } G \text{ } C \text{ } s0)$ 
       $-(\text{if } C = \text{Object then Skip else Init } (\text{super } c)) -n \rightarrow s1 \wedge$ 
       $G \vdash \text{set-lvars empty } s1 \text{ } -\text{init } c -n \rightarrow s2 \wedge$ 
       $s3 = \text{restore-lvars } s1 \text{ } s2)$ 
    by (auto intro: evaln-nonstrict le-maxI1 le-maxI2)
  ultimately have  $G \vdash \text{Norm } s0 \text{ } -\text{Init } C -n \rightarrow s3$ 
    by (rule evaln.Init)
  then show ?case ..
next
  case (NewC s0 C s1 a s2)
  then obtain n where
     $G \vdash \text{Norm } s0 \text{ } -\text{Init } C -n \rightarrow s1$ 
    by (iprover)
  with NewC

```

```

have  $G \vdash \text{Norm } s0 \text{ --NewC } C \text{ --}\succ \text{Addr } a \text{ --}n \rightarrow s2$ 
  by (iprover intro: evaln.NewC)
then show ?case ..
next
case (NewA s0 T s1 e i s2 a s3)
then obtain  $n1\ n2$  where
   $G \vdash \text{Norm } s0 \text{ --init-comp-ty } T \text{ --}n1 \rightarrow s1$ 
   $G \vdash s1 \text{ --}e \text{ --}\succ i \text{ --}n2 \rightarrow s2$ 
  by (iprover)
moreover
note  $\langle G \vdash \text{abupd } (\text{check-neg } i) \text{ } s2 \text{ --halloc Arr } T \text{ (the-Intg } i) \text{ --}\succ a \rightarrow s3 \rangle$ 
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 castT)
then obtain  $n$  where
   $G \vdash \text{Norm } s0 \text{ --}e \text{ --}\succ v \text{ --}n \rightarrow s1$ 
  by (iprover)
moreover
note  $\langle s2 = \text{abupd } (\text{raise-if } (\neg G, \text{snd } s1 \vdash v \text{ fits } \text{castT}) \text{ } \text{ClassCast}) \text{ } s1 \rangle$ 
ultimately
have  $G \vdash \text{Norm } s0 \text{ --Cast } \text{castT } e \text{ --}\succ v \text{ --}n \rightarrow s2$ 
  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  $\langle b = (v \neq \text{Null} \wedge G, \text{snd } s1 \vdash v \text{ fits } \text{RefT } T) \rangle$ 
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 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 binop e2 v2 s2)
then obtain  $n1\ n2$  where
   $G \vdash \text{Norm } s0 \text{ --}e1 \text{ --}\succ v1 \text{ --}n1 \rightarrow s1$ 
   $G \vdash s1 \text{ --(if need-second-arg binop } v1 \text{ then In1l } e2 \text{ else In1r Skip) --}\succ n2 \rightarrow (\text{In1 } v2, s2)$ 
  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  $G \vdash \text{Norm } s \multimap \text{Super} \multimap \text{val-this } s \multimap n \rightarrow \text{Norm } s$ 
    by (rule evaln.Super)
  then show ?case ..
next
  case (Acc s0 va v f s1)
  then obtain n where
     $G \vdash \text{Norm } s0 \multimap \text{va} = \multimap (v, f) \multimap n \rightarrow s1$ 
    by (iprover)
  then
  have  $G \vdash \text{Norm } s0 \multimap \text{Acc } \text{va} \multimap v \multimap n \rightarrow s1$ 
    by (rule evaln.Acc)
  then show ?case ..
next
  case (Ass s0 var w f s1 e v s2)
  then obtain n1 n2 where
     $G \vdash \text{Norm } s0 \multimap \text{var} = \multimap (w, f) \multimap n1 \rightarrow s1$ 
     $G \vdash s1 \multimap e \multimap v \multimap n2 \rightarrow s2$ 
    by (iprover)
  then
  have  $G \vdash \text{Norm } s0 \multimap \text{var} := e \multimap v \multimap \text{max } n1 \ n2 \rightarrow \text{assign } f \ v \ s2$ 
    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
     $G \vdash \text{Norm } s0 \multimap e0 \multimap b \multimap n1 \rightarrow s1$ 
     $G \vdash s1 \multimap (\text{if the-Bool } b \text{ then } e1 \text{ else } e2) \multimap v \multimap n2 \rightarrow s2$ 
    by (iprover)
  then
  have  $G \vdash \text{Norm } s0 \multimap e0 \ ? \ e1 : e2 \multimap v \multimap \text{max } n1 \ n2 \rightarrow s2$ 
    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
     $G \vdash \text{Norm } s0 \multimap e \multimap a' \multimap n1 \rightarrow s1$ 
     $G \vdash s1 \multimap \text{args} := \multimap \text{vs} \multimap n2 \rightarrow s2$ 
    by iprover
  moreover
  note  $\langle \text{invDeclC} = \text{invocation-declclass } G \text{ mode } (\text{store } s2) \ a' \ \text{statT} \ \langle \text{name} = \text{mn}, \text{parTs} = \text{pTs}' \rangle \rangle$ 
  moreover
  note  $\langle s3 = \text{init-lvars } G \ \text{invDeclC} \ \langle \text{name} = \text{mn}, \text{parTs} = \text{pTs}' \rangle \ \text{mode } a' \ \text{vs } s2 \rangle$ 
  moreover
  note  $\langle s3' = \text{check-method-access } G \ \text{accC}' \ \text{statT} \ \text{mode } \langle \text{name} = \text{mn}, \text{parTs} = \text{pTs}' \rangle \ a' \ s3 \rangle$ 
  moreover
  from Call.hyps
  obtain m where
     $G \vdash s3' \multimap \text{Methd } \text{invDeclC} \ \langle \text{name} = \text{mn}, \text{parTs} = \text{pTs}' \rangle \multimap v \multimap m \rightarrow s4$ 
    by iprover
  ultimately
  have  $G \vdash \text{Norm } s0 \multimap \{ \text{accC}', \text{statT}, \text{mode} \} e \cdot \text{mn} (\{ \text{pTs}' \} \text{args}) \multimap v \multimap \text{max } n1 \ (\text{max } n2 \ m) \rightarrow$ 
     $(\text{set-lvars } (\text{locals } (\text{store } s2))) \ s4$ 
    by (auto intro!: evaln.Call le-maxI1 le-max3I1 le-max3I2)
  thus ?case ..

```

```

next
  case (Methd s0 D sig v s1)
  then obtain n where
     $G \vdash \text{Norm } s0 \text{ --body } G D \text{ sig} \multimap v \text{ --} n \rightarrow s1$ 
    by iprover
  then have  $G \vdash \text{Norm } s0 \text{ --Methd } D \text{ sig} \multimap v \text{ --Suc } n \rightarrow s1$ 
    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  $\langle s3 = (\text{if } \exists l. \text{fst } s2 = \text{Some } (\text{Jump } (\text{Break } l)) \vee$ 
     $\text{fst } s2 = \text{Some } (\text{Jump } (\text{Cont } l))$ 
    then  $\text{abupd } (\lambda x. \text{Some } (\text{Error CrossMethodJump})) s2$ 
    else  $s2 \rangle$ 
  ultimately
  have
     $G \vdash \text{Norm } s0 \text{ --Body } D c \multimap \text{the } (\text{locals } (\text{store } s2) \text{ Result}) \text{ --max } n1 n2$ 
     $\rightarrow \text{abupd } (\text{absorb Ret}) s3$ 
    by (iprover intro: evaln.Body dest: evaln-max2)
  then show ?case ..
next
  case (LVar s vn )
  obtain n where
     $G \vdash \text{Norm } s \text{ --LVar } vn \multimap \text{lvar } vn s \text{ --} n \rightarrow \text{Norm } s$ 
    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
     $G \vdash \text{Norm } s0 \text{ --Init } \text{statDeclC} \text{ --} n1 \rightarrow s1$ 
     $G \vdash s1 \text{ --e--} a \text{ --} n2 \rightarrow s2$ 
    by iprover
  moreover
  note  $\langle s3 = \text{check-field-access } G \text{ accC statDeclC fn stat } a s2' \rangle$ 
  and  $\langle (v, s2') = \text{fvar statDeclC stat fn } a s2 \rangle$ 
  ultimately
  have  $G \vdash \text{Norm } s0 \text{ --}\{accC, \text{statDeclC}, \text{stat}\} e.. \text{fn} \multimap v \text{ --max } n1 n2 \rightarrow s3$ 
    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
     $G \vdash \text{Norm } s0 \text{ --e1--} a \text{ --} n1 \rightarrow s1$ 
     $G \vdash s1 \text{ --e2--} i \text{ --} n2 \rightarrow s2$ 
    by iprover
  moreover
  note  $\langle (v, s2') = \text{avar } G i a s2 \rangle$ 
  ultimately
  have  $G \vdash \text{Norm } s0 \text{ --e1.[e2]--} v \text{ --max } n1 n2 \rightarrow s2'$ 
    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
     $G \vdash \text{Norm } s0 \text{ } -e \text{ } \rightarrow v \text{ } -n1 \rightarrow s1$ 
     $G \vdash s1 \text{ } -es \text{ } \rightarrow vs \text{ } -n2 \rightarrow s2$ 
    by iprover
  then
    have  $G \vdash \text{Norm } s0 \text{ } -e \text{ } \# es \text{ } \rightarrow v \text{ } \# vs \text{ } -\max n1 n2 \rightarrow s2$ 
    by (blast intro!: evaln.Cons dest: evaln-max2)
  then show ?case ..
qed

end

```



# Chapter 21

## Trans

**theory** *Trans* **imports** *Evaln* **begin**

**constdefs** *groundVar*:: *var*  $\Rightarrow$  *bool*  
*groundVar* *v*  $\equiv$  (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* [consumes 1, case-names *LVar FVar AVar*]:

**assumes** *ground*: *groundVar* *v* **and**  
    *LVar*:  $\bigwedge$  *ln*.  $\llbracket v = \text{LVar } ln \rrbracket \Longrightarrow P$  **and**  
    *FVar*:  $\bigwedge$  *accC statDeclC stat a fn*.  
       $\llbracket v = \{accC, statDeclC, stat\} (Lit\ a) .. fn \rrbracket \Longrightarrow P$  **and**  
    *AVar*:  $\bigwedge$  *a i*.  $\llbracket v = (Lit\ a) .. [Lit\ i] \rrbracket \Longrightarrow P$

**shows** *P*

**proof** –

**from** *ground* *LVar FVar AVar*

**show** ?thesis

**apply** (cases *v*)

**apply** (simp add: *groundVar-def*)

**apply** (simp add: *groundVar-def*, blast)

**apply** (simp add: *groundVar-def*, blast)

**apply** (simp add: *groundVar-def*)

**done**

**qed**

**constdefs** *groundExprs*:: *expr* *list*  $\Rightarrow$  *bool*  
*groundExprs* *es*  $\equiv$  *list-all* ( $\lambda$  *e*.  $\exists$  *v*. *e*=*Lit* *v*) *es*

**consts** *the-val*:: *expr*  $\Rightarrow$  *val*

**primrec**

*the-val* (*Lit* *v*) = *v*

**consts** *the-var*:: *prog*  $\Rightarrow$  *state*  $\Rightarrow$  *var*  $\Rightarrow$  (*vvar*  $\times$  *state*)

**primrec**

*the-var* *G* *s* (*LVar* *ln*) = (*lvar* *ln* (*store* *s*), *s*)

*the-var-FVar-def*:

*the-var* *G* *s* ({*accC*,*statDeclC*,*stat*}*a*..*fn*) = *fvar* *statDeclC* *stat* *fn* (*the-val* *a*) *s*

*the-var-AVar-def*:

*the-var* *G* *s* (*a*..*i*) = *avar* *G* (*the-val* *i*) (*the-val* *a*) *s*

**lemma** *the-var-FVar-simp*[simp]:  
*the-var*  $G\ s\ (\{\text{acc}C, \text{statDecl}C, \text{stat}\}(\text{Lit } a)..fn) = \text{fvar statDecl}C\ \text{stat}\ fn\ a\ s$   
**by** (*simp*)  
**declare** *the-var-FVar-def* [simp del]

**lemma** *the-var-AVar-simp*:  
*the-var*  $G\ s\ ((\text{Lit } a).[\text{Lit } i]) = \text{avar } G\ i\ a\ s$   
**by** (*simp*)  
**declare** *the-var-AVar-def* [simp del]

**syntax** (*xsymbols*)  
 $\text{Ref} :: \text{loc} \Rightarrow \text{expr}$   
 $\text{SKIP} :: \text{expr}$

**translations**  
 $\text{Ref } a == \text{Lit } (\text{Addr } a)$   
 $\text{SKIP} == \text{Lit } \text{Unit}$

**inductive**  
 $\text{step} :: [\text{prog}, \text{term} \times \text{state}, \text{term} \times \text{state}] \Rightarrow \text{bool} \ (\vdash - \mapsto 1 \text{ } [61, 82, 82] \ 81)$   
**for**  $G :: \text{prog}$   
**where**

*Abrupt*:  
 $\llbracket \forall v. t \neq \langle \text{Lit } v \rangle; \forall t. t \neq \langle l \cdot \text{Skip} \rangle; \forall C\ vn\ c. t \neq \langle \text{Try Skip Catch}(C\ vn)\ c \rangle; \forall x\ c. t \neq \langle \text{Skip Finally } c \rangle \wedge xc \neq \text{Xcpt } x; \forall a\ c. t \neq \langle \text{FinA } a\ c \rangle \rrbracket$   
 $\implies$   
 $G \vdash (t, \text{Some } xc, s) \mapsto 1 \ (\langle \text{Lit arbitrary} \rangle, \text{Some } xc, s)$

| *InsInitE*:  $\llbracket G \vdash (\langle c \rangle, \text{Norm } s) \mapsto 1 \ (\langle c^\wedge, s' \rangle) \rrbracket$   
 $\implies$   
 $G \vdash (\langle \text{InsInitE } c\ e \rangle, \text{Norm } s) \mapsto 1 \ (\langle \text{InsInitE } c'\ e' \rangle, s')$

| *NewC*:  $G \vdash (\langle \text{NewC } C \rangle, \text{Norm } s) \mapsto 1 \ (\langle \text{InsInitE } (\text{Init } C) (\text{NewC } C) \rangle, \text{Norm } s)$   
| *NewCInitd*:  $\llbracket G \vdash \text{Norm } s \text{ } \text{--hallocc } (C\text{Inst } C) \succ a \rightarrow s' \rrbracket$   
 $\implies$   
 $G \vdash (\langle \text{InsInitE Skip } (\text{NewC } C) \rangle, \text{Norm } s) \mapsto 1 \ (\langle \text{Ref } a \rangle, s')$

| *NewA*:  
 $G \vdash (\langle \text{New } T[e] \rangle, \text{Norm } s) \mapsto 1 \ (\langle \text{InsInitE } (\text{init-comp-ty } T) (\text{New } T[e]) \rangle, \text{Norm } s)$   
| *InsInitNewAIdx*:  
 $\llbracket G \vdash (\langle e \rangle, \text{Norm } s) \mapsto 1 \ (\langle e^\wedge, s' \rangle) \rrbracket$   
 $\implies$   
 $G \vdash (\langle \text{InsInitE Skip } (\text{New } T[e]) \rangle, \text{Norm } s) \mapsto 1 \ (\langle \text{InsInitE Skip } (\text{New } T[e']) \rangle, s')$

- | *InsInitNewA*:  

$$\llbracket G \vdash \text{abupd } (\text{check-neg } i) \text{ (Norm } s) - \text{halloc } (\text{Arr } T \text{ (the-Intg } i)) \succ a \rightarrow s' \rrbracket$$

$$\implies$$

$$G \vdash (\langle \text{InsInitE Skip (New } T[\text{Lit } i]) \rangle, \text{Norm } s) \mapsto 1 \text{ } (\langle \text{Ref } a \rangle, s')$$
- | *CastE*:  

$$\llbracket G \vdash (\langle e \rangle, \text{Norm } s) \mapsto 1 \text{ } (\langle e \rangle^\wedge, s') \rrbracket$$

$$\implies$$

$$G \vdash (\langle \text{Cast } T \text{ } e \rangle, \text{None}, s) \mapsto 1 \text{ } (\langle \text{Cast } T \text{ } e \rangle^\wedge, s')$$
- | *Cast*:  

$$\llbracket s' = \text{abupd } (\text{raise-if } (\neg G, s \vdash v \text{ fits } T) \text{ ClassCast}) \text{ (Norm } s) \rrbracket$$

$$\implies$$

$$G \vdash (\langle \text{Cast } T \text{ (Lit } v) \rangle, \text{Norm } s) \mapsto 1 \text{ } (\langle \text{Lit } v \rangle, s')$$
- | *InstE*:  $\llbracket G \vdash (\langle e \rangle, \text{Norm } s) \mapsto 1 \text{ } (\langle e'::\text{expr} \rangle, s') \rrbracket$   

$$\implies$$

$$G \vdash (\langle e \text{ InstOf } T \rangle, \text{Norm } s) \mapsto 1 \text{ } (\langle e \rangle^\wedge, s')$$
- | *Inst*:  $\llbracket b = (v \neq \text{Null} \wedge G, s \vdash v \text{ fits RefT } T) \rrbracket$   

$$\implies$$

$$G \vdash (\langle (\text{Lit } v) \text{ InstOf } T \rangle, \text{Norm } s) \mapsto 1 \text{ } (\langle \text{Lit (Bool } b) \rangle, s')$$
- | *UnOpE*:  $\llbracket G \vdash (\langle e \rangle, \text{Norm } s) \mapsto 1 \text{ } (\langle e \rangle^\wedge, s') \rrbracket$   

$$\implies$$

$$G \vdash (\langle \text{UnOp unop } e \rangle, \text{Norm } s) \mapsto 1 \text{ } (\langle \text{UnOp unop } e \rangle^\wedge, s')$$
- | *UnOp*:  $G \vdash (\langle \text{UnOp unop (Lit } v) \rangle, \text{Norm } s) \mapsto 1 \text{ } (\langle \text{Lit (eval-unop unop } v) \rangle, \text{Norm } s)$
- | *BinOpE1*:  $\llbracket G \vdash (\langle e1 \rangle, \text{Norm } s) \mapsto 1 \text{ } (\langle e1 \rangle^\wedge, s') \rrbracket$   

$$\implies$$

$$G \vdash (\langle \text{BinOp binop } e1 \text{ } e2 \rangle, \text{Norm } s) \mapsto 1 \text{ } (\langle \text{BinOp binop } e1^\wedge \text{ } e2 \rangle, s')$$
- | *BinOpE2*:  $\llbracket \text{need-second-arg binop } v1; G \vdash (\langle e2 \rangle, \text{Norm } s) \mapsto 1 \text{ } (\langle e2 \rangle^\wedge, s') \rrbracket$   

$$\implies$$

$$G \vdash (\langle \text{BinOp binop (Lit } v1) \text{ } e2 \rangle, \text{Norm } s)$$

$$\mapsto 1 \text{ } (\langle \text{BinOp binop (Lit } v1) \text{ } e2 \rangle^\wedge, s')$$
- | *BinOpTerm*:  $\llbracket \neg \text{need-second-arg binop } v1 \rrbracket$   

$$\implies$$

$$G \vdash (\langle \text{BinOp binop (Lit } v1) \text{ } e2 \rangle, \text{Norm } s)$$

$$\mapsto 1 \text{ } (\langle \text{Lit } v1 \rangle, \text{Norm } s)$$
- | *BinOp*:  $G \vdash (\langle \text{BinOp binop (Lit } v1) \text{ (Lit } v2) \rangle, \text{Norm } s)$   

$$\mapsto 1 \text{ } (\langle \text{Lit (eval-binop binop } v1 \text{ } v2) \rangle, \text{Norm } s)$$
- | *Super*:  $G \vdash (\langle \text{Super} \rangle, \text{Norm } s) \mapsto 1 \text{ } (\langle \text{Lit (val-this } s) \rangle, \text{Norm } s)$
- | *AccVA*:  $\llbracket G \vdash (\langle va \rangle, \text{Norm } s) \mapsto 1 \text{ } (\langle va \rangle^\wedge, s') \rrbracket$   

$$\implies$$

$$G \vdash (\langle \text{Acc } va \rangle, \text{Norm } s) \mapsto 1 \text{ } (\langle \text{Acc } va \rangle^\wedge, s')$$
- | *Acc*:  $\llbracket \text{groundVar } va; ((v, vf), s') = \text{the-var } G \text{ (Norm } s) \text{ } va \rrbracket$   

$$\implies$$

$$G \vdash (\langle \text{Acc } va \rangle, \text{Norm } s) \mapsto 1 \text{ } (\langle \text{Lit } v \rangle, s')$$
- | *AssVA*:  $\llbracket G \vdash (\langle va \rangle, \text{Norm } s) \mapsto 1 \text{ } (\langle va \rangle^\wedge, s') \rrbracket$   

$$\implies$$

$$\begin{array}{l}
G \vdash (\langle va := e \rangle, Norm\ s) \mapsto 1\ (\langle va' := e \rangle, s') \\
| \text{ AssE: } \llbracket \text{groundVar } va; G \vdash (\langle e \rangle, Norm\ s) \mapsto 1\ (\langle e' \rangle, s') \rrbracket \\
\quad \Rightarrow \\
G \vdash (\langle va := e \rangle, Norm\ s) \mapsto 1\ (\langle va := e' \rangle, s') \\
| \text{ Ass: } \llbracket \text{groundVar } va; ((w, f), s') = \text{the-var } G\ (Norm\ s)\ va \rrbracket \\
\quad \Rightarrow \\
G \vdash (\langle va := (Lit\ v) \rangle, Norm\ s) \mapsto 1\ (\langle Lit\ v \rangle, \text{assign } f\ v\ s') \\
\\
| \text{ CondC: } \llbracket G \vdash (\langle e0 \rangle, Norm\ s) \mapsto 1\ (\langle e0' \rangle, s') \rrbracket \\
\quad \Rightarrow \\
G \vdash (\langle e0? e1:e2 \rangle, Norm\ s) \mapsto 1\ (\langle e0'? e1:e2 \rangle, s') \\
| \text{ Cond: } G \vdash (\langle Lit\ b? e1:e2 \rangle, Norm\ s) \mapsto 1\ (\langle \text{if the-Bool } b \text{ then } e1 \text{ else } e2 \rangle, Norm\ s) \\
\\
| \text{ CallTarget: } \llbracket G \vdash (\langle e \rangle, Norm\ s) \mapsto 1\ (\langle e' \rangle, s') \rrbracket \\
\quad \Rightarrow \\
G \vdash (\langle \{accC, statT, mode\} e \cdot mn(\{pTs\} args) \rangle, Norm\ s) \\
\quad \mapsto 1\ (\langle \{accC, statT, mode\} e' \cdot mn(\{pTs\} args) \rangle, s') \\
| \text{ CallArgs: } \llbracket G \vdash (\langle args \rangle, Norm\ s) \mapsto 1\ (\langle args' \rangle, s') \rrbracket \\
\quad \Rightarrow \\
G \vdash (\langle \{accC, statT, mode\} Lit\ a \cdot mn(\{pTs\} args) \rangle, Norm\ s) \\
\quad \mapsto 1\ (\langle \{accC, statT, mode\} Lit\ a \cdot mn(\{pTs\} args') \rangle, s') \\
| \text{ Call: } \llbracket \text{groundExprs } args; vs = \text{map the-val } args; \\
D = \text{invocation-declclass } G\ mode\ s\ a\ statT\ (\llbracket name=mn, parTs=pTs \rrbracket); \\
s' = \text{init-lvars } G\ D\ (\llbracket name=mn, parTs=pTs \rrbracket)\ mode\ a'\ vs\ (Norm\ s) \rrbracket \\
\quad \Rightarrow \\
G \vdash (\langle \{accC, statT, mode\} Lit\ a \cdot mn(\{pTs\} args) \rangle, Norm\ s) \\
\quad \mapsto 1\ (\langle \text{Callee } (locals\ s)\ (\text{Methd } D\ (\llbracket name=mn, parTs=pTs \rrbracket)) \rangle, s') \\
\\
| \text{ Callee: } \llbracket G \vdash (\langle e \rangle, Norm\ s) \mapsto 1\ (\langle e'::expr \rangle, s') \rrbracket \\
\quad \Rightarrow \\
G \vdash (\langle \text{Callee lcls-caller } e \rangle, Norm\ s) \mapsto 1\ (\langle e' \rangle, s') \\
\\
| \text{ CalleeRet: } G \vdash (\langle \text{Callee lcls-caller } (Lit\ v) \rangle, Norm\ s) \\
\quad \mapsto 1\ (\langle Lit\ v \rangle, (\text{set-lvars lcls-caller } (Norm\ s))) \\
\\
| \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 \\
\quad \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') \\
| \text{ InsInitBodyRet: } \\
G \vdash (\langle \text{InsInitE Skip } (\text{Body } D\ Skip) \rangle, Norm\ s) \\
\quad \mapsto 1\ (\langle Lit\ (\text{the } ((locals\ s)\ Result)) \rangle, \text{abupd } (\text{absorb Ret})\ (Norm\ s)) \\
\\
| \text{ FVar: } \llbracket \neg\ \text{inited statDeclC } (globs\ s) \rrbracket \\
\quad \Rightarrow \\
G \vdash (\langle \{accC, statDeclC, stat\} e..fn \rangle, Norm\ s) \\
\quad \mapsto 1\ (\langle \text{InsInitV } (Init\ statDeclC)\ (\{accC, statDeclC, stat\} e..fn) \rangle, Norm\ s) \\
| \text{ InsInitFVarE: } \\
\llbracket G \vdash (\langle e \rangle, Norm\ s) \mapsto 1\ (\langle e' \rangle, s') \rrbracket \\
\quad \Rightarrow \\
G \vdash (\langle \text{InsInitV Skip } (\{accC, statDeclC, stat\} e..fn) \rangle, Norm\ s) \\
\quad \mapsto 1\ (\langle \text{InsInitV Skip } (\{accC, statDeclC, stat\} e'..fn) \rangle, s')
\end{array}$$

| *InsInitFVar*:

$$\begin{aligned} & G \vdash (\langle \text{InsInitV Skip } (\{accC, statDeclC, stat\} Lit\ a..fn) \rangle, Norm\ s) \\ & \mapsto 1\ (\langle \{accC, statDeclC, stat\} Lit\ a..fn \rangle, Norm\ s) \end{aligned}$$

— Notice, that we do not have literal values for *vars*. The rules for accessing variables (*Acc*) and assigning to variables (*Ass*), test this with the predicate *groundVar*. After initialisation is done and the *FVar* is evaluated, we can't just throw away the *InsInitFVar* term and return a literal value, as in the cases of *New* or *NewC*. Instead we just return the evaluated *FVar* and test for initialisation in the rule *FVar*.

$$\begin{aligned} | \text{AVarE1}: & \llbracket G \vdash (\langle e1 \rangle, Norm\ s) \mapsto 1\ (\langle e1' \rangle, s') \rrbracket \\ & \implies \\ & G \vdash (\langle e1.[e2] \rangle, Norm\ s) \mapsto 1\ (\langle e1'.[e2] \rangle, s') \end{aligned}$$

$$\begin{aligned} | \text{AVarE2}: & G \vdash (\langle e2 \rangle, Norm\ s) \mapsto 1\ (\langle e2' \rangle, s') \\ & \implies \\ & G \vdash (\langle Lit\ a.[e2] \rangle, Norm\ s) \mapsto 1\ (\langle Lit\ a.[e2'] \rangle, s') \end{aligned}$$

— *Nil* is fully evaluated

$$\begin{aligned} | \text{ConsHd}: & \llbracket G \vdash (\langle e::expr \rangle, Norm\ s) \mapsto 1\ (\langle e'::expr \rangle, s') \rrbracket \\ & \implies \\ & G \vdash (\langle e\#es \rangle, Norm\ s) \mapsto 1\ (\langle e'\#es \rangle, s') \end{aligned}$$

$$\begin{aligned} | \text{ConsTl}: & \llbracket G \vdash (\langle es \rangle, Norm\ s) \mapsto 1\ (\langle es' \rangle, s') \rrbracket \\ & \implies \\ & G \vdash (\langle (Lit\ v)\#es \rangle, Norm\ s) \mapsto 1\ (\langle (Lit\ v)\#es' \rangle, s') \end{aligned}$$

$$| \text{Skip}: G \vdash (\langle Skip \rangle, Norm\ s) \mapsto 1\ (\langle SKIP \rangle, Norm\ s)$$

$$\begin{aligned} | \text{ExprE}: & \llbracket G \vdash (\langle e \rangle, Norm\ s) \mapsto 1\ (\langle e' \rangle, s') \rrbracket \\ & \implies \\ & G \vdash (\langle Expr\ e \rangle, Norm\ s) \mapsto 1\ (\langle Expr\ e' \rangle, s') \\ | \text{Expr}: & G \vdash (\langle Expr\ (Lit\ v) \rangle, Norm\ s) \mapsto 1\ (\langle Skip \rangle, Norm\ s) \end{aligned}$$

$$\begin{aligned} | \text{LabC}: & \llbracket G \vdash (\langle c \rangle, Norm\ s) \mapsto 1\ (\langle c' \rangle, s') \rrbracket \\ & \implies \\ & G \vdash (\langle l \cdot c \rangle, Norm\ s) \mapsto 1\ (\langle l \cdot c' \rangle, s') \\ | \text{Lab}: & G \vdash (\langle l \cdot Skip \rangle, s) \mapsto 1\ (\langle Skip \rangle, abupd\ (absorb\ l)\ s) \end{aligned}$$

$$\begin{aligned} | \text{CompC1}: & \llbracket G \vdash (\langle c1 \rangle, Norm\ s) \mapsto 1\ (\langle c1' \rangle, s') \rrbracket \\ & \implies \\ & G \vdash (\langle c1;; c2 \rangle, Norm\ s) \mapsto 1\ (\langle c1';; c2 \rangle, s') \end{aligned}$$

$$| \text{Comp}: G \vdash (\langle Skip;; c2 \rangle, Norm\ s) \mapsto 1\ (\langle c2 \rangle, Norm\ s)$$

$$\begin{aligned} | \text{IfE}: & \llbracket G \vdash (\langle e \rangle, Norm\ s) \mapsto 1\ (\langle e' \rangle, s') \rrbracket \\ & \implies \\ & G \vdash (\langle If(e)\ s1\ Else\ s2 \rangle, Norm\ s) \mapsto 1\ (\langle If(e')\ s1\ Else\ s2 \rangle, s') \\ | \text{If}: & G \vdash (\langle If(Lit\ v)\ s1\ Else\ s2 \rangle, Norm\ s) \end{aligned}$$

- $\mapsto 1 \ (\langle \text{if the-Bool } v \text{ then } s1 \text{ else } s2 \rangle, \text{Norm } s)$
- | *Loop*:  $G \vdash (\langle l \cdot \text{While}(e) \ c \rangle, \text{Norm } s) \mapsto 1 \ (\langle \text{If}(e) \ (\text{Cont } l \cdot c;; \ l \cdot \text{While}(e) \ c) \ \text{Else Skip} \rangle, \text{Norm } s)$
- | *Jmp*:  $G \vdash (\langle \text{Jmp } j \rangle, \text{Norm } s) \mapsto 1 \ (\langle \text{Skip} \rangle, (\text{Some } (\text{Jump } j), s))$
- | *ThrowE*:  $\llbracket G \vdash (\langle e \rangle, \text{Norm } s) \mapsto 1 \ (\langle e' \rangle, s') \rrbracket \Rightarrow G \vdash (\langle \text{Throw } e \rangle, \text{Norm } s) \mapsto 1 \ (\langle \text{Throw } e' \rangle, s')$
- | *Throw*:  $G \vdash (\langle \text{Throw } (\text{Lit } a) \rangle, \text{Norm } s) \mapsto 1 \ (\langle \text{Skip} \rangle, \text{abupd } (\text{throw } a) \ (\text{Norm } s))$
- | *TryC1*:  $\llbracket G \vdash (\langle c1 \rangle, \text{Norm } s) \mapsto 1 \ (\langle c1' \rangle, s') \rrbracket \Rightarrow G \vdash (\langle \text{Try } c1 \ \text{Catch}(C \ vn) \ c2 \rangle, \text{Norm } s) \mapsto 1 \ (\langle \text{Try } c1' \ \text{Catch}(C \ vn) \ c2 \rangle, s')$
- | *Try*:  $\llbracket G \vdash s \text{--}sxalloc \rightarrow s' \rrbracket \Rightarrow G \vdash (\langle \text{Try Skip Catch}(C \ vn) \ c2 \rangle, s) \mapsto 1 \ (\text{if } G, s \vdash \text{catch } C \text{ then } (\langle c2 \rangle, \text{new-}xcpt\text{-var } vn \ s') \text{ else } (\langle \text{Skip} \rangle, s'))$
- | *FinC1*:  $\llbracket G \vdash (\langle c1 \rangle, \text{Norm } s) \mapsto 1 \ (\langle c1' \rangle, s') \rrbracket \Rightarrow G \vdash (\langle c1 \ \text{Finally } c2 \rangle, \text{Norm } s) \mapsto 1 \ (\langle c1' \ \text{Finally } c2 \rangle, s')$
- | *Fin*:  $G \vdash (\langle \text{Skip Finally } c2 \rangle, (a, s)) \mapsto 1 \ (\langle \text{FinA } a \ c2 \rangle, \text{Norm } s)$
- | *FinAC*:  $\llbracket G \vdash (\langle c \rangle, s) \mapsto 1 \ (\langle c' \rangle, s') \rrbracket \Rightarrow G \vdash (\langle \text{FinA } a \ c \rangle, s) \mapsto 1 \ (\langle \text{FinA } a \ c' \rangle, s')$
- | *FinA*:  $G \vdash (\langle \text{FinA } a \ \text{Skip} \rangle, s) \mapsto 1 \ (\langle \text{Skip} \rangle, \text{abupd } (\text{abrupt-if } (a \neq \text{None}) \ a) \ s)$
- | *InitI1*:  $\llbracket \text{inited } C \ (\text{globs } s) \rrbracket \Rightarrow G \vdash (\langle \text{Init } C \rangle, \text{Norm } s) \mapsto 1 \ (\langle \text{Skip} \rangle, \text{Norm } s)$
- | *Init*:  $\llbracket \text{the } (\text{class } G \ C) = c; \neg \text{inited } C \ (\text{globs } s) \rrbracket \Rightarrow G \vdash (\langle \text{Init } C \rangle, \text{Norm } s) \mapsto 1 \ (\langle \text{if } C = \text{Object then Skip else } (\text{Init } (\text{super } c)));; \text{Expr } (\text{Callee } (\text{locals } s) \ (\text{InsInitE } (\text{init } c) \ \text{SKIP})) \rangle, \text{Norm } (\text{init-class-obj } G \ C \ s))$
- *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

*stepn*::  $[\text{prog}, \text{term} \times \text{state}, \text{nat}, \text{term} \times \text{state}] \Rightarrow \text{bool} \ (\vdash - \mapsto - [61, 82, 82] \ 81)$   
**where**  $G \vdash p \mapsto n \ p' \equiv (p, p') \in \{(x, y). \text{step } G \ x \ y\}^n$

### abbreviation

*steptr*::  $[\text{prog}, \text{term} \times \text{state}, \text{term} \times \text{state}] \Rightarrow \text{bool} \ (\vdash - \mapsto^* - [61, 82, 82] \ 81)$   
**where**  $G \vdash p \mapsto^* p' \equiv (p, p') \in \{(x, y). \text{step } G \ x \ y\}^*$

**lemma** *rtranc1-imp-rel-pow*:  $p \in R^{\wedge*} \Rightarrow \exists n. p \in R^{\wedge n}$

```

proof –
  assume  $p \in R^*$ 
  moreover obtain  $x\ y$  where  $p: p = (x,y)$  by (cases p)
  ultimately have  $(x,y) \in R^*$  by hypsubst
  hence  $\exists n. (x,y) \in R^n$ 
  proof induct
    fix  $a$  have  $(a,a) \in R^0$  by simp
    thus  $\exists n. (a,a) \in R^n$  ..
  next
    fix  $a\ b\ c$  assume  $\exists n. (a,b) \in R^n$ 
    then obtain  $n$  where  $(a,b) \in R^n$  ..
    moreover assume  $(b,c) \in R$ 
    ultimately have  $(a,c) \in R^{(Suc\ n)}$  by auto
    thus  $\exists n. (a,c) \in R^n$  ..
  qed
  with  $p$  show ?thesis by hypsubst
qed

```

**end**





## Chapter 22

### AxSem

## 50 Axiomatic semantics of Java expressions and statements (see also Eval.thy)

**theory** *AxSem* **imports** *Evaln TypeSafe* **begin**

design issues:

- a strong version of validity for triples with premises, namely one that takes the recursive depth needed to complete execution, enables correctness proof
- auxiliary variables are handled first-class (-i Thomas Kleymann)
- expressions not flattened to elementary assignments (as usual for axiomatic semantics) but treated first-class =i explicit result value handling
- intermediate values not on triple, but on assertion level (with result entry)
- multiple results with semantical substitution mechanism not requiring a stack
- because of dynamic method binding, terms need to be dependent on state. this is also useful for conditional expressions and statements
- result values in triples exactly as in eval relation (also for xcpt states)
- validity: additional assumption of state conformance and well-typedness, which is required for soundness and thus rule hazard required of completeness

restrictions:

- all triples in a derivation are of the same type (due to weak polymorphism)

**types** *res = vals* — result entry

**syntax**

*Val* :: *val*  $\Rightarrow$  *res*

*Var* :: *var*  $\Rightarrow$  *res*

*Vals* :: *val list*  $\Rightarrow$  *res*

**translations**

*Val* *x*  $\Rightarrow$  (*In1* *x*)

*Var* *x*  $\Rightarrow$  (*In2* *x*)

*Vals* *x*  $\Rightarrow$  (*In3* *x*)

**syntax**

-*Val* :: [*pttrn*]  $\Rightarrow$  *pttrn* (*Val*:- [951] 950)

-*Var* :: [*pttrn*]  $\Rightarrow$  *pttrn* (*Var*:- [951] 950)

-*Vals* :: [*pttrn*]  $\Rightarrow$  *pttrn* (*Vals*:- [951] 950)

**translations**

$\lambda Val:v . b == (\lambda v. b) \circ the-In1$

$\lambda Var:v . b == (\lambda v. b) \circ the-In2$

$\lambda Vals:v. b == (\lambda v. b) \circ the-In3$

— relation on result values, state and auxiliary variables

**types** '*a assn* = *res*  $\Rightarrow$  *state*  $\Rightarrow$  '*a*  $\Rightarrow$  *bool*

**translations**

*res*  $\leq$  (*type*) *AxSem.res*

*a assn*  $\leq$  (*type*) *vals*  $\Rightarrow$  *state*  $\Rightarrow$  *a*  $\Rightarrow$  *bool*

**constdefs**

*assn-imp* :: '*a assn*  $\Rightarrow$  '*a assn*  $\Rightarrow$  *bool* (infixr  $\Rightarrow$  25)

$P \Rightarrow Q \equiv \forall Y s Z. P Y s Z \longrightarrow Q Y s Z$

```

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

### 51 peek-and

```

constdefs
  peek-and :: 'a assn  $\Rightarrow$  (state  $\Rightarrow$  bool)  $\Rightarrow$  'a assn (infixl  $\wedge$ . 13)
   $P \wedge. p \equiv \lambda Y\ s\ Z. P\ Y\ s\ Z \wedge p\ s$ 

```

```

lemma peek-and-def2 [simp]:  $peek\text{-}and\ P\ p\ Y\ s = (\lambda Z. (P\ Y\ s\ Z \wedge p\ s))$ 
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]:  $peek\text{-}and\ (peek\text{-}and\ P\ p)\ p = peek\text{-}and\ P\ p$ 
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

```

```

syntax
  Normal :: 'a assn  $\Rightarrow$  'a assn

```

```

translations
  Normal  $P == P \wedge. normal$ 

```

```

lemma peek-and-Normal [simp]:  $peek\text{-}and\ (Normal\ P)\ p = Normal\ (peek\text{-}and\ P\ p)$ 
apply (rule ext)
apply (rule ext)
apply (rule ext)
apply auto
done

```

### 52 assn-supd

```

constdefs
  assn-supd :: 'a assn  $\Rightarrow$  (state  $\Rightarrow$  state)  $\Rightarrow$  'a assn (infixl  $;$ . 13)
   $P ;. f \equiv \lambda Y\ s'\ Z. \exists s. P\ Y\ s\ Z \wedge s' = f\ s$ 

```

```

lemma assn-supd-def2 [simp]: assn-supd P f Y s' Z = ( $\exists s. P\ Y\ s\ Z \wedge s' = f\ s$ )
apply (unfold assn-supd-def)
apply (simp (no-asm))
done

```

### 53 supd-assn

```

constdefs
  supd-assn :: (state  $\Rightarrow$  state)  $\Rightarrow$  'a assn  $\Rightarrow$  'a assn (infixr .; 13)
  f .; P  $\equiv \lambda Y\ s. P\ Y\ (f\ s)$ 

```

```

lemma supd-assn-def2 [simp]: (f .; P) Y s = P Y (f s)
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

```

### 54 subst-res

```

constdefs
  subst-res :: 'a assn  $\Rightarrow$  res  $\Rightarrow$  'a assn (infixr [-<- 60,61] 60)
  P [-<- w  $\equiv \lambda Y. P\ w$ 

```

```

lemma subst-res-def2 [simp]: (P [-<- w) Y = P w
apply (unfold subst-res-def)
apply (simp (no-asm))
done

```

```

lemma subst-subst-res [simp]: P [-<- w [-<- v = P [-<- w
apply (rule ext)
apply (simp (no-asm))
done

```

```

lemma peek-and-subst-res [simp]: (P  $\wedge$ . p) [-<- w = (P [-<- w  $\wedge$ . p)
apply (rule ext)
apply (rule ext)
apply (simp (no-asm))
done

```

### 55 subst-Bool

```

constdefs
  subst-Bool :: 'a assn  $\Rightarrow$  bool  $\Rightarrow$  'a assn (infixr [-<== 60,61] 60)

```

$$P \leftarrow = b \equiv \lambda Y s Z. \exists v. P (Val v) s Z \wedge (normal s \longrightarrow the-Bool v = b)$$

**lemma** *subst-Bool-def2* [simp]:  
 $(P \leftarrow = b) Y s Z = (\exists v. P (Val v) s Z \wedge (normal s \longrightarrow the-Bool v = b))$   
**apply** (unfold *subst-Bool-def*)  
**apply** (simp (no-asm))  
**done**

**lemma** *subst-Bool-the-BoolI*:  $P (Val b) s Z \implies (P \leftarrow = the-Bool b) Y s Z$   
**apply** *auto*  
**done**

## 56 peek-res

**constdefs**  
 $peek-res \quad :: (res \Rightarrow 'a\ assn) \Rightarrow 'a\ assn$   
 $peek-res\ Pf \equiv \lambda Y. Pf\ Y\ Y$

**syntax**  
 $@peek-res \quad :: pptrn \Rightarrow 'a\ assn \Rightarrow 'a\ assn \quad (\lambda \cdot. - [0,3] 3)$   
**translations**  
 $\lambda w. P \quad == peek-res (\lambda w. P)$

**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]:  $peek-res\ P \leftarrow w = P\ w \leftarrow w$   
**apply** (*rule ext*)  
**apply** (simp (no-asm))  
**done**

**lemma** *peek-subst-res-allI*:  
 $(\bigwedge a. T\ a\ (P\ (f\ a) \leftarrow f\ a)) \implies \forall a. T\ a\ (peek-res\ P \leftarrow f\ a)$   
**apply** (*rule allI*)  
**apply** (simp (no-asm))  
**apply** *fast*  
**done**

## 57 ign-res

**constdefs**  
 $ign-res \quad :: 'a\ assn \Rightarrow 'a\ assn \quad (-\downarrow [1000] 1000)$   
 $P \downarrow \quad \equiv \lambda Y s Z. \exists Y. P\ Y\ s\ Z$

**lemma** *ign-res-def2* [simp]:  $P \downarrow Y s Z = (\exists Y. P\ Y\ s\ Z)$   
**apply** (unfold *ign-res-def*)  
**apply** (simp (no-asm))  
**done**

```

lemma ign-ign-res [simp]:  $P \Downarrow = P \Downarrow$ 
apply (rule ext)
apply (rule ext)
apply (rule ext)
apply (simp (no-asm))
done

```

```

lemma ign-subst-res [simp]:  $P \Downarrow \leftarrow w = P \Downarrow$ 
apply (rule ext)
apply (rule ext)
apply (rule ext)
apply (simp (no-asm))
done

```

```

lemma peek-and-ign-res [simp]:  $(P \wedge. p) \Downarrow = (P \Downarrow \wedge. p)$ 
apply (rule ext)
apply (rule ext)
apply (rule ext)
apply (simp (no-asm))
done

```

## 58 peek-st

### constdefs

```

peek-st    :: (st  $\Rightarrow$  'a assn)  $\Rightarrow$  'a assn
peek-st P  $\equiv$   $\lambda Y s. P$  (store s) Y s

```

### syntax

```

@peek-st    :: pttrn  $\Rightarrow$  'a assn  $\Rightarrow$  'a assn          ( $\lambda \dots - [0,3] \ 3$ )

```

### translations

```

 $\lambda s.. P$   == peek-st ( $\lambda s. P$ )

```

```

lemma peek-st-def2 [simp]:  $(\lambda s.. Pf\ s)\ Y\ s = Pf\ (store\ s)\ Y\ s$ 
apply (unfold peek-st-def)
apply (simp (no-asm))
done

```

```

lemma peek-st-triv [simp]:  $(\lambda s.. P) = P$ 
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]:  $(\lambda s.. \lambda Y\ s'. P\ s\ Y\ s') = (\lambda Y\ s. P\ (store\ s)\ Y\ s)$ 
apply (rule ext)
apply (rule ext)
apply (simp (no-asm))

```

done

**lemma** *peek-st-subst-res* [simp]:  $(\lambda s.. P\ s) \leftarrow w = (\lambda s.. P\ s \leftarrow w)$   
**apply** (rule ext)  
**apply** (simp (no-asm))  
**done**

**lemma** *peek-st-Normal* [simp]:  $(\lambda s.. (Normal\ (P\ s))) = Normal\ (\lambda s.. P\ s)$   
**apply** (rule ext)  
**apply** (rule ext)  
**apply** (simp (no-asm))  
**done**

## 59 ign-res-eq

**constdefs**

*ign-res-eq* ::  $'a\ assn \Rightarrow res \Rightarrow 'a\ assn$  ( $\downarrow =$  [60,61] 60)  
 $P \downarrow = w \equiv \lambda Y.. P \downarrow \wedge. (\lambda s.. Y = w)$

**lemma** *ign-res-eq-def2* [simp]:  $(P \downarrow = w)\ Y\ s\ Z = ((\exists Y.. P\ Y\ s\ Z) \wedge Y = w)$   
**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**

## 60 RefVar

**constdefs**

*RefVar* ::  $(state \Rightarrow vvar \times state) \Rightarrow 'a\ assn \Rightarrow 'a\ assn$  (**infixr** ..; 13)  
 $vf\ ..; P \equiv \lambda Y\ s.. let\ (v, s') = vf\ s\ in\ P\ (Var\ v)\ s'$

**lemma** *RefVar-def2* [simp]:  $(vf\ ..; P)\ Y\ s = P\ (Var\ (fst\ (vf\ s)))\ (snd\ (vf\ s))$

**apply** (*unfold RefVar-def Let-def*)  
**apply** (*simp (no-asm) add: split-beta*)  
**done**

## 61 allocation

### constdefs

$Alloc \quad :: \text{prog} \Rightarrow \text{obj-tag} \Rightarrow 'a \text{ assn} \Rightarrow 'a \text{ assn}$   
 $Alloc \ G \ otag \ P \equiv \lambda Y \ s \ Z. \forall s' \ a. \ G \vdash s \text{ --halloc } otag \succ a \rightarrow s' \longrightarrow P \ (Val \ (Addr \ a)) \ s' \ Z$

$SXAlloc \quad :: \text{prog} \Rightarrow 'a \text{ assn} \Rightarrow 'a \text{ assn}$   
 $SXAlloc \ G \ P \equiv \lambda Y \ s \ Z. \forall s'. \ G \vdash s \text{ --salloc} \rightarrow s' \longrightarrow P \ Y \ s' \ Z$

**lemma** *Alloc-def2 [simp]:*  $Alloc \ G \ otag \ P \ Y \ s \ Z =$   
 $(\forall s' \ a. \ G \vdash s \text{ --halloc } otag \succ a \rightarrow s' \longrightarrow P \ (Val \ (Addr \ a)) \ s' \ Z)$   
**apply** (*unfold Alloc-def*)  
**apply** (*simp (no-asm)*)  
**done**

**lemma** *SXAlloc-def2 [simp]:*  
 $SXAlloc \ G \ P \ Y \ s \ Z = (\forall s'. \ G \vdash s \text{ --salloc} \rightarrow s' \longrightarrow P \ Y \ s' \ Z)$   
**apply** (*unfold SXAlloc-def*)  
**apply** (*simp (no-asm)*)  
**done**

### validity

#### constdefs

$type-ok \quad :: \text{prog} \Rightarrow \text{term} \Rightarrow \text{state} \Rightarrow \text{bool}$   
 $type-ok \ G \ t \ s \equiv$   
 $\exists L \ T \ C \ A. (normal \ s \longrightarrow (\llbracket prg=G, cls=C, lcl=L \rrbracket \vdash t :: T \wedge$   
 $\llbracket prg=G, cls=C, lcl=L \rrbracket \vdash dom \ (locals \ (store \ s)) \gg t \gg A) \wedge s :: \preceq (G, L)$

**datatype**  $'a \text{ triple} = triple \ ('a \text{ assn}) \text{ term} \ ('a \text{ assn})$   
 $(\{(1-)\} / \text{-->} / \{(1-)\} \quad [3,65,3] \ 75)$

**types**  $'a \text{ triples} = 'a \text{ triple set}$

### syntax

$var\text{-triple} \quad :: ['a \text{ assn}, var \quad , 'a \text{ assn}] \Rightarrow 'a \text{ triple}$   
 $(\{(1-)\} / \text{-->} / \{(1-)\} \quad [3,80,3] \ 75)$   
 $expr\text{-triple} \quad :: ['a \text{ assn}, expr \quad , 'a \text{ assn}] \Rightarrow 'a \text{ triple}$   
 $(\{(1-)\} / \text{-->} / \{(1-)\} \quad [3,80,3] \ 75)$   
 $exprs\text{-triple} \quad :: ['a \text{ assn}, expr \text{ list} \quad , 'a \text{ assn}] \Rightarrow 'a \text{ triple}$   
 $(\{(1-)\} / \text{--\#>} / \{(1-)\} \quad [3,65,3] \ 75)$   
 $stmt\text{-triple} \quad :: ['a \text{ assn}, stmt, \quad 'a \text{ assn}] \Rightarrow 'a \text{ triple}$   
 $(\{(1-)\} / \text{.-.} / \{(1-)\} \quad [3,65,3] \ 75)$

### syntax (*xsymbols*)

$triple \quad :: ['a \text{ assn}, term \quad , 'a \text{ assn}] \Rightarrow 'a \text{ triple}$   
 $(\{(1-)\} / \text{-->} / \{(1-)\} \quad [3,65,3] \ 75)$   
 $var\text{-triple} \quad :: ['a \text{ assn}, var \quad , 'a \text{ assn}] \Rightarrow 'a \text{ triple}$   
 $(\{(1-)\} / \text{-->} / \{(1-)\} \quad [3,80,3] \ 75)$



$expr\text{-}triple :: ['a\ assn, expr \quad , 'a\ assn] \Rightarrow 'a\ triple$   
 $(\{(1-)\} / \dashv\!\!\dashv\!\! / \{(1-)\} \quad [3,80,3] \ 75)$   
 $exprs\text{-}triple :: ['a\ assn, expr\ list \quad , 'a\ assn] \Rightarrow 'a\ triple$   
 $(\{(1-)\} / \dashv\!\!\dashv\!\! / \{(1-)\} \quad [3,65,3] \ 75)$

**translations**

$\{P\} \ e \dashv\!\!\dashv\!\! \{Q\} == \{P\} \ In1l \ e \succ \{Q\}$   
 $\{P\} \ e == \succ \{Q\} == \{P\} \ In2 \ e \succ \{Q\}$   
 $\{P\} \ e \doteq \succ \{Q\} == \{P\} \ In3 \ e \succ \{Q\}$   
 $\{P\} \ .c. \{Q\} == \{P\} \ In1r \ c \succ \{Q\}$

**lemma** *inj-triple*:  $inj \ (\lambda(P,t,Q). \{P\} \ t \succ \{Q\})$

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

**apply** *auto*

**done**

**lemma** *triple-inj-eq*:  $(\{P\} \ t \succ \{Q\} = \{P'\} \ t' \succ \{Q'\}) = (P=P' \wedge t=t' \wedge Q=Q')$

**apply** *auto*

**done**

**constdefs**

$mtriples :: ('c \Rightarrow 'sig \Rightarrow 'a\ assn) \Rightarrow ('c \Rightarrow 'sig \Rightarrow expr) \Rightarrow$   
 $( 'c \Rightarrow 'sig \Rightarrow 'a\ assn) \Rightarrow ('c \times 'sig) \ set \Rightarrow 'a\ triples$   
 $(\{ \{(1-)\} / \dashv\!\!\dashv\!\! / \{(1-)\} \mid - \} [3,65,3,65] \ 75)$   
 $\{\{P\} \ tf \dashv\!\!\dashv\!\! \{Q\} \mid ms\} \equiv (\lambda(C,sig). \{Normal(P \ C \ sig)\} \ tf \ C \ sig \dashv\!\!\dashv\!\! \{Q \ C \ sig\}) 'ms$

**consts**

$triple\text{-}valid :: prog \Rightarrow nat \Rightarrow \quad 'a\ triple \Rightarrow bool$   
 $( \quad \dashv\!\!\dashv\!\! \dashv\!\!\dashv\!\! [61,0, \ 58] \ 57)$   
 $ax\text{-}valids :: prog \Rightarrow 'b\ triples \Rightarrow 'a\ triples \Rightarrow bool$   
 $( -, \dashv\!\!\dashv\!\! \dashv\!\!\dashv\!\! [61,58,58] \ 57)$

**syntax**

$triples\text{-}valid :: prog \Rightarrow nat \Rightarrow \quad 'a\ triples \Rightarrow bool$   
 $( \quad \dashv\!\!\dashv\!\! \dashv\!\!\dashv\!\! [61,0, \ 58] \ 57)$   
 $ax\text{-}valid :: prog \Rightarrow 'b\ triples \Rightarrow 'a\ triple \Rightarrow bool$   
 $( -, \dashv\!\!\dashv\!\! \dashv\!\!\dashv\!\! [61,58,58] \ 57)$

**syntax** (*xsymbols*)

$triples\text{-}valid :: prog \Rightarrow nat \Rightarrow \quad 'a\ triples \Rightarrow bool$   
 $( \quad \dashv\!\!\dashv\!\! \dashv\!\!\dashv\!\! [61,0, \ 58] \ 57)$   
 $ax\text{-}valid :: prog \Rightarrow 'b\ triples \Rightarrow 'a\ triple \Rightarrow bool$   
 $( -, \dashv\!\!\dashv\!\! \dashv\!\!\dashv\!\! [61,58,58] \ 57)$

**defs** *triple-valid-def*:  $G \models n:t \equiv case \ t \ of \ \{P\} \ t \succ \{Q\} \Rightarrow$   
 $\forall Y \ s \ Z. \ P \ Y \ s \ Z \longrightarrow type\text{-}ok \ G \ t \ s \longrightarrow$   
 $(\forall Y' \ s'. \ G \vdash s \dashv\!\!\dashv\!\! \dashv\!\!\dashv\!\! (Y',s') \longrightarrow Q \ Y' \ s' \ Z)$

**translations**  $G \models n:ts == Ball \ ts \ (triple\text{-}valid \ G \ n)$

**defs** *ax-valids-def*:  $G, A \models ts \equiv \forall n. \ G \models n:A \longrightarrow G \models n:ts$

**translations**  $G, A \models t == G, A \models \{t\}$

**lemma** *triple-valid-def2*:  $G \models n:\{P\} \ t \succ \{Q\} =$

$(\forall Y \ s \ Z. \ P \ Y \ s \ Z$

```

  → (∃ L. (normal s → (∃ C T A. (prg=G,cls=C,lcl=L) ⊢ t :: T ∧
    (prg=G,cls=C,lcl=L) ⊢ dom (locals (store s)) » t » A)) ∧
    s :: ≤(G,L))
  → (∀ Y' s'. G ⊢ s -t>-n→ (Y',s') → Q Y' s' Z))
apply (unfold triple-valid-def type-ok-def)
apply (simp (no-asm))
done

declare split-paired-All [simp del] split-paired-Ex [simp del]
declare split-if [split del] split-if-asm [split del]
  option.split [split del] option.split-asm [split del]
declaration << K (Simplifier.map-ss (fn ss => ss delloop split-all-tac)) >>
declaration << K (Classical.map-cs (fn cs => cs delSWrapper split-all-tac)) >>

inductive
  ax-derivs :: prog ⇒ 'a triples ⇒ 'a triples ⇒ bool (-,|- [61,58,58] 57)
  and ax-deriv :: prog ⇒ 'a triples ⇒ 'a triple ⇒ bool (-,|- [61,58,58] 57)
  for G :: prog
where

  G,A ⊢ t ≡ G,A|⊢{t}

  | empty: G,A|⊢{}
  | insert: [G,A ⊢ t; G,A|⊢ts] ⇒
    G,A|⊢insert t ts

  | asm: ts ⊆ A ⇒ G,A|⊢ts

  | weaken: [G,A|⊢ts'; ts ⊆ ts'] ⇒ G,A|⊢ts

  | conseq: ∀ Y s Z . P Y s Z → (∃ P' Q'. G,A ⊢ {P'} t> {Q'} ∧ (∀ Y' s'.
    (∀ Y Z'. P' Y s Z' → Q' Y' s' Z') →
      Q Y' s' Z ))
    ⇒ G,A ⊢ {P } t> {Q }

  | hazard: G,A ⊢ {P ∧. Not ∘ type-ok G t} t> {Q}

  | Abrupt: G,A ⊢ {P ← (arbitrary3 t) ∧. Not ∘ normal} t> {P}

  — variables
  | LVar: G,A ⊢ {Normal (λs.. P ← Var (lvar vn s))} LVar vn => {P}

  | FVar: [G,A ⊢ {Normal P} .Init C. {Q};
    G,A ⊢ {Q} e -> {λ Val:a.. fvar C stat fn a ..; R}] ⇒
    G,A ⊢ {Normal P} {acc C,C,stat} e..fn => {R}

  | AVar: [G,A ⊢ {Normal P} e1 -> {Q};
    ∀ a. G,A ⊢ {Q ← Val a} e2 -> {λ Val:i.. avar G i a ..; R}] ⇒
    G,A ⊢ {Normal P} e1.[e2] => {R}

  — expressions

  | NewC: [G,A ⊢ {Normal P} .Init C. {Alloc G (CInst C) Q}] ⇒
    G,A ⊢ {Normal P} NewC C -> {Q}

  | NewA: [G,A ⊢ {Normal P} .init-comp-ty T. {Q}; G,A ⊢ {Q} e ->
    {λ Val:i.. abupd (check-neg i) .; Alloc G (Arr T (the-Intg i)) R}] ⇒
    G,A ⊢ {Normal P} New T[e] -> {R}

```

- | *Cast*:  $\llbracket G, A \vdash \{Normal\ P\} \ e \multimap \{\lambda Val:v:. \lambda s..$   
 $abupd \ (raise\text{-}if \ (\neg G, s \vdash v \text{ fits } T) \ ClassCast) \ .; \ Q \leftarrow Val \ v \rrbracket \Longrightarrow$   
 $G, A \vdash \{Normal\ P\} \ Cast \ T \ e \multimap \{Q\}$
- | *Inst*:  $\llbracket G, A \vdash \{Normal\ P\} \ e \multimap \{\lambda Val:v:. \lambda s..$   
 $Q \leftarrow Val \ (Bool \ (v \neq Null \wedge G, s \vdash v \text{ fits } RefT \ T)) \rrbracket \Longrightarrow$   
 $G, A \vdash \{Normal\ P\} \ e \ InstOf \ T \multimap \{Q\}$
- | *Lit*:  $G, A \vdash \{Normal\ (P \leftarrow Val \ v)\} \ Lit \ v \multimap \{P\}$
- | *UnOp*:  $\llbracket G, A \vdash \{Normal\ P\} \ e \multimap \{\lambda Val:v:. \ Q \leftarrow Val \ (eval\text{-}unop \ unop \ v)\} \rrbracket$   
 $\Longrightarrow$   
 $G, A \vdash \{Normal\ P\} \ UnOp \ unop \ e \multimap \{Q\}$
- | *BinOp*:  
 $\llbracket G, A \vdash \{Normal\ P\} \ e1 \multimap \{Q\};$   
 $\forall v1. \ G, A \vdash \{Q \leftarrow Val \ v1\}$   
 $(if \ need\text{-}second\text{-}arg \ binop \ v1 \ then \ (In1l \ e2) \ else \ (In1r \ Skip)) \multimap$   
 $\{\lambda Val:v2:. \ R \leftarrow Val \ (eval\text{-}binop \ binop \ v1 \ v2)\} \rrbracket$   
 $\Longrightarrow$   
 $G, A \vdash \{Normal\ P\} \ BinOp \ binop \ e1 \ e2 \multimap \{R\}$
- | *Super*:  $G, A \vdash \{Normal\ (\lambda s.. \ P \leftarrow Val \ (val\text{-}this \ s))\} \ Super \multimap \{P\}$
- | *Acc*:  $\llbracket G, A \vdash \{Normal\ P\} \ va \multimap \{\lambda Var:(v,f):. \ Q \leftarrow Val \ v\} \rrbracket \Longrightarrow$   
 $G, A \vdash \{Normal\ P\} \ Acc \ va \multimap \{Q\}$
- | *Ass*:  $\llbracket G, A \vdash \{Normal\ P\} \ va \multimap \{Q\};$   
 $\forall vf. \ G, A \vdash \{Q \leftarrow Var \ vf\} \ e \multimap \{\lambda Val:v:. \ assign \ (snd \ vf) \ v \ .; \ R\} \rrbracket \Longrightarrow$   
 $G, A \vdash \{Normal\ P\} \ va := e \multimap \{R\}$
- | *Cond*:  $\llbracket G, A \vdash \{Normal\ P\} \ e0 \multimap \{P'\};$   
 $\forall b. \ G, A \vdash \{P' \leftarrow =b\} \ (if \ b \ then \ e1 \ else \ e2) \multimap \{Q\} \rrbracket \Longrightarrow$   
 $G, A \vdash \{Normal\ P\} \ e0 \ ? \ e1 : e2 \multimap \{Q\}$
- | *Call*:  
 $\llbracket G, A \vdash \{Normal\ P\} \ e \multimap \{Q\}; \forall a. \ G, A \vdash \{Q \leftarrow Val \ a\} \ args \multimap \{R \ a\};$   
 $\forall a \ vs \ invC \ declC \ l. \ G, A \vdash \{(R \ a \leftarrow Vals \ vs \wedge.$   
 $(\lambda s. \ declC = invocation\text{-}declC \ class \ G \ mode \ (store \ s) \ a \ statT \ (\langle name=mn, parTs=pTs \rangle) \wedge$   
 $invC = invocation\text{-}class \ mode \ (store \ s) \ a \ statT \wedge$   
 $l = locals \ (store \ s)) \};$   
 $init\text{-}lvars \ G \ declC \ (\langle name=mn, parTs=pTs \rangle \ mode \ a \ vs) \wedge.$   
 $(\lambda s. \ normal \ s \longrightarrow G \vdash mode \rightarrow invC \preceq statT)\}$   
 $Methd \ declC \ (\langle name=mn, parTs=pTs \rangle) \multimap \{set\text{-}lvars \ l \ .; \ S\} \rrbracket \Longrightarrow$   
 $G, A \vdash \{Normal\ P\} \ \{accC, statT, mode\} \cdot mn(\{pTs\} args) \multimap \{S\}$
- | *Methd*:  $\llbracket G, A \cup \{\{P\} \ Methd \multimap \{Q\} \mid ms\} \vdash \{\{P\} \ body \ G \multimap \{Q\} \mid ms\} \rrbracket \Longrightarrow$   
 $G, A \vdash \{\{P\} \ Methd \multimap \{Q\} \mid ms\}$
- | *Body*:  $\llbracket G, A \vdash \{Normal\ P\} \ .Init \ D. \ \{Q\};$   
 $G, A \vdash \{Q\} \ .c. \ \{\lambda s.. \ abupd \ (absorb \ Ret) \ .; \ R \leftarrow (In1 \ (the \ (locals \ s \ Result)))\} \rrbracket$   
 $\Longrightarrow$   
 $G, A \vdash \{Normal\ P\} \ Body \ D \ c \multimap \{R\}$
- expression lists
- | *Nil*:  $G, A \vdash \{Normal\ (P \leftarrow Vals \ [])\} \ [] \multimap \{P\}$

	<i>Cons</i> : $\llbracket G, A \vdash \{ \text{Normal } P \} \ e \multimap \{ Q \};$ $\forall v. G, A \vdash \{ Q \leftarrow \text{Val } v \} \ es \multimap \{ \lambda \text{Vals:vs}.. R \leftarrow \text{Vals } (v \# \text{vs}) \} \rrbracket \implies$ $G, A \vdash \{ \text{Normal } P \} \ e \# es \multimap \{ R \}$
—	statements
	<i>Skip</i> : $G, A \vdash \{ \text{Normal } (P \leftarrow \Diamond) \} . \text{Skip} . \{ P \}$
	<i>Expr</i> : $\llbracket G, A \vdash \{ \text{Normal } P \} \ e \multimap \{ Q \leftarrow \Diamond \} \rrbracket \implies$ $G, A \vdash \{ \text{Normal } P \} . \text{Expr } e . \{ Q \}$
	<i>Lab</i> : $\llbracket G, A \vdash \{ \text{Normal } P \} . c . \{ \text{abupd } (\text{absorb } l) . ; Q \} \rrbracket \implies$ $G, A \vdash \{ \text{Normal } P \} . l . c . \{ Q \}$
	<i>Comp</i> : $\llbracket G, A \vdash \{ \text{Normal } P \} . c1 . \{ Q \};$ $G, A \vdash \{ Q \} . c2 . \{ R \} \rrbracket \implies$ $G, A \vdash \{ \text{Normal } P \} . c1 ; c2 . \{ R \}$
	<i>If</i> : $\llbracket G, A \vdash \{ \text{Normal } P \} \ e \multimap \{ P' \};$ $\forall b. G, A \vdash \{ P' \leftarrow b \} . (\text{if } b \text{ then } c1 \text{ else } c2) . \{ Q \} \rrbracket \implies$ $G, A \vdash \{ \text{Normal } P \} . \text{If}(e) \ c1 \ \text{Else } c2 . \{ Q \}$
	<i>Loop</i> : $\llbracket G, A \vdash \{ P \} \ e \multimap \{ P' \};$ $G, A \vdash \{ \text{Normal } (P' \leftarrow \text{True}) \} . c . \{ \text{abupd } (\text{absorb } (\text{Cont } l)) . ; P \} \rrbracket \implies$ $G, A \vdash \{ P \} . l . \text{While}(e) \ c . \{ (P' \leftarrow \text{False}) \downarrow = \Diamond \}$
	<i>Jmp</i> : $G, A \vdash \{ \text{Normal } (\text{abupd } (\lambda a. (\text{Some } (\text{Jump } j)))) . ; P \leftarrow \Diamond \} . \text{Jmp } j . \{ P \}$
	<i>Throw</i> : $\llbracket G, A \vdash \{ \text{Normal } P \} \ e \multimap \{ \lambda \text{Val:a}.. \text{abupd } (\text{throw } a) . ; Q \leftarrow \Diamond \} \rrbracket \implies$ $G, A \vdash \{ \text{Normal } P \} . \text{Throw } e . \{ Q \}$
	<i>Try</i> : $\llbracket G, A \vdash \{ \text{Normal } P \} . c1 . \{ \text{SXAlloc } G \ Q \};$ $G, A \vdash \{ Q \wedge (\lambda s. G, s \vdash \text{catch } C) ; . \text{new-xcpt-var } vn \} . c2 . \{ R \};$ $(Q \wedge (\lambda s. \neg G, s \vdash \text{catch } C)) \Rightarrow R \rrbracket \implies$ $G, A \vdash \{ \text{Normal } P \} . \text{Try } c1 \ \text{Catch}(C \ vn) \ c2 . \{ R \}$
	<i>Fin</i> : $\llbracket G, A \vdash \{ \text{Normal } P \} . c1 . \{ Q \};$ $\forall x. G, A \vdash \{ Q \wedge (\lambda s. x = \text{fst } s) ; . \text{abupd } (\lambda x. \text{None}) \}$ $. c2 . \{ \text{abupd } (\text{abrupt-if } (x \neq \text{None}) \ x) . ; R \} \rrbracket \implies$ $G, A \vdash \{ \text{Normal } P \} . c1 \ \text{Finally } c2 . \{ R \}$
	<i>Done</i> : $G, A \vdash \{ \text{Normal } (P \leftarrow \Diamond \wedge . \text{initd } C) \} . \text{Init } C . \{ P \}$
	<i>Init</i> : $\llbracket \text{the } (\text{class } G \ C) = c;$ $G, A \vdash \{ \text{Normal } ((P \wedge . \text{Not } \circ \text{initd } C) ; . \text{supd } (\text{init-class-obj } G \ C)) \}$ $. (\text{if } C = \text{Object then Skip else Init } (\text{super } c)) . \{ Q \};$ $\forall l. G, A \vdash \{ Q \wedge (\lambda s. l = \text{locals } (\text{store } s)) ; . \text{set-lvars empty} \}$ $. \text{init } c . \{ \text{set-lvars } l . ; R \} \rrbracket \implies$ $G, A \vdash \{ \text{Normal } (P \wedge . \text{Not } \circ \text{initd } C) \} . \text{Init } C . \{ R \}$

— Some dummy rules for the intermediate terms *Callee*, *InsInitE*, *InsInitV*, *FinA* only used by the smallstep semantics.

	<i>InsInitV</i> : $G, A \vdash \{ \text{Normal } P \} \ \text{InsInitV } c \ v \multimap \{ Q \}$
	<i>InsInitE</i> : $G, A \vdash \{ \text{Normal } P \} \ \text{InsInitE } c \ e \multimap \{ Q \}$
	<i>Callee</i> : $G, A \vdash \{ \text{Normal } P \} \ \text{Callee } l \ e \multimap \{ Q \}$
	<i>FinA</i> : $G, A \vdash \{ \text{Normal } P \} . \text{FinA } a \ c . \{ Q \}$

**constdefs**

*adapt-pre* :: 'a assn  $\Rightarrow$  'a assn  $\Rightarrow$  'a assn  $\Rightarrow$  'a assn  
*adapt-pre*  $P\ Q\ Q' \equiv \lambda Y\ s\ Z. \forall Y'\ s'. \exists Z'. P\ Y\ s\ Z' \wedge (Q\ Y'\ s'\ Z' \longrightarrow Q'\ Y'\ s'\ Z)$

### rules derived by induction

**lemma** *cut-valid*:  $\llbracket G, A' \rrbracket \models ts; G, A \models A \rrbracket \Longrightarrow G, A \models ts$

**apply** (*unfold ax-valids-def*)

**apply** *fast*

**done**

**lemma** *ax-thin* [*rule-format* (*no-asm*)]:

$G, (A' :: 'a\ triple\ set) \vdash (ts :: 'a\ triple\ set) \Longrightarrow \forall A. A' \subseteq A \longrightarrow G, A \vdash ts$

**apply** (*erule ax-derivs.induct*)

**apply** (*tactic ALLGOALS*(*EVERY* [*clarify-tac* @ {*claset*}, *REPEAT* o *smp-tac* 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* 3 1, *clarify, tactic smp-tac* 1 1, *rule exI, rule exI, erule* (1) *conjI*)

**prefer** 18

**apply** (*rule ax-derivs.Methd, drule spec, erule mp, fast*)

**apply** (*tactic*  $\ll$  *TRYALL* (*resolve-tac* ((*funpow* 5 *tl*) (*thms ax-derivs.intros*)))  $\gg$ )

**apply** *auto*

**done**

**lemma** *ax-thin-insert*:  $G, (A' :: 'a\ triple\ set) \vdash (t :: 'a\ triple) \Longrightarrow G, insert\ x\ A \vdash t$

**apply** (*erule ax-thin*)

**apply** *fast*

**done**

**lemma** *subset-mtriples-iff*:

$ts \subseteq \{\{P\}\ mb \multimap \{Q\} \mid ms\} = (\exists ms'. ms' \subseteq ms \wedge ts = \{\{P\}\ mb \multimap \{Q\} \mid ms'\})$

**apply** (*unfold mtriples-def*)

**apply** (*rule subset-image-iff*)

**done**

**lemma** *weaken*:

$G, (A' :: 'a\ triple\ set) \vdash (ts' :: 'a\ triple\ set) \Longrightarrow !ts. ts \subseteq ts' \longrightarrow G, A \vdash ts$

**apply** (*erule ax-derivs.induct*)

**apply** (*tactic ALLGOALS strip-tac*)

**apply** (*tactic*  $\ll$  *ALLGOALS*(*REPEAT* o (*EVERY* [*dtac* (*thm subset-singletonD*), *etac disjE, fast-tac* (@ {*claset*} *addSIs* [*thm ax-derivs.empty*])]))  $\gg$ )

**apply** (*tactic TRYALL hyp-subst-tac*)

**apply** (*simp, rule ax-derivs.empty*)

**apply** (*drule subset-insertD*)

**apply** (*blast intro: ax-derivs.insert*)

**apply** (*fast intro: ax-derivs.asm*)

**apply** (*fast intro: ax-derivs.weaken*)

**apply** (*rule ax-derivs.conseq, clarify, tactic smp-tac* 3 1, *blast*)

```

apply (tactic << TRYALL (resolve-tac ((funpow 5 tl) (thms ax-derivs.intros))
    THEN-ALL-NEW fast-tac @{claset})) >>)

```

```

apply (clarsimp simp add: subset-mtriples-iff)
apply (rule ax-derivs.Methd)
apply (erule 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:  $G, A \vdash \{P\} \text{ } t \succ \{Q\}$ . Given only the term above without annotations, Isabelle would infer a more general type were we could have different types of auxiliary variables in the assumption set ( $A$ ) and in the triple itself ( $P$  and  $Q$ ). But *ax-derivs.Methd* enforces the same type in the inductive definition of the derivation. So we have to restrict the types to be able to apply the rules.

```

lemma conseq12:  $\llbracket G, (A :: 'a \text{ triple set}) \vdash \{P :: 'a \text{ assn}\} \text{ } t \succ \{Q'\};$ 
 $\forall Y \ s \ Z. P \ Y \ s \ Z \longrightarrow (\forall Y' \ s'. (\forall Y \ Z'. P' \ Y \ s \ Z' \longrightarrow Q' \ Y' \ s' \ Z') \longrightarrow$ 
 $Q \ Y' \ s' \ Z) \rrbracket$ 
 $\implies G, A \vdash \{P :: 'a \text{ assn}\} \text{ } t \succ \{Q\}$ 
apply (rule ax-derivs.conseq)
apply clarsimp
apply blast
done

```

— Nice variant, since it is so symmetric we might be able to memorise it.

```

lemma conseq12':  $\llbracket G, (A :: 'a \text{ triple set}) \vdash \{P' :: 'a \text{ assn}\} \text{ } t \succ \{Q'\}; \forall s \ Y' \ s'.$ 
 $(\forall Y \ Z. P' \ Y \ s \ Z \longrightarrow Q' \ Y' \ s' \ Z) \longrightarrow$ 
 $(\forall Y \ Z. P \ Y \ s \ Z \longrightarrow Q \ Y' \ s' \ Z) \rrbracket$ 
 $\implies G, A \vdash \{P :: 'a \text{ assn}\} \text{ } t \succ \{Q\}$ 
apply (erule conseq12)
apply fast
done

```

```

lemma conseq12-from-conseq12':  $\llbracket G, (A :: 'a \text{ triple set}) \vdash \{P' :: 'a \text{ assn}\} \text{ } t \succ \{Q'\};$ 
 $\forall Y \ s \ Z. P \ Y \ s \ Z \longrightarrow (\forall Y' \ s'. (\forall Y \ Z'. P' \ Y \ s \ Z' \longrightarrow Q' \ Y' \ s' \ Z') \longrightarrow$ 
 $Q \ Y' \ s' \ Z) \rrbracket$ 
 $\implies G, A \vdash \{P :: 'a \text{ assn}\} \text{ } t \succ \{Q\}$ 
apply (erule conseq12')
apply blast
done

```

```

lemma conseq1:  $\llbracket G, (A :: 'a \text{ triple set}) \vdash \{P' :: 'a \text{ assn}\} \text{ } t \succ \{Q\}; P \Rightarrow P' \rrbracket$ 
 $\implies G, A \vdash \{P :: 'a \text{ assn}\} \text{ } t \succ \{Q\}$ 
apply (erule conseq12)
apply blast
done

```

```

lemma conseq2:  $\llbracket G, (A :: 'a \text{ triple set}) \vdash \{P :: 'a \text{ assn}\} \text{ } t \succ \{Q'\}; Q' \Rightarrow Q \rrbracket$ 
 $\implies G, A \vdash \{P :: 'a \text{ assn}\} \text{ } t \succ \{Q\}$ 

```

**apply** (*erule conseq12*)  
**apply** *blast*  
**done**

**lemma** *ax-escape*:  

$$\llbracket \forall Y\ s\ Z.\ P\ Y\ s\ Z \implies G,(A::'a\ triple\ set) \vdash \{\lambda Y'\ s'\ (Z'::'a).\ (Y',s') = (Y,s)\} \begin{matrix} t> \\ \{\lambda Y\ s\ Z'.\ Q\ Y\ s\ Z\} \end{matrix} \rrbracket$$
  

$$\implies G,A \vdash \{P::'a\ assn\} \begin{matrix} t> \\ \{Q::'a\ assn\} \end{matrix}$$
  
**apply** (*rule ax-derivs.conseq*)  
**apply** *force*  
**done**

**lemma** *ax-constant*:  $\llbracket C \implies G,(A::'a\ triple\ set) \vdash \{P::'a\ assn\} \begin{matrix} t> \\ \{Q\} \end{matrix} \rrbracket$   

$$\implies G,A \vdash \{\lambda Y\ s\ Z.\ C \wedge P\ Y\ s\ Z\} \begin{matrix} t> \\ \{Q\} \end{matrix}$$
  
**apply** (*rule ax-escape*)  
**apply** *clarify*  
**apply** (*rule conseq12*)  
**apply** *fast*  
**apply** *auto*  
**done**

**lemma** *ax-impossible* [*intro*]:  

$$G,(A::'a\ triple\ set) \vdash \{\lambda Y\ s\ Z.\ False\} \begin{matrix} t> \\ \{Q::'a\ assn\} \end{matrix}$$
  
**apply** (*rule ax-escape*)  
**apply** *clarify*  
**done**

**lemma** *ax-nochange-lemma*:  $\llbracket P\ Y\ s;\ All\ (op = w) \rrbracket \implies P\ w\ s$   
**apply** *auto*  
**done**

**lemma** *ax-nochange*:  

$$G,(A::(res \times state)\ triple\ set) \vdash \{\lambda Y\ s\ Z.\ (Y,s)=Z\} \begin{matrix} t> \\ \{\lambda Y\ s\ Z.\ (Y,s)=Z\} \end{matrix}$$
  

$$\implies G,A \vdash \{P::(res \times state)\ assn\} \begin{matrix} t> \\ \{P\} \end{matrix}$$
  
**apply** (*erule conseq12*)  
**apply** *auto*  
**apply** (*erule (1) ax-nochange-lemma*)  
**done**

**lemma** *ax-trivial*:  $G,(A::'a\ triple\ set) \vdash \{P::'a\ assn\} \begin{matrix} t> \\ \{\lambda Y\ s\ Z.\ True\} \end{matrix}$   
**apply** (*rule ax-derivs.conseq*)  
**apply** *auto*  
**done**

**lemma** *ax-disj*:

$\llbracket G, (A::'a \text{ triple set}) \vdash \{P1::'a \text{ assn}\} t \succ \{Q1\}; G, A \vdash \{P2::'a \text{ assn}\} t \succ \{Q2\} \rrbracket$   
 $\implies G, A \vdash \{\lambda Y s Z. P1 Y s Z \vee P2 Y s Z\} t \succ \{\lambda Y s Z. Q1 Y s Z \vee Q2 Y s Z\}$   
**apply** (*rule ax-escape*)  
**apply** *safe*  
**apply** (*erule conseq12, fast*) +  
**done**

**lemma** *ax-supd-shuffle*:

$(\exists Q. G, (A::'a \text{ triple set}) \vdash \{P::'a \text{ assn}\} .c1. \{Q\} \wedge G, A \vdash \{Q ;. f\} .c2. \{R\}) =$   
 $(\exists Q'. G, A \vdash \{P\} .c1. \{f ;. Q'\} \wedge G, A \vdash \{Q'\} .c2. \{R\})$   
**apply** (*best elim!: conseq1 conseq2*)  
**done**

**lemma** *ax-cases*:

$\llbracket G, (A::'a \text{ triple set}) \vdash \{P \wedge. C\} t \succ \{Q::'a \text{ assn}\};$   
 $G, A \vdash \{P \wedge. \text{Not} \circ C\} t \succ \{Q\} \rrbracket \implies G, A \vdash \{P\} t \succ \{Q\}$   
**apply** (*unfold peek-and-def*)  
**apply** (*rule ax-escape*)  
**apply** *clarify*  
**apply** (*case-tac C s*)  
**apply** (*erule conseq12, force*) +  
**done**

**lemma** *ax-adapt*:  $G, (A::'a \text{ triple set}) \vdash \{P::'a \text{ assn}\} t \succ \{Q\}$

$\implies G, A \vdash \{\text{adapt-pre } P Q Q'\} t \succ \{Q'\}$

**apply** (*unfold adapt-pre-def*)

**apply** (*erule conseq12*)

**apply** *fast*

**done**

**lemma** *adapt-pre-adapts*:  $G, (A::'a \text{ triple set}) \models \{P::'a \text{ assn}\} t \succ \{Q\}$

$\longrightarrow G, A \models \{\text{adapt-pre } P Q Q'\} t \succ \{Q'\}$

**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*:

$G, (A::'a \text{ triple set}) \vdash \{\text{Normal } P\} t \succ \{Q::'a \text{ assn}\}$



```

 $\impl G, A \vdash \{ \text{Normal } (P \wedge p) \} t \succ \{ Q \}$ 
apply (erule conseq1)
apply (simp (no-asm))
done

```

```

lemma peek-and-forget1:
 $G, (A :: 'a \text{ triple set}) \vdash \{ P :: 'a \text{ assn} \} t \succ \{ Q \}$ 
 $\impl G, A \vdash \{ P \wedge p \} t \succ \{ Q \}$ 
apply (erule conseq1)
apply (simp (no-asm))
done

```

```

lemmas ax-NormalD = peek-and-forget1 [of - - - - normal]

```

```

lemma peek-and-forget2:
 $G, (A :: 'a \text{ triple set}) \vdash \{ P :: 'a \text{ assn} \} t \succ \{ Q \wedge p \}$ 
 $\impl G, A \vdash \{ P \} t \succ \{ Q \}$ 
apply (erule conseq2)
apply (simp (no-asm))
done

```

```

lemma ax-subst-Val-allI:
 $\forall v. G, (A :: 'a \text{ triple set}) \vdash \{ (P' \quad v) \leftarrow \text{Val } v \} t \succ \{ (Q \ v) :: 'a \text{ assn} \}$ 
 $\impl \forall v. G, A \vdash \{ (\lambda w. P' (\text{the-In1 } w)) \leftarrow \text{Val } v \} t \succ \{ Q \ v \}$ 
apply (force elim!: conseq1)
done

```

```

lemma ax-subst-Var-allI:
 $\forall v. G, (A :: 'a \text{ triple set}) \vdash \{ (P' \quad v) \leftarrow \text{Var } v \} t \succ \{ (Q \ v) :: 'a \text{ assn} \}$ 
 $\impl \forall v. G, A \vdash \{ (\lambda w. P' (\text{the-In2 } w)) \leftarrow \text{Var } v \} t \succ \{ Q \ v \}$ 
apply (force elim!: conseq1)
done

```

```

lemma ax-subst-Vals-allI:
 $(\forall v. G, (A :: 'a \text{ triple set}) \vdash \{ (P' \quad v) \leftarrow \text{Vals } v \} t \succ \{ (Q \ v) :: 'a \text{ assn} \})$ 
 $\impl \forall v. G, A \vdash \{ (\lambda w. P' (\text{the-In3 } w)) \leftarrow \text{Vals } v \} t \succ \{ Q \ v \}$ 
apply (force elim!: conseq1)
done

```

## alternative axioms

```

lemma ax-Lit2:
 $G, (A :: 'a \text{ triple set}) \vdash \{ \text{Normal } P :: 'a \text{ assn} \} \text{Lit } v \multimap \{ \text{Normal } (P \downarrow = \text{Val } v) \}$ 
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*:  $G, (A::'a \text{ triple set}) \vdash \{ \text{Normal } P::'a \text{ assn} \} \text{ LVar } vn \Rightarrow \{ \text{Normal } (\lambda s.. P \downarrow = \text{Var } (lvar \text{ } vn \text{ } s)) \}$   
**apply** (*rule* *ax-derivs.LVar* [*THEN* *conseq1*])  
**apply** *force*  
**done**

**lemma** *ax-Super2*:  $G, (A::'a \text{ triple set}) \vdash \{ \text{Normal } P::'a \text{ assn} \} \text{ Super} \Rightarrow \{ \text{Normal } (\lambda s.. P \downarrow = \text{Val } (val\text{-this } s)) \}$   
**apply** (*rule* *ax-derivs.Super* [*THEN* *conseq1*])  
**apply** *force*  
**done**

**lemma** *ax-Nil2*:  
 $G, (A::'a \text{ triple set}) \vdash \{ \text{Normal } P::'a \text{ assn} \} [] \Rightarrow \{ \text{Normal } (P \downarrow = \text{Vals } []) \}$   
**apply** (*rule* *ax-derivs.Nil* [*THEN* *conseq1*])  
**apply** *force*  
**done**

### misc derived structural rules

**lemma** *ax-finite-mtriples-lemma*:  $\llbracket F \subseteq ms; \text{finite } ms; \forall (C, sig) \in ms. G, (A::'a \text{ triple set}) \vdash \{ \text{Normal } (P \text{ } C \text{ } sig)::'a \text{ assn} \} \text{ mb } C \text{ } sig \Rightarrow \{ Q \text{ } C \text{ } sig \} \rrbracket \Rightarrow$   
 $G, A \vdash \{ \{ P \} \text{ mb} \Rightarrow \{ Q \} \mid F \}$   
**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*:  
 $G, (A::'a \text{ triple set}) \vdash \text{insert } (t::'a \text{ triple}) \text{ } ts \Rightarrow G, A \vdash t \wedge G, A \vdash ts$   
**apply** (*fast* *intro*: *ax-derivs.weaken*)  
**done**

**lemma** *ax-methods-spec*:  
 $\llbracket G, (A::'a \text{ triple set}) \vdash \text{split } f \text{ } ms; (C, sig) \in ms \rrbracket \Rightarrow G, A \vdash ((f \text{ } C \text{ } sig)::'a \text{ triple})$   
**apply** (*erule* *ax-derivs.weaken*)  
**apply** (*force* *del*: *image-eqI* *intro*: *rev-image-eqI*)  
**done**

**lemma** *ax-finite-pointwise-lemma* [*rule-format*]:  $\llbracket F \subseteq ms; \text{finite } ms \rrbracket \Rightarrow$   
 $((\forall (C, sig) \in F. G, (A::'a \text{ triple set}) \vdash (f \text{ } C \text{ } sig)::'a \text{ triple})) \longrightarrow (\forall (C, sig) \in ms. G, A \vdash (g \text{ } C \text{ } sig)::'a \text{ triple})) \longrightarrow$   
 $G, A \vdash \text{split } f \text{ } F \longrightarrow G, A \vdash \text{split } g \text{ } F$   
**apply** (*frule* (1) *finite-subset*)  
**apply** (*erule* *rev-mp*)  
**apply** (*erule* *thin-rl*)  
**apply** (*erule* *finite-induct*)  
**apply** *clarsimp* +  
**apply** (*erule* *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 << bind-thms (ax-Abrupts, sum3-instantiate @{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, standard]

```

## derived rules for methd call

```

lemma ax-Call-known-DynT:
   $\llbracket G \vdash \text{IntVir} \rightarrow C \preceq \text{stat } T;$ 
   $\forall a \text{ vs } l. G, A \vdash \{(R \ a \leftarrow \text{Vals } vs \wedge. (\lambda s. l = \text{locals } (\text{store } s))) \};$ 
   $\text{init-lvars } G \ C \ (\llbracket \text{name}=mn, \text{parTs}=pTs \rrbracket) \ \text{IntVir } a \text{ vs}\}$ 
   $\text{Methd } C \ (\llbracket \text{name}=mn, \text{parTs}=pTs \rrbracket) \multimap \{\text{set-lvars } l \ .; S\};$ 
   $\forall a. G, A \vdash \{Q \leftarrow \text{Val } a\} \ \text{args} \multimap$ 
   $\{R \ a \wedge. (\lambda s. C = \text{obj-class } (\text{the } (\text{heap } (\text{store } s) \text{ (the-Addr } a))) \wedge$ 
   $C = \text{invocation-declclass}$ 
   $G \ \text{IntVir } (\text{store } s) \ a \ \text{stat } T \ (\llbracket \text{name}=mn, \text{parTs}=pTs \rrbracket) \};$ 
   $G, (A::'a \text{ triple set}) \vdash \{\text{Normal } P\} \ e \multimap \{Q::'a \text{ assn}\}$ 
   $\implies G, A \vdash \{\text{Normal } P\} \ \{\text{acc } C, \text{stat } T, \text{IntVir}\} e.mn(\{pTs\} \text{args}) \multimap \{S\}$ 
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*:

```


$$\llbracket \forall a \text{ vs } l. G, A \vdash \{R \leftarrow \text{Vals } vs \wedge (\lambda s. l = \text{locals } (\text{store } s))\} ;.$$


$$\text{init-lvars } G \ C \ (\llbracket \text{name}=\text{mn}, \text{parTs}=\text{pTs} \rrbracket \text{ Static any-Addr vs})$$


$$\text{Methd } C \ (\llbracket \text{name}=\text{mn}, \text{parTs}=\text{pTs} \rrbracket) \multimap \{\text{set-lvars } l \ ; \ S\};$$


$$G, A \vdash \{\text{Normal } P\} \ e \multimap \{Q\};$$


$$\forall a. G, (A::'a \text{ triple set}) \vdash \{Q \leftarrow \text{Val } a\} \ \text{args} \dot{\multimap} \{(R::\text{val} \Rightarrow 'a \text{ assn}) \ a$$


$$\wedge (\lambda s. C = \text{invocation-declclass}$$


$$G \text{ Static } (\text{store } s) \ a \ \text{statT } (\llbracket \text{name}=\text{mn}, \text{parTs}=\text{pTs} \rrbracket)\}$$


$$\rrbracket \implies G, A \vdash \{\text{Normal } P\} \ \{\text{accC}, \text{statT}, \text{Static}\} \cdot \text{mn}(\{pTs\} \text{args}) \multimap \{S\}$$

apply (erule ax-derivs.Call)
apply safe
apply (erule spec)
apply (rule ax-escape, clarsimp)
apply (erule-tac  $V = ?P \longrightarrow ?Q$  in thin-rl)
apply (drule spec, drule spec, drule spec, erule conseq12)
apply (force simp add: init-lvars-def Let-def)
done

```

**lemma** *ax-Methd1*:

```


$$\llbracket G, A \cup \{\{P\} \text{ Methd} \multimap \{Q\} \mid ms\} \vdash \{\{P\} \text{ body } G \multimap \{Q\} \mid ms\}; (C, sig) \in ms \rrbracket \implies$$


$$G, A \vdash \{\text{Normal } (P \ C \ sig)\} \text{ Methd } C \ sig \multimap \{Q \ C \ sig\}$$

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*:  $\llbracket \text{the } (\text{class } G \ C) = c; C \neq \text{Object};$

```


$$\forall l. G, A \vdash \{Q \wedge (\lambda s. l = \text{locals } (\text{store } s))\} ;. \text{set-lvars empty}\}$$


$$\text{.init } c. \{\text{set-lvars } l \ ; \ R\};$$


$$G, A \vdash \{\text{Normal } ((P \wedge. \text{Not } \circ \text{initd } C) ;. \text{supd } (\text{init-class-obj } G \ C))\}$$


$$\text{.Init } (\text{super } c). \{Q\} \rrbracket \implies$$


$$G, (A::'a \text{ triple set}) \vdash \{\text{Normal } (P \wedge. \text{Not } \circ \text{initd } C)\} \text{ .Init } C. \{R::'a \text{ assn}\}$$

apply (erule ax-derivs.Init)

```

**apply** (*simp* (*no-asm-simp*))  
**apply** *assumption*  
**done**

**lemma** *ax-Init-Skip-lemma*:  
 $\forall l. G, (A::'a \text{ triple set}) \vdash \{P \leftarrow \Diamond \wedge. (\lambda s. l = \text{locals } (store\ s)) \ ;. \text{set-lvars } l'\}$   
 $\text{.Skip. } \{(set-lvars\ l \ ;. P)::'a \text{ assn}\}$   
**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*:  $wf-prog\ G \Longrightarrow G, (A::'a \text{ triple set}) \vdash$   
 $\{Normal\ ((supd\ (init-class-obj\ G\ Object) \ ;. P \leftarrow \Diamond) \wedge. Not\ o\ initd\ Object)\}$   
 $.Init\ Object. \{(P \wedge. initd\ Object)::'a \text{ assn}\}$   
**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*:

$G, (A::'a \text{ triple set}) \vdash \{P::'a \text{ assn}\} \ t \succ$   
 $\{ \text{Normal } (\lambda Y \ (x,s) \ Z. (\forall a. \text{new-Addr } (\text{heap } s) = \text{Some } a \longrightarrow$   
 $Q \ (\text{Val } (\text{Addr } a)) \ (\text{Norm}(\text{init-obj } G \ (\text{CInst } C) \ (\text{Heap } a) \ s)) \ Z)) \ \wedge.$   
 $\text{heap-free } (\text{Suc } (\text{Suc } 0)) \}$   
 $\implies G, A \vdash \{P\} \ t \succ \{ \text{Alloc } G \ (\text{CInst } C) \ Q \}$

**apply** (*erule consec2*)

**apply** (*auto elim!:: halloc-elim-cases*)

**done**

**lemma** *ax-Alloc-Arr*:

$G, (A::'a \text{ triple set}) \vdash \{P::'a \text{ assn}\} \ t \succ$   
 $\{ \lambda \text{Val}:i:. \text{Normal } (\lambda Y \ (x,s) \ Z. \neg \text{the-Intg } i < 0 \ \wedge$   
 $(\forall a. \text{new-Addr } (\text{heap } s) = \text{Some } a \longrightarrow$   
 $Q \ (\text{Val } (\text{Addr } a)) \ (\text{Norm}(\text{init-obj } G \ (\text{Arr } T \ (\text{the-Intg } i)) \ (\text{Heap } a) \ s)) \ Z)) \ \wedge.$   
 $\text{heap-free } (\text{Suc } (\text{Suc } 0)) \}$

$\implies$

$G, A \vdash \{P\} \ t \succ \{ \lambda \text{Val}:i:. \text{abupd } (\text{check-neg } i) \ .; \text{Alloc } G \ (\text{Arr } T(\text{the-Intg } i)) \ Q \}$

**apply** (*erule consec2*)

**apply** (*auto elim!:: halloc-elim-cases*)

**done**

**lemma** *ax-SXAlloc-catch-SXcpt*:

$\llbracket G, (A::'a \text{ triple set}) \vdash \{P::'a \text{ assn}\} \ t \succ$   
 $\{ (\lambda Y \ (x,s) \ Z. x = \text{Some } (\text{Xcpt } (\text{Std } xn)) \ \wedge$   
 $(\forall a. \text{new-Addr } (\text{heap } s) = \text{Some } a \longrightarrow$   
 $Q \ Y \ (\text{Some } (\text{Xcpt } (\text{Loc } a)), \text{init-obj } G \ (\text{CInst } (\text{SXcpt } xn)) \ (\text{Heap } a) \ s) \ Z) \}$   
 $\wedge. \text{heap-free } (\text{Suc } (\text{Suc } 0)) \} \rrbracket$

$\implies$

$G, A \vdash \{P\} \ t \succ \{ \text{SXAlloc } G \ (\lambda Y \ s \ Z. Q \ Y \ s \ Z \ \wedge \ G, s \vdash \text{catch } \text{SXcpt } xn) \}$

**apply** (*erule consec2*)

**apply** (*auto elim!:: sxalloc-elim-cases halloc-elim-cases*)

**done**

**end**

## Chapter 23

# AxSound

## 62 Soundness proof for Axiomatic semantics of Java expressions and statements

theory *AxSound* imports *AxSem* begin

validity

consts

```
triple-valid2:: prog ⇒ nat ⇒          'a triple ⇒ bool
  ( -||=-::-[61,0, 58] 57)
ax-valids2:: prog ⇒ 'a triples ⇒ 'a triples ⇒ bool
  (-,||=-::-[61,58,58] 57)
```

```
defs triple-valid2-def: 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.

```
defs ax-valids2-def: 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 3 1)
apply (case-tac normal s)
apply clarsimp
apply (elim conjE impE)
apply blast
```

```
apply (tactic smp-tac 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:  $G \models 0::\{Normal\ P\}\ Methd\ C\ sig-\succ \{Q\}$ 
apply (clarsimp elim!: evaln-elim-cases simp add: triple-valid2-def2)
done

```

```

lemma Methd-triple-valid2-SucI:
 $\llbracket G \models n::\{Normal\ P\}\ body\ G\ C\ sig-\succ \{Q\} \rrbracket$ 
 $\implies G \models Suc\ n::\{Normal\ P\}\ Methd\ C\ sig-\succ \{Q\}$ 
apply (simp (no-asm-use) add: triple-valid2-def2)
apply (intro strip, tactic smp-tac 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:  $G, A \cup \{\{P\}\ Methd-\succ \{Q\} \mid ms\} \models::\{\{P\}\ body\ G-\succ \{Q\} \mid ms\}$ 
shows  $G, A \models::\{\{P\}\ Methd-\succ \{Q\} \mid ms\}$ 
proof -
{
  fix n
  assume recursive:  $\bigwedge n. \forall t \in (A \cup \{\{P\}\ Methd-\succ \{Q\} \mid ms\}). G \models n::t$ 
 $\implies \forall t \in \{\{P\}\ body\ G-\succ \{Q\} \mid ms\}. G \models n::t$ 
  have  $\forall t \in A. G \models n::t \implies \forall t \in \{\{P\}\ Methd-\succ \{Q\} \mid ms\}. G \models n::t$ 
  proof (induct n)
  case 0
  show  $\forall t \in \{\{P\}\ Methd-\succ \{Q\} \mid ms\}. G \models 0::t$ 
  proof -
  {

```

```

    fix C sig
    assume (C,sig) ∈ ms
    have G|=0::{Normal (P C sig)} Methd C sig-⋈ {Q C sig}
      by (rule Methd-triple-valid2-0)
  }
  thus ?thesis
    by (simp add: mtriples-def split-def)
qed
next
case (Suc m)
note hyp = ⟨∀ t∈A. G|=m::t ⟹ ∀ t∈{{P} Methd-⋈ {Q} | ms}. G|=m::t⟩
note prem = ⟨∀ t∈A. G|=Suc m::t⟩
show ∀ t∈{{P} Methd-⋈ {Q} | ms}. G|=Suc m::t
proof -
  {
    fix C sig
    assume m: (C,sig) ∈ ms
    have G|=Suc m::{Normal (P C sig)} Methd C sig-⋈ {Q C sig}
    proof -
      from prem have prem-m: ∀ t∈A. G|=m::t
        by (rule triples-valid2-Suc)
      hence ∀ t∈{{P} Methd-⋈ {Q} | ms}. G|=m::t
        by (rule hyp)
      with prem-m
      have ∀ t∈(A ∪ {{P} Methd-⋈ {Q} | ms}). G|=m::t
        by (simp add: ball-Un)
      hence ∀ t∈{{P} body G-⋈ {Q} | ms}. G|=m::t
        by (rule recursive)
      with m have G|=m::{Normal (P C sig)} body G C sig-⋈ {Q C sig}
        by (auto simp add: mtriples-def split-def)
      thus ?thesis
        by (rule Methd-triple-valid2-SucI)
    qed
  }
  thus ?thesis
    by (simp add: mtriples-def split-def)
qed
qed
qed
}
with recursive show ?thesis
  by (unfold ax-valids2-def) blast
qed

```

```

lemma valids2-inductI: ∀ s t n Y' s'. G⊢s-t⋈-n→ (Y',s') ⟶ t = c ⟶
  Ball A (triple-valid2 G n) ⟶ (∀ Y Z. P Y s Z ⟶
    (∀ L. s::⋈(G,L) ⟶
      (∀ T C A. (normal s ⟶ ((prg=G,cls=C,lcl=L)⊢t::T) ∧
        (prg=G,cls=C,lcl=L)⊢dom (locals (store s)))»t»A) ⟶
        Q Y' s' Z ∧ s'::⋈(G,L)))) ⟹
  G,A||=::{{P} c⋈ {Q}}
apply (simp (no-asm) add: ax-valids2-def triple-valid2-def2)
apply clarsimp
done

```

```

lemma da-good-approx-evalnE [consumes 4]:
  assumes evaln: G⊢s0 -t⋈-n→ (v, s1)

```

```

and    wt: ( $\llbracket \text{prg} = G, \text{cls} = C, \text{lcl} = L \rrbracket \vdash t :: T$ )
and    da: ( $\llbracket \text{prg} = G, \text{cls} = C, \text{lcl} = L \rrbracket \vdash \text{dom} (\text{locals} (\text{store } s0)) \gg t \gg A$ )
and    wf: wf-prog  $G$ 
and    elim: ( $\llbracket \text{normal } s1 \implies \text{nrm } A \subseteq \text{dom} (\text{locals} (\text{store } s1)) ;$ 
              $\wedge l. \llbracket \text{abrupt } s1 = \text{Some} (\text{Jump} (\text{Break } l)) ; \text{normal } s0 \rrbracket$ 
              $\implies \text{brk } A \ l \subseteq \text{dom} (\text{locals} (\text{store } s1)) ;$ 
              $\llbracket \text{abrupt } s1 = \text{Some} (\text{Jump } \text{Ret}) ; \text{normal } s0 \rrbracket$ 
              $\implies \text{Result} \in \text{dom} (\text{locals} (\text{store } s1))$ 
              $\rrbracket \implies P$ )
shows  $P$ 
proof –
  from evaln have  $G \vdash s0 \dashv t \dashv \rightarrow (v, s1)$ 
  by (rule evaln-eval)
  from this wt da wf elim show  $P$ 
  by (rule da-good-approxE') iprover+
qed

```

```

lemma validI:
  assumes  $I: \wedge n \ s0 \ L \ \text{acc}C \ T \ C \ v \ s1 \ Y \ Z.$ 
              $\llbracket \forall t \in A. G \models n :: t ; s0 :: \preceq (G, L) ;$ 
              $\text{normal } s0 \implies \llbracket \text{prg} = G, \text{cls} = \text{acc}C, \text{lcl} = L \rrbracket \vdash t :: T ;$ 
              $\text{normal } s0 \implies \llbracket \text{prg} = G, \text{cls} = \text{acc}C, \text{lcl} = L \rrbracket \vdash \text{dom} (\text{locals} (\text{store } s0)) \gg t \gg C ;$ 
              $G \vdash s0 \dashv t \dashv \rightarrow n \rightarrow (v, s1) ; P \ Y \ s0 \ Z \rrbracket \implies Q \ v \ s1 \ Z \wedge s1 :: \preceq (G, L)$ 
  shows  $G, A \models :: \{ \{ P \} \ t \gg \{ Q \} \}$ 
apply (simp add: ax-valids2-def triple-valid2-def2)
apply (intro allI impI)
apply (case-tac normal s)
apply clarsimp
apply (rule I, (assumption|simp)+)

apply (rule I, auto)
done

```

```

declare [[simproc add: wt-expr wt-var wt-exprs wt-stmt]]

```

```

lemma valid-stmtI:
  assumes  $I: \wedge n \ s0 \ L \ \text{acc}C \ C \ s1 \ Y \ Z.$ 
              $\llbracket \forall t \in A. G \models n :: t ; s0 :: \preceq (G, L) ;$ 
              $\text{normal } s0 \implies \llbracket \text{prg} = G, \text{cls} = \text{acc}C, \text{lcl} = L \rrbracket \vdash c :: \surd ;$ 
              $\text{normal } s0 \implies \llbracket \text{prg} = G, \text{cls} = \text{acc}C, \text{lcl} = L \rrbracket \vdash \text{dom} (\text{locals} (\text{store } s0)) \gg \langle c \rangle_s \gg C ;$ 
              $G \vdash s0 \dashv c \dashv \rightarrow s1 ; P \ Y \ s0 \ Z \rrbracket \implies Q \ \Diamond \ s1 \ Z \wedge s1 :: \preceq (G, L)$ 
  shows  $G, A \models :: \{ \{ P \} \ \langle c \rangle_s \gg \{ Q \} \}$ 
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  $I: \wedge n \ s0 \ L \ \text{acc}C \ C \ s1 \ Y \ Z.$ 
              $\llbracket \forall t \in A. G \models n :: t ; s0 :: \preceq (G, L) ; \text{normal } s0 ; \llbracket \text{prg} = G, \text{cls} = \text{acc}C, \text{lcl} = L \rrbracket \vdash c :: \surd ;$ 
              $\llbracket \text{prg} = G, \text{cls} = \text{acc}C, \text{lcl} = L \rrbracket \vdash \text{dom} (\text{locals} (\text{store } s0)) \gg \langle c \rangle_s \gg C ;$ 

```

$$G \vdash s0 \text{ } -c-n \rightarrow s1; (Normal\ P)\ Y\ s0\ Z \Longrightarrow Q \Diamond s1\ Z \wedge s1::\preceq(G,L)$$
**shows**  $G, A \models::\{ \{ Normal\ P \} \langle c \rangle_s \succ \{ Q \} \}$   
**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**  $I: \bigwedge n\ s0\ L\ accC\ T\ C\ vf\ s1\ Y\ Z.$   

$$\llbracket \forall t \in A. G \models n::t; s0::\preceq(G,L); normal\ s0;$$
  

$$(\llbracket prg=G, cls=accC, lcl=L \rrbracket \vdash t::=T;$$
  

$$(\llbracket prg=G, cls=accC, lcl=L \rrbracket \vdash dom\ (locals\ (store\ s0))) \gg \langle t \rangle_v \gg C;$$
  

$$G \vdash s0 \text{ } -t=\succ vf-n \rightarrow s1; (Normal\ P)\ Y\ s0\ Z \rrbracket$$
  

$$\Longrightarrow Q\ (In2\ vf)\ s1\ Z \wedge s1::\preceq(G,L)$$
  
**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**  $I: \bigwedge n\ s0\ L\ accC\ T\ C\ v\ s1\ Y\ Z.$   

$$\llbracket \forall t \in A. G \models n::t; s0::\preceq(G,L); normal\ s0;$$
  

$$(\llbracket prg=G, cls=accC, lcl=L \rrbracket \vdash t::=T;$$
  

$$(\llbracket prg=G, cls=accC, lcl=L \rrbracket \vdash dom\ (locals\ (store\ s0))) \gg \langle t \rangle_e \gg C;$$
  

$$G \vdash s0 \text{ } -t=\succ v-n \rightarrow s1; (Normal\ P)\ Y\ s0\ Z \rrbracket$$
  

$$\Longrightarrow Q\ (In1\ v)\ s1\ Z \wedge s1::\preceq(G,L)$$
  
**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**  $I: \bigwedge n\ s0\ L\ accC\ T\ C\ vs\ s1\ Y\ Z.$   

$$\llbracket \forall t \in A. G \models n::t; s0::\preceq(G,L); normal\ s0;$$
  

$$(\llbracket prg=G, cls=accC, lcl=L \rrbracket \vdash t::=T;$$
  

$$(\llbracket prg=G, cls=accC, lcl=L \rrbracket \vdash dom\ (locals\ (store\ s0))) \gg \langle t \rangle_l \gg C;$$
  

$$G \vdash s0 \text{ } -t=\succ vs-n \rightarrow s1; (Normal\ P)\ Y\ s0\ Z \rrbracket$$
  

$$\Longrightarrow Q\ (In3\ vs)\ s1\ Z \wedge s1::\preceq(G,L)$$
  
**shows**  $G, A \models::\{ \{ Normal\ P \} \langle t \rangle_l \succ \{ Q \} \}$   
**apply** (*simp add: ax-valids2-def triple-valid2-def2*)  
**apply** (*intro allI impI*)  
**apply** (*elim exE conjE*)  
**apply** *simp*  
**apply** (*rule I*)  
**by** *auto*

**lemma** *validE* [*consumes 5*]:

**assumes** *valid*:  $G, A \models::\{ \{ P \} t \succ \{ Q \} \}$

**and**  $P: P \ Y \ s0 \ Z$   
**and**  $valid-A: \forall t \in A. G \models n::t$   
**and**  $conf: s0::\preceq(G, L)$   
**and**  $eval: G \vdash s0 \rightarrow -t \succ -n \rightarrow (v, s1)$   
**and**  $wt: normal \ s0 \implies (\text{prg}=G, \text{cls}=\text{acc}C, \text{lcl}=L) \vdash t::T$   
**and**  $da: normal \ s0 \implies (\text{prg}=G, \text{cls}=\text{acc}C, \text{lcl}=L) \vdash \text{dom} \ (\text{locals} \ (\text{store} \ s0)) \gg t \gg C$   
**and**  $elim: \llbracket Q \ v \ s1 \ Z; s1::\preceq(G, L) \rrbracket \implies \text{concl}$   
**shows**  $\text{concl}$   
**using**  $\text{prems}$   
**by**  $(\text{simp} \ \text{add: ax-valids2-def triple-valid2-def2}) \ \text{fast}$

**lemma**  $\text{all-empty:} \ (!x. P) = P$

**by**  $\text{simp}$

**corollary**  $\text{evaln-type-sound:}$

**assumes**  $evaln: G \vdash s0 \rightarrow -t \succ -n \rightarrow (v, s1)$  **and**  
 $wt: (\text{prg}=G, \text{cls}=\text{acc}C, \text{lcl}=L) \vdash t::T$  **and**  
 $da: (\text{prg}=G, \text{cls}=\text{acc}C, \text{lcl}=L) \vdash \text{dom} \ (\text{locals} \ (\text{store} \ s0)) \gg t \gg A$  **and**  
 $conf-s0: s0::\preceq(G, L)$  **and**  
 $wf: wf\text{-prog} \ G$   
**shows**  $s1::\preceq(G, L) \wedge (normal \ s1 \longrightarrow G, L, \text{store} \ s1 \vdash t \succ v::\preceq T) \wedge$   
 $(\text{error-free} \ s0 = \text{error-free} \ s1)$

**proof**  $-$

**from**  $evaln$  **have**  $G \vdash s0 \rightarrow -t \succ -n \rightarrow (v, s1)$   
**by**  $(\text{rule} \ \text{evaln-eval})$   
**from**  $this \ wt \ da \ wf \ conf-s0$  **show**  $?thesis$   
**by**  $(\text{rule} \ \text{eval-type-sound})$

**qed**

**corollary**  $\text{dom-locals-evaln-mono-elim} \ [\text{consumes} \ 1]:$

**assumes**  
 $evaln: G \vdash s0 \rightarrow -t \succ -n \rightarrow (v, s1)$  **and**  
 $\text{hyps: } \llbracket \text{dom} \ (\text{locals} \ (\text{store} \ s0)) \subseteq \text{dom} \ (\text{locals} \ (\text{store} \ s1));$   
 $\wedge \ v \ s \ \text{val}. \llbracket v = \text{In2} \ vv; normal \ s1 \rrbracket$   
 $\implies \text{dom} \ (\text{locals} \ (\text{store} \ s))$   
 $\subseteq \text{dom} \ (\text{locals} \ (\text{store} \ ((\text{snd} \ vv) \ \text{val} \ s))) \rrbracket \implies P$

**shows**  $P$

**proof**  $-$

**from**  $evaln$  **have**  $G \vdash s0 \rightarrow -t \succ -n \rightarrow (v, s1)$  **by**  $(\text{rule} \ \text{evaln-eval})$   
**from**  $this \ \text{hyps}$  **show**  $?thesis$   
**by**  $(\text{rule} \ \text{dom-locals-eval-mono-elim}) \ \text{iprover}+$

**qed**

**lemma**  $\text{evaln-no-abrupt:}$

$\wedge s \ s'. \llbracket G \vdash s \rightarrow -t \succ -n \rightarrow (w, s') \rrbracket \implies normal \ s$   
**by**  $(\text{erule} \ \text{evaln-cases, auto})$

**declare**  $\text{inj-term-simps} \ [\text{simp}]$

**lemma**  $\text{ax-sound2:}$

**assumes**  $wf: wf\text{-prog} \ G$   
**and**  $\text{deriv: } G, A \vdash ts$   
**shows**  $G, A \models ts$   
**using**  $\text{deriv}$

```

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  $\text{valid-ts} = \langle G, A \models:: ts \rangle$ 
  {
    fix n assume  $\text{valid-A: } \forall t \in A. G \models n:: t$ 
    have  $G \models n:: t$  and  $\forall t \in ts. G \models n:: t$ 
    proof –
      from valid-A valid-t show  $G \models n:: t$ 
      by (simp add: ax-valids2-def)
    next
      from valid-A valid-ts show  $\forall t \in ts. G \models n:: t$ 
      by (unfold ax-valids2-def) blast
    qed
    hence  $\forall t' \in \text{insert } t \text{ } ts. G \models n:: t'$ 
    by simp
  }
  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  $\text{con} = \langle \forall Y \ s \ Z. P \ Y \ s \ Z \longrightarrow$ 
     $(\exists P' \ Q'.$ 
     $(G, A \vdash \{P'\} \ t \succ \{Q'\} \wedge G, A \models:: \{ \{P'\} \ t \succ \{Q'\} \}) \wedge$ 
     $(\forall Y' \ s'. (\forall Y \ Z'. P' \ Y \ s \ Z' \longrightarrow Q' \ Y' \ s' \ Z') \longrightarrow Q \ Y' \ s' \ Z)) \rangle$ 
  show  $G, A \models:: \{ \{P\} \ t \succ \{Q\} \}$ 
  proof (rule validI)
    fix n s0 L accC T C v s1 Y Z
    assume  $\text{valid-A: } \forall t \in A. G \models n:: t$ 
    assume  $\text{conf: } s0:: \preceq (G, L)$ 
    assume  $\text{wt: normal } s0 \implies (\text{prg}=G, \text{cls}=\text{accC}, \text{lcl}=L) \vdash t:: T$ 
    assume  $\text{da: normal } s0$ 
     $\implies (\text{prg}=G, \text{cls}=\text{accC}, \text{lcl}=L) \vdash_{\text{dom}} (\text{locals } (\text{store } s0)) \gg t \gg C$ 
    assume  $\text{eval: } G \vdash s0 \ -t \succ -n \longrightarrow (v, s1)$ 
    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 3 1)
    apply clarify
    apply (tactic smp-tac 1 1)

```

```

    apply (erule allE,erule allE, erule mp)
    apply (intro strip)
    apply (tactic smp-tac 3 1)
    apply (tactic smp-tac 2 1)
    apply (tactic smp-tac 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  $G, A \models::\{ \{ P \wedge. \text{Not} \circ \text{type-ok } G \ t \} \ t \succ \{ Q \} \}$ 
  by (simp add: ax-valids2-def triple-valid2-def2 type-ok-def) fast
next
  case (Abrupt A P t)
  show  $G, A \models::\{ \{ P \leftarrow \text{arbitrary3 } t \wedge. \text{Not} \circ \text{normal} \} \ t \succ \{ P \} \}$ 
  proof (rule validI)
    fix n s0 L accC T C v s1 Y Z
    assume conf-s0:  $s0::\preceq(G, L)$ 
    assume eval:  $G \vdash s0 \multimap t \multimap \neg n \rightarrow (v, s1)$ 
    assume  $(P \leftarrow \text{arbitrary3 } t \wedge. \text{Not} \circ \text{normal}) \ Y \ s0 \ Z$ 
    then obtain P:  $P \ (\text{arbitrary3 } t) \ s0 \ Z$  and abrupt-s0:  $\neg \text{normal } s0$ 
    by simp
    from eval abrupt-s0 obtain  $s1=s0$  and  $v=\text{arbitrary3 } t$ 
    by auto
    with P conf-s0
    show  $P \ v \ s1 \ Z \wedge s1::\preceq(G, L)$ 
    by simp
  qed
next
  case (LVar A P vn)
  show  $G, A \models::\{ \{ \text{Normal } (\lambda s.. P \leftarrow \text{In2 } (\text{lvar } vn \ s)) \} \ LVar \ vn \Rightarrow \{ P \} \}$ 
  proof (rule valid-var-NormalI)
    fix n s0 L accC T C vf s1 Y Z
    assume conf-s0:  $s0::\preceq(G, L)$ 
    assume normal-s0: normal s0
    assume wt:  $(\text{prg} = G, \text{cls} = \text{accC}, \text{lcl} = L) \vdash LVar \ vn::=T$ 
    assume da:  $(\text{prg}=G, \text{cls}=\text{accC}, \text{lcl}=L) \vdash \text{dom } (\text{locals } (\text{store } s0)) \gg \langle LVar \ vn \rangle_v \gg C$ 
    assume eval:  $G \vdash s0 \multimap LVar \ vn \Rightarrow vf \multimap \neg n \rightarrow s1$ 
    assume P:  $(\text{Normal } (\lambda s.. P \leftarrow \text{In2 } (\text{lvar } vn \ s))) \ Y \ s0 \ Z$ 
    show  $P \ (\text{In2 } vf) \ s1 \ Z \wedge s1::\preceq(G, L)$ 
    proof
      from eval normal-s0 obtain  $s1=s0 \ vf=\text{lvar } vn \ (\text{store } s0)$ 
      by (fastsimp elim: evaln-elim-cases)
      with P show  $P \ (\text{In2 } vf) \ s1 \ Z$ 
      by simp
    next

```

```

from eval wt da conf-s0 wf
show  $s1::\preceq(G, L)$ 
  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 =  $\langle G, A \rangle \models \{ \{ \text{Normal } P \} . \text{Init statDeclC} . \{ Q \} \}$ 
note valid-e =  $\langle G, A \rangle \models \{ \{ Q \} \} e \multimap \{ \lambda \text{Val} : a . \text{fvar statDeclC stat fn a} . ; R \}$ 
show  $G, A \models \{ \{ \text{Normal } P \} \{ \text{accC}, \text{statDeclC}, \text{stat} \} e . \text{fn} \multimap \{ R \} \}$ 
proof (rule valid-var-NormalI)
  fix n s0 L accC' T V vf 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:  $(\text{prg} = G, \text{cls} = \text{accC}', \text{lcl} = L) \vdash \{ \text{accC}, \text{statDeclC}, \text{stat} \} e . \text{fn}::T$ 
  assume da:  $(\text{prg} = G, \text{cls} = \text{accC}', \text{lcl} = L) \vdash \text{dom}(\text{locals}(\text{store } s0)) \gg \{ \text{accC}, \text{statDeclC}, \text{stat} \} e . \text{fn} \rangle_v \gg V$ 
  assume eval:  $G \vdash s0 \multimap \{ \text{accC}, \text{statDeclC}, \text{stat} \} e . \text{fn} \multimap \text{vf} \multimap n \multimap s3$ 
  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:  $(\text{prg} = G, \text{cls} = \text{accC}, \text{lcl} = L) \vdash e::\text{Class statC}$  and
    accfield: accfield G accC statC fn = Some (statDeclC, f) and
    eq-accC: accC = accC' and
    stat: stat = is-static f and
    T: T = (type f)
    by (cases) (auto simp add: member-is-static-simp)
  from da eq-accC
  have da-e:  $(\text{prg} = G, \text{cls} = \text{accC}, \text{lcl} = L) \vdash \text{dom}(\text{locals}(\text{store } s0)) \gg \langle e \rangle_e \gg V$ 
    by cases simp
  from eval obtain a s1 s2 s2' where
    eval-init:  $G \vdash s0 \multimap \text{Init statDeclC} \multimap n \multimap s1$  and
    eval-e:  $G \vdash s1 \multimap e \multimap a \multimap n \multimap s2$  and
    fvar:  $(\text{vf}, s2') = \text{fvar statDeclC stat fn a s2}$  and
    s3: s3 = check-field-access G accC statDeclC fn stat a s2'
    using normal-s0 by (fastsimp elim: evaln-elim-cases)
  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:  $(\text{prg} = G, \text{cls} = \text{accC}, \text{lcl} = L) \vdash \text{dom}(\text{locals}(\text{store } s0)) \gg (\text{Init statDeclC})_s \gg I$ 
    by (auto intro: da-Init [simplified] assigned.select-convs)
from valid-init P valid-A conf-s0 eval-init wt-init da-init
obtain Q:  $Q \diamond s1 Z$  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  $\longrightarrow G, \text{store } s2 \vdash a::\preceq \text{Class statC}$ 

```



```

proof (cases normal s1)
  case True
  obtain V' where
    da-e':
      ( $\langle \text{prg} = G, \text{cls} = \text{acc}C, \text{lcl} = L \rangle \vdash \text{dom} (\text{locals} (\text{store } s1)) \rangle \langle e \rangle_e \rangle V'$ )
  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  $\longrightarrow G, \text{store } s2 \vdash a::\preceq \text{Class stat}C$ 
  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  $\lfloor \text{vf} \rfloor_v s2' Z$  and  $s2::\preceq(G, L)$ 
  by (cases rule: validE) (simp add: fvar [symmetric])
moreover from False eval-e have  $\neg \text{normal } s2$ 
  by auto
hence normal s2  $\longrightarrow G, \text{store } s2 \vdash a::\preceq \text{Class stat}C$ 
  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 =  $\langle G, A \mid \models::\{ \{ \text{Normal } P \} e1 \multimap \{ Q \} \} \rangle$ 
have valid-e2:  $\bigwedge a. G, A \mid \models::\{ \{ Q \leftarrow \text{In1 } a \} e2 \multimap \{ \lambda \text{Val}:i::\text{avar } G \ i \ a \ ..; R \} \}$ 
  using AVar.hyps by simp
show  $G, A \mid \models::\{ \{ \text{Normal } P \} e1.[e2] \multimap \{ R \} \}$ 
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:  $\langle \text{prg} = G, \text{cls} = \text{acc}C, \text{lcl} = L \rangle \vdash e1.[e2]::=T$ 
  assume da:  $\langle \text{prg} = G, \text{cls} = \text{acc}C, \text{lcl} = L \rangle$ 
     $\vdash \text{dom} (\text{locals} (\text{store } s0)) \rangle \langle e1.[e2] \rangle_v \rangle V$ 
  assume eval:  $G \vdash s0 \multimap e1.[e2] \multimap \text{vf} \multimap n \longrightarrow s2'$ 
  assume P: (Normal P) Y s0 Z
  show  $R \lfloor \text{vf} \rfloor_v s2' Z \wedge s2'::\preceq(G, L)$ 
  proof –

```

```

from wt obtain
  wt-e1: ( $\text{prg}=G, \text{cls}=\text{acc}C, \text{lcl}=L$ )  $\vdash e1 :: -T.$  and
  wt-e2: ( $\text{prg}=G, \text{cls}=\text{acc}C, \text{lcl}=L$ )  $\vdash e2 :: -\text{Prim}T \text{ Integer}$ 
by (rule wt-elim-cases) simp
from da obtain E1 where
  da-e1: ( $\text{prg}=G, \text{cls}=\text{acc}C, \text{lcl}=L$ )  $\vdash \text{dom} (\text{locals} (\text{store } s0)) \gg \langle e1 \rangle_e \gg E1$  and
  da-e2: ( $\text{prg}=G, \text{cls}=\text{acc}C, \text{lcl}=L$ )  $\vdash \text{nrm } E1 \gg \langle e2 \rangle_e \gg V$ 
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 (fastsimp elim: evaln-elim-cases)
from valid-e1 P valid-A conf-s0 eval-e1 wt-e1 da-e1
obtain Q:  $Q \ [a]_e \ s1 \ Z$  and conf-s1:  $s1 :: \preceq (G, L)$ 
by (rule validE)
from Q have Q':  $\bigwedge v. (Q \leftarrow \text{In1 } a) \ v \ s1 \ Z$ 
by simp
have R  $\lfloor \text{vf} \rfloor_v \ s2' \ Z$ 
proof (cases normal s1)
  case True
    obtain V' where
      ( $\text{prg}=G, \text{cls}=\text{acc}C, \text{lcl}=L$ )  $\vdash \text{dom} (\text{locals} (\text{store } s1)) \gg \langle e2 \rangle_e \gg V'$ 
    proof –
      from eval-e1 wt-e1 da-e1 wf True
      have  $\text{nrm } E1 \subseteq \text{dom} (\text{locals} (\text{store } s1))$ 
      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-init =  $\langle G, A \mid :: \{ \{ \text{Normal } P \} . \text{Init } C. \{ \text{Alloc } G \ (C \text{Inst } C) \ Q \} \} \rangle$ 
show  $G, A \mid :: \{ \{ \text{Normal } P \} \text{ NewC } C -\succ \{ Q \} \}$ 
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: ( $\text{prg}=G, \text{cls}=\text{acc}C, \text{lcl}=L$ )  $\vdash \text{NewC } C :: -T$ 
  assume da: ( $\text{prg}=G, \text{cls}=\text{acc}C, \text{lcl}=L$ )
     $\vdash \text{dom} (\text{locals} (\text{store } s0)) \gg \langle \text{NewC } C \rangle_e \gg E$ 
  assume eval:  $G \vdash s0 -\text{NewC } C -\succ v -n \rightarrow s2$ 

```

```

assume  $P$ : (Normal  $P$ )  $Y$   $s0$   $Z$ 
show  $Q \ [v]_e \ s2 \ Z \wedge s2::\preceq(G, L)$ 
proof –
  from  $wt$  obtain  $is\text{-}cls\text{-}C$ :  $is\text{-}class \ G \ C$ 
    by (rule  $wt\text{-}elim\text{-}cases$ ) (auto  $dest$ :  $is\text{-}acc\text{-}classD$ )
  hence  $wt\text{-}init$ :  $(\text{prg}=G, \text{cls}=accC, \text{lcl}=L) \vdash \text{Init } C::\checkmark$ 
    by auto
  obtain  $I$  where
     $da\text{-}init$ :  $(\text{prg}=G, \text{cls}=accC, \text{lcl}=L) \vdash \text{dom } (locals \ (store \ s0)) \gg \langle \text{Init } C \rangle_s \gg I$ 
    by (auto  $intro$ :  $da\text{-}Init$  [simplified]  $assigned.select\text{-}convs$ )
  from  $eval$  obtain  $s1 \ a$  where
     $eval\text{-}init$ :  $G \vdash s0 \text{ --Init } C \text{ --} n \rightarrow s1$  and
     $alloc$ :  $G \vdash s1 \text{ --halloc } C \text{Inst } C \succ a \rightarrow s2$  and
     $v$ :  $v = Addr \ a$ 
    using  $normal\text{-}s0$  by (fastsimp  $elim$ :  $evaln\text{-}elim\text{-}cases$ )
  from  $valid\text{-}init \ P \ valid\text{-}A \ conf\text{-}s0 \ eval\text{-}init \ wt\text{-}init \ da\text{-}init$ 
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  $eval \ wt \ da \ conf\text{-}s0 \ wf$ 
    have  $s2::\preceq(G, L)$ 
      by (rule  $evaln\text{-}type\text{-}sound$  [elim-format]) simp
    ultimately show  $?thesis \ ..$ 
qed
qed
next
case ( $NewA \ A \ P \ T \ Q \ e \ R$ )
note  $valid\text{-}init = \langle G, A \rangle \models:: \{ \{ Normal \ P \} .init\text{-}comp\text{-}ty \ T. \ \{ Q \} \}$ 
note  $valid\text{-}e = \langle G, A \rangle \models:: \{ \{ Q \} \ e \text{ --} \succ \{ \lambda Val: i. .abupd \ (check\text{-}neg \ i) \} .;$ 
       $Alloc \ G \ (Arr \ T \ (the\text{-}Intg \ i)) \ R \}$ 
show  $G, A \models:: \{ \{ Normal \ P \} \ New \ T[e] \text{ --} \succ \{ R \} \}$ 
proof (rule  $valid\text{-}expr\text{-}NormalI$ )
  fix  $n \ s0 \ L \ accC \ arrT \ E \ v \ s3 \ Y \ Z$ 
  assume  $valid\text{-}A$ :  $\forall t \in A. \ G \models n::t$ 
  assume  $conf\text{-}s0$ :  $s0::\preceq(G, L)$ 
  assume  $normal\text{-}s0$ :  $normal \ s0$ 
  assume  $wt$ :  $(\text{prg}=G, \text{cls}=accC, \text{lcl}=L) \vdash New \ T[e]::\text{--}arrT$ 
  assume  $da$ :  $(\text{prg}=G, \text{cls}=accC, \text{lcl}=L) \vdash \text{dom } (locals \ (store \ s0)) \gg \langle New \ T[e] \rangle_e \gg E$ 
  assume  $eval$ :  $G \vdash s0 \text{ --New } T[e] \text{ --} \succ v \text{ --} n \rightarrow s3$ 
  assume  $P$ : (Normal  $P$ )  $Y \ s0 \ Z$ 
  show  $R \ [v]_e \ s3 \ Z \wedge s3::\preceq(G, L)$ 
proof –
    from  $wt$  obtain
       $wt\text{-}init$ :  $(\text{prg}=G, \text{cls}=accC, \text{lcl}=L) \vdash init\text{-}comp\text{-}ty \ T::\checkmark$  and
       $wt\text{-}e$ :  $(\text{prg}=G, \text{cls}=accC, \text{lcl}=L) \vdash e::\text{--}PrimT \ Integer$ 
      by (rule  $wt\text{-}elim\text{-}cases$ ) (auto  $intro$ :  $wt\text{-}init\text{-}comp\text{-}ty$ )
    from  $da$  obtain
       $da\text{-}e$ :  $(\text{prg}=G, \text{cls}=accC, \text{lcl}=L) \vdash \text{dom } (locals \ (store \ s0)) \gg \langle e \rangle_e \gg E$ 
      by cases simp
    from  $eval$  obtain  $s1 \ i \ s2 \ a$  where
       $eval\text{-}init$ :  $G \vdash s0 \text{ --init-comp-ty } T \text{ --} n \rightarrow s1$  and
       $eval\text{-}e$ :  $G \vdash s1 \text{ --}e \text{ --} \succ i \text{ --} n \rightarrow s2$  and
       $alloc$ :  $G \vdash abupd \ (check\text{-}neg \ i) \ s2 \text{ --halloc } Arr \ T \ (the\text{-}Intg \ i) \succ a \rightarrow s3$  and
       $v$ :  $v = Addr \ a$ 
      using  $normal\text{-}s0$  by (fastsimp  $elim$ :  $evaln\text{-}elim\text{-}cases$ )
    obtain  $I$  where
       $da\text{-}init$ :

```

```

( $\langle \text{prg} = G, \text{cls} = \text{acc} C, \text{lcl} = L \rangle \vdash \text{dom} (\text{locals} (\text{store } s0)) \rangle \langle \text{init-comp-ty } T \rangle_s \rangle I$ )
proof (cases  $\exists C. T = \text{Class } C$ )
  case True
  thus ?thesis
    by - (rule that, (auto intro: da-Init [simplified]
      assigned.select-convs
      simp add: init-comp-ty-def))

next
  case False
  thus ?thesis
    by - (rule that, (auto intro: da-Skip [simplified]
      assigned.select-convs
      simp add: init-comp-ty-def))

qed
with valid-init P valid-A conf-s0 eval-init wt-init
obtain Q:  $Q \Diamond s1 Z$  and conf-s1:  $s1 :: \preceq (G, L)$ 
  by (rule validE)
obtain E' where
  ( $\langle \text{prg} = G, \text{cls} = \text{acc} C, \text{lcl} = L \rangle \vdash \text{dom} (\text{locals} (\text{store } s1)) \rangle \langle e \rangle_e \rangle E'$ )
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
have ( $\lambda \text{Val} : i. \text{abupd} (\text{check-neg } i) ;$ 
   $\text{Alloc } G (\text{Arr } T (\text{the-Intg } i)) R \lfloor i \rfloor_e s2 Z$ )
  by (rule validE)
with alloc v have  $R \lfloor v \rfloor_e s3 Z$ 
  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 (Cast A P e T Q)
note valid-e =  $\langle G, A \rangle \models :: \{ \{ \text{Normal } P \} e \multimap$ 
   $\{ \lambda \text{Val} : v. \lambda s. \text{abupd} (\text{raise-if } (\neg G, s \vdash v \text{ fits } T) \text{ ClassCast}) ;$ 
   $Q \leftarrow \text{In1 } v \} \}$ 
show  $G, A \models :: \{ \{ \text{Normal } P \} \text{Cast } T e \multimap \{ Q \} \}$ 
proof (rule valid-expr-NormalI)
  fix n s0 L accC castT 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: ( $\langle \text{prg} = G, \text{cls} = \text{acc} C, \text{lcl} = L \rangle \vdash \text{Cast } T e :: - \text{castT}$ )
  assume da: ( $\langle \text{prg} = G, \text{cls} = \text{acc} C, \text{lcl} = L \rangle \vdash \text{dom} (\text{locals} (\text{store } s0)) \rangle \langle \text{Cast } T e \rangle_e \rangle E$ )
  assume eval:  $G \vdash s0 - \text{Cast } T e \multimap v - n \rightarrow s2$ 
  assume P: (Normal P) Y s0 Z
  show  $Q \lfloor v \rfloor_e s2 Z \wedge s2 :: \preceq (G, L)$ 
proof -
  from wt obtain eT where

```

```

  wt-e: ( $\text{prg} = G, \text{cls} = \text{accC}, \text{lcl} = L$ )  $\vdash e :: -eT$ 
  by cases simp
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 where
  eval-e:  $G \vdash s0 -e -\succ v -n \rightarrow s1$  and
  s2:  $s2 = \text{abupd} (\text{raise-if } (\neg G, \text{snd } s1 \vdash v \text{ fits } T) \text{ ClassCast}) s1$ 
  using normal-s0 by (fastsimp elim: evaln-elim-cases)
from valid-e P valid-A conf-s0 eval-e wt-e da-e
have ( $\lambda \text{Val}.v. \lambda s.. \text{abupd} (\text{raise-if } (\neg G, s \vdash v \text{ fits } T) \text{ ClassCast}) .;$ 
   $Q \leftarrow \text{In1 } v$ )  $\lfloor v \rfloor_e s1 Z$ 
  by (rule validE)
with s2 have  $Q \lfloor v \rfloor_e s2 Z$ 
  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 (Inst A P e Q T)
assume valid-e:  $G, A \models :: \{ \{ \text{Normal } P \} e -\succ$ 
   $\{ \lambda \text{Val}.v.: \lambda s.. Q \leftarrow \text{In1} (\text{Bool } (v \neq \text{Null} \wedge G, s \vdash v \text{ fits } \text{RefT } T)) \} \}$ 
show  $G, A \models :: \{ \{ \text{Normal } P \} e \text{ InstOf } T -\succ \{ Q \} \}$ 
proof (rule valid-expr-NormalI)
  fix n s0 L accC instT 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: ( $\text{prg} = G, \text{cls} = \text{accC}, \text{lcl} = L$ )  $\vdash e \text{ InstOf } T :: -\text{instT}$ 
  assume da: ( $\text{prg} = G, \text{cls} = \text{accC}, \text{lcl} = L$ )  $\vdash \text{dom} (\text{locals} (\text{store } s0)) \gg \langle e \text{ InstOf } T \rangle_e \gg E$ 
  assume eval:  $G \vdash s0 -e \text{ InstOf } T -\succ v -n \rightarrow s1$ 
  assume P: (Normal P) Y s0 Z
  show  $Q \lfloor v \rfloor_e s1 Z \wedge s1 :: \preceq (G, L)$ 
  proof -
    from wt obtain eT where
      wt-e: ( $\text{prg} = G, \text{cls} = \text{accC}, \text{lcl} = L$ )  $\vdash e :: -eT$ 
      by cases simp
    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 a where
      eval-e:  $G \vdash s0 -e -\succ a -n \rightarrow s1$  and
      v:  $v = \text{Bool } (a \neq \text{Null} \wedge G, \text{store } s1 \vdash a \text{ fits } \text{RefT } T)$ 
      using normal-s0 by (fastsimp elim: evaln-elim-cases)
    from valid-e P valid-A conf-s0 eval-e wt-e da-e
    have ( $\lambda \text{Val}.v.: \lambda s.. Q \leftarrow \text{In1} (\text{Bool } (v \neq \text{Null} \wedge G, s \vdash v \text{ fits } \text{RefT } T))$ )
       $\lfloor a \rfloor_e s1 Z$ 
      by (rule validE)
    with v have  $Q \lfloor v \rfloor_e s1 Z$ 
      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 ..
  end
end

```

```

    qed
  qed
next
  case (Lit A P v)
  show  $G, A \models :: \{ \{ \text{Normal } (P \leftarrow \text{In1 } v) \} \text{ Lit } v \multimap \{ P \} \}$ 
  proof (rule valid-expr-NormalI)
    fix n L s0 s1 v' Y Z
    assume conf-s0:  $s0 :: \preceq (G, L)$ 
    assume normal-s0: normal s0
    assume eval:  $G \vdash s0 \multimap \text{Lit } v \multimap v' \multimap n \rightarrow s1$ 
    assume P:  $(\text{Normal } (P \leftarrow \text{In1 } v)) \text{ Y s0 Z}$ 
    show  $P \llbracket v' \rrbracket_e s1 Z \wedge s1 :: \preceq (G, L)$ 
    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 accC 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:  $(\text{prg} = G, \text{cls} = \text{accC}, \text{lcl} = L) \vdash \text{UnOp unop } e :: -T$ 
    assume da:  $(\text{prg} = G, \text{cls} = \text{accC}, \text{lcl} = L) \vdash \text{dom } (\text{locals } (\text{store } s0)) \gg \langle e \rangle_e \gg E$ 
    assume eval:  $G \vdash s0 \multimap \text{UnOp unop } e \multimap v \multimap n \rightarrow s1$ 
    assume P:  $(\text{Normal } P) \text{ Y s0 Z}$ 
    show  $Q \llbracket v \rrbracket_e s1 Z \wedge s1 :: \preceq (G, L)$ 
    proof -
      from wt obtain eT where
        wt-e:  $(\text{prg} = G, \text{cls} = \text{accC}, \text{lcl} = L) \vdash e :: -eT$ 
      by cases simp
      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 ve where
        eval-e:  $G \vdash s0 \multimap e \multimap ve \multimap n \rightarrow s1$  and
        v:  $v = \text{eval-unop unop } ve$ 
      using normal-s0 by (fastsimp elim: evaln-elim-cases)
      from valid-e P valid-A conf-s0 eval-e wt-e da-e
      have  $(\lambda \text{Val}. v. Q \leftarrow \text{In1 } (\text{eval-unop unop } v)) \llbracket ve \rrbracket_e s1 Z$ 
      by (rule validE)
      with v have  $Q \llbracket v \rrbracket_e s1 Z$ 
      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) >
      {λ Val:v2:. R←In1 (eval-binop binop v1 v2)} }
  using BinOp.hyps by simp
show G,A ⊨ :: { {Normal P} BinOp binop e1 e2 -> {R} }
proof (rule valid-expr-NormalI)
  fix n s0 L accC T E v s2 Y Z
  assume valid-A: ∀ t ∈ A. G ⊨ n :: t
  assume conf-s0: s0 :: ≤(G, L)
  assume normal-s0: normal s0
  assume wt: (prg = G, cls = accC, lcl = L) ⊢ BinOp binop e1 e2 :: - T
  assume da: (prg = G, cls = accC, lcl = L)
    ⊢ dom (locals (store s0)) » ⟨BinOp binop e1 e2⟩e » E
  assume eval: G ⊢ s0 - BinOp binop e1 e2 -> v - n → s2
  assume P: (Normal P) Y s0 Z
  show R [v]e s2 Z ∧ s2 :: ≤(G, L)
proof -
  from wt obtain e1T e2T where
    wt-e1: (prg = G, cls = accC, lcl = L) ⊢ e1 :: - e1T and
    wt-e2: (prg = G, cls = accC, lcl = L) ⊢ e2 :: - e2T and
    wt-binop: wt-binop G binop e1T e2T
  by cases simp
  have wt-Skip: (prg = G, cls = accC, lcl = L) ⊢ Skip :: ✓
  by simp

  from da obtain E1 where
    da-e1: (prg = G, cls = accC, lcl = L) ⊢ dom (locals (store s0)) » ⟨e1⟩e » E1
  by cases simp+
  from eval obtain v1 s1 v2 where
    eval-e1: G ⊢ s0 - e1 -> v1 - n → s1 and
    eval-e2: G ⊢ s1 - (if need-second-arg binop v1 then ⟨e2⟩e else ⟨Skip⟩s)
      -> - n → ([v2]e, s2) and
    v: v = eval-binop binop v1 v2
  using normal-s0 by (fastsimp elim: evaln-elim-cases)
  from valid-e1 P valid-A conf-s0 eval-e1 wt-e1 da-e1
  obtain Q: Q [v1]e s1 Z and conf-s1: s1 :: ≤(G, L)
  by (rule validE)
  from Q have Q': ∧ v. (Q←In1 v1) v s1 Z
  by simp
  have (λ Val:v2:. R←In1 (eval-binop binop v1 v2)) [v2]e s2 Z
  proof (cases normal s1)
    case True
    from eval-e1 wt-e1 da-e1 conf-s0 wf
    have conf-v1: G, store s1 ⊢ v1 :: ≤e1T
    by (rule evaln-type-sound [elim-format]) (insert True, simp)
    from eval-e1
    have G ⊢ s0 - e1 -> v1 → s1
    by (rule evaln-eval)
    from da wt-e1 wt-e2 wt-binop conf-s0 True this conf-v1 wf
    obtain E2 where
      da-e2: (prg = G, cls = accC, lcl = L) ⊢ dom (locals (store s1))
        » (if need-second-arg binop v1 then ⟨e2⟩e else ⟨Skip⟩s) » E2
    by (rule da-e2-BinOp [elim-format]) iprover
    from wt-e2 wt-Skip obtain T2
    where (prg = G, cls = accC, lcl = L)
      ⊢ (if need-second-arg binop v1 then ⟨e2⟩e else ⟨Skip⟩s) :: T2
    by (cases need-second-arg binop v1) auto
    note ve=validE [OF valid-e2, OF Q' valid-A conf-s1 eval-e2 this da-e2]

  thus ?thesis

```

```

      by (rule ve)
    next
      case False
      note ve=validE [OF valid-e2, OF Q' valid-A conf-s1 eval-e2]
      with False show ?thesis
      by iprover
    qed
  with v have R [v]e s2 Z
  by simp
  moreover
  from eval wt da conf-s0 wf
  have s2::≼(G, L)
  by (rule evaln-type-sound [elim-format]) simp
  ultimately show ?thesis ..
qed
qed
next
  case (Super A P)
  show G, A ⊨ :: { {Normal (λs.. P ← In1 (val-this s))} Super-⋈ {P} }
  proof (rule valid-expr-NormalI)
    fix n L s0 s1 v Y Z
    assume conf-s0: s0::≼(G, L)
    assume normal-s0: normal s0
    assume eval: G ⊢ s0 -Super-⋈ v -n → s1
    assume P: (Normal (λs.. P ← In1 (val-this s))) Y s0 Z
    show P [v]e s1 Z ∧ s1::≼(G, L)
    proof -
      from eval have s1=s0 and v=val-this (store s0)
      using normal-s0 by (auto elim: evaln-elim-cases)
      with P conf-s0 show ?thesis by simp
    qed
  qed
next
  case (Acc A P var Q)
  note valid-var = ⟨G, A ⊨ :: { {Normal P} var=⋈ {λVar:(v, f):. Q ← In1 v} }⟩
  show G, A ⊨ :: { {Normal P} Acc var-⋈ {Q} }
  proof (rule valid-expr-NormalI)
    fix n s0 L accC T E v s1 Y Z
    assume valid-A: ∀ t ∈ A. G ⊨ n::t
    assume conf-s0: s0::≼(G, L)
    assume normal-s0: normal s0
    assume wt: (⊢ prg=G, cls=accC, lcl=L) ⊢ Acc var::-T
    assume da: (⊢ prg=G, cls=accC, lcl=L) ⊢ dom (locals (store s0)) » ⟨Acc var⟩e E
    assume eval: G ⊢ s0 -Acc var-⋈ v -n → s1
    assume P: (Normal P) Y s0 Z
    show Q [v]e s1 Z ∧ s1::≼(G, L)
    proof -
      from wt obtain
        wt-var: (⊢ prg=G, cls=accC, lcl=L) ⊢ var::=T
      by cases simp
      from da obtain V where
        da-var: (⊢ prg=G, cls=accC, lcl=L) ⊢ dom (locals (store s0)) » ⟨var⟩v V
      by (cases ∃ n. var=LVar n) (insert da.LVar, auto elim!: da-elim-cases)
      from eval obtain w upd where
        eval-var: G ⊢ s0 -var=⋈(v, upd)-n → s1
      using normal-s0 by (fastsimp elim: evaln-elim-cases)
      from valid-var P valid-A conf-s0 eval-var wt-var da-var
      have (λVar:(v, f):. Q ← In1 v) [(v, upd)]v s1 Z
      by (rule validE)
    qed
  qed

```



```

then have  $Q \lfloor v \rfloor_e s1 Z$ 
  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 (Ass A P var Q e R)
note valid-var =  $\langle G, A \models :: \{ \{ Normal P \} \text{ var} = \succ \{ Q \} \} \rangle$ 
have valid-e:  $\bigwedge vf. G, A \models :: \{ \{ Q \leftarrow In2 vf \} \text{ e} - \succ \{ \lambda Val: v. \text{ assign } (snd vf) v .; R \} \}$ 
  using Ass.hyps by simp
show  $G, A \models :: \{ \{ Normal P \} \text{ var} := e - \succ \{ R \} \}$ 
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:  $\langle prg = G, cls = accC, lcl = L \rangle \vdash \text{var} := e :: - T$ 
  assume da:  $\langle prg = G, cls = accC, lcl = L \rangle \vdash \text{dom } (locals (store s0)) \gg \langle \text{var} := e \rangle_e \gg E$ 
  assume eval:  $G \vdash s0 - \text{var} := e - \succ v - n \rightarrow s3$ 
  assume P: (Normal P) Y s0 Z
  show  $R \lfloor v \rfloor_e s3 Z \wedge s3 :: \preceq(G, L)$ 
proof -
  from wt obtain varT where
    wt-var:  $\langle prg = G, cls = accC, lcl = L \rangle \vdash \text{var} :: \text{varT}$  and
    wt-e:  $\langle prg = G, cls = accC, lcl = L \rangle \vdash e :: - T$ 
  by cases simp
  from eval obtain  $w \text{ upd } s1 s2$  where
    eval-var:  $G \vdash s0 - \text{var} = \succ (w, \text{upd}) - n \rightarrow s1$  and
    eval-e:  $G \vdash s1 - e - \succ v - n \rightarrow s2$  and
    s3:  $s3 = \text{assign upd } v s2$ 
  using normal-s0 by (auto elim: evaln-elim-cases)
  have  $R \lfloor v \rfloor_e s3 Z$ 
proof (cases  $\exists vn. \text{var} = LVar vn$ )
  case False
  with da obtain V where
    da-var:  $\langle prg = G, cls = accC, lcl = L \rangle \vdash \text{dom } (locals (store s0)) \gg \langle \text{var} \rangle_v \gg V$  and
    da-e:  $\langle prg = G, cls = accC, lcl = L \rangle \vdash \text{nrm } V \gg \langle e \rangle_e \gg E$ 
  by cases simp +
  from valid-var P valid-A conf-s0 eval-var wt-var da-var
obtain  $Q: Q \lfloor (w, \text{upd}) \rfloor_v s1 Z$  and conf-s1:  $s1 :: \preceq(G, L)$ 
  by (rule validE)
  hence  $Q': \bigwedge v. (Q \leftarrow In2 (w, \text{upd})) v s1 Z$ 
  by simp
  have  $(\lambda Val: v. \text{ assign } (snd (w, \text{upd})) v .; R) \lfloor v \rfloor_e s2 Z$ 
proof (cases normal s1)
  case True
  obtain E' where
    da-e':  $\langle prg = G, cls = accC, lcl = L \rangle \vdash \text{dom } (locals (store s1)) \gg \langle e \rangle_e \gg E'$ 
  proof -
  from eval-var wt-var da-var wf True
  have  $\text{nrm } V \subseteq \text{dom } (locals (store s1))$ 
  by (cases rule: da-good-approx-evalnE) iprover
  with da-e show thesis

```

```

    by (rule da-weakenE) (rule that)
  qed
  note ve=validE [OF valid-e, OF Q' valid-A conf-s1 eval-e wt-e da-e]
  show ?thesis
    by (rule ve)
next
  case False
  note ve=validE [OF valid-e, OF Q' valid-A conf-s1 eval-e]
  with False show ?thesis
    by iprover
qed
with s3 show R [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: (prg=G, cls=accC, lcl=L) ⊢ dom (locals (store s0)) » ⟨e⟩e E
    by cases simp+
  from da.LVar vn obtain V where
    da-var: (prg=G, cls=accC, lcl=L)
      ⊢ dom (locals (store s0)) » ⟨var⟩v V
    by auto
  from valid-var P valid-A conf-s0 eval-var wt-var da-var
  obtain Q: Q [(w, upd)]v s1 Z and conf-s1: s1 :: ⌊(G, L)
    by (rule validE)
  hence Q': ∧ v. (Q ← In2 (w, upd)) v s1 Z
    by simp
  have (λ Val:v. assign (snd (w, upd)) v .; R) [v]e s2 Z
  proof (cases normal s1)
    case True
    obtain E' where
      da-e': (prg=G, cls=accC, lcl=L)
        ⊢ dom (locals (store s1)) » ⟨e⟩e E'
    proof -
      from eval-var
      have dom (locals (store s0)) ⊆ dom (locals (store (s1)))
        by (rule dom-locals-evaln-mono-elim)
      with da-e show thesis
        by (rule da-weakenE) (rule that)
    qed
    note ve=validE [OF valid-e, OF Q' valid-A conf-s1 eval-e wt-e da-e]
    show ?thesis
      by (rule ve)
  next
    case False
    note ve=validE [OF valid-e, OF Q' valid-A conf-s1 eval-e]
    with False show ?thesis
      by iprover
  qed
  with s3 show R [v]e s3 Z
    by simp
qed
moreover
from eval wt da conf-s0 wf
have s3 :: ⌊(G, L)
  by (rule evaln-type-sound [elim-format]) simp

```

```

ultimately show ?thesis ..
qed
qed
next
case (Cond A P e0 P' e1 e2 Q)
note valid-e0 = ⟨G,A⟩⊨::{ {Normal P} e0-⋈ {P'} }
have valid-then-else: ∧ b. ⟨G,A⟩⊨::{ {P'←=b} (if b then e1 else e2)-⋈ {Q} }
  using Cond.hyps by simp
show ⟨G,A⟩⊨::{ {Normal P} e0 ? e1 : e2-⋈ {Q} }
proof (rule valid-expr-NormalI)
  fix n s0 L accC T E v s2 Y Z
  assume valid-A: ∀ t∈A. G⊨n::t
  assume conf-s0: s0::⊆(G,L)
  assume normal-s0: normal s0
  assume wt: (prg=G,cls=accC,lcl=L)⊢e0 ? e1 : e2::-T
  assume da: (prg=G,cls=accC,lcl=L)⊢dom (locals (store s0))»⟨e0 ? e1:e2⟩e E
  assume eval: G⊢s0 -e0 ? e1 : e2-⋈v-n→ s2
  assume P: (Normal P) Y s0 Z
  show Q [v]e s2 Z ∧ s2::⊆(G, L)
proof -
  from wt obtain T1 T2 where
    wt-e0: (prg=G,cls=accC,lcl=L)⊢e0::-PrimT Boolean and
    wt-e1: (prg=G,cls=accC,lcl=L)⊢e1::-T1 and
    wt-e2: (prg=G,cls=accC,lcl=L)⊢e2::-T2
  by cases simp
  from da obtain E0 E1 E2 where
    da-e0: (prg=G,cls=accC,lcl=L)⊢dom (locals (store s0)) »⟨e0⟩e E0 and
    da-e1: (prg=G,cls=accC,lcl=L)
      ⊢(dom (locals (store s0)) ∪ assigns-if True e0)»⟨e1⟩e E1 and
    da-e2: (prg=G,cls=accC,lcl=L)
      ⊢(dom (locals (store s0)) ∪ assigns-if False e0)»⟨e2⟩e E2
  by cases simp+
  from eval obtain b s1 where
    eval-e0: G⊢s0 -e0-⋈b-n→ s1 and
    eval-then-else: G⊢s1 -(if the-Bool b then e1 else e2)-⋈v-n→ s2
  using normal-s0 by (fastsimp elim: evaln-elim-cases)
  from valid-e0 P valid-A conf-s0 eval-e0 wt-e0 da-e0
  obtain P' [b]e s1 Z and conf-s1: s1::⊆(G,L)
  by (rule validE)
  hence P': ∧ v. (P'←=(the-Bool b)) v s1 Z
  by (cases normal s1) auto
  have Q [v]e s2 Z
  proof (cases normal s1)
    case True
    note normal-s1=this
    from wt-e1 wt-e2 obtain T' where
      wt-then-else:
        (prg=G,cls=accC,lcl=L)⊢(if the-Bool b then e1 else e2)::-T'
      by (cases the-Bool b) simp+
    have s0-s1: dom (locals (store s0))
      ∪ assigns-if (the-Bool b) e0 ⊆ dom (locals (store s1))
    proof -
      from eval-e0
      have eval-e0': G⊢s0 -e0-⋈b→ s1
      by (rule evaln-eval)
      hence
        dom (locals (store s0)) ⊆ dom (locals (store s1))
      by (rule dom-locals-eval-mono-elim)
    moreover

```

```

from eval-e0' True wt-e0
have assigns-if (the-Bool b) e0  $\subseteq$  dom (locals (store s1))
  by (rule assigns-if-good-approx')
ultimately show ?thesis by (rule Un-least)
qed
obtain E' where
  da-then-else:
  ( $\text{prg} = G, \text{cls} = \text{accC}, \text{lcl} = L$ )
   $\vdash \text{dom (locals (store s1))} \gg \langle \text{if the-Bool } b \text{ then } e1 \text{ else } e2 \rangle_e \gg E'$ 
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 =  $\langle G, A \mid \vdash :: \{ \text{Normal } P \} \ e \multimap \{ Q \} \}$ 
have valid-args:  $\bigwedge a. G, A \mid \vdash :: \{ \{ Q \leftarrow \text{In1 } a \} \ \text{args} \dot{\multimap} \{ R \ a \} \}$ 
  using Call.hyps by simp
have valid-methd:  $\bigwedge a \ \text{vs} \ \text{invC} \ \text{declC} \ l.$ 
   $G, A \mid \vdash :: \{ \{ R \ a \leftarrow \text{In3 } \text{vs} \ \wedge.$ 
     $(\lambda s. \text{declC} =$ 
       $\text{invocation-declclass } G \ \text{mode} \ (\text{store } s) \ a \ \text{statT}$ 
       $(\text{name} = \text{mn}, \text{parTs} = \text{pTs}') \wedge$ 
       $\text{invC} = \text{invocation-class } \text{mode} \ (\text{store } s) \ a \ \text{statT} \wedge$ 
       $l = \text{locals} \ (\text{store } s)) ;.$ 
       $\text{init-lvars } G \ \text{declC} \ (\text{name} = \text{mn}, \text{parTs} = \text{pTs}') \ \text{mode} \ a \ \text{vs} \wedge.$ 
       $(\lambda s. \text{normal } s \longrightarrow G \vdash \text{mode} \rightarrow \text{invC} \preceq \text{statT}) \}$ 
       $\text{Methd } \text{declC} \ (\text{name} = \text{mn}, \text{parTs} = \text{pTs}') \multimap \{ \text{set-lvars } l \ ; \ S \} \}$ 
  using Call.hyps by simp
show  $G, A \mid \vdash :: \{ \text{Normal } P \} \ \{ \text{accC}', \text{statT}, \text{mode} \} e \cdot \text{mn} \ (\{ \text{pTs}' \} \text{args}) \multimap \{ S \}$ 
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:  $(\text{prg} = G, \text{cls} = \text{accC}, \text{lcl} = L) \vdash \{ \text{accC}', \text{statT}, \text{mode} \} e \cdot \text{mn} \ (\{ \text{pTs}' \} \text{args}) :: -T$ 
  assume da:  $(\text{prg} = G, \text{cls} = \text{accC}, \text{lcl} = L) \vdash \text{dom} \ (\text{locals} \ (\text{store } s0))$ 
     $\gg \langle \{ \text{accC}', \text{statT}, \text{mode} \} e \cdot \text{mn} \ (\{ \text{pTs}' \} \text{args}) \rangle_e \gg E$ 

```

```

assume eval:  $G \vdash s0 \multimap \{accC', statT, mode\} e \cdot mn(\{pTs'\} args) \multimap v \multimap n \rightarrow s5$ 
assume P: (Normal P)  $Y s0 Z$ 
show  $S \lfloor v \rfloor_e s5 Z \wedge s5 :: \preceq(G, L)$ 
proof –
  from wt obtain pTs statDeclT statM where
    wt-e:  $(\lfloor prg = G, cls = accC, lcl = L \rfloor) \vdash e :: \multimap RefT statT$  and
    wt-args:  $(\lfloor prg = G, cls = accC, lcl = L \rfloor) \vdash args :: \multimap pTs$  and
    statM:  $max\text{-}spec\ G\ accC\ statT\ (\lfloor name = mn, parTs = pTs \rfloor)$ 
       $= \{((statDeclT, statM), pTs')\}$  and
    mode: mode = invmode statM e and
    T: T = (resTy statM) and
    eq-accC-accC': accC = accC'
  by cases fastsimp+
from da obtain C where
    da-e:  $(\lfloor prg = G, cls = accC, lcl = L \rfloor) \vdash (dom (locals (store s0))) \gg \langle e \rangle_e \gg C$  and
    da-args:  $(\lfloor prg = G, cls = accC, lcl = L \rfloor) \vdash nrm\ C \gg \langle args \rangle_l \gg E$ 
  by cases simp
from eval eq-accC-accC' obtain a s1 vs s2 s3 s3' s4 invDeclC where
    evaln-e:  $G \vdash s0 \multimap e \multimap a \multimap n \rightarrow s1$  and
    evaln-args:  $G \vdash s1 \multimap args \multimap vs \multimap n \rightarrow s2$  and
    invDeclC: invDeclC = invocation-declclass
       $G\ mode\ (store\ s2)\ a\ statT\ (\lfloor name = mn, parTs = pTs' \rfloor)$  and
    s3: s3 = init-lvars G invDeclC  $(\lfloor name = mn, parTs = pTs' \rfloor)\ mode\ a\ vs\ s2$  and
    check: s3' = check-method-access G
       $accC'\ statT\ mode\ (\lfloor name = mn, parTs = pTs' \rfloor)\ a\ s3$  and
    evaln-methd:
       $G \vdash s3' \multimap Methd\ invDeclC\ (\lfloor name = mn, parTs = pTs' \rfloor) \multimap v \multimap n \rightarrow s4$  and
    s5: s5 = (set-lvars (locals (store s2))) s4
  using normal-s0 by (auto elim: evaln-elim-cases)

from evaln-e
have eval-e:  $G \vdash s0 \multimap e \multimap a \rightarrow s1$ 
by (rule evaln-eval)

from eval-e - wt-e wf
have s1-no-return: abrupt s1  $\neq Some\ (Jump\ Ret)$ 
by (rule eval-expression-no-jump
  [where ?Env =  $(\lfloor prg = G, cls = accC, lcl = L \rfloor, simplified)$ ]
  (insert normal-s0, 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:  $\bigwedge v. (Q \leftarrow In1\ a)\ v\ s1\ Z$ 
by simp
obtain
  R:  $(R\ a)\ \lfloor vs \rfloor_l s2 Z$  and
  conf-s2:  $s2 :: \preceq(G, L)$  and
  s2-no-return: abrupt s2  $\neq Some\ (Jump\ Ret)$ 
proof (cases normal s1)
  case True
obtain E' where
    da-args':
       $(\lfloor prg = G, cls = accC, lcl = L \rfloor) \vdash dom\ (locals\ (store\ s1)) \gg \langle args \rangle_l \gg E'$ 
  proof –
    from evaln-e wt-e da-e wf True
    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
with valid-args Q valid-A conf-s1 evaln-args wt-args
obtain (R a) [vs]l s2 Z s2::≼(G,L)
  by (rule validE)
moreover
from evaln-args
have e: G ⊢ s1 -args ≐> vs → s2
  by (rule evaln-eval)
from this s1-no-return wt-args wf
have abrupt s2 ≠ Some (Jump Ret)
  by (rule eval-expression-list-no-jump
    [where ?Env=(|prg=G,cls=accC,lcl=L|),simplified])
ultimately show ?thesis ..
next
case False
with valid-args Q valid-A conf-s1 evaln-args
obtain (R a) [vs]l s2 Z s2::≼(G,L)
  by (cases rule: validE) iprover+
moreover
from False evaln-args have s2=s1
  by auto
with s1-no-return have abrupt s2 ≠ Some (Jump Ret)
  by simp
ultimately show ?thesis ..
qed

obtain invC where
  invC: invC = invocation-class mode (store s2) a statT
  by simp
with s3
have invC': invC = (invocation-class mode (store s3) a statT)
  by (cases s2,cases mode) (auto simp add: init-lvars-def2 )
obtain l where
  l: l = locals (store s2)
  by simp

from eval wt da conf-s0 wf
have conf-s5: s5::≼(G, L)
  by (rule evaln-type-sound [elim-format]) simp
let PROP ?R = ∧ v.
  (R a ← In3 vs ∧.
    (λs. invDeclC = invocation-declclass G mode (store s) a statT
      (|name = mn, parTs = pTs'|) ∧
      invC = invocation-class mode (store s) a statT ∧
      l = locals (store s)) ;.
      init-lvars G invDeclC (|name = mn, parTs = pTs'|) mode a vs ∧.
      (λs. normal s → G ⊢ mode → invC ≼ statT)
    ) v s3' Z
  )
{
  assume abrupt-s3: ¬ normal s3
  have S [v]e s5 Z
  proof -
    from abrupt-s3 check have eq-s3'-s3: s3'=s3
      by (auto simp add: check-method-access-def Let-def)
    with R s3 invDeclC invC l abrupt-s3
    have R': PROP ?R
      by auto
    have conf-s3': s3'::≼(G, empty)

```

```

proof –
  from s2-no-return s3
  have abrupt s3 ≠ Some (Jump Ret)
    by (cases s2) (auto simp add: init-lvars-def2 split: split-if-asm)
  moreover
  obtain abr2 str2 where s2: s2=(abr2,str2)
    by (cases s2)
  from s3 s2 conf-s2 have (abrupt s3,str2):: $\preceq(G, L)$ 
    by (auto simp add: init-lvars-def2 split: split-if-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)  $\lfloor v \rfloor_e s4 Z$ 
  by (cases rule: validE) simp+
with s5 l show ?thesis
  by simp
qed
} note abrupt-s3-lemma = this

have S  $\lfloor v \rfloor_e s5 Z$ 
proof (cases normal s2)
  case False
  with s3 have abrupt-s3: ¬ normal s3
    by (cases s2) (simp add: init-lvars-def2)
  thus ?thesis
    by (rule abrupt-s3-lemma)
next
  case True
  note normal-s2 = this
  with evaln-args
  have normal-s1: normal s1
    by (rule evaln-no-abrupt)
  obtain E' where
    da-args':
    ( $\lfloor prg=G, cls=accC, lcl=L \rfloor \vdash \text{dom}(\text{locals}(\text{store } s1)) \gg \langle args \rangle_l \gg E'$ )
  proof –
    from evaln-e wt-e da-e wf normal-s1
    have  $\text{nrm } C \subseteq \text{dom}(\text{locals}(\text{store } s1))$ 
      by (cases rule: da-good-approx-evalnE) iprover
    with da-args show thesis
      by (rule da-weakenE) (rule that)
    qed
  from evaln-args
  have eval-args: G ⊢ s1 − args ≧ vs → s2
    by (rule evaln-eval)
  from evaln-e wt-e da-e conf-s0 wf
  have conf-a: G, store s1 ⊢ a :: ≧ RefT statT
    by (rule evaln-type-sound [elim-format]) (insert normal-s1,simp)
  with normal-s1 normal-s2 eval-args
  have conf-a-s2: G, store s2 ⊢ a :: ≧ RefT statT
    by (auto dest: eval-gext intro: conf-gext)
  from evaln-args wt-args da-args' conf-s1 wf
  have conf-args: list-all2 (conf G (store s2)) vs pTs
    by (rule evaln-type-sound [elim-format]) (insert normal-s2,simp)

```

```

from statM
obtain
  statM': (statDeclT, statM) ∈ mheads G accC statT (name=mn, parTs=pTs')
  and
  pTs-widen:  $G \vdash pTs [\preceq] pTs'$ 
  by (blast dest: max-spec2mheads)
show ?thesis
proof (cases normal s3)
  case False
  thus ?thesis
  by (rule abrupt-s3-lemma)
next
  case True
  note normal-s3 = this
  with s3 have notNull: mode = IntVir  $\longrightarrow$  a  $\neq$  Null
  by (cases s2) (auto simp add: init-lvars-def2)
  from conf-s2 conf-a-s2 wf notNull invC
  have dynT-prop:  $G \vdash mode \rightarrow invC \preceq statT$ 
  by (cases s2) (auto intro: DynT-propI)

  with wt-e statM' invC mode wf
  obtain dynM where
    dynM: dynlookup G statT invC (name=mn, parTs=pTs') = Some dynM and
    acc-dynM:  $G \vdash Methd$  (name=mn, parTs=pTs') dynM
      in invC dyn-accessible-from accC
    by (force dest!: call-access-ok)
  with invC' check eq-accC-accC'
  have eq-s3'-s3: s3' = s3
  by (auto simp add: check-method-access-def Let-def)

  with dynT-prop R s3 invDeclC invC l
  have R': PROP ?R
  by auto

from dynT-prop wf wt-e statM' mode invC invDeclC dynM
obtain
  dynM: dynlookup G statT invC (name=mn, parTs=pTs') = Some dynM and
  wf-dynM: wf-mdecl G invDeclC (name=mn, parTs=pTs'), mthd dynM) and
  dynM': methd G invDeclC (name=mn, parTs=pTs') = Some dynM and
  iscls-invDeclC: is-class G invDeclC and
  invDeclC': invDeclC = declclass dynM and
  invC-widen:  $G \vdash invC \preceq_C invDeclC$  and
  resTy-widen:  $G \vdash resTy$  dynM  $\preceq_{resTy}$  statM and
  is-static-eq: is-static dynM = is-static statM and
  involved-classes-prop:
    (if invmode statM e = IntVir
      then  $\forall statC. statT = ClassT$  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
        VNam v
         $\Rightarrow$  (table-of (lcls (mbody (mthd dynM))))
        (pars (mthd dynM)  $\vdash pTs'$ )) v

```



```

      | Res  $\Rightarrow$  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
    ( $\langle$ 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-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 intro: conf-gext dest: eval-gext)
  apply simp
  apply simp
  apply (simp add: invC)
  apply (simp add: invDeclC)
  apply (simp add: normal-s2)
  apply (cases s2, simp add: L' init-lvars-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
  ( $\langle$ 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
  wt-methd:
  ( $\langle$ prg=G,cls=invDeclC,lcl=L')
   $\vdash$  (Methd invDeclC ( $\langle$ name = mn, parTs = pTs') $\rangle$ )::-mthdT
  by (auto intro: wt.Methd)
obtain M where
  da-methd:
  ( $\langle$ prg=G,cls=invDeclC,lcl=L')
   $\vdash$  dom (locals (store s3'))
   $\gg$  (Methd invDeclC ( $\langle$ name=mn,parTs=pTs') $\rangle$ ) $\gg_e$  M
proof -
  from wf-dynM
  obtain M' where
    da-body:
    ( $\langle$ prg=G, cls=invDeclC
      ,lcl=callee-lcl invDeclC ( $\langle$ name = mn, parTs = pTs') $\rangle$  (mthd dynM)
     $\rangle$   $\vdash$  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
    ( $\langle$ 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-def)
    also from pTs-widen
    have ... = length pTs'
      by (simp add: widens-def list-all2-def)
    also from wf-dynM
    have ... = length (pars (mthd dynM))
      by (simp add: wf-mdecl-def wf-mhead-def)
    finally show ?thesis ..
  qed
moreover note s3 dynM' is-static-eq normal-s2 mode
ultimately
have parameters (mthd dynM) = dom (locals (store s3))
  using dom-locals-init-lvars
  [of mthd dynM G invDeclC ⟨name=mn,parTs=pTs'⟩ vs e a s2]
  by simp
thus ?thesis using eq-s3'-s3 by simp
qed
ultimately obtain M2 where
  da:
  ⟨prg=G, cls=invDeclC,lcl=L'⟩
  ⊢ dom (locals (store s3')) » ⟨stmt (mbody (mthd dynM))⟩ M2 and
  M2: nrm M' ⊆ nrm M2
  by (rule da-weakenE)
from res M2 have Result ∈ nrm M2
  by blast
moreover from wf-dynM
have jumpNestingOkS {Ret} (stmt (mbody (mthd dynM)))
  by (rule wf-mdeclE)
ultimately
obtain M3 where
  ⟨prg=G, cls=invDeclC,lcl=L'⟩ ⊢ dom (locals (store s3'))
    » ⟨Body (declclass dynM) (stmt (mbody (mthd dynM)))⟩ M3
  using da
  by (iprover intro: da.Body assigned.select-convs)
from - this [simplified]
show thesis
  by (rule da.Methd [simplified,elim-format])
    (auto intro: dynM' that)
qed
from valid-methd R' valid-A conf-s3' evaln-methd wt-methd da-methd
have (set-lvars l .; S) [v]e s4 Z
  by (cases rule: validE) iprover+
with s5 l show ?thesis
  by simp
qed
qed

```

```

  with conf-s5 show ?thesis by iprover
qed
qed
next
case (Methd A P Q ms)
note valid-body =  $\langle G, A \cup \{\{P\} \text{ Methd} \multimap \{Q\} \mid ms\} \models \{\{P\} \text{ body } G \multimap \{Q\} \mid ms\} \rangle$ 
show  $G, A \models \{\{P\} \text{ Methd} \multimap \{Q\} \mid ms\}$ 
  by (rule Methd-sound) (rule Methd.hyps)
next
case (Body A P D Q c R)
note valid-init =  $\langle G, A \models \{\{ \text{Normal } P \} \text{ .Init } D. \{Q\} \} \rangle$ 
note valid-c =  $\langle G, A \models \{\{Q\} \text{ .c.} \}$ 
   $\{\lambda s.. \text{abupd (absorb Ret) .; } R \leftarrow \text{In1 (the (locals s Result))}\} \rangle$ 
show  $G, A \models \{\{ \text{Normal } P \} \text{ Body } D \text{ c} \multimap \{R\} \}$ 
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:  $(\text{prg} = G, \text{cls} = \text{accC}, \text{lcl} = L) \vdash \text{Body } D \text{ c} :: - T$ 
  assume da:  $(\text{prg} = G, \text{cls} = \text{accC}, \text{lcl} = L) \vdash \text{dom (locals (store s0))} \gg \langle \text{Body } D \text{ c} \rangle_e \gg E$ 
  assume eval:  $G \vdash s0 \multimap \text{Body } D \text{ c} \multimap v \multimap n \rightarrow s4$ 
  assume P:  $(\text{Normal } P) \text{ Y s0 Z}$ 
  show  $R \text{ [v]}_e \text{ s4 Z} \wedge s4 :: \preceq (G, L)$ 
proof -
  from wt obtain
    iscls-D: is-class G D and
    wt-init:  $(\text{prg} = G, \text{cls} = \text{accC}, \text{lcl} = L) \vdash \text{Init } D :: \checkmark$  and
    wt-c:  $(\text{prg} = G, \text{cls} = \text{accC}, \text{lcl} = L) \vdash c :: \checkmark$ 
  by cases auto
  obtain I where
    da-init:  $(\text{prg} = G, \text{cls} = \text{accC}, \text{lcl} = L) \vdash \text{dom (locals (store s0))} \gg \langle \text{Init } D \rangle_s \gg I$ 
  by (auto intro: da-Init [simplified] assigned.select-convs)
  from da obtain C where
    da-c:  $(\text{prg} = G, \text{cls} = \text{accC}, \text{lcl} = L) \vdash (\text{dom (locals (store s0))}) \gg \langle c \rangle_s \gg C$  and
    jmpOk:  $\text{jumpNestingOkS } \{ \text{Ret} \} \text{ c}$ 
  by cases simp
  from eval obtain s1 s2 s3 where
    eval-init:  $G \vdash s0 \multimap \text{Init } D \multimap n \rightarrow s1$  and
    eval-c:  $G \vdash s1 \multimap c \multimap n \rightarrow s2$  and
    v:  $v = \text{the (locals (store s2) Result)}$  and
    s3:  $s3 = (\text{if } \exists l. \text{abrupt } s2 = \text{Some (Jump (Break l))} \vee$ 
       $\text{abrupt } s2 = \text{Some (Jump (Cont l))}$ 
       $\text{then abupd } (\lambda x. \text{Some (Error CrossMethodJump)}) \text{ s2 else s2})$  and
    s4:  $s4 = \text{abupd (absorb Ret) s3}$ 
  using normal-s0 by (fastsimp elim: evaln-elim-cases)
  obtain C' where
    da-c':  $(\text{prg} = G, \text{cls} = \text{accC}, \text{lcl} = L) \vdash (\text{dom (locals (store s1))}) \gg \langle c \rangle_s \gg C'$ 
  proof -
    from eval-init
    have  $(\text{dom (locals (store s0))}) \subseteq (\text{dom (locals (store s1))})$ 
      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:  $(\lambda s.. \text{abupd (absorb Ret) .; } R \leftarrow \text{In1 (the (locals s Result))})$ 

```

```

    ◇ s2 Z
  by (rule validE)
have s3=s2
proof -
  from eval-init [THEN evaln-eval] wf
  have s1-no-jmp:  $\bigwedge j. \text{abrupt } s1 \neq \text{Some } (\text{Jump } j)$ 
    by - (rule eval-statement-no-jump [OF - - wt-init],
          insert normal-s0, auto)
  from eval-c [THEN evaln-eval] - wt-c wf
  have  $\bigwedge j. \text{abrupt } s2 = \text{Some } (\text{Jump } j) \implies j = \text{Ret}$ 
    by (rule jumpNestingOk-evalE) (auto intro: jmpOk simp add: s1-no-jmp)
  moreover note s3
  ultimately show ?thesis
    by (force split: split-if)
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  $G, A \models :: \{ \{ \text{Normal } (P \leftarrow [\ ]_l) \} \} \dot{\supset} \{ P \} \}$ 
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:  $G \vdash s0 - [\ ] \dot{\supset} vs - n \rightarrow s1$ 
  assume P:  $(\text{Normal } (P \leftarrow [\ ]_l)) \ Y \ s0 \ Z$ 
  show  $P \ [vs]_l \ s1 \ Z \wedge s1::\preceq(G, L)$ 
  proof -
    from eval obtain vs=[ ] s1=s0
    using normal-s0 by (auto elim: evaln-elim-cases)
    with P conf-s0 show ?thesis
      by simp
  qed
qed
next
case (Cons A P e Q es R)
note valid-e =  $\langle G, A \models :: \{ \{ \text{Normal } P \} \} e - \supset \{ Q \} \rangle$ 
have valid-es:  $\bigwedge v. G, A \models :: \{ \{ Q \leftarrow [v]_e \} \} es \dot{\supset} \{ \lambda \text{Vals:vs}.. R \leftarrow [(v \# vs)]_l \} \}$ 
  using Cons.hyps by simp
show  $G, A \models :: \{ \{ \text{Normal } P \} \} e \# es \dot{\supset} \{ R \} \}$ 
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:  $(\text{prg} = G, \text{cls} = \text{accC}, \text{lcl} = L) \vdash e \# es::\dot{=} T$ 
  assume da:  $(\text{prg} = G, \text{cls} = \text{accC}, \text{lcl} = L) \vdash \text{dom } (\text{locals } (\text{store } s0)) \gg \langle e \# es \rangle_l \gg E$ 
  assume eval:  $G \vdash s0 - e \# es \dot{\supset} v - n \rightarrow s2$ 
  assume P:  $(\text{Normal } P) \ Y \ s0 \ Z$ 
  show  $R \ [v]_l \ s2 \ Z \wedge s2::\preceq(G, L)$ 
  proof -

```

```

from wt obtain eT esT where
  wt-e:  $(\text{prg}=G, \text{cls}=\text{acc}C, \text{lcl}=L) \vdash e :: -eT$  and
  wt-es:  $(\text{prg}=G, \text{cls}=\text{acc}C, \text{lcl}=L) \vdash es :: \dot{=} esT$ 
by cases simp
from da obtain E1 where
  da-e:  $(\text{prg}=G, \text{cls}=\text{acc}C, \text{lcl}=L) \vdash (\text{dom } (\text{locals } (\text{store } s0))) \gg \langle e \rangle_e \gg E1$  and
  da-es:  $(\text{prg}=G, \text{cls}=\text{acc}C, \text{lcl}=L) \vdash \text{nrm } E1 \gg \langle es \rangle_l \gg E$ 
by cases simp
from eval obtain s1 ve vs where
  eval-e:  $G \vdash s0 -e -\succ ve -n \rightarrow s1$  and
  eval-es:  $G \vdash s1 -es \dot{=} \succ vs -n \rightarrow s2$  and
  v:  $v = ve \# vs$ 
using normal-s0 by (fastsimp elim: evaln-elim-cases)
from valid-e P valid-A conf-s0 eval-e wt-e da-e
obtain Q:  $Q \lfloor ve \rfloor_e s1 Z$  and conf-s1:  $s1 :: \preceq(G, L)$ 
by (rule validE)
from Q have Q':  $\bigwedge v. (Q \leftarrow \lfloor ve \rfloor_e) v s1 Z$ 
by simp
have  $(\lambda \text{Vals:vs}.. R \leftarrow \lfloor (ve \# vs) \rfloor_l) \lfloor vs \rfloor_l s2 Z$ 
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  $R \lfloor v \rfloor_l s2 Z$ 
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 (fastsimp elim: evaln-elim-cases)
    with P conf-s0 show ?thesis
    by simp
qed
qed
next
case (Expr A P e Q)
note valid-e =  $\langle G, A \mid \vdash :: \{ \{ Normal P \} e \multimap \{ Q \leftarrow \Diamond \} \} \rangle$ 
show  $G, A \mid \vdash :: \{ \{ Normal P \} . Expr e . \{ Q \} \}$ 
proof (rule valid-stmt-NormalI)
  fix n s0 L accC C s1 Y Z
  assume valid-A:  $\forall t \in A. G \mid \vdash n :: t$ 
  assume conf-s0:  $s0 :: \preceq (G, L)$ 
  assume normal-s0: normal s0
  assume wt:  $(\mid prg = G, cls = accC, lcl = L) \vdash Expr e :: \checkmark$ 
  assume da:  $(\mid prg = G, cls = accC, lcl = L) \vdash dom (locals (store s0)) \gg \langle Expr e \rangle_s \gg C$ 
  assume eval:  $G \vdash s0 - Expr e - n \rightarrow s1$ 
  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:  $(\mid prg = G, cls = accC, lcl = L) \vdash e :: -eT$ 
    by cases simp
    from da obtain E where
      da-e:  $(\mid prg = G, cls = accC, lcl = L) \vdash dom (locals (store s0)) \gg \langle e \rangle_e \gg E$ 
    by cases simp
    from eval obtain v where
      eval-e:  $G \vdash s0 - e \multimap v - n \rightarrow s1$ 
    using normal-s0 by (fastsimp elim: evaln-elim-cases)
    from valid-e P valid-A conf-s0 eval-e wt-e da-e
    obtain Q:  $(Q \leftarrow \Diamond) \lfloor v \rfloor_e s1 Z$  and  $s1 :: \preceq (G, L)$ 
    by (rule validE)
    thus ?thesis by simp
  qed
qed
qed
next
case (Lab A P c l Q)
note valid-c =  $\langle G, A \mid \vdash :: \{ \{ Normal P \} . c . \{ abupd (absorb l) . ; Q \} \} \rangle$ 
show  $G, A \mid \vdash :: \{ \{ Normal P \} . l . c . \{ Q \} \}$ 
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:  $(\mid prg = G, cls = accC, lcl = L) \vdash l . c :: \checkmark$ 
  assume da:  $(\mid prg = G, cls = accC, lcl = L) \vdash dom (locals (store s0)) \gg \langle l . c \rangle_s \gg C$ 
  assume eval:  $G \vdash s0 - l . c - n \rightarrow s2$ 
  assume P:  $(Normal P) Y s0 Z$ 
  show  $Q \Diamond s2 Z \wedge s2 :: \preceq (G, L)$ 
  proof -
    from wt obtain
      wt-c:  $(\mid prg = G, cls = accC, lcl = L) \vdash c :: \checkmark$ 
    by cases simp
    from da obtain E where
      da-c:  $(\mid prg = G, cls = accC, lcl = L) \vdash dom (locals (store s0)) \gg \langle c \rangle_s \gg E$ 
    by cases simp
    from eval obtain s1 where
      eval-c:  $G \vdash s0 - c - n \rightarrow s1$  and

```

```

    s2: s2 = abupd (absorb l) s1
    using normal-s0 by (fastsimp elim: evaln-elim-cases)
  from valid-c P valid-A conf-s0 eval-c wt-c da-c
  obtain Q: (abupd (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 =  $\langle G, A \mid \vdash :: \{ \{ Normal\ P \} .c1. \{ Q \} \} \rangle$ 
note valid-c2 =  $\langle G, A \mid \vdash :: \{ \{ Q \} .c2. \{ R \} \} \rangle$ 
show  $G, A \mid \vdash :: \{ \{ Normal\ P \} .c1;; c2. \{ R \} \}$ 
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:  $(\mid prg = G, cls = accC, lcl = L) \vdash (c1;; c2)::\checkmark$ 
  assume da:  $(\mid prg = G, cls = accC, lcl = L) \vdash dom\ (locals\ (store\ s0)) \gg \langle c1;; c2 \rangle_s \gg C$ 
  assume eval:  $G \vdash s0 \multimap c1;; c2 \multimap n \rightarrow s2$ 
  assume P:  $(Normal\ P)\ Y\ s0\ Z$ 
  show  $R \Diamond s2\ Z \wedge s2::\preceq(G, L)$ 
proof -
  from eval obtain s1 where
    eval-c1:  $G \vdash s0 \multimap c1 \multimap n \rightarrow s1$  and
    eval-c2:  $G \vdash s1 \multimap c2 \multimap n \rightarrow s2$ 
  using normal-s0 by (fastsimp elim: evaln-elim-cases)
  from wt obtain
    wt-c1:  $(\mid prg = G, cls = accC, lcl = L) \vdash c1::\checkmark$  and
    wt-c2:  $(\mid prg = G, cls = accC, lcl = L) \vdash c2::\checkmark$ 
  by cases simp
  from da obtain C1 C2 where
    da-c1:  $(\mid prg = G, cls = accC, lcl = L) \vdash dom\ (locals\ (store\ s0)) \gg \langle c1 \rangle_s \gg C1$  and
    da-c2:  $(\mid prg = G, cls = accC, lcl = L) \vdash nrm\ C1 \gg \langle c2 \rangle_s \gg C2$ 
  by cases simp
  from valid-c1 P valid-A conf-s0 eval-c1 wt-c1 da-c1
  obtain Q:  $Q \Diamond s1\ Z$  and conf-s1:  $s1::\preceq(G, L)$ 
    by (rule validE)
  have  $R \Diamond s2\ Z$ 
  proof (cases normal s1)
    case True
    obtain C2' where
       $(\mid prg = G, cls = accC, lcl = L) \vdash dom\ (locals\ (store\ s1)) \gg \langle c2 \rangle_s \gg C2'$ 
    proof -
      from eval-c1 wt-c1 da-c1 wf True
      have  $nrm\ C1 \subseteq dom\ (locals\ (store\ s1))$ 
        by (cases rule: da-good-approx-evalnE) iprover
      with da-c2 show thesis
        by (rule da-weakenE) (rule that)
    qed
  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:: $\preceq(G, L)$ 
    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 =  $\langle G, A \mid \vdash :: \{ \{ \text{Normal } P \} \ e \multimap \{ P' \} \} \rangle$ 
  have valid-then-else:  $\bigwedge b. G, A \mid \vdash :: \{ \{ P' \leftarrow b \} \cdot (\text{if } b \text{ then } c1 \text{ else } c2) \cdot \{ Q \} \}$ 
    using If.hyps by simp
  show  $G, A \mid \vdash :: \{ \{ \text{Normal } P \} \cdot \text{If}(e) \ c1 \ \text{Else } c2 \cdot \{ Q \} \}$ 
  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:  $(\text{prg} = G, \text{cls} = \text{accC}, \text{lcl} = L) \vdash \text{If}(e) \ c1 \ \text{Else } c2 :: \checkmark$ 
    assume da:  $(\text{prg} = G, \text{cls} = \text{accC}, \text{lcl} = L) \vdash \text{dom}(\text{locals}(\text{store } s0)) \gg \langle \text{If}(e) \ c1 \ \text{Else } c2 \rangle_s \gg C$ 
    assume eval:  $G \vdash s0 \multimap \text{If}(e) \ c1 \ \text{Else } c2 \multimap n \rightarrow s2$ 
    assume P:  $(\text{Normal } P) \ Y \ s0 \ Z$ 
    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:  $G \vdash s1 \multimap (\text{if the-Bool } b \text{ then } c1 \text{ else } c2) \multimap n \rightarrow s2$ 
    using normal-s0 by (auto elim: evaln-elim-cases)
    from wt obtain
      wt-e:  $(\text{prg} = G, \text{cls} = \text{accC}, \text{lcl} = L) \vdash e :: \text{--PrimT Boolean}$  and
      wt-then-else:  $(\text{prg} = G, \text{cls} = \text{accC}, \text{lcl} = L) \vdash (\text{if the-Bool } b \text{ then } c1 \text{ else } c2) :: \checkmark$ 
    by cases (simp split: split-if)
    from da obtain E S where
      da-e:  $(\text{prg} = G, \text{cls} = \text{accC}, \text{lcl} = L) \vdash \text{dom}(\text{locals}(\text{store } s0)) \gg \langle e \rangle_e \gg E$  and
      da-then-else:
         $(\text{prg} = G, \text{cls} = \text{accC}, \text{lcl} = L) \vdash$ 
         $(\text{dom}(\text{locals}(\text{store } s0)) \cup \text{assigns-if}(\text{the-Bool } b) \ e)$ 
         $\gg \langle \text{if the-Bool } b \text{ then } c1 \text{ else } c2 \rangle_s \gg S$ 
    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':  $\bigwedge v. (P' \leftarrow \text{the-Bool } b) \ v \ s1 \ Z$ 
    by (cases normal s1) auto
    have  $Q \diamond s2 \ Z$ 
  proof (cases normal s1)
    case True
    have s0-s1:  $\text{dom}(\text{locals}(\text{store } s0))$ 
       $\cup \text{assigns-if}(\text{the-Bool } b) \ e \subseteq \text{dom}(\text{locals}(\text{store } s1))$ 
  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
   $(\text{prg}=G, \text{cls}=\text{acc } C, \text{lcl}=L)$ 
   $\vdash \text{dom}(\text{locals}(\text{store } s1)) \gg (\text{if } \text{the-Bool } b \text{ then } c1 \text{ else } c2)_s \gg S'$ 
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  $s2 :: \preceq(G, L)$ 
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 =  $\langle G, A \mid \vdash :: \{ \{P\} \ e \multimap \{P'\} \} \rangle$ 
note valid-c =  $\langle G, A \mid \vdash :: \{ \text{Normal } (P' \leftarrow \text{True}) \}$ 
  .c.
   $\{ \text{abupd } (\text{absorb } (\text{Cont } l)) \ .; P \} \rangle$ 
show  $G, A \mid \vdash :: \{ \{P\} \ .l \cdot \text{While}(e) \ .c \cdot \{P' \leftarrow \text{False} \downarrow = \diamond \} \}$ 
proof (rule valid-stmtI)
fix  $n \ s0 \ L \ \text{acc } C \ s3 \ Y \ Z$ 
assume valid-A:  $\forall t \in A. \ G \models n :: t$ 
assume conf-s0:  $s0 :: \preceq(G, L)$ 
assume wt:  $\text{normal } s0 \implies (\text{prg}=G, \text{cls}=\text{acc } C, \text{lcl}=L) \vdash l \cdot \text{While}(e) \ c :: \checkmark$ 
assume da:  $\text{normal } s0 \implies (\text{prg}=G, \text{cls}=\text{acc } C, \text{lcl}=L)$ 
   $\vdash \text{dom}(\text{locals}(\text{store } s0)) \gg (l \cdot \text{While}(e) \ c)_s \gg C$ 
assume eval:  $G \vdash s0 \multimap l \cdot \text{While}(e) \ c \multimap n \rightarrow s3$ 
assume P:  $P \ Y \ s0 \ Z$ 
show  $(P' \leftarrow \text{False} \downarrow = \diamond) \diamond s3 \ Z \wedge s3 :: \preceq(G, L)$ 
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  $(P' \leftarrow \text{False} \downarrow = \diamond) \diamond s3 \ Z$  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.
  {
    fix  $t \ s \ s' \ v$ 
    assume  $G \vdash s \multimap t \multimap n \rightarrow (v, s')$ 
    hence  $\bigwedge Y' \ T \ E.$ 
  }

```

```


$$\llbracket t = \langle l \cdot \text{While}(e) \ c \rangle_s; \forall t \in A. G \models n :: t; P \ Y' \ s \ Z; s :: \preceq (G, L);$$


$$\text{normal } s \implies (\text{prg} = G, \text{cls} = \text{acc} C, \text{lcl} = L) \vdash t :: T;$$


$$\text{normal } s \implies (\text{prg} = G, \text{cls} = \text{acc} C, \text{lcl} = L) \vdash \text{dom} (\text{locals} (\text{store } s)) \gg t \gg E$$


$$\llbracket \implies (P' \leftarrow \text{False} \downarrow = \Diamond) \ v \ s' \ Z$$

(is PROP ?Hyp n t s v s')
proof (induct)
  case (Loop s0' e' b n' s1' c' s2' l' s3' Y' T E)
  note while =  $\langle \langle l \cdot \text{While}(e') \ c' \rangle_s :: \text{term} \rangle = \langle l \cdot \text{While}(e) \ c \rangle_s$ 
  hence eqs: l' = l e' = e c' = c by simp-all
  note valid-A =  $\langle \forall t \in A. G \models n' :: t \rangle$ 
  note P =  $\langle P \ Y' \ (\text{Norm } s0') \ Z \rangle$ 
  note conf-s0' =  $\langle \text{Norm } s0' :: \preceq (G, L) \rangle$ 
  have wt:  $(\text{prg} = G, \text{cls} = \text{acc} C, \text{lcl} = L) \vdash \langle l \cdot \text{While}(e) \ c \rangle_s :: T$ 
    using Loop.premis eqs by simp
  have da:  $(\text{prg} = G, \text{cls} = \text{acc} C, \text{lcl} = L) \vdash$ 
     $\text{dom} (\text{locals} (\text{store} ((\text{Norm } s0') :: \text{state}))) \gg \langle l \cdot \text{While}(e) \ c \rangle_s \gg E$ 
    using Loop.premis eqs by simp
  have evaln-e:  $G \vdash \text{Norm } s0' - e - \succ b - n' \rightarrow s1'$ 
    using Loop.hyps eqs by simp
  show  $(P' \leftarrow \text{False} \downarrow = \Diamond) \ \Diamond \ s3' \ Z$ 
proof -
  from wt obtain
    wt-e:  $(\text{prg} = G, \text{cls} = \text{acc} C, \text{lcl} = L) \vdash e :: \text{PrimT Boolean}$  and
    wt-c:  $(\text{prg} = G, \text{cls} = \text{acc} C, \text{lcl} = L) \vdash c :: \checkmark$ 
    by cases (simp add: eqs)
  from da obtain E S where
    da-e:  $(\text{prg} = G, \text{cls} = \text{acc} C, \text{lcl} = L) \vdash$ 
       $\text{dom} (\text{locals} (\text{store} ((\text{Norm } s0') :: \text{state}))) \gg \langle e \rangle_e \gg E$  and
    da-c:  $(\text{prg} = G, \text{cls} = \text{acc} C, \text{lcl} = L) \vdash$ 
       $(\text{dom} (\text{locals} (\text{store} ((\text{Norm } s0') :: \text{state}))) \cup \text{assigns-if True } e) \gg \langle c \rangle_s \gg S$ 
    by cases (simp add: eqs)
  from evaln-e
  have eval-e:  $G \vdash \text{Norm } s0' - e - \succ b \rightarrow s1'$ 
    by (rule evaln-eval)
  from valid-e P valid-A conf-s0' evaln-e wt-e da-e
  obtain P':  $P' \ [b]_e \ s1' \ Z$  and conf-s1':  $s1' :: \preceq (G, L)$ 
    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: PROP ?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 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':
     $(\text{prg}=G, \text{cls}=\text{acc}C, \text{lcl}=L) \vdash (\text{dom } (\text{locals } (\text{store } s1'))) \gg \langle c \rangle_s \gg S'$ 
proof –
  note  $s0'-s1'$ 
  moreover
from eval-e normal-s1' wt-e
have  $\text{assigns-if True } e \subseteq \text{dom } (\text{locals } (\text{store } s1'))$ 
by (rule assigns-if-good-approx' [elim-format])
    (simp add: True)
ultimately
have  $\text{dom } (\text{locals } (\text{store } ((\text{Norm } s0')::\text{state})))$ 
       $\cup \text{assigns-if True } e \subseteq \text{dom } (\text{locals } (\text{store } s1'))$ 
by (rule Un-least)
with da-c show thesis
by (rule da-weakenE) (rule that)
qed
with 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  $P-s2': P \Diamond (\text{abupd } (\text{absorb } (\text{Cont } l)) s2') Z$ 
by simp
from conf-s2'
have conf-absorb:  $\text{abupd } (\text{absorb } (\text{Cont } l)) s2'::\preceq(G, L)$ 
by (cases s2') (auto intro: conforms-absorb)
moreover
obtain  $E'$  where
  da-while':
     $(\text{prg}=G, \text{cls}=\text{acc}C, \text{lcl}=L) \vdash$ 
       $\text{dom } (\text{locals } (\text{store } (\text{abupd } (\text{absorb } (\text{Cont } l)) s2')))$ 
       $\gg \langle l \cdot \text{While}(e) \ c \rangle_s \gg E'$ 
proof –
  note  $s0'-s1'$ 
also
from eval-c
have  $G \vdash s1' - c \rightarrow s2'$ 
by (rule evaln-eval)
hence  $\text{dom } (\text{locals } (\text{store } s1')) \subseteq \text{dom } (\text{locals } (\text{store } s2'))$ 
by (rule dom-locals-eval-mono-elim)
also
have  $\dots \subseteq \text{dom } (\text{locals } (\text{store } (\text{abupd } (\text{absorb } (\text{Cont } l)) s2')))$ 
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
from valid-A P-s2' conf-absorb wt da-while'
show  $(P' \leftarrow \text{False} \downarrow = \Diamond) \Diamond s3' Z$ 
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=( $\lfloor \text{prg} = G, \text{cls} = \text{acc } C, \text{lcl} = L \rfloor$ ), 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 abupd (absorb (Cont l)) s2'=s2'
        by (cases s1') (simp add: absorb-def)
      ultimately show ?thesis
        using eval-while abr
        by auto
    next
      case False
      with Loop.hyps show ?thesis by simp
    qed
  qed
with P' False show ?thesis
  by auto
qed
qed
next
  case (Abrupt abr s t' n' Y' T E)
  note t' =  $\langle t' = \langle l \cdot \text{While}(e) \ c \rangle_s \rangle$ 
  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  $(P' \leftarrow \text{False} \downarrow = \Diamond) \ (\text{arbitrary3 } t') \ (\text{Some } \text{abr}, s) \ Z$ 
  proof -
    have eval-e:
       $G \vdash (\text{Some } \text{abr}, s) - \langle e \rangle_e \succ - n' \rightarrow (\text{arbitrary3 } \langle e \rangle_e, (\text{Some } \text{abr}, s))$ 
      by auto
    from valid-e P valid-A conf eval-e
    have P' (arbitrary3  $\langle e \rangle_e$ ) (Some abr, s) Z
      by (cases rule: validE [where ?P=P]) simp+
    with t' show ?thesis
      by auto
  qed
qed simp-all
} note generalized=this

```

```

from eval - valid-A P conf-s0 wt da
have ( $P' \leftarrow \text{False} \downarrow = \Diamond$ )  $\Diamond$  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 (Jump A j P)
show  $G, A \models :: \{ \text{Normal } (\text{abupd } (\lambda a. \text{Some } (\text{Jump } j))) .; P \leftarrow \Diamond \} . \text{Jump } j. \{P\} \}$ 
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:  $G \vdash s0 \rightarrow \text{Jump } j \rightarrow n \rightarrow s1$ 
  assume P:  $(\text{Normal } (\text{abupd } (\lambda a. \text{Some } (\text{Jump } j))) .; P \leftarrow \Diamond) Y s0 Z$ 
  show  $P \Diamond s1 Z \wedge s1 :: \preceq(G, L)$ 
  proof –
    from eval obtain s where
      s:  $s0 = \text{Norm } s \ s1 = (\text{Some } (\text{Jump } j), s)$ 
    using normal-s0 by (auto elim: evaln-elim-cases)
    with P have  $P \Diamond s1 Z$ 
    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 (Throw A P e Q)
note valid-e =  $\langle G, A \models :: \{ \text{Normal } P \} e \rightarrow \{ \lambda \text{Val}:a. \text{abupd } (\text{throw } a) .; Q \leftarrow \Diamond \} \}$ 
show  $G, A \models :: \{ \text{Normal } P \} . \text{Throw } e. \{Q\} \}$ 
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:  $G \vdash s0 \multimap \text{Throw } e \multimap n \rightarrow s2$ 
assume P: (Normal P)  $Y s0 Z$ 
show  $Q \Diamond s2 Z \wedge s2 :: \preceq (G, L)$ 
proof –
  from eval obtain s1 a where
    eval-e:  $G \vdash s0 \multimap e \multimap \text{throw } a \multimap n \rightarrow s1$  and
    s2:  $s2 = \text{abupd } (\text{throw } a) s1$ 
    using normal-s0 by (auto elim: evaln-elim-cases)
  from wt obtain T where
    wt-e:  $(\text{prg} = G, \text{cls} = \text{acc } C, \text{lcl} = L) \vdash e :: - T$ 
    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 valid-e P valid-A conf-s0 eval-e wt-e da-e
obtain  $(\lambda \text{Val} : a. \text{abupd } (\text{throw } a) .; Q \leftarrow \Diamond) \lfloor a \rfloor_e s1 Z$ 
    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 =  $\langle G, A \models :: \{ \{ \text{Normal } P \} . c1 . \{ \text{SXAlloc } G \ Q \} \} \rangle$ 
note valid-c2 =  $\langle G, A \models :: \{ \{ Q \wedge . (\lambda s. G, s \vdash \text{catch } C) ;. \text{new-xcpt-var } vn \} . c2 . \{ R \} \} \rangle$ 
note Q-R =  $\langle (Q \wedge . (\lambda s. \neg G, s \vdash \text{catch } C)) \Rightarrow R \rangle$ 
show  $G, A \models :: \{ \{ \text{Normal } P \} . \text{Try } c1 \text{ Catch}(C \text{ vn}) c2 . \{ R \} \}$ 
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:  $(\text{prg} = G, \text{cls} = \text{acc } C, \text{lcl} = L) \vdash \text{Try } c1 \text{ Catch}(C \text{ vn}) c2 :: \checkmark$ 
  assume da:  $(\text{prg} = G, \text{cls} = \text{acc } C, \text{lcl} = L) \vdash \text{dom } (\text{locals } (\text{store } s0)) \gg \langle \text{Try } c1 \text{ Catch}(C \text{ vn}) c2 \rangle_s \gg E$ 
  assume eval:  $G \vdash s0 \multimap \text{Try } c1 \text{ Catch}(C \text{ vn}) c2 \multimap n \rightarrow s3$ 
  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:  $G \vdash s0 \multimap c1 \multimap n \rightarrow s1$  and
      sxalloc:  $G \vdash s1 \multimap \text{sxalloc} \rightarrow s2$  and
      s3: if  $G, s2 \vdash \text{catch } C$ 
        then  $G \vdash \text{new-xcpt-var } vn s2 \multimap c2 \multimap n \rightarrow s3$ 
        else  $s3 = s2$ 
      using normal-s0 by (fastsimp elim: evaln-elim-cases)
    from wt obtain
      wt-c1:  $(\text{prg} = G, \text{cls} = \text{acc } C, \text{lcl} = L) \vdash c1 :: \checkmark$  and
      wt-c2:  $(\text{prg} = G, \text{cls} = \text{acc } C, \text{lcl} = L (\text{VName } vn \mapsto \text{Class } C)) \vdash c2 :: \checkmark$ 
      by cases simp
    from da obtain C1 C2 where
      da-c1:  $(\text{prg} = G, \text{cls} = \text{acc } C, \text{lcl} = L) \vdash \text{dom } (\text{locals } (\text{store } s0)) \gg \langle c1 \rangle_s \gg C1$  and

```

```

da-c2: (⟦prg=G,cls=accC,lcl=L(VName vn↦Class C)⟧)
  ⊢ (dom (locals (store s0)) ∪ {VName vn}) »⟨c2⟩s C2
by cases simp
from valid-c1 P valid-A conf-s0 eval-c1 wt-c1 da-c1
obtain sxQ: (SXAlloc G Q) ◇ s1 Z and conf-s1: s1::≼(G,L)
  by (rule validE)
from xalloc sxQ
have Q: Q ◇ s2 Z
  by auto
have R ◇ s3 Z
proof (cases ∃ x. abrupt s1 = Some (Xcpt x))
case False
from xalloc wf
have s2=s1
  by (rule xalloc-type-sound [elim-format])
  (insert False, auto split: option.splits abrupt.splits )
with False
have no-catch: ¬ G,s2⊢catch C
  by (simp add: catch-def)
moreover
from no-catch s3
have s3=s2
  by simp
ultimately show ?thesis
  using Q Q-R by simp
next
case True
note exception-s1 = this
show ?thesis
proof (cases G,s2⊢catch C)
case False
with s3
have s3=s2
  by simp
with False Q Q-R show ?thesis
  by simp
next
case True
with s3 have eval-c2: G⊢new-xcpt-var vn s2 -c2-n→ s3
  by simp
from conf-s1 xalloc wf
have conf-s2: s2::≼(G, L)
  by (auto dest: xalloc-type-sound
    split: option.splits abrupt.splits)
from exception-s1 xalloc wf
obtain a
  where xcpt-s2: abrupt s2 = Some (Xcpt (Loc a))
  by (auto dest!: xalloc-type-sound
    split: option.splits abrupt.splits)
with True
have G⊢obj-ty (the (globs (store s2) (Heap a)))≼Class C
  by (cases s2) simp
with xcpt-s2 conf-s2 wf
have conf-new-xcpt: new-xcpt-var vn s2 ::≼(G, L(VName vn↦Class C))
  by (auto dest: Try-lemma)
obtain C2' where
da-c2':
  (⟦prg=G,cls=accC,lcl=L(VName vn↦Class C)⟧)
  ⊢ (dom (locals (store (new-xcpt-var vn s2)))) »⟨c2⟩s C2'

```

```

proof –
  have ( $\text{dom } (\text{locals } (\text{store } s0)) \cup \{VName\ vn\}$ )
     $\subseteq \text{dom } (\text{locals } (\text{store } (\text{new-xcpt-var } vn\ s2)))$ 
  proof –
    from eval-c1
    have  $\text{dom } (\text{locals } (\text{store } s0))$ 
       $\subseteq \text{dom } (\text{locals } (\text{store } s1))$ 
      by (rule dom-locals-evaln-mono-elim)
    also
    from sxalloc
    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  $\{VName\ vn\} \subseteq \dots$ 
      by (cases s2) simp
    ultimately show ?thesis
      by (rule Un-least)
  qed
  with da-c2 show thesis
    by (rule da-weakenE) (rule that)
  qed
from Q eval-c2 True
have ( $Q \wedge (\lambda s. G, s \vdash \text{catch } C) ; \text{new-xcpt-var } vn$ )
   $\Diamond (\text{new-xcpt-var } vn\ s2)\ Z$ 
  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 =  $\langle G, A \mid \vdash :: \{ \{Normal\ P\} .c1. \{Q\} \} \rangle$ 
have valid-c2:  $\bigwedge \text{abr. } G, A \mid \vdash :: \{ \{Q \wedge (\lambda s. \text{abr} = \text{fst } s) ; \text{abupd } (\lambda x. \text{None})\} .c2. \{ \text{abupd } (\text{abrupt-if } (\text{abr} \neq \text{None}) \text{abr}) ; R \} \}$ 
  using Fin.hyps by simp
show  $G, A \mid \vdash :: \{ \{Normal\ P\} .c1\ \text{Finally}\ c2. \{R\} \}$ 
proof (rule valid-stmt-NormalI)
  fix  $n\ s0\ L\ \text{accC}\ E\ s3\ Y\ Z$ 
  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:  $(\text{prg} = G, \text{cls} = \text{accC}, \text{lcl} = L) \vdash c1\ \text{Finally}\ c2 :: \checkmark$ 
  assume da:  $(\text{prg} = G, \text{cls} = \text{accC}, \text{lcl} = L)$ 
     $\vdash \text{dom } (\text{locals } (\text{store } s0)) \gg \langle c1\ \text{Finally}\ c2 \rangle_s \gg E$ 
  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:  $G \vdash s0 \rightarrow c1 \rightarrow n \rightarrow (abr1, s1)$  and
  eval-c2:  $G \vdash Norm\ s1 \rightarrow c2 \rightarrow n \rightarrow s2$  and
  s3: s3 = (if  $\exists err. abr1 = Some\ (Error\ err)$ 
    then (abr1, s1)
    else abupd (abrupt-if (abr1  $\neq None$ ) abr1) s2)
using normal-s0 by (fastsimp elim: evaln-elim-cases)
from wt obtain
  wt-c1:  $(\langle prg = G, cls = accC, lcl = L \rangle) \vdash c1 :: \checkmark$  and
  wt-c2:  $(\langle prg = G, cls = accC, lcl = L \rangle) \vdash c2 :: \checkmark$ 
by cases simp
from da obtain C1 C2 where
  da-c1:  $(\langle prg = G, cls = accC, lcl = L \rangle) \vdash dom\ (locals\ (store\ s0)) \gg \langle c1 \rangle_s \gg C1$  and
  da-c2:  $(\langle prg = G, cls = accC, lcl = L \rangle) \vdash dom\ (locals\ (store\ s0)) \gg \langle c2 \rangle_s \gg C2$ 
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':  $(Q \wedge. (\lambda s. abr1 = fst\ s) ;. abupd\ (\lambda x. None)) \Diamond (Norm\ s1)\ Z$ 
by auto
from eval-c1 wt-c1 da-c1 conf-s0 wf
have error-free (abr1, s1)
by (rule evaln-type-sound [elim-format]) (insert normal-s0, simp)
with s3 have s3': s3 = abupd (abrupt-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-if (abr1  $\neq None$ ) abr1) ;. 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:  $s3 :: \preceq (G, L)$ 
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  $wt: (\text{prg}=G, cls=accC, lcl=L) \vdash Init\ C::\checkmark$ 
assume  $da: (\text{prg}=G, cls=accC, lcl=L)$ 
 $\vdash dom\ (locals\ (store\ s0)) \gg \langle Init\ C \rangle_s \gg E$ 
assume  $eval: G \vdash s0 \rightarrow Init\ C \rightarrow n \rightarrow s3$ 
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  $(Init\ C\ c\ A\ P\ Q\ R)$ 
note  $c = \langle the\ (class\ G\ C) = c \rangle$ 
note  $valid-super =$ 
 $\langle G, A \models::\{ \{ Normal\ (P \wedge. Not \circ initd\ C ;. supd\ (init-class-obj\ G\ C)) \}$ 
 $.(if\ C = Object\ then\ Skip\ else\ Init\ (super\ c)).$ 
 $\{ Q \} \} \rangle$ 
have  $valid-init:$ 
 $\bigwedge l. G, A \models::\{ \{ Q \wedge. (\lambda s. l = locals\ (snd\ s)) ;. set-lvars\ empty \}$ 
 $.init\ c.$ 
 $\{ set-lvars\ l ;. R \} \}$ 
using  $Init.hyps$  by  $simp$ 
show  $G, A \models::\{ \{ Normal\ (P \wedge. Not \circ initd\ C) \} .Init\ C. \{ R \} \}$ 
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  $wt: (\text{prg}=G, cls=accC, lcl=L) \vdash Init\ C::\checkmark$ 
assume  $da: (\text{prg}=G, cls=accC, lcl=L)$ 
 $\vdash dom\ (locals\ (store\ s0)) \gg \langle Init\ C \rangle_s \gg E$ 
assume  $eval: G \vdash s0 \rightarrow Init\ C \rightarrow n \rightarrow s3$ 
assume  $P: (Normal\ (P \wedge. Not \circ initd\ C))\ Y\ s0\ Z$ 
show  $R \Diamond s3\ Z \wedge s3::\preceq(G, L)$ 
proof –
  from  $P$  have  $not-initd: \neg initd\ C\ (globs\ (store\ s0))$  by  $simp$ 
  with  $eval\ c$  obtain  $s1\ s2$  where
     $eval-super:$ 
 $G \vdash Norm\ ((init-class-obj\ G\ C)\ (store\ s0))$ 
 $\neg (if\ C = Object\ then\ Skip\ else\ Init\ (super\ c)) \rightarrow n \rightarrow s1$  and
     $eval-init: G \vdash (set-lvars\ empty)\ s1 \rightarrow init\ c \rightarrow n \rightarrow s2$  and
 $s3: s3 = (set-lvars\ (locals\ (store\ s1)))\ s2$ 
    using  $normal-s0$  by  $(auto\ elim!: evaln-elim-cases)$ 
  from  $wt\ c$  have
 $cls-C: class\ G\ C = Some\ c$ 
  by  $cases\ auto$ 
  from  $wf\ cls-C$  have
 $wt-super: (\text{prg}=G, cls=accC, lcl=L)$ 
 $\vdash (if\ C = Object\ then\ Skip\ else\ Init\ (super\ c))::\checkmark$ 
  by  $(cases\ C=Object)$ 

```

```

(auto dest: wf-prog-cdecl wf-cdecl-supD is-acc-classD)
obtain  $S$  where
  da-super:
    ( $\text{prg} = G, \text{cls} = \text{acc}C, \text{lcl} = L$ )
     $\vdash \text{dom} (\text{locals} (\text{store} ((\text{Norm} ((\text{init-class-obj } G \ C) (\text{store } s0))))::\text{state}))$ 
    »( $\text{if } C = \text{Object} \text{ then } \text{Skip} \text{ else } \text{Init } (\text{super } c)$ ) $_s$ »  $S$ 
proof (cases  $C = \text{Object}$ )
  case True
    with da-Skip show ?thesis
    using that by (auto intro: assigned.select-convs)
  next
    case False
    with da-Init show ?thesis
    by – (rule that, auto intro: assigned.select-convs)
qed
from normal-s0 conf-s0 wf cls-C not-inited
have conf-init-cls: ( $\text{Norm} ((\text{init-class-obj } G \ C) (\text{store } s0))))::\preceq(G, L)$ 
  by (auto intro: conforms-init-class-obj)
from  $P$ 
have  $P'$ : ( $\text{Normal} (P \wedge. \text{Not} \circ \text{initd } C ;. \text{supd } (\text{init-class-obj } G \ C)))$ 
   $Y (\text{Norm} ((\text{init-class-obj } G \ C) (\text{store } s0))) Z$ 
  by auto

from valid-super  $P'$  valid-A conf-init-cls eval-super wt-super da-super
obtain  $Q$ :  $Q \diamond s1 Z$  and conf-s1:  $s1::\preceq(G, L)$ 
  by (rule validE)

from cls-C wf have wt-init: ( $\text{prg} = G, \text{cls} = C, \text{lcl} = \text{empty}$ ) $\vdash (\text{init } c)::\checkmark$ 
  by (rule wf-prog-cdecl [THEN wf-cdecl-wt-init])
from cls-C wf obtain  $I$  where
  ( $\text{prg} = G, \text{cls} = C, \text{lcl} = \text{empty}$ ) $\vdash \{ \} \gg (\text{init } c)_s \gg I$ 
  by (rule wf-prog-cdecl [THEN wf-cdeclE,simplified]) blast

then obtain  $I'$  where
  da-init:
    ( $\text{prg} = G, \text{cls} = C, \text{lcl} = \text{empty}$ ) $\vdash \text{dom} (\text{locals} (\text{store} ((\text{set-lvars empty}) s1)))$ 
    »( $\text{init } c$ ) $_s$ »  $I'$ 
  by (rule da-weakenE) simp
have conf-s1-empty: ( $\text{set-lvars empty}$ )  $s1::\preceq(G, \text{empty})$ 
proof –
  from eval-super have
     $G \vdash \text{Norm} ((\text{init-class-obj } G \ C) (\text{store } s0))$ 
    –( $\text{if } C = \text{Object} \text{ then } \text{Skip} \text{ else } \text{Init } (\text{super } c)$ ) $\rightarrow s1$ 
    by (rule evaln-eval)
  from this wt-super wf
  have s1-no-ret:  $\bigwedge j. \text{abrupt } s1 \neq \text{Some } (\text{Jump } j)$ 
    by – (rule eval-statement-no-jump
      [where ?Env = ( $\text{prg} = G, \text{cls} = \text{acc}C, \text{lcl} = L$ )], auto split: split-if)
  with conf-s1
  show ?thesis
    by (cases  $s1$ ) (auto intro: conforms-set-locals)
qed

obtain  $l$  where  $l: l = \text{locals} (\text{store } s1)$ 
  by simp
with  $Q$ 
have  $Q'$ : ( $Q \wedge. (\lambda s. l = \text{locals } (\text{snd } s)) ;. \text{set-lvars empty}$ )
   $\diamond ((\text{set-lvars empty}) s1) Z$ 

```

```

    by auto
  from valid-init Q' valid-A conf-s1-empty eval-init wt-init da-init
  have (set-lvars l .; R)  $\Diamond$  s2 Z
    by (rule validE)
  with s3 l 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 (InsInitV A P c v Q)
show G,A $\models$ ::{ {Normal P} InsInitV c v $\multimap$  {Q} }
proof (rule valid-var-NormalI)
  fix s0 vf n s1 L Z
  assume normal s0
  moreover
  assume G $\vdash$ s0  $\neg$ InsInitV c v $\multimap$ vf $\neg$ n $\rightarrow$  s1
  ultimately have False
    by (cases s0) (simp add: evaln-InsInitV)
  thus Q [vf]v s1 Z  $\wedge$  s1:: $\preceq$ (G, L)..
qed
next
case (InsInitE A P c e Q)
show G,A $\models$ ::{ {Normal P} InsInitE c e $\multimap$  {Q} }
proof (rule valid-expr-NormalI)
  fix s0 v n s1 L Z
  assume normal s0
  moreover
  assume G $\vdash$ s0  $\neg$ InsInitE c e $\multimap$ v $\neg$ n $\rightarrow$  s1
  ultimately have False
    by (cases s0) (simp add: evaln-InsInitE)
  thus Q [v]e s1 Z  $\wedge$  s1:: $\preceq$ (G, L)..
qed
next
case (Callee A P l e Q)
show G,A $\models$ ::{ {Normal P} Callee l e $\multimap$  {Q} }
proof (rule valid-expr-NormalI)
  fix s0 v n s1 L Z
  assume normal s0
  moreover
  assume G $\vdash$ s0  $\neg$ Callee l e $\multimap$ v $\neg$ n $\rightarrow$  s1
  ultimately have False
    by (cases s0) (simp add: evaln-Callee)
  thus Q [v]e s1 Z  $\wedge$  s1:: $\preceq$ (G, L)..
qed
next
case (FinA A P a c Q)
show G,A $\models$ ::{ {Normal P} .FinA a c. {Q} }
proof (rule valid-stmt-NormalI)
  fix s0 v n s1 L Z
  assume normal s0
  moreover
  assume G $\vdash$ s0  $\neg$ FinA a c $\neg$ n $\rightarrow$  s1
  ultimately have 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:
  wf-prog  $G \implies G, (A::'a\ triple\ set) \vdash (ts::'a\ triple\ set) \implies G, A \models ts$ 
apply (subst ax-valids2-eq [symmetric])
apply assumption
apply (erule (1) ax-sound2)
done

```

```

lemma sound-valid2-lemma:
   $\llbracket \forall v\ n.\ Ball\ A\ (triple-valid2\ G\ n) \longrightarrow P\ v\ n;\ Ball\ A\ (triple-valid2\ G\ n) \rrbracket$ 
   $\implies P\ v\ n$ 
by blast

end

```



## Chapter 24

### AxCompl

### 63 Completeness proof for Axiomatic semantics of Java expressions and state-ments

theory *AxCompl* imports *AxSem* begin

design issues:

- proof structured by Most General Formulas (-i Thomas Kleymann)

set of not yet initialized classes

constdefs

*nyinitcls* :: *prog*  $\Rightarrow$  *state*  $\Rightarrow$  *qname* *set*  
*nyinitcls* *G* *s*  $\equiv$  { *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*], *standard*]

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*)

apply (unfold *nyinitcls-def*)

apply (*simp* (*no-asm*))

done

lemma *nyinitcls-abrupt-cong* [*simp*]: *nyinitcls* *G* (*f* *x*, *y*) = *nyinitcls* *G* (*x*, *y*)

apply (unfold *nyinitcls-def*)

apply (*simp* (*no-asm*))

done

lemma *nyinitcls-abupd-cong* [*simp*]:!!*s*. *nyinitcls* *G* (*abupd* *f* *s*) = *nyinitcls* *G* *s*

apply (unfold *nyinitcls-def*)

apply (*simp* (*no-asm-simp*) *only*: *split-tupled-all*)

apply (*simp* (*no-asm*))

done

lemma *card-nyinitcls-abrupt-congE* [*elim*!]:

*card* (*nyinitcls* *G* (*x*, *s*))  $\leq$  *n*  $\Longrightarrow$  *card* (*nyinitcls* *G* (*y*, *s*))  $\leq$  *n*

apply (unfold *nyinitcls-def*)

apply *auto*

done

lemma *nyinitcls-new-xcpt-var* [*simp*]:



```

nyinitcls G (new-xcpt-var vn s) = nyinitcls G s
apply (unfold nyinitcls-def)
apply (induct-tac s)
apply (simp (no-asm))
done

```

```

lemma nyinitcls-init-lvars [simp]:
  nyinitcls G ((init-lvars G C sig mode a' pvs) s) = nyinitcls G s
apply (induct-tac s)
apply (simp (no-asm) add: init-lvars-def2 split add: split-if)
done

```

```

lemma nyinitcls-emptyD:  $\llbracket \text{nyinitcls } G \text{ } s = \{\}; \text{is-class } G \text{ } C \rrbracket \implies \text{initd } C \text{ } s$ 
apply (unfold nyinitcls-def)
apply fast
done

```

```

lemma card-Suc-lemma:
   $\llbracket \text{card } (\text{insert } a \text{ } A) \leq \text{Suc } n; a \notin A; \text{finite } A \rrbracket \implies \text{card } A \leq n$ 
apply clarsimp
done

```

```

lemma nyinitcls-le-SucD:
   $\llbracket \text{card } (\text{nyinitcls } G \text{ } (x,s)) \leq \text{Suc } n; \neg \text{initd } C \text{ } (\text{globs } s); \text{class } G \text{ } C = \text{Some } y \rrbracket \implies$ 
   $\text{card } (\text{nyinitcls } G \text{ } (x, \text{init-class-obj } G \text{ } C \text{ } s)) \leq n$ 
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 in thin-rl)
apply (rule card-Suc-lemma [OF - - finite-nyinitcls])
apply (auto dest!: not-initdD elim!:
  simp add: nyinitcls-def initd-def split add: split-if-asm)
done

```

```

lemma initd-gext':  $\llbracket s \leq |s'|; \text{initd } C \text{ } (\text{globs } s) \rrbracket \implies \text{initd } C \text{ } (\text{globs } s')$ 
by (rule initd-gext)

```

```

lemma nyinitcls-gext:  $\text{snd } s \leq | \text{snd } s' \implies \text{nyinitcls } G \text{ } s' \subseteq \text{nyinitcls } G \text{ } s$ 
apply (unfold nyinitcls-def)
apply (force dest!: initd-gext')
done

```

```

lemma card-nyinitcls-gext:
   $\llbracket \text{snd } s \leq | \text{snd } s'; \text{card } (\text{nyinitcls } G \text{ } s) \leq n \rrbracket \implies \text{card } (\text{nyinitcls } G \text{ } s') \leq n$ 
apply (rule le-trans)
apply (rule card-mono)
apply (rule finite-nyinitcls)
apply (erule nyinitcls-gext)
apply assumption
done

```

**init-le****constdefs**

$init-le :: prog \Rightarrow nat \Rightarrow state \Rightarrow bool$  ( $\vdash init \leq$  [51,51] 50)  
 $G \vdash init \leq n \equiv \lambda s. card (nyinitcls\ G\ s) \leq n$

**lemma** *init-le-def2* [simp]:  $(G \vdash init \leq n) \ s = (card\ (nyinitcls\ G\ s) \leq n)$   
**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 consequ1*)  
**apply** *clarsimp*  
**apply** (*rule card-nyinitcls-bound*)  
**done**

**Most General Triples and Formulas****constdefs**

$remember-init-state :: state\ assn$  ( $\dot{=}$ )  
 $\dot{=} \equiv \lambda Y\ s\ Z. s = Z$

**lemma** *remember-init-state-def2* [simp]:  $\dot{=}\ Y = op =$   
**apply** (*unfold remember-init-state-def*)  
**apply** (*simp (no-asm)*)  
**done**

**consts**

$MGF :: [state\ assn, term, prog] \Rightarrow state\ triple\ (\{-\} \dashv\succ \{-\rightarrow\})$  [3,65,3] 62)  
 $MGFn :: [nat, term, prog] \Rightarrow state\ triple\ (\{=: \dashv\succ \{-\rightarrow\})$  [3,65,3] 62)

**defs**

*MGF-def*:

$\{P\} \ t \succ \{G \rightarrow\} \equiv \{P\} \ t \succ \{\lambda Y\ s' \ s. G \vdash s - t \rightarrow (Y, s')\}$

*MGFn-def*:

$\{=: n\} \ t \succ \{G \rightarrow\} \equiv \{\dot{=}\ \wedge. G \vdash init \leq n\} \ t \succ \{G \rightarrow\}$

**lemma** *MGF-valid*:  $wf-prog\ G \implies G, \{\dot{=}\} = \{\dot{=}\} \ t \succ \{G \rightarrow\}$   
**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]:

$$(\forall Y' Y s. Y = Y' \wedge P s \longrightarrow Q s) = (\forall s. P s \longrightarrow Q s)$$

**apply** *auto*

**done**

**lemma** *MGFn-def2*:

$$G, A \vdash \{=:n\} \ t \succ \{G \rightarrow\} = G, A \vdash \{\dot{=}\} \wedge. G \vdash \text{init} \leq n \\ t \succ \{\lambda Y s' s. G \vdash s - t \succ \rightarrow (Y, s')\}$$

**apply** (*unfold MGFn-def MGF-def*)

**apply** *fast*

**done**

**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 (no-asm-use) add: MGFn-def2 MGF-def*)

**apply** *safe*

**apply** (*erule-tac* [2] *All-init-leD*)

**apply** (*erule conseq1*)

**apply** *clarsimp*

**done**

**lemma** *MGFnD*:

$$G, (A::\text{state triple set}) \vdash \{=:n\} \ t \succ \{G \rightarrow\} \implies \\ G, A \vdash \{(\lambda Y' s' s. s' = s \wedge P s) \wedge. G \vdash \text{init} \leq n\} \\ t \succ \{(\lambda Y' s' s. G \vdash s - t \succ \rightarrow (Y', s') \wedge P s) \wedge. G \vdash \text{init} \leq n\}$$

**apply** (*unfold init-le-def*)

**apply** (*simp (no-asm-use) add: MGFn-def2*)

**apply** (*erule conseq12*)

**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*:  $G, A \vdash \{\text{Normal} \dot{=}\} \ t \succ \{G \rightarrow\} \implies$

$$G, (A::\text{state triple set}) \vdash \{\dot{=}\} \ t \succ \{G \rightarrow\}$$

**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*:

$$G, (A::\text{state triple set}) \vdash \{\dot{=}\} \ t \succ \{G \rightarrow\} \implies G, A \vdash \{\text{Normal} \dot{=}\} \ t \succ \{G \rightarrow\}$$

**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*:

$G, (A::\text{state triple set}) \vdash \{ \text{Normal}((\lambda Y' s' s. s'=s \wedge \text{normal } s) \wedge. G \vdash \text{init} \leq n) \} t \succ$   
 $\{ \lambda Y s' s. G \vdash s - t \succ \rightarrow (Y, s') \} \implies G, A \vdash \{ =:n \} t \succ \{ G \rightarrow \}$   
**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*:  
 $(\exists T L C. (\text{prg}=G, \text{cls}=C, \text{lcl}=L) \vdash t::T)$   
 $\longrightarrow G, (A::\text{state triple set}) \vdash \{ =:n \} t \succ \{ G \rightarrow \}$   
 $\implies G, A \vdash \{ =:n \} t \succ \{ G \rightarrow \}$   
**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*:  
 $(\forall T L C. (\text{prg}=G, \text{cls}=C, \text{lcl}=L) \vdash t::T)$   
 $\longrightarrow G, (A::\text{state triple set})$   
 $\vdash \{ \text{Normal}((\lambda Y' s' s. s'=s \wedge \text{normal } s) \wedge. G \vdash \text{init} \leq n) \wedge. (\lambda s. s::\preceq(G, L)) \}$   
 $t \succ$   
 $\{ \lambda Y s' s. G \vdash s - t \succ \rightarrow (Y, s') \}$   
 $\implies G, A \vdash \{ =:n \} t \succ \{ G \rightarrow \}$   
**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*:  
 $(\forall T L C B. (\text{prg}=G, \text{cls}=C, \text{lcl}=L) \vdash t::T)$   
 $\longrightarrow G, (A::\text{state triple set})$   
 $\vdash \{ \text{Normal}((\lambda Y' s' s. s'=s \wedge \text{normal } s) \wedge. G \vdash \text{init} \leq n) \wedge. (\lambda s. s::\preceq(G, L))$   
 $\wedge. (\lambda s. (\text{prg}=G, \text{cls}=C, \text{lcl}=L) \vdash \text{dom } (\text{locals } (\text{store } s)) \gg t \gg B) \}$   
 $t \succ$   
 $\{ \lambda Y s' s. G \vdash s - t \succ \rightarrow (Y, s') \}$   
 $\implies G, A \vdash \{ =:n \} t \succ \{ G \rightarrow \}$   
**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 spec, drule (1) mp)
apply (erule conseq12)
apply blast
done

```

## main lemmas

**lemma** *MGFn-Init*:

```

assumes mgf-hyp:  $\forall m. \text{Suc } m \leq n \longrightarrow (\forall t. G, A \vdash \{=:m\} t \succ \{G \rightarrow\})$ 
shows  $G, (A::\text{state triple set}) \vdash \{=:n\} \langle \text{Init } C \rangle_s \succ \{G \rightarrow\}$ 
proof (rule MGFn-free-wt [rule-format],elim exE,rule MGFn-NormalI)
  fix T L accC
  assume  $(\text{prg}=G, \text{cls}=\text{accC}, \text{lcl}=L) \vdash \langle \text{Init } C \rangle_s :: T$ 
  hence is-cls: is-class G C
  by cases simp
  show  $G, A \vdash \{ \text{Normal } ((\lambda Y' s' s. s' = s \wedge \text{normal } s) \wedge. G \vdash \text{init} \leq n) \}$ 
     $. \text{Init } C.$ 
     $\{ \lambda Y' s' s. G \vdash s - \langle \text{Init } C \rangle_s \rightarrow (Y, s') \}$ 
    (is  $G, A \vdash \{ \text{Normal } ?P \} . \text{Init } C. \{ ?R \}$ )
  proof (rule ax-cases [where ?C=initd C])
    show  $G, A \vdash \{ \text{Normal } ?P \wedge. \text{initd } C \} . \text{Init } C. \{ ?R \}$ 
    by (rule ax-derivs.Done [THEN conseq1]) (fastsimp intro: init-done)
  next
    have  $G, A \vdash \{ \text{Normal } (?P \wedge. \text{Not } \circ \text{initd } C) \} . \text{Init } C. \{ ?R \}$ 
    proof (cases n)
      case 0
      with is-cls
      show ?thesis
      by - (rule ax-impossible [THEN conseq1],fastsimp dest: nyinitcls-emptyD)
    next
      case (Suc m)
      with mgf-hyp have mgf-hyp':  $\bigwedge t. G, A \vdash \{=:m\} t \succ \{G \rightarrow\}$ 
      by simp
      from is-cls obtain c where c: the (class G C) = c
      by auto
      let ?Q =  $(\lambda Y' s' (x,s) .$ 
         $G \vdash (x, \text{init-class-obj } G C s)$ 
         $- (\text{if } C = \text{Object then Skip else Init (super (the (class } G C))) \rightarrow s'$ 
         $\wedge x = \text{None} \wedge \neg \text{initd } C (\text{globs } s)) \wedge. G \vdash \text{init} \leq m$ 
      from c
      show ?thesis
      proof (rule ax-derivs.Init [where ?Q=?Q])
        let ?P' =  $\text{Normal } ((\lambda Y' s' s. s' = \text{supd } (\text{init-class-obj } G C) s$ 
           $\wedge \text{normal } s \wedge \neg \text{initd } C s) \wedge. G \vdash \text{init} \leq m)$ 
        show  $G, A \vdash \{ \text{Normal } (?P \wedge. \text{Not } \circ \text{initd } C ;. \text{supd } (\text{init-class-obj } G C)) \}$ 
           $. (\text{if } C = \text{Object then Skip else Init (super } c)).$ 
           $\{ ?Q \}$ 
        proof (rule conseq1 [where ?P'=?P'])
          show  $G, A \vdash \{ ?P' \} . (\text{if } C = \text{Object then Skip else Init (super } c)). \{ ?Q \}$ 
          proof (cases C=Object)
            case True
            have  $G, A \vdash \{ ?P' \} . \text{Skip}. \{ ?Q \}$ 
            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  $G, A \vdash \{?P'\} . \text{Init} (\text{super } c). \{?Q\}$ 
    by (rule MGFnD' [THEN conseq12]) (fastsimp simp add: False c)
  with False show ?thesis
  by simp
qed
next
  from Suc is-cls
  show Normal ( $?P \wedge . \text{Not} \circ \text{initd } C ; . \text{supd} (\text{init-class-obj } G \ C)$ )
     $\Rightarrow ?P'$ 
    by (fastsimp elim: nyinitcls-le-SucD)
qed
next
  from mgf-hyp'
  show  $\forall l. G, A \vdash \{?Q \wedge . (\lambda s. l = \text{locals } (\text{snd } s)) ; . \text{set-lvars empty}\}$ 
    .init c.
    {set-lvars l ; ?R}
  apply (rule MGFnD' [THEN conseq12, THEN allI])
  apply (clarsimp simp add: split-paired-all)
  apply (rule eval.Init [OF c])
  apply (insert c)
  apply auto
done
qed
qed
thus  $G, A \vdash \{\text{Normal } ?P \wedge . \text{Not} \circ \text{initd } C\} . \text{Init } C. \{?R\}$ 
  by clarsimp
qed
qed
lemmas MGFn-InitD = MGFn-Init [THEN MGFnD, THEN ax-NormalD]

```

**lemma** *MGFn-Call*:

```

  assumes mgf-methds:
     $\forall C \text{ sig}. G, (A::\text{state triple set}) \vdash \{=:n\} \langle (\text{Methd } C \text{ sig}) \rangle_e \succ \{G \rightarrow\}$ 
  and mgf-e:  $G, A \vdash \{=:n\} \langle e \rangle_e \succ \{G \rightarrow\}$ 
  and mgf-ps:  $G, A \vdash \{=:n\} \langle ps \rangle_l \succ \{G \rightarrow\}$ 
  and wf: wf-prog G
  shows  $G, A \vdash \{=:n\} \langle \{accC, statT, mode\} e \cdot mn(\{pTs'\} ps) \rangle_e \succ \{G \rightarrow\}$ 
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:  $(\text{prg}=G, \text{cls}=accC, \text{lcl}=L) \vdash e :: -\text{RefT } statT$  and
    wt-args:  $(\text{prg}=G, \text{cls}=accC, \text{lcl}=L) \vdash ps :: \dot{=} pTs$  and
    statM:  $\text{max-spec } G \text{ accC } statT (\text{name}=mn, \text{parTs}=pTs)$ 
      =  $\{((\text{statDeclT}, statM), pTs')\}$  and
    mode:  $\text{mode} = \text{invmode } statM \ e$  and
    T:  $T = \text{Inl } (\text{resTy } statM)$  and
    eq-accC-accC':  $accC = accC'$ 
  by cases fastsimp+
let ?Q =  $(\lambda Y \ s1 \ (x, s) . x = \text{None} \wedge$ 
     $(\exists a. G \vdash \text{Norm } s -e- \succ a \rightarrow s1 \wedge$ 
     $(\text{normal } s1 \rightarrow G, \text{store } s1 \vdash a :: \preceq \text{RefT } statT))$ 

```

```

    ∧ Y = In1 a) ∧
    (∃ P. normal s1
      → (⟦prg=G,cls=accC',lcl=L⟧ ⊢ dom (locals (store s1)) » ⟨ps⟩l » P))
    ∧. G ⊢ init ≤ n ∧. (λ s. s :: ≤(G, L)) :: state assn
let ?R = λ a. ((λ Y (x2,s2) (x,s) . x = None ∧
    (∃ s1 pvs. G ⊢ Norm s -e- > a → s1 ∧
      (normal s1 → G, store s1 ⊢ a :: ≤ RefT statT) ∧
      Y = [pvs]l ∧ G ⊢ s1 -ps- > pvs → (x2,s2)))
    ∧. G ⊢ init ≤ n ∧. (λ s. s :: ≤(G, L)) :: state assn

show G, A ⊢ {Normal ((λ Y' s' s. s' = s ∧ abrupt s = None) ∧. G ⊢ init ≤ n ∧.
    (λ s. s :: ≤(G, L)) ∧.
    (λ s. (⟦prg=G,cls=accC',lcl=L⟧ ⊢ dom (locals (store s))
      » ⟨{accC,statT,mode} e.mn( {pTs'}ps)⟩e » E)))}
    {accC,statT,mode} e.mn( {pTs'}ps) ->
    {λ Y s' s. ∃ v. Y = [v]e ∧
      G ⊢ s -{accC,statT,mode} e.mn( {pTs'}ps) -> v → s'}
    (is G, A ⊢ {Normal ?P} {accC,statT,mode} e.mn( {pTs'}ps) -> {?S})
proof (rule ax-derivs.Call [where ?Q=?Q and ?R=?R])
from mgf-e
show G, A ⊢ {Normal ?P} e -> {?Q}
proof (rule MGFnD' [THEN conseq12], clarsimp)
fix s0 s1 a
assume conf-s0: Norm s0 :: ≤(G, L)
assume da: (⟦prg=G,cls=accC',lcl=L⟧ ⊢
    dom (locals s0) » ⟨{accC,statT,mode} e.mn( {pTs'}ps)⟩e » E
assume eval-e: G ⊢ Norm s0 -e- > a → s1
show (abrupt s1 = None → G, store s1 ⊢ a :: ≤ RefT statT) ∧
    (abrupt s1 = None →
      (∃ P. (⟦prg=G,cls=accC',lcl=L⟧ ⊢ dom (locals (store s1)) » ⟨ps⟩l » P))
      ∧ s1 :: ≤(G, L)
proof -
from da obtain C where
    da-e: (⟦prg=G,cls=accC,lcl=L⟧ ⊢
      dom (locals (store ((Norm s0)::state))) » ⟨e⟩e » C and
    da-ps: (⟦prg=G,cls=accC,lcl=L⟧ ⊢ nrm C » ⟨ps⟩l » E
    by cases (simp add: eq-accC-accC')
from eval-e conf-s0 wt-e da-e wf
obtain (abrupt s1 = None → G, store s1 ⊢ a :: ≤ RefT statT)
    and s1 :: ≤(G, L)
    by (rule eval-type-soundE) simp
moreover
    {
      assume normal-s1: normal s1
      have ∃ P. (⟦prg=G,cls=accC,lcl=L⟧ ⊢ dom (locals (store s1)) » ⟨ps⟩l » P
      proof -
        from eval-e wt-e da-e wf normal-s1
        have nrm C ⊆ dom (locals (store s1))
        by (cases rule: da-good-approxE') iprover
        with da-ps show ?thesis
        by (rule da-weakenE) iprover
      qed
    }
    ultimately show ?thesis
    using eq-accC-accC' by simp
  qed
qed
qed
next
show ∀ a. G, A ⊢ {?Q ← In1 a} ps ≡ > {?R a} (is ∀ a. ?PS a)

```

```

proof
  fix  $a$ 
  show  $?PS\ a$ 
  proof (rule  $MGFnD'$  [OF  $mgf-ps$ , THEN  $conseq12$ ],
    clarsimp simp add: eq-accC-accC' [symmetric])
    fix  $s0\ s1\ s2\ vs$ 
    assume  $conf-s1: s1::\preceq(G, L)$ 
    assume  $eval-e: G \vdash Norm\ s0 \ -e-\succ a \rightarrow s1$ 
    assume  $conf-a: abrupt\ s1 = None \longrightarrow G, store\ s1 \vdash a::\preceq RefT\ statT$ 
    assume  $eval-ps: G \vdash s1 \ -ps-\succ vs \rightarrow s2$ 
    assume  $da-ps: abrupt\ s1 = None \longrightarrow$ 
      ( $\exists P. (\text{prg}=G, \text{cls}=\text{accC}, \text{lcl}=L) \vdash$ 
         $\text{dom}(\text{locals}(\text{store}\ s1)) \gg \langle ps \rangle_l \gg P$ )
    show ( $\exists s1. G \vdash Norm\ s0 \ -e-\succ a \rightarrow s1 \wedge$ 
      ( $\text{abrupt}\ s1 = None \longrightarrow G, store\ s1 \vdash a::\preceq RefT\ statT$ )  $\wedge$ 
       $G \vdash s1 \ -ps-\succ vs \rightarrow s2$ )  $\wedge$ 
       $s2::\preceq(G, L)$ )
    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  $\forall a\ vs\ invC\ declC\ l.$ 
     $G, A \vdash \{ ?R\ a \leftarrow [vs]_l \wedge.$ 
      ( $\lambda s. \text{declC} =$ 
         $\text{invocation-declclass}\ G\ mode\ (\text{store}\ s)\ a\ statT$ 
        ( $\text{name}=mn, \text{parTs}=pTs'$ )  $\wedge$ 
         $invC = \text{invocation-class}\ mode\ (\text{store}\ s)\ a\ statT \wedge$ 
         $l = \text{locals}(\text{store}\ s)) ;.$ 
         $\text{init-lvars}\ G\ declC\ (\text{name}=mn, \text{parTs}=pTs')\ mode\ a\ vs \wedge.$ 
        ( $\lambda s. \text{normal}\ s \longrightarrow G \vdash mode \rightarrow invC \preceq statT$ )  $\}$ 
         $\text{Methd}\ declC\ (\text{name}=mn, \text{parTs}=pTs') \rightarrow$ 
         $\{ \text{set-lvars}\ l\ ;\ ?S \}$ 
      ) ( $\text{is}\ \forall\ a\ vs\ invC\ declC\ l. ?METHD\ a\ vs\ invC\ declC\ l$ )
  proof (intro allI)
    fix  $a\ vs\ invC\ declC\ l$ 
    from  $mgf-methods$  [rule-format]
    show  $?METHD\ a\ vs\ invC\ declC\ l$ 
    proof (rule  $MGFnD'$  [THEN  $conseq12$ ], clarsimp)
      fix  $s4\ s2\ s1::state$ 
      fix  $s0\ v$ 
      let  $?D = \text{invocation-declclass}\ G\ mode\ (\text{store}\ s2)\ a\ statT$ 
        ( $\text{name}=mn, \text{parTs}=pTs'$ )

```



```

let ?s3 = init-lvars  $G \text{ ?}D \ (\llbracket \text{name}=\text{mn}, \text{parTs}=\text{pTs}' \rrbracket) \text{ mode } a \text{ vs } s2$ 
assume inv-prop: abrupt ?s3 = None
   $\longrightarrow G \vdash \text{mode} \rightarrow \text{invocation-class mode (store } s2) \text{ } a \text{ statT} \preceq \text{statT}$ 
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:  $G \vdash \text{Norm } s0 \text{ } -e \rightarrow a \rightarrow s1$ 
assume eval-ps:  $G \vdash s1 \text{ } -ps \rightarrow vs \rightarrow s2$ 
assume eval-mthd:  $G \vdash ?s3 \text{ } -\text{Methd } ?D \ (\llbracket \text{name}=\text{mn}, \text{parTs}=\text{pTs}' \rrbracket) \text{ } -\text{v} \rightarrow s4$ 
show  $G \vdash \text{Norm } s0 \text{ } -\{\text{accC}, \text{statT}, \text{mode}\} e \cdot \text{mn}(\{pTs'\}ps) \text{ } -\text{v}$ 
   $\longrightarrow (\text{set-lvars (locals (store } s2))) \text{ } s4$ 

proof –
  obtain  $D$  where  $D: D = ?D$  by simp
  obtain  $s3$  where  $s3: s3 = ?s3$  by simp
  obtain  $s3'$  where
     $s3': s3' = \text{check-method-access } G \text{ accC statT mode}$ 
     $(\llbracket \text{name}=\text{mn}, \text{parTs}=\text{pTs}' \rrbracket) \text{ } a \text{ } s3$ 
    by simp
  have eq-s3'-s3:  $s3' = s3$ 
  proof –
    from inv-prop  $s3 \text{ mode}$ 
    have normal  $s3 \implies$ 
       $G \vdash \text{invmode statM } e \rightarrow \text{invocation-class mode (store } s2) \text{ } a \text{ statT} \preceq \text{statT}$ 
      by auto
    with eval-ps wt-e statM conf-s2 conf-a [rule-format]
    have check-method-access  $G \text{ accC statT (invmode statM } e)$ 
       $(\llbracket \text{name}=\text{mn}, \text{parTs}=\text{pTs}' \rrbracket) \text{ } a \text{ } s3 = s3$ 
      by (rule error-free-call-access) (auto simp add: s3 mode wf)
    thus ?thesis
      by (simp add: s3' mode)
  qed
  with eval-mthd  $D \text{ } s3$ 
  have  $G \vdash s3' \text{ } -\text{Methd } D \ (\llbracket \text{name}=\text{mn}, \text{parTs}=\text{pTs}' \rrbracket) \text{ } -\text{v} \rightarrow s4$ 
    by simp
  with eval-e eval-ps  $D \text{ } - s3'$ 
  show ?thesis
    by (rule eval-Call) (auto simp add: s3 mode D)
  qed
qed
qed
qed
qed

```

```

lemma eval-expression-no-jump':
  assumes eval:  $G \vdash s0 \text{ } -e \rightarrow v \rightarrow s1$ 
  and no-jmp: abrupt s0  $\neq \text{Some (Jump } j)$ 
  and wt:  $(\llbracket \text{prg}=G, \text{cls}=C, \text{lcl}=L \rrbracket) \vdash e :: -T$ 
  and wf: wf-prog  $G$ 
shows abrupt s1  $\neq \text{Some (Jump } j)$ 
using eval no-jmp wt wf
by – (rule eval-expression-no-jump
  [where  $?Env = (\llbracket \text{prg}=G, \text{cls}=C, \text{lcl}=L \rrbracket), \text{simplified}$ ], auto)

```

To derive the most general formula for the loop statement, we need to come up with a proper loop invariant, which intuitively states that we are currently inside the evaluation of the loop. To define such an invariant, we unroll the loop in iterated evaluations of the expression and evaluations of the loop body.

**constdefs**

$unroll:: prog \Rightarrow label \Rightarrow expr \Rightarrow stmt \Rightarrow (state \times state) set$

$unroll\ G\ l\ e\ c \equiv \{(s, t). \exists v\ s1\ s2. \\ G \vdash s -e-\triangleright v \rightarrow s1 \wedge the-Bool\ v \wedge normal\ s1 \wedge \\ G \vdash s1 -c\rightarrow s2 \wedge t=(abupd\ (absorb\ (Cont\ l))\ s2)\}$

**lemma** *unroll-while*:

**assumes** *unroll*:  $(s, t) \in (unroll\ G\ l\ e\ c)^*$

**and** *eval-e*:  $G \vdash t -e-\triangleright v \rightarrow s'$

**and** *normal-termination*:  $normal\ s' \longrightarrow \neg the-Bool\ v$

**and** *wt*:  $(\langle prg=G, cls=C, lcl=L \rangle) \vdash e::\neg T$

**and** *wf*: *wf-prog* *G*

**shows**  $G \vdash s -l\cdot While(e)\ c \rightarrow s'$

**using** *unroll*

**proof** (*induct rule: converse-rtrancl-induct*)

**show**  $G \vdash t -l\cdot While(e)\ c \rightarrow s'$

**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*:  $abrupt\ s' \neq Some\ (Jump\ (Cont\ l))$

**by** (*rule eval-expression-no-jump'*) (*insert normal-t,simp*)

**have**

*if the-Bool v*

*then*  $(G \vdash s' -c\rightarrow s' \wedge$

$G \vdash (abupd\ (absorb\ (Cont\ l))\ s') -l\cdot While(e)\ c \rightarrow s')$

*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*:  $(abupd\ (absorb\ (Cont\ l))\ s')=s'$

**by** (*cases s'*) (*simp add: absorb-def split: split-if*)

**moreover**

**from** *no-absorb abrupt-s'*

**have**  $G \vdash (abupd\ (absorb\ (Cont\ l))\ s') -l\cdot While(e)\ c \rightarrow s'$

**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) ∈ unroll G l e c
  assume while: G ⊢ s3 -l• While(e) c → s'
  show G ⊢ s -l• While(e) c → s'
  proof -
    from unroll obtain v s1 s2 where
      normal-s1: normal s1 and
      eval-e: G ⊢ s -e-⋗ v → s1 and
      continue: the-Bool v and
      eval-c: G ⊢ s1 -c→ s2 and
      s3: s3=(abupd (absorb (Cont l)) s2)
    by (unfold unroll-def) fast
    from eval-e normal-s1 have
      normal s
    by (rule eval-no-abrupt-lemma [rule-format])
    with while eval-e continue eval-c s3 show ?thesis
    by (auto intro!: eval.Loop)
  qed
qed

```

lemma MGFn-Loop:

```

  assumes mfg-e: G,(A::state triple set) ⊢ {=:n} ⟨e⟩e⋗ {G→}
  and mfg-c: G,A ⊢ {=:n} ⟨c⟩s⋗ {G→}
  and wf: wf-prog G
  shows G,A ⊢ {=:n} ⟨l• While(e) c⟩s⋗ {G→}
  proof (rule MGFn-free-wt [rule-format], elim exE)
    fix T L C
    assume wt: (|prg = G, cls = C, lcl = L|) ⊢ ⟨l• While(e) c⟩s::T
    then obtain eT where
      wt-e: (|prg = G, cls = C, lcl = L|) ⊢ e::-eT
    by cases simp
    show ?thesis
    proof (rule MGFn-NormalI)
      show G,A ⊢ {Normal ((λ Y' s' s. s' = s ∧ normal s) ∧. G ⊢ init ≤ n)}
        .l• While(e) c.
        {λ Y s' s. G ⊢ s -In1r (l• While(e) c)⋗→ (Y, s')}
    proof (rule conseq12
      [where ?P'=(λ Y s' s. (s,s') ∈ (unroll G l e c)* ) ∧. G ⊢ init ≤ n
        and ?Q'=((λ Y s' s. (∃ t b. (s,t) ∈ (unroll G l e c)* ∧
          Y = [b]e ∧ G ⊢ t -e-⋗ b → s')
          ∧. G ⊢ init ≤ n) ←= False ↓ = ◇)])
      show G,A ⊢ {(λ Y s' s. (s, s') ∈ (unroll G l e c)* ) ∧. G ⊢ init ≤ n}
        .l• While(e) c.
        {((λ Y s' s. (∃ t b. (s, t) ∈ (unroll G l e c)* ∧
          Y = In1 b ∧ G ⊢ t -e-⋗ b → s')
          ∧. G ⊢ init ≤ n) ←= False ↓ = ◇)}
    proof (rule ax-derivs.Loop)
      from mfg-e
      show G,A ⊢ {(λ Y s' s. (s, s') ∈ (unroll G l e c)* ) ∧. G ⊢ init ≤ n}
        e-⋗
        {(λ Y s' s. (∃ t b. (s, t) ∈ (unroll G l e c)* ∧
          Y = In1 b ∧ G ⊢ t -e-⋗ b → s')
          ∧. G ⊢ init ≤ n}
    proof (rule MGFnD' [THEN conseq12], clarsimp)

```

```

    fix s Z s' v
    assume (Z, s) ∈ (unroll G l e c)*
    moreover
    assume  $G \vdash s -e-\succ v \rightarrow s'$ 
    ultimately
    show  $\exists t. (Z, t) \in (\text{unroll } G \text{ l e c})^* \wedge G \vdash t -e-\succ v \rightarrow s'$ 
      by blast
  qed
next
  from mfg-c
  show  $G, A \vdash \{ \text{Normal } (((\lambda Y s' s. \exists t b. (s, t) \in (\text{unroll } G \text{ l e c})^* \wedge$ 
     $Y = \lfloor b \rfloor_e \wedge G \vdash t -e-\succ b \rightarrow s')$ 
     $\wedge. G \vdash \text{init} \leq n) \leftarrow \text{True}) \}$ 
    .c.
     $\{ \text{abupd } (\text{absorb } (\text{Cont } l)) \} .;$ 
     $((\lambda Y s' s. (s, s') \in (\text{unroll } G \text{ l e c})^*) \wedge. G \vdash \text{init} \leq n) \}$ 
  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:  $G \vdash t -e-\succ v \rightarrow \text{Norm } s$ 
    assume true: the-Bool v
    assume eval-c:  $G \vdash \text{Norm } s -c \rightarrow s'$ 
    show  $(Z, \text{abupd } (\text{absorb } (\text{Cont } l)) s') \in (\text{unroll } G \text{ l e c})^*$ 
    proof -
      note unroll
    also
      from eval-e true eval-c
      have  $(t, \text{abupd } (\text{absorb } (\text{Cont } l)) s') \in \text{unroll } G \text{ l e c}$ 
      by (unfold unroll-def) force
    ultimately show ?thesis ..
  qed
qed
qed
next
  show
     $\forall Y s Z.$ 
     $(\text{Normal } (((\lambda Y' s' s. s' = s \wedge \text{normal } s) \wedge. G \vdash \text{init} \leq n)) Y s Z$ 
     $\longrightarrow (\forall Y' s'.$ 
     $(\forall Y Z'.$ 
     $((\lambda Y s' s. (s, s') \in (\text{unroll } G \text{ l e c})^*) \wedge. G \vdash \text{init} \leq n) Y s Z'$ 
     $\longrightarrow (((\lambda Y s' s. \exists t b. (s, t) \in (\text{unroll } G \text{ l e c})^*$ 
     $\wedge Y = \lfloor b \rfloor_e \wedge G \vdash t -e-\succ b \rightarrow s')$ 
     $\wedge. G \vdash \text{init} \leq n) \leftarrow \text{False} \downarrow = \Diamond) Y' s' Z')$ 
     $\longrightarrow G \vdash Z -\langle l \cdot \text{While}(e) \ c \rangle_s \succ \rightarrow (Y', s'))$ 
  proof (clarsimp)
    fix Y' s' s
    assume asm:
       $\forall Z'. (Z', \text{Norm } s) \in (\text{unroll } G \text{ l e c})^*$ 
       $\longrightarrow \text{card } (\text{nyinitcls } G s') \leq n \wedge$ 
       $(\exists v. (\exists t. (Z', t) \in (\text{unroll } G \text{ l e c})^* \wedge G \vdash t -e-\succ v \rightarrow s') \wedge$ 
       $(\text{fst } s' = \text{None} \longrightarrow \neg \text{the-Bool } v)) \wedge Y' = \Diamond$ 
    show  $Y' = \Diamond \wedge G \vdash \text{Norm } s -l \cdot \text{While}(e) \ c \rightarrow s'$ 
    proof -
      from asm obtain v t where
        —  $Z'$  gets instantiated with  $\text{Norm } s$ 
      unroll:  $(\text{Norm } s, t) \in (\text{unroll } G \text{ l e c})^*$  and
      eval-e:  $G \vdash t -e-\succ v \rightarrow s'$  and
      normal-termination:  $\text{normal } s' \longrightarrow \neg \text{the-Bool } v$  and
      Y':  $Y' = \Diamond$ 

```

```

    by auto
  from unroll eval-e normal-termination wt-e wf
  have  $G \vdash \text{Norm } s \dashv \vdash \text{While}(e) \ c \rightarrow s'$ 
    by (rule unroll-while)
  with  $Y'$ 
  show ?thesis
    by simp
qed
qed
qed
qed
qed

```

**lemma** *MGFn-FVar*:

```

  fixes  $A :: \text{state triple set}$ 
  assumes mgf-init:  $G, A \vdash \{=:n\} \langle \text{Init statDeclC} \rangle_s \succ \{G \rightarrow\}$ 
  and mgf-e:  $G, A \vdash \{=:n\} \langle e \rangle_e \succ \{G \rightarrow\}$ 
  and wf:  $\text{wf-prog } G$ 
  shows  $G, A \vdash \{=:n\} \langle \{accC, statDeclC, stat\} e..fn \rangle_v \succ \{G \rightarrow\}$ 
proof (rule MGFn-free-wt-da-NormalConformI [rule-format], clarsimp)
  note inj-term-simps [simp]
  fix  $T \ L \ accC' \ V$ 
  assume wt:  $\langle \text{prg} = G, \text{cls} = accC', \text{lcl} = L \rangle \vdash \{accC, statDeclC, stat\} e..fn \rangle_v :: T$ 
  then obtain  $statC \ f$  where
    wt-e:  $\langle \text{prg} = G, \text{cls} = accC', \text{lcl} = L \rangle \vdash e :: \text{Class } statC$  and
    accfield:  $\text{accfield } G \ accC' \ statC \ fn = \text{Some } (statDeclC, f)$  and
    eq-accC:  $accC = accC'$  and
    stat:  $stat = \text{is-static } f$ 
  by (cases) (auto simp add: member-is-static-simp)
  let ?Q =  $(\lambda Y \ s1 \ (x, s) . x = \text{None} \wedge$ 
     $(G \vdash \text{Norm } s \dashv \vdash \text{Init statDeclC} \rightarrow s1) \wedge$ 
     $(\exists E. \langle \text{prg} = G, \text{cls} = accC', \text{lcl} = L \rangle \vdash \text{dom } (locals \ (store \ s1)) \gg \langle e \rangle_e \gg E))$ 
     $\wedge. G \vdash \text{init} \leq n \wedge. (\lambda s. s :: \preceq (G, L))$ 
  show  $G, A \vdash \{Normal\}$ 
     $((\lambda Y' \ s' \ s. s' = s \wedge \text{abrupt } s = \text{None}) \wedge. G \vdash \text{init} \leq n \wedge.$ 
     $(\lambda s. s :: \preceq (G, L)) \wedge.$ 
     $(\lambda s. \langle \text{prg} = G, \text{cls} = accC', \text{lcl} = L \rangle$ 
     $\vdash \text{dom } (locals \ (store \ s)) \gg \langle \{accC, statDeclC, stat\} e..fn \rangle_v \gg V))$ 
     $\} \{accC, statDeclC, stat\} e..fn = \succ$ 
     $\{ \lambda Y \ s' \ s. \exists vf. Y = \lfloor vf \rfloor_v \wedge$ 
     $G \vdash s \dashv \vdash \{accC, statDeclC, stat\} e..fn = \succ vf \rightarrow s' \}$ 
  (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  $G, A \vdash \{Normal \ ?P\} . \text{Init statDeclC} . \{?Q\}$ 
proof (rule MGFnD' [THEN conseq12], clarsimp)
  fix  $s \ s'$ 
  assume conf-s:  $\text{Norm } s :: \preceq (G, L)$ 
  assume da:  $\langle \text{prg} = G, \text{cls} = accC', \text{lcl} = L \rangle$ 
     $\vdash \text{dom } (locals \ s) \gg \langle \{accC, statDeclC, stat\} e..fn \rangle_v \gg V$ 
  assume eval-init:  $G \vdash \text{Norm } s \dashv \vdash \text{Init statDeclC} \rightarrow s'$ 
  show  $(\exists E. \langle \text{prg} = G, \text{cls} = accC', \text{lcl} = L \rangle \vdash \text{dom } (locals \ (store \ s')) \gg \langle e \rangle_e \gg E) \wedge$ 
     $s' :: \preceq (G, L)$ 
proof -
  from da
  obtain  $E$  where
     $\langle \text{prg} = G, \text{cls} = accC', \text{lcl} = L \rangle \vdash \text{dom } (locals \ s) \gg \langle e \rangle_e \gg E$ 
  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
   $(\text{prg}=G, \text{cls}=\text{acc}C', \text{lcl}=L) \vdash \text{dom } (\text{locals } (\text{store } s')) \gg \langle e \rangle_e \gg E'$ 
  by (rule da-weakenE)
moreover
have  $s'::\preceq(G, L)$ 
proof –
  have wt-init:  $(\text{prg}=G, \text{cls}=\text{acc}C, \text{lcl}=L) \vdash (\text{Init statDecl}C)::\checkmark$ 
  proof –
    from wf wt-e
    have iscls-statC: is-class  $G \text{ stat}C$ 
      by (auto dest: ty-expr-is-type type-is-class)
    with wf accfield
    have iscls-statDeclC: is-class  $G \text{ statDecl}C$ 
      by (auto dest!: accfield-fields dest: fields-declC)
    thus ?thesis by simp
  qed
obtain  $I$  where
   $\text{da-init}: (\text{prg}=G, \text{cls}=\text{acc}C, \text{lcl}=L) \vdash \text{dom } (\text{locals } (\text{store } ((\text{Norm } s)::\text{state}))) \gg \langle \text{Init statDecl}C \rangle_s \gg I$ 
  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  $G, A \vdash \{?Q\} \ e \multimap \{ \lambda \text{Val}:a. \text{fvar statDecl}C \text{ stat fn } a \ ..; ?R \}$ 
proof (rule MGFnD' [THEN conseq12], clarsimp)
  fix  $s0 \ s1 \ s2 \ E \ a$ 
  let  $?fvar = \text{fvar statDecl}C \text{ stat fn } a \ s2$ 
  assume eval-init:  $G \vdash \text{Norm } s0 \multimap \text{Init statDecl}C \rightarrow s1$ 
  assume eval-e:  $G \vdash s1 \multimap e \multimap a \rightarrow s2$ 
  assume conf-s1:  $s1::\preceq(G, L)$ 
  assume da-e:  $(\text{prg}=G, \text{cls}=\text{acc}C', \text{lcl}=L) \vdash \text{dom } (\text{locals } (\text{store } s1)) \gg \langle e \rangle_e \gg E$ 
  show  $G \vdash \text{Norm } s0 \multimap \{ \text{acc}C, \text{statDecl}C, \text{stat} \} e.. \text{fn} \multimap \text{fst } ?fvar \rightarrow \text{snd } ?fvar$ 
  proof –
    obtain  $v \ s2'$  where
       $v: v = \text{fst } ?fvar \text{ and } s2': s2' = \text{snd } ?fvar$ 
      by simp
    obtain  $s3$  where
       $s3: s3 = \text{check-field-access } G \text{ acc}C' \text{ statDecl}C \text{ fn stat } a \ s2'$ 
      by simp
    have eq-s3-s2':  $s3 = s2'$ 
    proof –
      from eval-e conf-s1 wt-e da-e wf obtain
        conf-s2:  $s2::\preceq(G, L)$  and
        conf-a: normal  $s2 \implies G, \text{store } s2 \vdash a::\preceq \text{Class stat}C$ 
        by (rule eval-type-soundE) simp
      from accfield wt-e eval-init eval-e conf-s2 conf-a - wf
      show ?thesis
      by (rule error-free-field-access
        [where ?v=v and ?s2'=s2', elim-format])

```

```

      (simp add: s3 v s2' stat)+
    qed
  from eval-init eval-e
  show ?thesis
    apply (rule eval.FVar [where ?s2'=s2'])
    apply (simp add: s2')
    apply (simp add: s3 [symmetric] eq-s3-s2' eq-accC s2' [symmetric])
    done
  qed
qed
qed
qed

```

**lemma** *MGFn-Fin*:

```

  assumes wf: wf-prog G
  and   mgf-c1: G, A ⊢ {=:n} ⟨c1⟩s > {G→}
  and   mgf-c2: G, A ⊢ {=:n} ⟨c2⟩s > {G→}
  shows G, (A::state triple set) ⊢ {=:n} ⟨c1 Finally c2⟩s > {G→}
proof (rule MGFn-free-wt-da-NormalConformI [rule-format], clarsimp)
  fix T L accC C
  assume wt: (prg=G, cls=accC, lcl=L) ⊢ In1r (c1 Finally c2)::T
  then obtain
    wt-c1: (prg=G, cls=accC, lcl=L) ⊢ c1::√ and
    wt-c2: (prg=G, cls=accC, lcl=L) ⊢ c2::√
  by cases simp
  let ?Q = (λY' s' s. normal s ∧ G ⊢ s - c1 → s' ∧
    (∃ C1. (prg=G, cls=accC, lcl=L) ⊢ dom (locals (store s)) » ⟨c1⟩s » C1)
    ∧ s::⊢(G, L))
    ∧. G ⊢ init ≤ n
  show G, A ⊢ {Normal
    ((λY' s' s. s' = s ∧ abrupt s = None) ∧. G ⊢ init ≤ n ∧.
    (λs. s::⊢(G, L)) ∧.
    (λs. (prg=G, cls=accC, lcl=L)
      ⊢ dom (locals (store s)) » ⟨c1 Finally c2⟩s » C))}
    .c1 Finally c2.
    {λY s' s. Y = ◇ ∧ G ⊢ s - c1 Finally c2 → s'}}
  (is G, A ⊢ {Normal ?P} .c1 Finally c2. {?R})
proof (rule ax-derivs.Fin [where ?Q=?Q])
  from mgf-c1
  show G, A ⊢ {Normal ?P} .c1. {?Q}
proof (rule MGFnD' [THEN conseq12], clarsimp)
  fix s0
  assume (prg=G, cls=accC, lcl=L) ⊢ dom (locals s0) » ⟨c1 Finally c2⟩s » C
  thus ∃ C1. (prg=G, cls=accC, lcl=L) ⊢ dom (locals s0) » ⟨c1⟩s » C1
  by cases (auto simp add: inj-term-simps)
  qed
next
  from mgf-c2
  show ∀ abr. G, A ⊢ {?Q ∧. (λs. abr = abrupt s) ∴ abupd (λabr. None)} .c2.
    {abupd (abrupt-if (abr ≠ None) abr) ∴ ?R}
proof (rule MGFnD' [THEN conseq12, THEN allI], clarsimp)
  fix s0 s1 s2 C1
  assume da-c1: (prg=G, cls=accC, lcl=L) ⊢ dom (locals s0) » ⟨c1⟩s » C1
  assume conf-s0: Norm s0::⊢(G, L)
  assume eval-c1: G ⊢ Norm s0 - c1 → s1
  assume eval-c2: G ⊢ abupd (λabr. None) s1 - c2 → s2
  show G ⊢ Norm s0 - c1 Finally c2

```

```

      → abupd (abrupt-if (∃ y. abrupt s1 = Some y) (abrupt s1)) s2
proof -
  obtain abr1 str1 where s1: s1=(abr1,str1)
  by (cases s1)
  with eval-c1 eval-c2 obtain
    eval-c1': G⊢ Norm s0 -c1→ (abr1,str1) and
    eval-c2': G⊢ Norm str1 -c2→ s2
  by simp
  obtain s3 where
    s3: s3 = (if ∃ err. abr1 = Some (Error err)
      then (abr1, str1)
      else abupd (abrupt-if (abr1 ≠ None) abr1) s2)
  by simp
  from eval-c1' conf-s0 wt-c1 - wf
  have error-free (abr1,str1)
  by (rule eval-type-soundE) (insert da-c1,auto)
  with s3 have eq-s3: s3=abupd (abrupt-if (abr1 ≠ None) abr1) s2
  by (simp add: error-free-def)
  from eval-c1' eval-c2' s3
  show ?thesis
  by (rule eval.Fin [elim-format]) (simp add: s1 eq-s3)
qed
qed
qed
qed

```

**lemma** *Body-no-break*:

```

assumes eval-init: G⊢ Norm s0 -Init D→ s1
and      eval-c: G⊢ s1 -c→ s2
and      jmpOk: jumpNestingOkS {Ret} c
and      wt-c: (|prg=G, cls=C, lcl=L|)⊢ c::√
and      clsD: class G D=Some d
and      wf: wf-prog G
shows ∀ l. abrupt s2 ≠ Some (Jump (Break l)) ∧
      abrupt s2 ≠ Some (Jump (Cont l))
proof
  fix l show abrupt s2 ≠ Some (Jump (Break l)) ∧
    abrupt s2 ≠ Some (Jump (Cont l))
  proof -
    fix accC from clsD have wt-init: (|prg=G, cls=accC, lcl=L|)⊢ (Init D)::√
    by auto
    from eval-init wf
    have s1-no-jmp: ∧ j. abrupt s1 ≠ Some (Jump j)
    by - (rule eval-statement-no-jump [OF - - wt-init],auto)
    from eval-c - wt-c wf
    show ?thesis
    apply (rule jumpNestingOk-eval [THEN conjE, elim-format])
    using jmpOk s1-no-jmp
    apply auto
    done
  qed
qed

```

**lemma** *MGFn-Body*:

```

assumes wf: wf-prog G
and      mgf-init: G, A⊢ {=:n} ⟨Init D⟩s⋗ {G→}
and      mgf-c: G, A⊢ {=:n} ⟨c⟩s⋗ {G→}

```



**shows**  $G, (A::\text{state triple set}) \vdash \{=:n\} \langle \text{Body } D \ c \rangle_e \succ \{G \rightarrow\}$   
**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**  $?Q = (\lambda Y' \ s' \ s. \text{normal } s \wedge G \vdash s - \text{Init } D \rightarrow s' \wedge \text{jumpNestingOkS } \{\text{Ret}\} \ c)$   
 $\quad \wedge. G \vdash \text{init} \leq n$   
**show**  $G, A \vdash \{ \text{Normal}$   
 $\quad ((\lambda Y' \ s' \ s. s' = s \wedge \text{fst } s = \text{None}) \wedge. G \vdash \text{init} \leq n \wedge.$   
 $\quad (\lambda s. s :: \preceq (G, L)) \wedge.$   
 $\quad (\lambda s. (\text{prg}=G, \text{cls}=\text{accC}, \text{lcl}=L))$   
 $\quad \vdash \text{dom } (\text{locals } (\text{store } s)) \gg \langle \text{Body } D \ c \rangle_e \gg E))\}$   
 $\quad \text{Body } D \ c \rightarrow$   
 $\quad \{\lambda Y \ s' \ s. \exists v. Y = \text{In1 } v \wedge G \vdash s - \text{Body } D \ c \rightarrow v \rightarrow s'\}$   
**(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**  $G \vdash \text{Norm } s0 - \text{Body } D \ c \rightarrow \text{the } (\text{locals } (\text{store } s2) \ \text{Result})$   
 $\quad \rightarrow \text{abupd } (\text{absorb } \text{Ret}) \ s2$   
**proof** –  
**from**  $\text{wt}$  **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**  
 $s3: s3 = (\text{if } \exists l. \text{fst } s2 = \text{Some } (\text{Jump } (\text{Break } l)) \vee$   
 $\quad \text{fst } s2 = \text{Some } (\text{Jump } (\text{Cont } l))$   
 $\quad \text{then } \text{abupd } (\lambda x. \text{Some } (\text{Error } \text{CrossMethodJump})) \ s2$   
 $\quad \text{else } s2)$   
**by** *simp*  
**from**  $\text{eval-init } \text{eval-c } \text{nestingOk } \text{wt-c } d \ \text{wf}$   
**have**  $\text{eq-s3-s2}: s3 = s2$   
**by** (rule *Body-no-break* [elim-format]) (*simp add: s3*)  
**from**  $\text{eval-init } \text{eval-c } s3$   
**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
       $G, A \vdash \{=:n\} \ \langle es \rangle_l \succ \{G \rightarrow\}$ 
    proof (induct rule: var-expr-stmt.inducts)
      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 (fastsimp 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 (fastsimp 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 add: split-if)
  }

```

```

    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 (fastsimp 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 (fastsimp 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 (fastsimp intro: eval.Inst)
  done
next
case (Lit v)
show ?case
  apply -
  apply (rule MGFn-NormalI)
  apply (rule ax-derivs.Lit [THEN conseq1])
  apply (fastsimp 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 (fastsimp 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 (fastsimp 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 (fastsimp intro: eval.BinOp)
    done
next
  case Super
  show ?case
    apply -
    apply (rule MGFn-NormalI)
    apply (rule ax-derivs.Super [THEN conseq1])
    apply (fastsimp 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 (fastsimp intro: eval.Acc simp add: split-paired-all)
    done
next
  case (Ass v e)
  thus  $G, \text{At}\{=:n\} \langle v:=e \rangle_e \succ \{G \rightarrow\}$ 
    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 (fastsimp intro: eval.Ass simp add: split-paired-all)
    done
next
  case (Cond e1 e2 e3)
  thus  $G, \text{At}\{=:n\} \langle e1 \ ? \ e2 : e3 \rangle_e \succ \{G \rightarrow\}$ 
    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 (fastsimp intro: eval.Cond)
    apply (case-tac b)
    apply simp
    apply (erule MGFnD'[THEN conseq12])
    apply (fastsimp intro: eval.Cond)
    apply simp
    apply (erule MGFnD'[THEN conseq12])
    apply (fastsimp intro: eval.Cond)
    done
next
  case (Call accC statT mode e mn pTs' ps)
  note mgf-e =  $\langle G, \text{At}\{=:n\} \langle e \rangle_e \succ \{G \rightarrow\} \rangle$ 
  note mgf-ps =  $\langle G, \text{At}\{=:n\} \langle ps \rangle_l \succ \{G \rightarrow\} \rangle$ 
  from mgf-methds mgf-e mgf-ps wf
  show  $G, \text{At}\{=:n\} \langle \{accC, statT, mode\} e \cdot mn(\{pTs' ps\}) \rangle_e \succ \{G \rightarrow\}$ 
    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 (fastsimp 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 (fastsimp 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 (fastsimp 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 (fastsimp intro: eval.Comp)
  done
next
  case (If' e c1 c2)
  thus  $G, A \vdash \{=:n\} \langle \text{If}(e) \text{ c1 Else c2} \rangle_s \succ \{G \rightarrow\}$ 
  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 (fastsimp intro: eval.If)
    apply (case-tac b)
    apply simp
    apply (erule MGFnD' [THEN conseq12])
    apply (fastsimp intro: eval.If)
    apply simp
    apply (erule MGFnD' [THEN conseq12])
    apply (fastsimp 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 simp add: abupd-def2)
  done
next
case (Throw e)
thus ?case
  apply -
  apply (rule MGFn-NormalI)
  apply (erule MGFnD' [THEN conseq12, THEN ax-derivs.Throw])
  apply (fastsimp 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 =  $(\lambda Y' s' s. \text{normal } s \wedge (\exists s''. G \vdash s - \langle c1 \rangle_s \rightarrow (Y', s'') \wedge G \vdash s'' - s \text{xalloc} \rightarrow s')) \wedge. G \vdash \text{init} \leq n]$ )
  apply (erule MGFnD [THEN ax-NormalD, THEN conseq2])
  apply (fastsimp elim: sxalloc-geat [THEN card-nyinitcls-geat])
  apply (erule MGFnD' [THEN conseq12])
  apply (fastsimp intro: eval.Try)
  apply (fastsimp 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 (fastsimp 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 (fastsimp intro: eval.Cons)
    done
qed
}
thus ?thesis
  by (cases rule: term-cases) auto
qed
qed

```

**lemma** *MGF-asm:*

```

 $\llbracket \forall C \text{ sig. is-methd } G \ C \text{ sig} \longrightarrow G, A \vdash \{\dot{=}\} \text{ In1l } (\text{Methd } C \text{ sig}) \succ \{G \rightarrow\}; \text{ wf-prog } G \rrbracket$ 
 $\implies G, (A::\text{state triple set}) \vdash \{\dot{=}\} t \succ \{G \rightarrow\}$ 
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 @{clasimpset}))
apply (tactic << dtac (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 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:  $\text{wf-prog } G \implies G, (\{\} :: \text{state triple set}) \vdash \{ \doteq \} \ t \succ \{ G \rightarrow \}$ 
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  $\implies$ 
   $G, A \cup (\lambda(C, sig). \{Normal \doteq\} \langle Methd \ C \ sig \rangle_e \succ \{G \rightarrow\}) \text{ ' } ms$ 
 $\vdash (\lambda(C, sig). \{Normal \doteq\} \langle body \ G \ C \ sig \rangle_e \succ \{G \rightarrow\}) \text{ ' } ms \implies$ 
 $G, A \vdash (\lambda(C, sig). \{Normal \doteq\} \langle Methd \ C \ sig \rangle_e \succ \{G \rightarrow\}) \text{ ' } ms$ 
apply (unfold MGF-def)
apply (rule ax-derivs.Methd [unfolded mtriples-def])
apply (erule ax-finite-pointwise)
prefer 2
apply (rule ax-derivs.asm)
apply fast
apply clarsimp
apply (rule conseq2)
apply (erule (1) ax-methods-spec)
apply clarsimp
apply (erule eval-Methd)
done

```

```

lemma MGF-simult-Methd: wf-prog G  $\implies$ 
   $G, (\{ \} :: state \ triple \ set) \vdash (\lambda(C, sig). \{Normal \doteq\} \langle Methd \ C \ sig \rangle_e \succ \{G \rightarrow\})$ 
 $\text{ ' } Collect \ (split \ (is-methd \ G))$ 
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:  $\llbracket G \vdash s \rightarrow t \succ \rightarrow (Y', s') ; type-ok \ G \ t \ s ; wf-prog \ G \rrbracket$ 
 $\implies \exists n. G \vdash s \rightarrow t \succ \rightarrow n \rightarrow (Y', s')$ 
apply (cases normal s)
apply (force simp add: type-ok-def intro: eval-evaln)
apply (force intro: evaln.Abrupt)
done

```

```

lemma MGF-complete:
  assumes valid:  $G, \{ \} \models \{P\} \ t \succ \{Q\}$ 
  and mgf:  $G, (\{ \} :: state \ triple \ set) \vdash \{ \} \ t \succ \{G \rightarrow\}$ 
  and wf: wf-prog G
  shows  $G, (\{ \} :: state \ triple \ set) \vdash \{P :: state \ assn\} \ t \succ \{Q\}$ 
proof (rule ax-no-hazard)
  from mgf
  have  $G, (\{ \} :: state \ triple \ set) \vdash \{ \} \ t \succ \{ \lambda Y \ s' \ s. G \vdash s \rightarrow t \succ \rightarrow (Y, s') \}$ 
  by (unfold MGF-def)
  thus  $G, (\{ \} :: state \ triple \ set) \vdash \{P \ \wedge \ type-ok \ G \ t\} \ t \succ \{Q\}$ 
proof (rule conseq12, clarsimp)
  fix  $Y \ s \ Z \ Y' \ s'$ 
  assume  $P: P \ Y \ s \ Z$ 
  assume type-ok: type-ok G t s

```

```

assume eval-t:  $G \vdash s - t \succ \rightarrow (Y', s')$ 
show  $Q \ Y' \ s' \ Z$ 
proof –
  from eval-t type-ok wf
  obtain  $n$  where evaln:  $G \vdash s - t \succ - n \rightarrow (Y', s')$ 
    by (rule eval-to-evaln [elim-format]) iprover
  from valid have
    valid-expanded:
     $\forall n \ Y \ s \ Z. \ P \ Y \ s \ Z \longrightarrow \text{type-ok } G \ t \ s$ 
       $\longrightarrow (\forall Y' \ s'. \ G \vdash s - t \succ - n \rightarrow (Y', s') \longrightarrow Q \ Y' \ s' \ Z)$ 
    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  $G, (\{\} :: \text{state triple set}) \vdash \{\doteq\} \ t \succ \{G \rightarrow\}$ 
    by (rule MGF-deriv)
  from valid this wf
  show ?thesis
    by (rule MGF-complete)
qed

end

```

## Chapter 25

### AxExample

## 64 Example of a proof based on the Bali axiomatic semantics

**theory** *AxExample* **imports** *AxSem Example* **begin**

**constdefs**

```

  arr-inv :: st  $\Rightarrow$  bool
  arr-inv  $\equiv \lambda s. \exists \text{obj } a \ T \ \text{el. } \text{globs } s \ (\text{Stat } \text{Base}) = \text{Some } \text{obj} \wedge$ 
                                     values obj (Inl (arr, Base)) = Some (Addr a)  $\wedge$ 
                                     heap s a = Some ( $\langle \text{tag}=\text{Arr } T \ 2, \text{values}=\text{el} \rangle$ )

```

**lemma** *arr-inv-new-obj*:

$\bigwedge a. \llbracket \text{arr-inv } s; \text{new-Addr } (\text{heap } s) = \text{Some } a \rrbracket \Longrightarrow \text{arr-inv } (\text{gupd}(\text{Inl } a \mapsto x) \ s)$

**apply** (*unfold arr-inv-def*)

**apply** (*force dest!:: new-AddrD2*)

**done**

**lemma** *arr-inv-set-locals* [*simp*]:  $\text{arr-inv } (\text{set-locals } l \ s) = \text{arr-inv } s$

**apply** (*unfold arr-inv-def*)

**apply** (*simp (no-asm)*)

**done**

**lemma** *arr-inv-gupd-Stat* [*simp*]:

$\text{Base} \neq C \Longrightarrow \text{arr-inv } (\text{gupd}(\text{Stat } C \mapsto \text{obj}) \ s) = \text{arr-inv } s$

**apply** (*unfold arr-inv-def*)

**apply** (*simp (no-asm-simp)*)

**done**

**lemma** *ax-inv-lupd* [*simp*]:  $\text{arr-inv } (\text{lupd}(x \mapsto y) \ s) = \text{arr-inv } s$

**apply** (*unfold arr-inv-def*)

**apply** (*simp (no-asm)*)

**done**

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

**declare** *lvar-def* [*simp*]

**ML**  $\langle\langle$

*local*

```

  val ax-Skip = thm ax-Skip;
  val ax-StatRef = thm ax-StatRef;
  val ax-MethdN = thm ax-MethdN;
  val ax-Alloc = thm ax-Alloc;
  val ax-Alloc-Arr = thm ax-Alloc-Arr;
  val ax-SXAlloc-Normal = thm ax-SXAlloc-Normal;
  val ax-derivs-intros = funpow 7 tl (thms ax-derivs.intros);

```

*in*

**fun** *inst1-tac* *s t st* =

```

  case AList.lookup (op =) (rev (Term.add-varnames (prop-of st) [])) s of
    SOME i => Tactic.instantiate-tac' [((s, i), t)] st | NONE => Seq.empty;

```

**val** *ax-tac* =

*REPEAT* *o rtac allI THEN'*

```

  resolve-tac (ax-Skip :: ax-StatRef :: ax-MethdN :: ax-Alloc ::
    ax-Alloc-Arr :: ax-SXAlloc-Normal :: ax-derivs-intros);

```

end;  
 >>

```

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 1 )
defer
apply (tactic ax-tac 1 )
defer
apply (tactic << inst1-tac 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↓=◇ in conseq2)
prefer 2
apply simp
apply (tactic ax-tac 1 )
prefer 2
apply (rule ax-impossible [THEN conseq1], clarsimp)
apply (rule-tac P' = Normal ?P in conseq1)
prefer 2
apply clarsimp
apply (tactic ax-tac 1 )
apply (tactic ax-tac 1 )
prefer 2
apply (rule ax-subst-Val-allI)
apply (tactic << inst1-tac P' λu a. Normal (?PP a←?x) u >>)
apply (simp del: avar-def2 peek-and-def2)
apply (tactic ax-tac 1 )
apply (tactic ax-tac 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 1 )
apply (tactic ax-tac 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 1 )
apply (rule-tac C = Ext in ax-Call-known-DynT)
apply (unfold DynT-prop-def)
apply (simp (no-asm))
apply (intro strip)
apply (rule-tac P' = Normal ?P in conseq1)
apply (tactic ax-tac 1 )

```

```

apply    (rule ax-thin [OF - empty-subsetI])
apply    (simp (no-asm) add: body-def2)
apply    (tactic ax-tac 1 )

defer
apply    (simp (no-asm))
apply    (tactic ax-tac 1)

apply    (rule-tac [2] ax-derivs.Abrupt)

apply    (rule ax-derivs.Expr)
apply    (tactic ax-tac 1)
prefer 2
apply    (rule ax-subst-Var-allI)
apply    (tactic ⟨ inst1-tac P' λa vs l vf. ?PP a vs l vf ← ?x ∧. ?p ⟩)
apply    (rule allI)
apply    (tactic ⟨ simp-tac (@{simpset} delloop split-all-tac delsimps [@{thm peek-and-def2}, @{thm heap-def2}, @{thm subst-res-def2}, @{thm normal-def2}]) 1 ⟩)
apply    (rule ax-derivs.Abrupt)
apply    (simp (no-asm))
apply    (tactic ax-tac 1 )
apply    (tactic ax-tac 2, tactic ax-tac 2, tactic ax-tac 2)
apply    (tactic ax-tac 1)
apply    (tactic ⟨ inst1-tac R λa'. Normal ((λ Vals:vs (x, s) Z. arr-inv s ∧ initd Ext (globs s) ∧ a' ≠ Null ∧ vs = [Null]) ∧. heap-free two) ⟩)
apply    fastsimp
prefer 4
apply    (rule ax-derivs.Done [THEN consequ1],force)
apply    (rule ax-subst-Val-allI)
apply    (tactic ⟨ inst1-tac P' λu a. Normal (?PP a ← ?x) u ⟩)
apply    (simp (no-asm) del: peek-and-def2 heap-free-def2 normal-def2 o-apply)
apply    (tactic ax-tac 1)
prefer 2
apply    (rule ax-subst-Val-allI)
apply    (tactic ⟨ inst1-tac P' λaa v. Normal (?QQ aa v ← ?y) ⟩)
apply    (simp del: peek-and-def2 heap-free-def2 normal-def2)
apply    (tactic ax-tac 1)
apply    (tactic ax-tac 1)
apply    (tactic ax-tac 1)
apply    (tactic ax-tac 1)

apply (simp (no-asm))

apply (rule-tac Q' = Normal ((λ Y (x, s) Z. arr-inv s ∧ (∃ a. the (locals s (VName e)) = Addr a ∧ obj-class (the (globs s (Inl a))) = Ext ∧ invocation-declclass tprg IntVir s (the (locals s (VName e))) (ClassT Base) (name = foo, parTs = [Class Base]) = Ext)) ∧. initd Ext ∧. heap-free two)
in consequ2)
prefer 2
apply clarsimp
apply (tactic ax-tac 1)
apply (tactic ax-tac 1)
defer
apply (rule ax-subst-Var-allI)
apply (tactic ⟨ inst1-tac P' λu vf. Normal (?PP vf ∧. ?p) u ⟩)
apply (simp (no-asm) del: split-paired-All peek-and-def2 initd-def2 heap-free-def2 normal-def2)
apply (tactic ax-tac 1 )
apply (tactic ax-tac 1 )

```

```

apply (rule-tac  $Q' = \text{Normal } ((\lambda Y s Z. \text{arr-inv } (\text{store } s) \wedge \text{vf}=\text{lvar } (\text{VName } e) (\text{store } s)) \wedge. \text{heap-free tree} \wedge. \text{initd Ext})$  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 (@{simpset} delloop split-all-tac) 1 >>))
apply (rule ax-Init-Skip-lemma)
apply (tactic << simp-tac (@{simpset} delloop split-all-tac) 1 >>))
apply (rule ax-InitS [THEN conseq1])
apply force
apply (simp (no-asm))
apply (unfold arr-viewed-from-def)
apply (rule allI)
apply (rule-tac  $P' = \text{Normal } ?P$  in conseq1)
apply (tactic << simp-tac (@{simpset} delloop split-all-tac) 1 >>))
apply (tactic ax-tac 1)
apply (tactic ax-tac 1)
apply (rule-tac [2] ax-subst-Var-allI)
apply (tactic << inst1-tac  $P' \lambda \text{vf } l \text{ vfa. Normal } (?P \text{ vf } l \text{ vfa})$  >>))
apply (tactic << simp-tac (@{simpset} delloop split-all-tac delsimps [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 2)
apply (tactic ax-tac 3)
apply (tactic ax-tac 3)
apply (tactic << inst1-tac  $P \lambda \text{vf } l \text{ vfa. Normal } (?P \text{ vf } l \text{ vfa} \leftarrow \Diamond)$  >>))
apply (tactic << simp-tac (@{simpset} delloop split-all-tac) 2 >>))
apply (tactic ax-tac 2)
apply (tactic ax-tac 1)
apply (tactic ax-tac 2)
apply (rule ax-derivs.Done [THEN conseq1])
apply (tactic << inst1-tac  $Q \lambda \text{vf. Normal } ((\lambda Y s Z. \text{vf}=\text{lvar } (\text{VName } e) (\text{snd } s)) \wedge. \text{heap-free four} \wedge. \text{initd Base} \wedge. \text{initd Ext})$  >>))
apply (clarsimp split del: split-if)
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 (@{simpset} delloop split-all-tac) 1 >>))
apply (rule ax-triv-Init-Object [THEN peek-and-forget2, THEN conseq1])
apply (rule wf-tprg)
apply clarsimp
apply (tactic << inst1-tac  $P \lambda \text{vf. Normal } ((\lambda Y s Z. \text{vf} = \text{lvar } (\text{VName } e) (\text{snd } s)) \wedge. \text{heap-free four} \wedge. \text{initd Ext})$  >>))
apply clarsimp
apply (tactic << inst1-tac  $PP \lambda \text{vf. Normal } ((\lambda Y s Z. \text{vf} = \text{lvar } (\text{VName } e) (\text{snd } s)) \wedge. \text{heap-free four} \wedge. \text{Not } \circ \text{initd Base})$  >>))
apply clarsimp

apply (rule conseq1)
apply (tactic ax-tac 1)
apply clarsimp
done

```

**lemma** *Loop-Xcpt-benchmark:*

```

  Q = (λY (x,s) Z. x ≠ None → the-Bool (the (locals s i))) ⇒
    G,({::'a triple set})⊢{Normal (λY s Z::'a. True)}
    .lab1• While(Lit (Bool True)) (If(Acc (LVar i)) (Throw (Acc (LVar xcpt))) Else
      (Expr (Ass (LVar i) (Acc (LVar j))))). {Q}
  apply (rule-tac P' = Q and Q' = Q ←= False ↓ = ◇ in conseq12)
  apply safe
  apply (tactic ax-tac 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 1)
  apply clarsimp
  prefer 2
  apply clarsimp
  apply (tactic ax-tac 1 )
  apply (tactic
    ⟨ inst1-tac P' Normal (λs.. (λY s Z. True) ↓ = Val (the (locals s i))) ⟩)
  apply (tactic ax-tac 1)
  apply (rule conseq1)
  apply (tactic ax-tac 1)
  apply clarsimp
  apply (rule allI)
  apply (rule ax-escape)
  apply auto
  apply (rule conseq1)
  apply (tactic ax-tac 1 )
  apply (tactic ax-tac 1)
  apply (tactic ax-tac 1)
  apply clarsimp
  apply (rule-tac Q' = Normal (λY s Z. True) in conseq2)
  prefer 2
  apply clarsimp
  apply (rule conseq1)
  apply (tactic ax-tac 1)
  apply (tactic ax-tac 1)
  prefer 2
  apply (rule ax-subst-Var-allI)
  apply (tactic ⟨ inst1-tac P' λb Y ba Z vf. λY (x,s) Z. x= None ∧ snd vf = snd (lvar i s) ⟩)
  apply (rule allI)
  apply (rule-tac P' = Normal ?P in conseq1)
  prefer 2
  apply clarsimp
  apply (tactic ax-tac 1)
  apply (rule conseq1)
  apply (tactic ax-tac 1)
  apply clarsimp
  apply (tactic ax-tac 1)
  apply clarsimp
  done

end

```