

An Operational Semantics and Type Safety Proof for Multiple Inheritance in C++ (CoreC++)

Daniel Wasserrab
Fakultät für Mathematik und Informatik
Universität Passau
<http://www.infosun.fmi.uni-passau.de/st/staff/wasserra/>



December 12, 2009

Abstract

We present an operational semantics and type safety proof for multiple inheritance in C++. The semantics models the behavior of method calls, field accesses, and two forms of casts. For explanations see [1].

Contents

1	Auxiliary Definitions	6
1.1	<i>distinct-fst</i>	7
1.2	Using <i>list-all2</i> for relations	8
2	CoreC++ types	10
3	CoreC++ values	12
4	Expressions	14
4.1	The expressions	14
4.2	Free Variables	15
5	Class Declarations and Programs	16
6	The subclass relation	19

7	Definition of Subobjects	21
7.1	General definitions	21
7.2	Subobjects according to Rossie-Friedman	21
7.3	Subobject handling and lemmas	23
7.4	Paths	26
7.5	Appending paths	26
7.6	The relation on paths	27
7.7	Member lookups	28
8	Objects and the Heap	31
8.1	Objects	31
8.2	Heap	32
9	Exceptions	33
9.1	Exceptions	33
9.2	System exceptions	33
9.3	<i>preallocated</i>	33
9.4	<i>start-heap</i>	34
10	Syntax	35
11	Program State	36
12	Big Step Semantics	37
12.1	The rules	37
12.2	Final expressions	42
13	Small Step Semantics	44
13.1	Some pre-definitions	44
13.2	The rules	44
13.3	The reflexive transitive closure	49
13.4	Some easy lemmas	50
14	System Classes	51
15	The subtype relation	52
16	Well-typedness of CoreC++ expressions	53
16.1	The rules	53
16.2	Easy consequences	55
17	Generic Well-formedness of programs	57
17.1	Well-formedness lemmas	57
17.2	Well-formedness subclass lemmas	58
17.3	Well-formedness <i>leq_path</i> lemmas	59

17.4	Lemmas concerning Subobjs	60
17.5	Well-formedness and appendPath	62
17.6	Path and program size	63
17.7	Well-formedness and Path	64
17.8	Well-formedness and member lookup	66
17.9	Well formedness and widen	69
17.10	Well formedness and well typing	69
18	Weak well-formedness of CoreC++ programs	70
19	Equivalence of Big Step and Small Step Semantics	71
19.1	Some casts-lemmas	71
19.2	Small steps simulate big step	72
19.3	Cast	72
19.4	LAss	73
19.5	BinOp	74
19.6	FAcc	75
19.7	FAss	75
19.8	::	76
19.9	If	76
19.10	While	77
19.11	Throw	77
19.12	InitBlock	78
19.13	Block	78
19.14	List	79
19.15	Call	79
19.16	The main Theorem	83
19.17	Big steps simulates small step	83
19.18	Equivalence	85
20	Definite assignment	87
20.1	Hypersets	87
20.2	Definite assignment	88
21	Runtime Well-typedness	90
21.1	Run time types	90
21.2	The rules	90
21.3	Easy consequences	93
21.4	Some interesting lemmas	94
22	Conformance Relations for Proofs	95
22.1	Value conformance $:\leq$	95
22.2	Value list conformance $[:\leq]$	96
22.3	Field conformance $(:\leq)$	96

22.4	Heap conformance	96
22.5	Local variable conformance	97
22.6	Environment conformance	97
22.7	Type conformance	97
23	Progress of Small Step Semantics	99
23.1	Some pre-definitions	99
23.2	The theorem <i>progress</i>	101
24	Heap Extension	103
24.1	The Heap Extension	103
24.2	\trianglelefteq and preallocated	103
24.3	\trianglelefteq in Small- and BigStep	104
24.4	\trianglelefteq and conformance	104
24.5	\trianglelefteq in the runtime type system	105
25	Well-formedness Constraints	106
26	Type Safety Proof	107
26.1	Basic preservation lemmas	107
26.2	Subject reduction	108
26.3	Lifting to \rightarrow^*	109
26.4	Lifting to \Rightarrow	110
26.5	The final polish	111
27	Determinism Proof	112
27.1	Some lemmas	112
27.2	The proof	113
28	Program annotation	114
28.1	Various additional list functions	115
28.2	Various additional set functions	115
28.3	Basic set operations	115
28.4	Functorial set operations	116
28.5	Derived set operations	117
28.6	Lifting	117
28.7	Basic operations	118
28.8	Derived operations	120
28.9	Functorial operations	120
28.10	Misc operations	121
28.11	Simplified simprules	121
28.12	Preprocessor setup	122
28.13	Code generator setup	123

29 Code generation for Semantics and Type System	125
29.1 General redefinitions	125
29.2 Rewriting lemmas for Semantic rules	127
29.3 Rewriting lemmas for Type rules	132
29.4 Code generation	133
Bibliography	140

1 Auxiliary Definitions

theory *Aux*
imports *Main While-Combinator*
begin

declare
option.splits[*split*]
Let-def[*simp*]
subset-insertI2 [*simp*]
Cons-eq-map-conv [*iff*]

lemma *nat-add-max-le*[*simp*]:
 $((n::nat) + \max i j \leq m) = (n + i \leq m \wedge n + j \leq m)$
<proof>

lemma *Suc-add-max-le*[*simp*]:
 $(\text{Suc}(n + \max i j) \leq m) = (\text{Suc}(n + i) \leq m \wedge \text{Suc}(n + j) \leq m)$
<proof>

notation *Some* ($([-])$)

lemma *butlast-tail*:
 $\text{butlast } (Xs@[X, Y]) = Xs@[X]$
<proof>

lemma *butlast-noteq*: $Cs \neq [] \implies \text{butlast } Cs \neq Cs$
<proof>

lemma *app-hd-tl*: $\llbracket Cs \neq []; Cs = Cs' @ \text{tl } Cs \rrbracket \implies Cs' = [\text{hd } Cs]$
<proof>

lemma *only-one-append*: $\llbracket C' \notin \text{set } Cs; C' \notin \text{set } Cs'; Ds @ C' \# Ds' = Cs @ C' \# Cs' \rrbracket$
 $\implies Cs = Ds \wedge Cs' = Ds'$
<proof>

constdefs
pick :: 'a set \Rightarrow 'a
pick A \equiv SOME x. x \in A

lemma *pick-is-element*: $x \in A \implies \text{pick } A \in A$
 <proof>

constdefs

set2list :: 'a set \Rightarrow 'a list
set2list A \equiv fst (while ($\lambda(Es,S). S \neq \{\}$)
 ($\lambda(Es,S). \text{let } x = \text{pick } S \text{ in } (x \# Es, S - \{x\})$)
 ($[], A$))

lemma *card-pick*: $\llbracket \text{finite } A; A \neq \{\} \rrbracket \implies \text{Suc}(\text{card}(A - \{\text{pick}(A)\})) = \text{card } A$
 <proof>

lemma *set2list-prop*: $\llbracket \text{finite } A; A \neq \{\} \rrbracket \implies$

$\exists xs. \text{while } (\lambda(Es,S). S \neq \{\})$
 ($\lambda(Es,S). \text{let } x = \text{pick } S \text{ in } (x \# Es, S - \{x\})$)
 ($[], A$) = ($xs, \{\}$) \wedge ($\text{set } xs \cup \{\} = A$)

<proof>

lemma *set2list-correct*: $\llbracket \text{finite } A; A \neq \{\}; \text{set2list } A = xs \rrbracket \implies \text{set } xs = A$
 <proof>

1.1 distinct-fst

constdefs

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

lemma *distinct-fst-Nil* [simp]:

distinct-fst []

<proof>

lemma *distinct-fst-Cons* [simp]:

distinct-fst ((k,x)#kxs) = (*distinct-fst* kxs \wedge ($\forall y. (k,y) \notin \text{set } kxs$))

<proof>

lemma *map-of-SomeI*:

$\llbracket \text{distinct-fst } kxs; (k,x) \in \text{set } kxs \rrbracket \implies \text{map-of } kxs \ k = \text{Some } x$
 <proof>

1.2 Using *list-all2* for relations

constdefs

fun-of :: ('a × 'b) set ⇒ 'a ⇒ 'b ⇒ bool
fun-of S ≡ λx y. (x,y) ∈ S

Convenience lemmas

declare *fun-of-def* [*simp*]

lemma *rel-list-all2-Cons* [*iff*]:

list-all2 (*fun-of* S) (x#xs) (y#ys) =
 ((x,y) ∈ S ∧ *list-all2* (*fun-of* S) xs ys)
 ⟨*proof*⟩

lemma *rel-list-all2-Cons1*:

list-all2 (*fun-of* S) (x#xs) ys =
 (∃ z zs. ys = z#zs ∧ (x,z) ∈ S ∧ *list-all2* (*fun-of* S) xs zs)
 ⟨*proof*⟩

lemma *rel-list-all2-Cons2*:

list-all2 (*fun-of* S) xs (y#ys) =
 (∃ z zs. xs = z#zs ∧ (z,y) ∈ S ∧ *list-all2* (*fun-of* S) zs ys)
 ⟨*proof*⟩

lemma *rel-list-all2-refl*:

(∧x. (x,x) ∈ S) ⇒ *list-all2* (*fun-of* S) xs xs
 ⟨*proof*⟩

lemma *rel-list-all2-antisym*:

[[(∧x y. [(x,y) ∈ S; (y,x) ∈ T] ⇒ x = y);
list-all2 (*fun-of* S) xs ys; *list-all2* (*fun-of* T) ys xs] ⇒ xs = ys
 ⟨*proof*⟩

lemma *rel-list-all2-trans*:

[[∧a b c. [(a,b) ∈ R; (b,c) ∈ S] ⇒ (a,c) ∈ T;
list-all2 (*fun-of* R) as bs; *list-all2* (*fun-of* S) bs cs]
 ⇒ *list-all2* (*fun-of* T) as cs
 ⟨*proof*⟩

lemma *rel-list-all2-update-cong*:

[[i < size xs; *list-all2* (*fun-of* S) xs ys; (x,y) ∈ S]
 ⇒ *list-all2* (*fun-of* S) (xs[i:=x]) (ys[i:=y])
 ⟨*proof*⟩

lemma *rel-list-all2-nthD*:

[[*list-all2* (*fun-of* S) xs ys; p < size xs] ⇒ (xs!p,ys!p) ∈ S
 ⟨*proof*⟩

lemma *rel-list-all2I*:

[[length a = length b; ∧n. n < length a ⇒ (a!n,b!n) ∈ S] ⇒ *list-all2* (*fun-of*

S) *a b*
<proof>

declare *fun-of-def* [*simp del*]

end

2 CoreC++ types

theory *Type* **imports** *Aux* **begin**

types

cname = *string* — class names
mname = *string* — method name
vname = *string* — names for local/field variables

constdefs

this :: *vname*
this \equiv "this"

— types

datatype *ty*
= *Void* — type of statements
| *Boolean*
| *Integer*
| *NT* — null type
| *Class cname* — class type

datatype *base* — superclass

= *Repeats cname* — repeated (nonvirtual) inheritance
| *Shares cname* — shared (virtual) inheritance

consts

getbase :: *base* \Rightarrow *cname*
isRepBase :: *base* \Rightarrow *bool*
isShBase :: *base* \Rightarrow *bool*

primrec

getbase (*Repeats C*) = *C*
getbase (*Shares C*) = *C*

primrec

isRepBase (*Repeats C*) = *True*
isRepBase (*Shares C*) = *False*

primrec

isShBase (*Repeats C*) = *False*
isShBase (*Shares C*) = *True*

constdefs

is-refT :: *ty* \Rightarrow *bool*
is-refT *T* \equiv $T = NT \vee (\exists C. T = \text{Class } C)$

lemma [*iff*]: *is-refT* *NT*
(*proof*)

lemma [iff]: *is-refT*(Class *C*)
⟨*proof*⟩

lemma *refTE*:
[[*is-refT* *T*; *T* = *NT* \implies *Q*; $\bigwedge C. T = \text{Class } C \implies Q$]] \implies *Q*
⟨*proof*⟩

lemma *not-refTE*:
[[\neg *is-refT* *T*; *T* = *Void* \vee *T* = *Boolean* \vee *T* = *Integer* \implies *Q*]] \implies *Q*
⟨*proof*⟩

types
env = *vname* \rightarrow *ty*

end

3 CoreC++ values

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

types

addr = *nat*
path = *cname list* — Path-component in subobjects
reference = *addr* × *path*

datatype *val*

= *Unit* — dummy result value of void expressions
| *Null* — null reference
| *Bool bool* — Boolean value
| *Intg int* — integer value
| *Ref reference* — Address on the heap and subobject-path

consts

the-Intg :: *val* ⇒ *int*
the-addr :: *val* ⇒ *addr*
the-path :: *val* ⇒ *path*

primrec

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

primrec

the-addr (*Ref r*) = *fst r*

primrec

the-path (*Ref r*) = *snd r*

consts

default-val :: *ty* ⇒ *val* — default value for all types

primrec

default-val Void = *Unit*
default-val Boolean = *Bool False*
default-val Integer = *Intg 0*
default-val NT = *Null*
default-val (Class C) = *Null*

lemma *default-val-no-Ref:default-val T = Ref(a,Cs) ⇒ False*
(*proof*)

consts

typeof :: *val* ⇒ *ty option*

primrec

typeof Unit = *Some Void*
typeof Null = *Some NT*

typeof (Bool *b*) = *Some Boolean*
typeof (Intg *i*) = *Some Integer*
typeof (Ref *r*) = *None*

lemma [*simp*]: (*typeof v = Some Boolean*) = ($\exists b. v = \text{Bool } b$)
<proof>

lemma [*simp*]: (*typeof v = Some Integer*) = ($\exists i. v = \text{Intg } i$)
<proof>

lemma [*simp*]: (*typeof v = Some NT*) = (*v = Null*)
<proof>

lemma [*simp*]: (*typeof v = Some Void*) = (*v = Unit*)
<proof>

end

4 Expressions

theory *Expr* **imports** *Value* **begin**

4.1 The expressions

datatype *bop* = *Eq* | *Add* — names of binary operations

datatype *expr*

= *new cname* — class instance creation
 | *Cast cname expr* — dynamic type cast
 | *StatCast cname expr* — static type cast
 (λ - [80,81] 80)
 | *Val val* — value
 | *BinOp expr bop expr* (\ll -> - [80,0,81] 80)
 — binary operation
 | *Var vname* — local variable
 | *LAss vname expr* ($:=$ - [70,70] 70)
 — local assignment
 | *FAcc expr vname path* ($\cdot\cdot$ -{ } [10,90,99] 90)
 — field access
 | *FAss expr vname path expr* ($\cdot\cdot$ -{ } := - [10,70,99,70] 70)
 — field assignment
 | *Call expr cname option mname expr list*
 — method call
 | *Block vname ty expr* ($\{$ -;-; -})
 | *Seq expr expr* (-; / - [61,60] 60)
 | *Cond expr expr expr* (*if* '(-) -/ *else* - [80,79,79] 70)
 | *While expr expr* (*while* '(-) - [80,79] 70)
 | *throw expr*

abbreviation (*input*)

DynCall :: *expr* \Rightarrow *mname* \Rightarrow *expr list* \Rightarrow *expr* ($\cdot\cdot$ -'(-) [90,99,0] 90) **where**
e·*M*(*es*) == *Call e None M es*

abbreviation (*input*)

StaticCall :: *expr* \Rightarrow *cname* \Rightarrow *mname* \Rightarrow *expr list* \Rightarrow *expr*
 (\cdot -'(-::')-'(-) [90,99,99,0] 90) **where**
e·(*C*::*M*(*es*)) == *Call e (Some C) M es*

The semantics of binary operators:

consts

binop :: *bop* \times *val* \times *val* \Rightarrow *val option*

recdef *binop* { }

binop(*Eq*, *v*₁, *v*₂) = *Some*(*Bool* (*v*₁ = *v*₂))
binop(*Add*, *Intg* *i*₁, *Intg* *i*₂) = *Some*(*Intg*(*i*₁+*i*₂))
binop(*bop*, *v*₁, *v*₂) = *None*

lemma [*simp*]:

$(\text{binop}(\text{Add}, v_1, v_2) = \text{Some } v) = (\exists i_1 i_2. v_1 = \text{Intg } i_1 \wedge v_2 = \text{Intg } i_2 \wedge v = \text{Intg}(i_1 + i_2))$

$\langle \text{proof} \rangle$

lemma *binop-not-ref*[*simp*]:

$\text{binop}(\text{bop}, v_1, v_2) = \text{Some } (\text{Ref } r) \implies \text{False}$

$\langle \text{proof} \rangle$

4.2 Free Variables

consts

$\text{fv} :: \text{expr} \Rightarrow \text{vname set}$

$\text{fvs} :: \text{expr list} \Rightarrow \text{vname set}$

primrec

$\text{fv}(\text{new } C) = \{\}$

$\text{fv}(\text{Cast } C e) = \text{fv } e$

$\text{fv}(\downarrow C) e = \text{fv } e$

$\text{fv}(\text{Val } v) = \{\}$

$\text{fv}(e_1 \ll \text{bop} \gg e_2) = \text{fv } e_1 \cup \text{fv } e_2$

$\text{fv}(\text{Var } V) = \{V\}$

$\text{fv}(V := e) = \{V\} \cup \text{fv } e$

$\text{fv}(e \cdot F\{Cs\}) = \text{fv } e$

$\text{fv}(e_1 \cdot F\{Cs\} := e_2) = \text{fv } e_1 \cup \text{fv } e_2$

$\text{fv}(\text{Call } e \text{ Copt } M \text{ es}) = \text{fv } e \cup \text{fvs } es$

$\text{fv}(\{V:T; e\}) = \text{fv } e - \{V\}$

$\text{fv}(e_1 ;; e_2) = \text{fv } e_1 \cup \text{fv } e_2$

$\text{fv}(\text{if } (b) e_1 \text{ else } e_2) = \text{fv } b \cup \text{fv } e_1 \cup \text{fv } e_2$

$\text{fv}(\text{while } (b) e) = \text{fv } b \cup \text{fv } e$

$\text{fv}(\text{throw } e) = \text{fv } e$

$\text{fvs}(\[]) = \{\}$

$\text{fvs}(e \# es) = \text{fv } e \cup \text{fvs } es$

lemma [*simp*]: $\text{fvs}(es_1 @ es_2) = \text{fvs } es_1 \cup \text{fvs } es_2$

$\langle \text{proof} \rangle$

lemma [*simp*]: $\text{fvs}(\text{map } \text{Val } vs) = \{\}$

$\langle \text{proof} \rangle$

end

5 Class Declarations and Programs

theory *Decl* **imports** *Expr* **begin**

types

$fdecl = vname \times ty$ — field declaration
 $method = ty\ list \times ty \times (vname\ list \times expr)$ — arg. types, return type, params,
 body
 $mdecl = mname \times method$ — method declaration
 $class = base\ list \times fdecl\ list \times mdecl\ list$ — class = superclasses, fields, methods

 $cdecl = cname \times class$ — classa declaration
 $prog = cdecl\ list$ — program

translations

$fdecl \leq (type) vname \times ty$
 $mdecl \leq (type) mname \times ty\ list \times ty \times (vname\ list \times expr)$
 $class \leq (type) cname \times fdecl\ list \times mdecl\ list$
 $cdecl \leq (type) cname \times class$
 $prog \leq (type) cdecl\ list$

constdefs

$class :: prog \Rightarrow cname \rightarrow class$
 $class \equiv map\ of$

 $is\ class :: prog \Rightarrow cname \Rightarrow bool$
 $is\ class\ P\ C \equiv class\ P\ C \neq None$

 $base\ Classes :: base\ list \Rightarrow cname\ set$
 $base\ Classes\ Bs \equiv set\ ((map\ getbase)\ Bs)$

 $Rep\ Bases :: base\ list \Rightarrow cname\ set$
 $Rep\ Bases\ Bs \equiv set\ ((map\ getbase)\ (filter\ isRepBase\ Bs))$

 $Shared\ Bases :: base\ list \Rightarrow cname\ set$
 $Shared\ Bases\ Bs \equiv set\ ((map\ getbase)\ (filter\ isShBase\ Bs))$

lemma *not-getbase-repeats*:

$D \notin set\ (map\ getbase\ xs) \implies Repeats\ D \notin set\ xs$
<proof>

lemma *not-getbase-shares*:

$D \notin set\ (map\ getbase\ xs) \implies Shares\ D \notin set\ xs$
<proof>

lemma *RepBaseclass-isBaseclass*:
 $\llbracket \text{class } P \ C = \text{Some}(Bs, fs, ms); \text{Repeats } D \in \text{set } Bs \rrbracket$
 $\implies D \in \text{baseClasses } Bs$
 $\langle \text{proof} \rangle$

lemma *ShBaseclass-isBaseclass*:
 $\llbracket \text{class } P \ C = \text{Some}(Bs, fs, ms); \text{Shares } D \in \text{set } Bs \rrbracket$
 $\implies D \in \text{baseClasses } Bs$
 $\langle \text{proof} \rangle$

lemma *base-repeats-or-shares*:
 $\llbracket B \in \text{set } Bs; D = \text{getbase } B \rrbracket$
 $\implies \text{Repeats } D \in \text{set } Bs \vee \text{Shares } D \in \text{set } Bs$
 $\langle \text{proof} \rangle$

lemma *baseClasses-repeats-or-shares*:
 $D \in \text{baseClasses } Bs \implies \text{Repeats } D \in \text{set } Bs \vee \text{Shares } D \in \text{set } Bs$
 $\langle \text{proof} \rangle$

lemma *finite-is-class*: *finite* $\{C. \text{is-class } P \ C\}$
 $\langle \text{proof} \rangle$

lemma *finite-baseClasses*:
 $\text{class } P \ C = \text{Some}(Bs, fs, ms) \implies \text{finite } (\text{baseClasses } Bs)$
 $\langle \text{proof} \rangle$

constdefs
 $\text{is-type} :: \text{prog} \Rightarrow \text{ty} \Rightarrow \text{bool}$
 $\text{is-type } P \ T \equiv$
 $(\text{case } T \text{ of } \text{Void} \Rightarrow \text{True} \mid \text{Boolean} \Rightarrow \text{True} \mid \text{Integer} \Rightarrow \text{True} \mid \text{NT} \Rightarrow \text{True}$
 $\mid \text{Class } C \Rightarrow \text{is-class } P \ C)$

lemma *is-type-simps* [*simp*]:
 $\text{is-type } P \ \text{Void} \wedge \text{is-type } P \ \text{Boolean} \wedge \text{is-type } P \ \text{Integer} \wedge$
 $\text{is-type } P \ \text{NT} \wedge \text{is-type } P \ (\text{Class } C) = \text{is-class } P \ C$
 $\langle \text{proof} \rangle$

abbreviation
 $\text{types } P == \text{Collect } (\text{CONST } \text{is-type } P)$

lemma *typeof-lit-is-type*:
 $\text{typeof } v = \text{Some } T \implies \text{is-type } P \ T$

⟨proof⟩

end

6 The subclass relation

theory *ClassRel* **imports** *Decl* **begin**

— direct repeated subclass

inductive-set

subclsR :: *prog* \Rightarrow (*cname* \times *cname*) *set*

and *subclsR'* :: *prog* \Rightarrow [*cname*, *cname*] \Rightarrow *bool* ($- \vdash - \prec_R -$ [71,71,71] 70)

for *P* :: *prog*

where

$P \vdash C \prec_R D \equiv (C, D) \in \text{subclsR } P$

| *subclsRI*: $\llbracket \text{class } P \ C = \text{Some } (Bs, \text{rest}); \text{Repeats}(D) \in \text{set } Bs \rrbracket \Longrightarrow P \vdash C \prec_R D$

— direct shared subclass

inductive-set

subclsS :: *prog* \Rightarrow (*cname* \times *cname*) *set*

and *subclsS'* :: *prog* \Rightarrow [*cname*, *cname*] \Rightarrow *bool* ($- \vdash - \prec_S -$ [71,71,71] 70)

for *P* :: *prog*

where

$P \vdash C \prec_S D \equiv (C, D) \in \text{subclsS } P$

| *subclsSI*: $\llbracket \text{class } P \ C = \text{Some } (Bs, \text{rest}); \text{Shares}(D) \in \text{set } Bs \rrbracket \Longrightarrow P \vdash C \prec_S D$

— direct subclass

inductive-set

subcls1 :: *prog* \Rightarrow (*cname* \times *cname*) *set*

and *subcls1'* :: *prog* \Rightarrow [*cname*, *cname*] \Rightarrow *bool* ($- \vdash - \prec^1 -$ [71,71,71] 70)

for *P* :: *prog*

where

$P \vdash C \prec^1 D \equiv (C, D) \in \text{subcls1 } P$

| *subcls1I*: $\llbracket \text{class } P \ C = \text{Some } (Bs, \text{rest}); D \in \text{baseClasses } Bs \rrbracket \Longrightarrow P \vdash C \prec^1 D$

abbreviation

subcls :: *prog* \Rightarrow [*cname*, *cname*] \Rightarrow *bool* ($- \vdash - \preceq^* -$ [71,71,71] 70) **where**

$P \vdash C \preceq^* D \equiv (C, D) \in (\text{subcls1 } P)^*$

lemma *subclsRD*:

$P \vdash C \prec_R D \Longrightarrow \exists fs \ ms \ Bs. (\text{class } P \ C = \text{Some } (Bs, fs, ms)) \wedge (\text{Repeats}(D) \in \text{set } Bs)$

<proof>

lemma *subclsSD*:

$P \vdash C \prec_S D \Longrightarrow \exists fs \ ms \ Bs. (\text{class } P \ C = \text{Some } (Bs, fs, ms)) \wedge (\text{Shares}(D) \in \text{set } Bs)$

<proof>

lemma *subcls1D*:

$P \vdash C \prec^1 D \Longrightarrow \exists fs \ ms \ Bs. (\text{class } P \ C = \text{Some } (Bs, fs, ms)) \wedge (D \in \text{baseClasses } Bs)$

Bs)
<proof>

lemma *subclsR-subcls1*:
 $P \vdash C \prec_R D \implies P \vdash C \prec^1 D$
<proof>

lemma *subclsS-subcls1*:
 $P \vdash C \prec_S D \implies P \vdash C \prec^1 D$
<proof>

lemma *subcls1-subclsR-or-subclsS*:
 $P \vdash C \prec^1 D \implies P \vdash C \prec_R D \vee P \vdash C \prec_S D$
<proof>

lemma *finite-subcls1*: *finite (subcls1 P)*
<proof>

lemma *finite-subclsR*: *finite (subclsR P)*
<proof>

lemma *finite-subclsS*: *finite (subclsS P)*
<proof>

lemma *subcls1-class*:
 $P \vdash C \prec^1 D \implies \text{is-class } P \ C$
<proof>

lemma *subcls-is-class*:
 $\llbracket P \vdash D \preceq^* C; \text{is-class } P \ C \rrbracket \implies \text{is-class } P \ D$
<proof>

end

7 Definition of Subobjects

theory *SubObj* **imports** *ClassRel* **begin**

7.1 General definitions

types

$subobj = cname \times path$

consts

$mdc :: subobj \Rightarrow cname$

$ldc :: subobj \Rightarrow cname$

defs *mdc-def*:

$mdc\ S == fst\ S$

defs *ldc-def*:

$ldc\ S == last\ (snd\ S)$

lemma *mdc-tuple* [*simp*]: $mdc\ (C, Cs) = C$

<proof>

lemma *ldc-tuple* [*simp*]: $ldc\ (C, Cs) = last\ Cs$

<proof>

7.2 Subobjects according to Rossie-Friedman

consts

$is-subobj :: prog \times subobj \Rightarrow bool$ — legal subobject to class hierarchy

recdef *is-subobj* *measure* $(\lambda (P, (C, Xs)). length\ Xs)$

$is-subobj\ (P, (C, [])) = False$

$is-subobj\ (P, (C, [D])) = ((is-class\ P\ C \wedge C = D) \vee (\exists X. P \vdash C \preceq^* X \wedge P \vdash X \prec_S D))$

$is-subobj\ (P, (C, D\#E\#Xs)) = (let\ Ys = butlast\ (D\#E\#Xs);$

$Y = last\ (D\#E\#Xs);$

$X = last\ Ys$

$in\ is-subobj\ (P, (C, Ys)) \wedge P \vdash X \prec_R Y)$

lemma *subobj-aux-rev*:

assumes $1: is-subobj(P, (C, C'\#rev\ Cs@[C']))$

shows $is-subobj(P, (C, C'\#rev\ Cs))$

<proof>

lemma *subobj-aux*:

assumes $1: is-subobj(P, (C, C'\#Cs@[C']))$

shows $is-subobj(P, (C, C'\#Cs))$

<proof>

lemma *isSubobj-isClass*:
assumes $1:is-subobj (P,R)$
shows *is-class* $P (mdc R)$

<proof>

lemma *isSubobjs-subclsR-rev*:
assumes $1:is-subobj (P,(C,Cs@[D,D']@(rev Cs')))$
shows $P \vdash D \prec_R D'$
<proof>

lemma *isSubobjs-subclsR*:
assumes $1:is-subobj (P,(C,Cs@[D,D']@Cs'))$
shows $P \vdash D \prec_R D'$

<proof>

lemma *mdc-leq-ldc-aux*:
assumes $1:is-subobj(P,(C,C'\#rev Cs'))$
shows $P \vdash C \preceq^* last (C'\#rev Cs')$
<proof>

lemma *mdc-leq-ldc*:
assumes $1:is-subobj(P,R)$
shows $P \vdash mdc R \preceq^* ldc R$

<proof>

Next three lemmas show subobject property as presented in literature

lemma *class-isSubobj*:
is-class $P C \implies is-subobj (P,(C,[C]))$
<proof>

lemma *repSubobj-isSubobj*:
assumes $1:is-subobj (P,(C,Xs@[X]))$ **and** $2:P \vdash X \prec_R Y$

shows $is\text{-subobj} (P, (C, Xs@[X, Y]))$

$\langle proof \rangle$

lemma $shSubobj\text{-}isSubobj$:

assumes $1: is\text{-subobj} (P, (C, Xs@[X]))$ **and** $2: P \vdash X \prec_S Y$

shows $is\text{-subobj} (P, (C, [Y]))$

$\langle proof \rangle$

Auxiliary lemmas

lemma $build\text{-}rec\text{-}isSubobj\text{-}rev$:

assumes $1: is\text{-subobj} (P, (D, D\#rev Cs))$ **and** $2: P \vdash C \prec_R D$

shows $is\text{-subobj} (P, (C, C\#D\#rev Cs))$

$\langle proof \rangle$

lemma $build\text{-}rec\text{-}isSubobj$:

assumes $1: is\text{-subobj} (P, (D, D\#Cs))$ **and** $2: P \vdash C \prec_R D$

shows $is\text{-subobj} (P, (C, C\#D\#Cs))$

$\langle proof \rangle$

lemma $isSubobj\text{-}isSubobj\text{-}isSubobj\text{-}rev$:

assumes $1: is\text{-subobj} (P, (C, [D]))$ **and** $2: is\text{-subobj} (P, (D, D\#(rev Cs)))$

shows $is\text{-subobj} (P, (C, D\#(rev Cs)))$

$\langle proof \rangle$

lemma $isSubobj\text{-}isSubobj\text{-}isSubobj$:

assumes $1: is\text{-subobj} (P, (C, [D]))$ **and** $2: is\text{-subobj} (P, (D, D\#Cs))$

shows $is\text{-subobj} (P, (C, D\#Cs))$

$\langle proof \rangle$

7.3 Subobject handling and lemmas

Subobjects consisting of repeated inheritance relations only:

inductive $Subobjs_R :: prog \Rightarrow cname \Rightarrow path \Rightarrow bool$ **for** $P :: prog$

where

$SubobjsR\text{-}Base: is\text{-class} P C \implies Subobjs_R P C [C]$

$| SubobjsR\text{-}Rep: [P \vdash C \prec_R D; Subobjs_R P D Cs] \implies Subobjs_R P C (C \# Cs)$

All subobjects:

inductive *Subobjs* :: *prog* \Rightarrow *cname* \Rightarrow *path* \Rightarrow *bool* **for** *P* :: *prog*
where

Subobjs-Rep: $Subobjs_R P C Cs \Longrightarrow Subobjs P C Cs$
| *Subobjs-Sh*: $\llbracket P \vdash C \preceq^* C'; P \vdash C' \prec_S D; Subobjs_R P D Cs \rrbracket$
 $\Longrightarrow Subobjs P C Cs$

lemma *Subobjs-Base:is-class* $P C \Longrightarrow Subobjs P C [C]$
 $\langle proof \rangle$

lemma *SubobjsR-nonempty*: $Subobjs_R P C Cs \Longrightarrow Cs \neq []$
 $\langle proof \rangle$

lemma *Subobjs-nonempty*: $Subobjs P C Cs \Longrightarrow Cs \neq []$
 $\langle proof \rangle$

lemma *hd-SubobjsR*:
 $Subobjs_R P C Cs \Longrightarrow \exists Cs'. Cs = C \# Cs'$
 $\langle proof \rangle$

lemma *SubobjsR-subclassRep*:
 $Subobjs_R P C Cs \Longrightarrow (C, last Cs) \in (subclsR P)^*$
 $\langle proof \rangle$

lemma *SubobjsR-subclass*: $Subobjs_R P C Cs \Longrightarrow P \vdash C \preceq^* last Cs$
 $\langle proof \rangle$

lemma *Subobjs-subclass*: $Subobjs P C Cs \Longrightarrow P \vdash C \preceq^* last Cs$
 $\langle proof \rangle$

lemma *Subobjs-notSubobjsR*:
 $\llbracket Subobjs P C Cs; \neg Subobjs_R P C Cs \rrbracket$
 $\Longrightarrow \exists C' D. P \vdash C \preceq^* C' \wedge P \vdash C' \prec_S D \wedge Subobjs_R P D Cs$
 $\langle proof \rangle$

lemma assumes *subo*: $Subobjs_R P (hd (Cs @ C' \# Cs')) (Cs @ C' \# Cs')$

shows $SubobjsR\text{-}Subobjs:Subobjs\ P\ C'\ (C'\#Cs')$
 $\langle proof \rangle$

lemma $Subobjs\text{-}Subobjs:Subobjs\ P\ C\ (Cs@ C'\#Cs') \implies Subobjs\ P\ C'\ (C'\#Cs')$
 $\langle proof \rangle$

lemma $SubobjsR\text{-}isClass:$
assumes $subo:Subobjs_R\ P\ C\ Cs$
shows $is\text{-}class\ P\ C$
 $\langle proof \rangle$

lemma $Subobjs\text{-}isClass:$
assumes $subo:Subobjs\ P\ C\ Cs$
shows $is\text{-}class\ P\ C$
 $\langle proof \rangle$

lemma $Subobjs\text{-}subclsR:$
assumes $subo:Subobjs\ P\ C\ (Cs@[D,D']@Cs')$
shows $P \vdash D \prec_R D'$
 $\langle proof \rangle$

lemma **assumes** $subo:Subobjs_R\ P\ (hd\ Cs)\ (Cs@[D])$ **and** $notempty:Cs \neq []$
shows $butlast\text{-}Subobjs\text{-}Rep:Subobjs_R\ P\ (hd\ Cs)\ Cs$
 $\langle proof \rangle$

lemma **assumes** $subo:Subobjs\ P\ C\ (Cs@[D])$ **and** $notempty:Cs \neq []$
shows $butlast\text{-}Subobjs:Subobjs\ P\ C\ Cs$
 $\langle proof \rangle$

lemma **assumes** $subo:Subobjs\ P\ C\ (Cs@(rev\ Cs'))$ **and** $notempty:Cs \neq []$
shows $rev\text{-}appendSubobj:Subobjs\ P\ C\ Cs$
 $\langle proof \rangle$

lemma *appendSubobj*:
assumes *subo*:*Subobjs* *P C (Cs@Cs')* **and** *notempty*:*Cs* \neq []
shows *Subobjs* *P C Cs*

<proof>

lemma *SubobjsR-isSubobj*:
Subobjs_R P C Cs \implies *is-subobj*(*P,(C,Cs)*)
<proof>

lemma *leq-SubobjsR-isSubobj*:
 $\llbracket P \vdash C \preceq^* C'; P \vdash C' \prec_S D; \text{Subobjs}_R P D Cs \rrbracket$
 \implies *is-subobj* (*P,(C,Cs)*)

<proof>

lemma *Subobjs-isSubobj*:
Subobjs P C Cs \implies *is-subobj*(*P,(C,Cs)*)
<proof>

7.4 Paths

7.5 Appending paths

Avoided name clash by calling one path *Path*.

constdefs

path-via :: *prog* \Rightarrow *cname* \Rightarrow *cname* \Rightarrow *path* \Rightarrow *bool*
 $(- \vdash \text{Path} - \text{to} - \text{via} - [51,51,51,51] 50)$
 $P \vdash \text{Path } C \text{ to } D \text{ via } Cs \equiv \text{Subobjs } P C Cs \wedge \text{last } Cs = D$

path-unique :: *prog* \Rightarrow *cname* \Rightarrow *cname* \Rightarrow *bool*
 $(- \vdash \text{Path} - \text{to} - \text{unique} [51,51,51] 50)$
 $P \vdash \text{Path } C \text{ to } D \text{ unique} \equiv \exists! Cs. \text{Subobjs } P C Cs \wedge \text{last } Cs = D$

appendPath :: *path* \Rightarrow *path* \Rightarrow *path* (**infixr** $@_p$ 65)
 $Cs @_p Cs' \equiv \text{if } (\text{last } Cs = \text{hd } Cs') \text{ then } Cs @ (\text{tl } Cs') \text{ else } Cs'$

lemma *appendPath-last*: *Cs* \neq [] \implies *last Cs* = *last (Cs'@_pCs)*
<proof>

inductive

$casts\text{-}to :: prog \Rightarrow ty \Rightarrow val \Rightarrow val \Rightarrow bool$
 $(- \vdash - casts\text{-}to - [51,51,51,51] 50)$

for $P :: prog$

where

$casts\text{-}prim: \forall C. T \neq Class\ C \Longrightarrow P \vdash T\ casts\ v\ to\ v$

| $casts\text{-}null: P \vdash Class\ C\ casts\ Null\ to\ Null$

| $casts\text{-}ref: \llbracket P \vdash Path\ last\ Cs\ to\ C\ via\ Cs';\ Ds = Cs@_p\ Cs' \rrbracket$
 $\Longrightarrow P \vdash Class\ C\ casts\ Ref(a,Cs)\ to\ Ref(a,Ds)$

inductive

$Casts\text{-}to :: prog \Rightarrow ty\ list \Rightarrow val\ list \Rightarrow val\ list \Rightarrow bool$
 $(- \vdash - Casts\text{-}to - [51,51,51,51] 50)$

for $P :: prog$

where

$Casts\text{-}Nil: P \vdash []\ Casts\ []\ to\ []$

| $Casts\text{-}Cons: \llbracket P \vdash T\ casts\ v\ to\ v';\ P \vdash Ts\ Casts\ vs\ to\ vs' \rrbracket$
 $\Longrightarrow P \vdash (T\#Ts)\ Casts\ (v\#vs)\ to\ (v'\#vs')$

lemma *length-Casts-vs:*

$P \vdash Ts\ Casts\ vs\ to\ vs' \Longrightarrow length\ Ts = length\ vs$
 $\langle proof \rangle$

lemma *length-Casts-vs':*

$P \vdash Ts\ Casts\ vs\ to\ vs' \Longrightarrow length\ Ts = length\ vs'$
 $\langle proof \rangle$

7.6 The relation on paths

inductive-set

$leq\text{-}path1 :: prog \Rightarrow cname \Rightarrow (path \times path)\ set$

and $leq\text{-}path1' :: prog \Rightarrow cname \Rightarrow [path, path] \Rightarrow bool\ (-, - \vdash - \sqsubset^1 - [71,71,71]$
 $70)$

for $P :: prog$ **and** $C :: cname$

where

$P, C \vdash Cs \sqsubset^1 Ds \equiv (Cs, Ds) \in leq\text{-}path1\ P\ C$

| $leq\text{-}pathRep: \llbracket Subobjs\ P\ C\ Cs; Subobjs\ P\ C\ Ds; Cs = butlast\ Ds \rrbracket$
 $\Longrightarrow P, C \vdash Cs \sqsubset^1 Ds$

| $leq\text{-}pathSh: \llbracket Subobjs\ P\ C\ Cs; P \vdash last\ Cs \prec_S D \rrbracket$
 $\Longrightarrow P, C \vdash Cs \sqsubset^1 [D]$

abbreviation

$leq\text{-}path :: prog \Rightarrow cname \Rightarrow [path, path] \Rightarrow bool$ ($-, - \vdash - \sqsubseteq -$ [71,71,71] 70)

where

$P, C \vdash Cs \sqsubseteq Ds \equiv (Cs, Ds) \in (leq\text{-}path1\ P\ C)^*$

lemma *leq-path-rep*:

$\llbracket Subobjs\ P\ C\ (Cs@[C']);\ Subobjs\ P\ C\ (Cs@[C',C'']) \rrbracket$
 $\implies P, C \vdash (Cs@[C']) \sqsubseteq^1 (Cs@[C',C''])$
 $\langle proof \rangle$

lemma *leq-path-sh*:

$\llbracket Subobjs\ P\ C\ (Cs@[C']);\ P \vdash C' \prec_S C'' \rrbracket$
 $\implies P, C \vdash (Cs@[C']) \sqsubseteq^1 [C'']$
 $\langle proof \rangle$

7.7 Member lookups

constdefs

$FieldDecls :: prog \Rightarrow cname \Rightarrow vname \Rightarrow (path \times ty)\ set$

$FieldDecls\ P\ C\ F \equiv$

$\{(Cs, T). Subobjs\ P\ C\ Cs \wedge (\exists Bs\ fs\ ms. class\ P\ (last\ Cs) = Some(Bs, fs, ms)$
 $\wedge map\text{-}of\ fs\ F = Some\ T)\}$

$LeastFieldDecl :: prog \Rightarrow cname \Rightarrow vname \Rightarrow ty \Rightarrow path \Rightarrow bool$

($-\vdash -$ has least $-$ via - [51,0,0,0,51] 50)

$P \vdash C$ has least $F:T$ via $Cs \equiv$

$(Cs, T) \in FieldDecls\ P\ C\ F \wedge$
 $(\forall (Cs', T') \in FieldDecls\ P\ C\ F. P, C \vdash Cs \sqsubseteq Cs')$

$MethodDecls :: prog \Rightarrow cname \Rightarrow mname \Rightarrow (path \times method)\ set$

$MethodDecls\ P\ C\ M \equiv$

$\{(Cs, mthd). Subobjs\ P\ C\ Cs \wedge (\exists Bs\ fs\ ms. class\ P\ (last\ Cs) = Some(Bs, fs, ms)$
 $\wedge map\text{-}of\ ms\ M = Some\ mthd)\}$

— needed for well formed criterion

$HasMethodDef :: prog \Rightarrow cname \Rightarrow mname \Rightarrow method \Rightarrow path \Rightarrow bool$

($-\vdash -$ has $- = -$ via - [51,0,0,0,51] 50)

$P \vdash C$ has $M = mthd$ via $Cs \equiv (Cs, mthd) \in MethodDecls\ P\ C\ M$

$LeastMethodDef :: prog \Rightarrow cname \Rightarrow mname \Rightarrow method \Rightarrow path \Rightarrow bool$

($-\vdash -$ has least $- = -$ via - [51,0,0,0,51] 50)

$P \vdash C$ has least $M = mthd$ via $Cs \equiv$

$(Cs, mthd) \in MethodDecls\ P\ C\ M \wedge$
 $(\forall (Cs', mthd') \in MethodDecls\ P\ C\ M. P, C \vdash Cs \sqsubseteq Cs')$

$MinimalMethodDefs :: prog \Rightarrow cname \Rightarrow mname \Rightarrow (path \times method)set$
 $MinimalMethodDefs P C M \equiv$
 $\{(Cs, mthd). (Cs, mthd) \in MethodDefs P C M \wedge$
 $(\forall (Cs', mthd') \in MethodDefs P C M. P, C \vdash Cs' \sqsubseteq Cs \longrightarrow Cs' = Cs)\}$

$OverriderMethodDefs :: prog \Rightarrow subobj \Rightarrow mname \Rightarrow (path \times method)set$
 $OverriderMethodDefs P R M \equiv$
 $\{(Cs, mthd). \exists Cs' mthd'. P \vdash (lde R) \text{ has least } M = mthd' \text{ via } Cs' \wedge$
 $(Cs, mthd) \in MinimalMethodDefs P (mdc R) M \wedge$
 $P, mdc R \vdash Cs \sqsubseteq (snd R)@_p Cs'\}$

$FinalOverriderMethodDef :: prog \Rightarrow subobj \Rightarrow mname \Rightarrow method \Rightarrow path \Rightarrow bool$
 $(- \vdash - \text{ has overrider } - = - \text{ via } - [51, 0, 0, 0, 51] 50)$
 $P \vdash R \text{ has overrider } M = mthd \text{ via } Cs \equiv$
 $(Cs, mthd) \in OverriderMethodDefs P R M \wedge$
 $card(OverriderMethodDefs P R M) = 1$

inductive

$SelectMethodDef :: prog \Rightarrow cname \Rightarrow path \Rightarrow mname \Rightarrow method \Rightarrow path \Rightarrow bool$
 $(- \vdash '(-, -)' \text{ selects } - = - \text{ via } - [51, 0, 0, 0, 51] 50)$

for $P :: prog$

where

dyn-unique:

$P \vdash C \text{ has least } M = mthd \text{ via } Cs' \implies P \vdash (C, Cs) \text{ selects } M = mthd \text{ via } Cs'$

| *dyn-ambiguous:*

$\llbracket \forall mthd Cs'. \neg P \vdash C \text{ has least } M = mthd \text{ via } Cs';$

$P \vdash (C, Cs) \text{ has overrider } M = mthd \text{ via } Cs' \rrbracket$

$\implies P \vdash (C, Cs) \text{ selects } M = mthd \text{ via } Cs'$

lemma sees-fields-fun:

$(Cs, T) \in FieldDecls P C F \implies (Cs, T') \in FieldDecls P C F \implies T = T'$

<proof>

lemma sees-field-fun:

$\llbracket P \vdash C \text{ has least } F:T \text{ via } Cs; P \vdash C \text{ has least } F:T' \text{ via } Cs \rrbracket$

$\implies T = T'$

<proof>

lemma has-least-method-has-method:

$P \vdash C \text{ has least } M = mthd \text{ via } Cs \implies P \vdash C \text{ has } M = mthd \text{ via } Cs$

<proof>

lemma *visible-methods-exist*:

$(Cs, mthd) \in \text{MethodDefs } P \ C \ M \implies$
 $(\exists Bs \ fs \ ms. \text{ class } P \ (\text{last } Cs) = \text{Some}(Bs, fs, ms) \wedge \text{map-of } ms \ M = \text{Some } mthd)$
 $\langle \text{proof} \rangle$

lemma *sees-methods-fun*:

$(Cs, mthd) \in \text{MethodDefs } P \ C \ M \implies (Cs, mthd') \in \text{MethodDefs } P \ C \ M \implies mthd$
 $= mthd'$
 $\langle \text{proof} \rangle$

lemma *sees-method-fun*:

$\llbracket P \vdash C \text{ has least } M = mthd \text{ via } Cs; P \vdash C \text{ has least } M = mthd' \text{ via } Cs \rrbracket$
 $\implies mthd = mthd'$
 $\langle \text{proof} \rangle$

lemma *overrider-method-fun*:

assumes *overrider*: $P \vdash (C, Cs) \text{ has overrider } M = mthd \text{ via } Cs'$
and *overrider'*: $P \vdash (C, Cs) \text{ has overrider } M = mthd' \text{ via } Cs''$
shows $mthd = mthd' \wedge Cs' = Cs''$
 $\langle \text{proof} \rangle$

end

8 Objects and the Heap

theory *Objects* imports *SubObj* begin

8.1 Objects

types

$subo = (path \times (vname \rightarrow val))$ — subobjects realized on the heap
 $obj = cname \times subo\ set$ — mdc and subobject

constdefs

$init-class-fieldmap :: prog \Rightarrow cname \Rightarrow (vname \rightarrow val)$
 $init-class-fieldmap\ P\ C \equiv$
 $map-of\ (map\ (\lambda(F,T).(F,default-val\ T))\ (fst(snd(the(class\ P\ C))))\)$

inductive

$init-obj :: prog \Rightarrow cname \Rightarrow (path \times (vname \rightarrow val)) \Rightarrow bool$
for $P :: prog$ **and** $C :: cname$

where

$Subobjs\ P\ C\ Cs \Longrightarrow init-obj\ P\ C\ (Cs,init-class-fieldmap\ P\ (last\ Cs))$

lemma *init-obj-nonempty*: $init-obj\ P\ C\ (Cs,fs) \Longrightarrow Cs \neq []$
(*proof*)

lemma *init-obj-no-Ref*:

$\llbracket init-obj\ P\ C\ (Cs,fs); fs\ F = Some(Ref(a',Cs')) \rrbracket \Longrightarrow False$
(*proof*)

lemma *SubobjsSet-init-objSet*:

$\{Cs.\ Subobjs\ P\ C\ Cs\} = \{Cs.\ \exists\ vmap.\ init-obj\ P\ C\ (Cs,vmap)\}$
(*proof*)

constdefs

$obj-ty :: obj \Rightarrow ty$
 $obj-ty\ obj \equiv Class\ (fst\ obj)$

— a new, blank object with default values in all fields:

$blank :: prog \Rightarrow cname \Rightarrow obj$
 $blank\ P\ C \equiv (C,\ Collect\ (init-obj\ P\ C))$

lemma [*simp*]: $obj-ty\ (C,S) = Class\ C$
(*proof*)

8.2 Heap

types $heap = addr \rightarrow obj$

abbreviation

$cname-of :: heap \Rightarrow addr \Rightarrow cname$ **where**
 $cname-of\ hp\ a == fst\ (the\ (hp\ a))$

constdefs

$new-Addr :: heap \Rightarrow addr\ option$
 $new-Addr\ h \equiv if\ \exists a. h\ a = None\ then\ Some(SOME\ a.\ h\ a = None)\ else\ None$

lemma *new-Addr-SomeD*:

$new-Addr\ h = Some\ a \implies h\ a = None$
<proof>

end

9 Exceptions

theory *Exceptions* **imports** *Objects* **begin**

9.1 Exceptions

constdefs

NullPointer :: *cname*
NullPointer \equiv "*NullPointer*"

ClassCast :: *cname*
ClassCast \equiv "*ClassCast*"

OutOfMemory :: *cname*
OutOfMemory \equiv "*OutOfMemory*"

sys-xcpts :: *cname set*
sys-xcpts \equiv {*NullPointer*, *ClassCast*, *OutOfMemory*}

addr-of-sys-xcpt :: *cname* \Rightarrow *addr*
addr-of-sys-xcpt *s* \equiv if *s* = *NullPointer* then 0 else
if *s* = *ClassCast* then 1 else
if *s* = *OutOfMemory* then 2 else undefined

start-heap :: *prog* \Rightarrow *heap*
start-heap *P* \equiv empty (*addr-of-sys-xcpt* *NullPointer* \mapsto blank *P* *NullPointer*)
(*addr-of-sys-xcpt* *ClassCast* \mapsto blank *P* *ClassCast*)
(*addr-of-sys-xcpt* *OutOfMemory* \mapsto blank *P* *OutOfMemory*)

preallocated :: *heap* \Rightarrow *bool*
preallocated *h* \equiv $\forall C \in \text{sys-xcpts}. \exists S. h (\text{addr-of-sys-xcpt } C) = \text{Some } (C, S)$

9.2 System exceptions

lemma [*simp*]:

NullPointer \in *sys-xcpts* \wedge *OutOfMemory* \in *sys-xcpts* \wedge *ClassCast* \in *sys-xcpts*
(*proof*)

lemma *sys-xcpts-cases* [*consumes 1, cases set*]:

$\llbracket C \in \text{sys-xcpts}; P \text{ NullPointer}; P \text{ OutOfMemory}; P \text{ ClassCast} \rrbracket \Longrightarrow P C$
(*proof*)

9.3 preallocated

lemma *preallocated-dom* [*simp*]:

$\llbracket \text{preallocated } h; C \in \text{sys-xcpts} \rrbracket \Longrightarrow \text{addr-of-sys-xcpt } C \in \text{dom } h$
(*proof*)

lemma *preallocatedD*:

$\llbracket \text{preallocated } h; C \in \text{sys-xcpts} \rrbracket \implies \exists S. h (\text{addr-of-sys-xcpt } C) = \text{Some } (C,S)$
<proof>

lemma *preallocatedE* [*elim?*]:

$\llbracket \text{preallocated } h; C \in \text{sys-xcpts}; \bigwedge S. h (\text{addr-of-sys-xcpt } C) = \text{Some}(C,S) \implies P \ h \ C \rrbracket$
 $\implies P \ h \ C$
<proof>

lemma *cname-of-xcp* [*simp*]:

$\llbracket \text{preallocated } h; C \in \text{sys-xcpts} \rrbracket \implies \text{cname-of } h (\text{addr-of-sys-xcpt } C) = C$
<proof>

lemma *preallocated-start*:

preallocated (start-heap P)
<proof>

9.4 *start-heap*

lemma *start-Subobj*:

$\llbracket \text{start-heap } P \ a = \text{Some}(C, S); (Cs,fs) \in S \rrbracket \implies \text{Subobjs } P \ C \ Cs$
<proof>

lemma *start-SuboSet*:

$\llbracket \text{start-heap } P \ a = \text{Some}(C, S); \text{Subobjs } P \ C \ Cs \rrbracket \implies \exists fs. (Cs,fs) \in S$
<proof>

lemma *start-init-obj*: *start-heap P a = Some(C,S) $\implies S = \text{Collect } (\text{init-obj } P \ C)$*

<proof>

lemma *start-subobj*:

$\llbracket \text{start-heap } P \ a = \text{Some}(C, S); \exists fs. (Cs, fs) \in S \rrbracket \implies \text{Subobjs } P \ C \ Cs$
<proof>

end

10 Syntax

theory *Syntax* **imports** *Exceptions* **begin**

Syntactic sugar

abbreviation (*input*)

InitBlock :: *vname* \Rightarrow *ty* \Rightarrow *expr* \Rightarrow *expr* \Rightarrow *expr* ((*l*'{-: - := -;/ -})) **where**
InitBlock *V T e1 e2* == { *V:T*; *V := e1*;; *e2*}

abbreviation *unit* **where** *unit* == *Val Unit*

abbreviation *null* **where** *null* == *Val Null*

abbreviation *ref r* == *Val(Ref r)*

abbreviation *true* == *Val(Bool True)*

abbreviation *false* == *Val(Bool False)*

abbreviation

Throw :: *reference* \Rightarrow *expr* **where**

Throw r == *throw(ref r)*

abbreviation (*input*)

THROW :: *cname* \Rightarrow *expr* **where**

THROW xc == *Throw(addr-of-sys-xcpt xc,[xc])*

end

11 Program State

theory *State* **imports** *Exceptions* **begin**

types

locals = *vname* \rightarrow *val* — local vars, incl. params and “this”
state = *heap* \times *locals*

constdefs

hp :: *state* \Rightarrow *heap*
hp \equiv *fst*
lcl :: *state* \Rightarrow *locals*
lcl \equiv *snd*

declare *hp-def*[*simp*] *lcl-def*[*simp*]

end

12 Big Step Semantics

theory *BigStep* **imports** *Syntax State* **begin**

12.1 The rules

inductive

eval :: *prog* \Rightarrow *env* \Rightarrow *expr* \Rightarrow *state* \Rightarrow *expr* \Rightarrow *state* \Rightarrow *bool*
 ($\cdot, - \vdash ((1\langle \cdot, / - \rangle) \Rightarrow / (1\langle \cdot, / - \rangle))$) [51,0,0,0,0] 81)

and *evals* :: *prog* \Rightarrow *env* \Rightarrow *expr list* \Rightarrow *state* \Rightarrow *expr list* \Rightarrow *state* \Rightarrow *bool*
 ($\cdot, - \vdash ((1\langle \cdot, / - \rangle) [\Rightarrow] / (1\langle \cdot, / - \rangle))$) [51,0,0,0,0] 81)

for *P* :: *prog*

where

New:

\llbracket *new-Addr* *h* = *Some a*; *h'* = *h*(*a* \mapsto (*C*, *Collect (init-obj P C)*)) \rrbracket
 $\Longrightarrow P, E \vdash \langle \text{new } C, (h, l) \rangle \Rightarrow \langle \text{ref } (a, [C]), (h', l) \rangle$

| *NewFail*:

new-Addr h = *None* \Longrightarrow
 $P, E \vdash \langle \text{new } C, (h, l) \rangle \Rightarrow \langle \text{THROW OutOfMemory}, (h, l) \rangle$

| *StaticUpCast*:

$\llbracket P, E \vdash \langle e, s_0 \rangle \Rightarrow \langle \text{ref } (a, Cs), s_1 \rangle; P \vdash \text{Path last } Cs \text{ to } C \text{ via } Cs'; Ds = Cs @_p Cs' \rrbracket$
 $\Longrightarrow P, E \vdash \langle \llbracket C \rrbracket e, s_0 \rangle \Rightarrow \langle \text{ref } (a, Ds), s_1 \rangle$

| *StaticDownCast*:

$P, E \vdash \langle e, s_0 \rangle \Rightarrow \langle \text{ref } (a, Cs @ [C] @ Cs'), s_1 \rangle$
 $\Longrightarrow P, E \vdash \langle \llbracket C \rrbracket e, s_0 \rangle \Rightarrow \langle \text{ref } (a, Cs @ [C]), s_1 \rangle$

| *StaticCastNull*:

$P, E \vdash \langle e, s_0 \rangle \Rightarrow \langle \text{null}, s_1 \rangle \Longrightarrow$
 $P, E \vdash \langle \llbracket C \rrbracket e, s_0 \rangle \Rightarrow \langle \text{null}, s_1 \rangle$

| *StaticCastFail*:

$\llbracket P, E \vdash \langle e, s_0 \rangle \Rightarrow \langle \text{ref } (a, Cs), s_1 \rangle; \neg P \vdash (\text{last } Cs) \preceq^* C; C \notin \text{set } Cs \rrbracket$
 $\Longrightarrow P, E \vdash \langle \llbracket C \rrbracket e, s_0 \rangle \Rightarrow \langle \text{THROW ClassCast}, s_1 \rangle$

| *StaticCastThrow*:

$P, E \vdash \langle e, s_0 \rangle \Rightarrow \langle \text{throw } e', s_1 \rangle \Longrightarrow$
 $P, E \vdash \langle \llbracket C \rrbracket e, s_0 \rangle \Rightarrow \langle \text{throw } e', s_1 \rangle$

| *StaticUpDynCast*:

$\llbracket P, E \vdash \langle e, s_0 \rangle \Rightarrow \langle \text{ref } (a, Cs), s_1 \rangle; P \vdash \text{Path last } Cs \text{ to } C \text{ unique};$
 $P \vdash \text{Path last } Cs \text{ to } C \text{ via } Cs'; Ds = Cs @_p Cs' \rrbracket$
 $\Longrightarrow P, E \vdash \langle \text{Cast } C \ e, s_0 \rangle \Rightarrow \langle \text{ref } (a, Ds), s_1 \rangle$

| *StaticDownDynCast*:

$P, E \vdash \langle e, s_0 \rangle \Rightarrow \langle \text{ref } (a, Cs @ [C] @ Cs'), s_1 \rangle$

$$\implies P, E \vdash \langle \text{Cast } C \ e, s_0 \rangle \Rightarrow \langle \text{ref } (a, Cs@[C]), s_1 \rangle$$

| *DynCast*:

$$\begin{aligned} & \llbracket P, E \vdash \langle e, s_0 \rangle \Rightarrow \langle \text{ref } (a, Cs), (h, l) \rangle; h \ a = \text{Some}(D, S); \\ & \quad P \vdash \text{Path } D \text{ to } C \text{ via } Cs'; P \vdash \text{Path } D \text{ to } C \text{ unique} \rrbracket \\ & \implies P, E \vdash \langle \text{Cast } C \ e, s_0 \rangle \Rightarrow \langle \text{ref } (a, Cs'), (h, l) \rangle \end{aligned}$$

| *DynCastNull*:

$$\begin{aligned} & P, E \vdash \langle e, s_0 \rangle \Rightarrow \langle \text{null}, s_1 \rangle \implies \\ & P, E \vdash \langle \text{Cast } C \ e, s_0 \rangle \Rightarrow \langle \text{null}, s_1 \rangle \end{aligned}$$

| *DynCastFail*:

$$\begin{aligned} & \llbracket P, E \vdash \langle e, s_0 \rangle \Rightarrow \langle \text{ref } (a, Cs), (h, l) \rangle; h \ a = \text{Some}(D, S); \neg P \vdash \text{Path } D \text{ to } C \\ & \text{unique}; \\ & \quad \neg P \vdash \text{Path last } Cs \text{ to } C \text{ unique}; C \notin \text{set } Cs \rrbracket \\ & \implies P, E \vdash \langle \text{Cast } C \ e, s_0 \rangle \Rightarrow \langle \text{null}, (h, l) \rangle \end{aligned}$$

| *DynCastThrow*:

$$\begin{aligned} & P, E \vdash \langle e, s_0 \rangle \Rightarrow \langle \text{throw } e', s_1 \rangle \implies \\ & P, E \vdash \langle \text{Cast } C \ e, s_0 \rangle \Rightarrow \langle \text{throw } e', s_1 \rangle \end{aligned}$$

| *Val*:

$$P, E \vdash \langle \text{Val } v, s \rangle \Rightarrow \langle \text{Val } v, s \rangle$$

| *BinOp*:

$$\begin{aligned} & \llbracket P, E \vdash \langle e_1, s_0 \rangle \Rightarrow \langle \text{Val } v_1, s_1 \rangle; P, E \vdash \langle e_2, s_1 \rangle \Rightarrow \langle \text{Val } v_2, s_2 \rangle; \\ & \quad \text{binop}(bop, v_1, v_2) = \text{Some } v \rrbracket \\ & \implies P, E \vdash \langle e_1 \ll bop \gg e_2, s_0 \rangle \Rightarrow \langle \text{Val } v, s_2 \rangle \end{aligned}$$

| *BinOpThrow1*:

$$\begin{aligned} & P, E \vdash \langle e_1, s_0 \rangle \Rightarrow \langle \text{throw } e, s_1 \rangle \implies \\ & P, E \vdash \langle e_1 \ll bop \gg e_2, s_0 \rangle \Rightarrow \langle \text{throw } e, s_1 \rangle \end{aligned}$$

| *BinOpThrow2*:

$$\begin{aligned} & \llbracket P, E \vdash \langle e_1, s_0 \rangle \Rightarrow \langle \text{Val } v_1, s_1 \rangle; P, E \vdash \langle e_2, s_1 \rangle \Rightarrow \langle \text{throw } e, s_2 \rangle \rrbracket \\ & \implies P, E \vdash \langle e_1 \ll bop \gg e_2, s_0 \rangle \Rightarrow \langle \text{throw } e, s_2 \rangle \end{aligned}$$

| *Var*:

$$\begin{aligned} & l \ V = \text{Some } v \implies \\ & P, E \vdash \langle \text{Var } V, (h, l) \rangle \Rightarrow \langle \text{Val } v, (h, l) \rangle \end{aligned}$$

| *LAss*:

$$\begin{aligned} & \llbracket P, E \vdash \langle e, s_0 \rangle \Rightarrow \langle \text{Val } v, (h, l) \rangle; E \ V = \text{Some } T; \\ & \quad P \vdash T \text{ casts } v \text{ to } v'; l' = l(V \mapsto v') \rrbracket \\ & \implies P, E \vdash \langle V := e, s_0 \rangle \Rightarrow \langle \text{Val } v', (h, l') \rangle \end{aligned}$$

| *LAssThrow*:

$$\begin{aligned} & P, E \vdash \langle e, s_0 \rangle \Rightarrow \langle \text{throw } e', s_1 \rangle \implies \\ & P, E \vdash \langle V := e, s_0 \rangle \Rightarrow \langle \text{throw } e', s_1 \rangle \end{aligned}$$

| *FAcc*:
 $\llbracket P, E \vdash \langle e, s_0 \rangle \Rightarrow \langle \text{ref } (a, Cs'), (h, l) \rangle; h a = \text{Some}(D, S);$
 $Ds = Cs' @_p Cs; (Ds, fs) \in S; fs F = \text{Some } v \rrbracket$
 $\Rightarrow P, E \vdash \langle e \cdot F \{Cs\}, s_0 \rangle \Rightarrow \langle \text{Val } v, (h, l) \rangle$

| *FAccNull*:
 $P, E \vdash \langle e, s_0 \rangle \Rightarrow \langle \text{null}, s_1 \rangle \Rightarrow$
 $P, E \vdash \langle e \cdot F \{Cs\}, s_0 \rangle \Rightarrow \langle \text{THROW NullPointer}, s_1 \rangle$

| *FAccThrow*:
 $P, E \vdash \langle e, s_0 \rangle \Rightarrow \langle \text{throw } e', s_1 \rangle \Rightarrow$
 $P, E \vdash \langle e \cdot F \{Cs\}, s_0 \rangle \Rightarrow \langle \text{throw } e', s_1 \rangle$

| *FAss*:
 $\llbracket P, E \vdash \langle e_1, s_0 \rangle \Rightarrow \langle \text{ref } (a, Cs'), s_1 \rangle; P, E \vdash \langle e_2, s_1 \rangle \Rightarrow \langle \text{Val } v, (h_2, l_2) \rangle;$
 $h_2 a = \text{Some}(D, S); P \vdash (\text{last } Cs' \text{ has least } F:T \text{ via } Cs; P \vdash T \text{ casts } v \text{ to } v');$
 $Ds = Cs' @_p Cs; (Ds, fs) \in S; fs' = fs(F \mapsto v');$
 $S' = S - \{(Ds, fs)\} \cup \{(Ds, fs')\}; h_2' = h_2(a \mapsto (D, S')) \rrbracket$
 $\Rightarrow P, E \vdash \langle e_1 \cdot F \{Cs\} := e_2, s_0 \rangle \Rightarrow \langle \text{Val } v', (h_2', l_2) \rangle$

| *FAssNull*:
 $\llbracket P, E \vdash \langle e_1, s_0 \rangle \Rightarrow \langle \text{null}, s_1 \rangle; P, E \vdash \langle e_2, s_1 \rangle \Rightarrow \langle \text{Val } v, s_2 \rangle \rrbracket \Rightarrow$
 $P, E \vdash \langle e_1 \cdot F \{Cs\} := e_2, s_0 \rangle \Rightarrow \langle \text{THROW NullPointer}, s_2 \rangle$

| *FAssThrow1*:
 $P, E \vdash \langle e_1, s_0 \rangle \Rightarrow \langle \text{throw } e', s_1 \rangle \Rightarrow$
 $P, E \vdash \langle e_1 \cdot F \{Cs\} := e_2, s_0 \rangle \Rightarrow \langle \text{throw } e', s_1 \rangle$

| *FAssThrow2*:
 $\llbracket P, E \vdash \langle e_1, s_0 \rangle \Rightarrow \langle \text{Val } v, s_1 \rangle; P, E \vdash \langle e_2, s_1 \rangle \Rightarrow \langle \text{throw } e', s_2 \rangle \rrbracket$
 $\Rightarrow P, E \vdash \langle e_1 \cdot F \{Cs\} := e_2, s_0 \rangle \Rightarrow \langle \text{throw } e', s_2 \rangle$

| *CallObjThrow*:
 $P, E \vdash \langle e, s_0 \rangle \Rightarrow \langle \text{throw } e', s_1 \rangle \Rightarrow$
 $P, E \vdash \langle \text{Call } e \text{ Copt } M \text{ } es, s_0 \rangle \Rightarrow \langle \text{throw } e', s_1 \rangle$

| *CallParamsThrow*:
 $\llbracket P, E \vdash \langle e, s_0 \rangle \Rightarrow \langle \text{Val } v, s_1 \rangle; P, E \vdash \langle es, s_1 \rangle [\Rightarrow] \langle \text{map Val } vs @ \text{throw } ex \# es', s_2 \rangle$
 \rrbracket
 $\Rightarrow P, E \vdash \langle \text{Call } e \text{ Copt } M \text{ } es, s_0 \rangle \Rightarrow \langle \text{throw } ex, s_2 \rangle$

| *Call*:
 $\llbracket P, E \vdash \langle e, s_0 \rangle \Rightarrow \langle \text{ref } (a, Cs), s_1 \rangle; P, E \vdash \langle ps, s_1 \rangle [\Rightarrow] \langle \text{map Val } vs, (h_2, l_2) \rangle;$
 $h_2 a = \text{Some}(C, S); P \vdash \text{last } Cs \text{ has least } M = (Ts', T', pns', \text{body}') \text{ via } Ds;$
 $P \vdash (C, Cs @_p Ds) \text{ selects } M = (Ts, T, pns, \text{body}) \text{ via } Cs'; \text{length } vs = \text{length}$
 $pns;$
 $P \vdash Ts \text{ Casts } vs \text{ to } vs'; l_2' = [\text{this} \mapsto \text{Ref } (a, Cs'), pns[\mapsto] vs'];$
 $\text{new-body} = (\text{case } T' \text{ of Class } D \Rightarrow ([D])\text{body} \quad | - \Rightarrow \text{body});$

$$\begin{aligned} & P, E(\text{this} \mapsto \text{Class}(\text{last } Cs'), \text{pns}[\mapsto] Ts) \vdash \langle \text{new-body}, (h_2, l_2') \rangle \Rightarrow \langle e', (h_3, l_3) \rangle \text{]} \\ \Rightarrow & P, E \vdash \langle e \cdot M(\text{ps}), s_0 \rangle \Rightarrow \langle e', (h_3, l_2) \rangle \end{aligned}$$

| *StaticCall*:

$$\begin{aligned} & \text{[} P, E \vdash \langle e, s_0 \rangle \Rightarrow \langle \text{ref } (a, Cs), s_1 \rangle; P, E \vdash \langle \text{ps}, s_1 \rangle [\Rightarrow] \langle \text{map Val } vs, (h_2, l_2) \rangle; \\ & P \vdash \text{Path } (\text{last } Cs) \text{ to } C \text{ unique}; P \vdash \text{Path } (\text{last } Cs) \text{ to } C \text{ via } Cs''; \\ & P \vdash C \text{ has least } M = (Ts, T, \text{pns}, \text{body}) \text{ via } Cs'; Ds = (Cs @_p Cs'') @_p Cs'; \\ & \text{length } vs = \text{length } \text{pns}; P \vdash Ts \text{ Casts } vs \text{ to } vs'; \\ & l_2' = [\text{this} \mapsto \text{Ref } (a, Ds), \text{pns}[\mapsto] vs \uparrow]; \\ & P, E(\text{this} \mapsto \text{Class}(\text{last } Ds), \text{pns}[\mapsto] Ts) \vdash \langle \text{body}, (h_2, l_2') \rangle \Rightarrow \langle e', (h_3, l_3) \rangle \text{]} \\ \Rightarrow & P, E \vdash \langle e \cdot (C::)M(\text{ps}), s_0 \rangle \Rightarrow \langle e', (h_3, l_2) \rangle \end{aligned}$$

| *CallNull*:

$$\begin{aligned} & \text{[} P, E \vdash \langle e, s_0 \rangle \Rightarrow \langle \text{null}, s_1 \rangle; P, E \vdash \langle \text{es}, s_1 \rangle [\Rightarrow] \langle \text{map Val } vs, s_2 \rangle \text{]} \\ \Rightarrow & P, E \vdash \langle \text{Call } e \text{ Copt } M \text{ es}, s_0 \rangle \Rightarrow \langle \text{THROW NullPointer}, s_2 \rangle \end{aligned}$$

| *Block*:

$$\begin{aligned} & \text{[} P, E(V \mapsto T) \vdash \langle e_0, (h_0, l_0(V := \text{None})) \rangle \Rightarrow \langle e_1, (h_1, l_1) \rangle \text{]} \Rightarrow \\ & P, E \vdash \langle \{V:T; e_0\}, (h_0, l_0) \rangle \Rightarrow \langle e_1, (h_1, l_1(V := l_0 V)) \rangle \end{aligned}$$

| *Seq*:

$$\begin{aligned} & \text{[} P, E \vdash \langle e_0, s_0 \rangle \Rightarrow \langle \text{Val } v, s_1 \rangle; P, E \vdash \langle e_1, s_1 \rangle \Rightarrow \langle e_2, s_2 \rangle \text{]} \\ \Rightarrow & P, E \vdash \langle e_0;;e_1, s_0 \rangle \Rightarrow \langle e_2, s_2 \rangle \end{aligned}$$

| *SeqThrow*:

$$\begin{aligned} & P, E \vdash \langle e_0, s_0 \rangle \Rightarrow \langle \text{throw } e, s_1 \rangle \Rightarrow \\ & P, E \vdash \langle e_0;;e_1, s_0 \rangle \Rightarrow \langle \text{throw } e, s_1 \rangle \end{aligned}$$

| *CondT*:

$$\begin{aligned} & \text{[} P, E \vdash \langle e, s_0 \rangle \Rightarrow \langle \text{true}, s_1 \rangle; P, E \vdash \langle e_1, s_1 \rangle \Rightarrow \langle e', s_2 \rangle \text{]} \\ \Rightarrow & P, E \vdash \langle \text{if } (e) e_1 \text{ else } e_2, s_0 \rangle \Rightarrow \langle e', s_2 \rangle \end{aligned}$$

| *CondF*:

$$\begin{aligned} & \text{[} P, E \vdash \langle e, s_0 \rangle \Rightarrow \langle \text{false}, s_1 \rangle; P, E \vdash \langle e_2, s_1 \rangle \Rightarrow \langle e', s_2 \rangle \text{]} \\ \Rightarrow & P, E \vdash \langle \text{if } (e) e_1 \text{ else } e_2, s_0 \rangle \Rightarrow \langle e', s_2 \rangle \end{aligned}$$

| *CondThrow*:

$$\begin{aligned} & P, E \vdash \langle e, s_0 \rangle \Rightarrow \langle \text{throw } e', s_1 \rangle \Rightarrow \\ & P, E \vdash \langle \text{if } (e) e_1 \text{ else } e_2, s_0 \rangle \Rightarrow \langle \text{throw } e', s_1 \rangle \end{aligned}$$

| *WhileF*:

$$\begin{aligned} & P, E \vdash \langle e, s_0 \rangle \Rightarrow \langle \text{false}, s_1 \rangle \Rightarrow \\ & P, E \vdash \langle \text{while } (e) c, s_0 \rangle \Rightarrow \langle \text{unit}, s_1 \rangle \end{aligned}$$

| *WhileT*:

$$\begin{aligned} & \text{[} P, E \vdash \langle e, s_0 \rangle \Rightarrow \langle \text{true}, s_1 \rangle; P, E \vdash \langle c, s_1 \rangle \Rightarrow \langle \text{Val } v_1, s_2 \rangle; \\ & P, E \vdash \langle \text{while } (e) c, s_2 \rangle \Rightarrow \langle e_3, s_3 \rangle \text{]} \\ \Rightarrow & P, E \vdash \langle \text{while } (e) c, s_0 \rangle \Rightarrow \langle e_3, s_3 \rangle \end{aligned}$$

| *WhileCondThrow*:

$$\begin{aligned} P, E \vdash \langle e, s_0 \rangle &\Rightarrow \langle \text{throw } e', s_1 \rangle \Longrightarrow \\ P, E \vdash \langle \text{while } (e) \ c, s_0 \rangle &\Rightarrow \langle \text{throw } e', s_1 \rangle \end{aligned}$$

| *WhileBodyThrow*:

$$\begin{aligned} \llbracket P, E \vdash \langle e, s_0 \rangle &\Rightarrow \langle \text{true}, s_1 \rangle; P, E \vdash \langle c, s_1 \rangle \Rightarrow \langle \text{throw } e', s_2 \rangle \rrbracket \\ \Longrightarrow P, E \vdash \langle \text{while } (e) \ c, s_0 \rangle &\Rightarrow \langle \text{throw } e', s_2 \rangle \end{aligned}$$

| *Throw*:

$$\begin{aligned} P, E \vdash \langle e, s_0 \rangle &\Rightarrow \langle \text{ref } r, s_1 \rangle \Longrightarrow \\ P, E \vdash \langle \text{throw } e, s_0 \rangle &\Rightarrow \langle \text{Throw } r, s_1 \rangle \end{aligned}$$

| *ThrowNull*:

$$\begin{aligned} P, E \vdash \langle e, s_0 \rangle &\Rightarrow \langle \text{null}, s_1 \rangle \Longrightarrow \\ P, E \vdash \langle \text{throw } e, s_0 \rangle &\Rightarrow \langle \text{THROW NullPointer}, s_1 \rangle \end{aligned}$$

| *ThrowThrow*:

$$\begin{aligned} P, E \vdash \langle e, s_0 \rangle &\Rightarrow \langle \text{throw } e', s_1 \rangle \Longrightarrow \\ P, E \vdash \langle \text{throw } e, s_0 \rangle &\Rightarrow \langle \text{throw } e', s_1 \rangle \end{aligned}$$

| *Nil*:

$$P, E \vdash \langle [], s \rangle [\Rightarrow] \langle [], s \rangle$$

| *Cons*:

$$\begin{aligned} \llbracket P, E \vdash \langle e, s_0 \rangle &\Rightarrow \langle \text{Val } v, s_1 \rangle; P, E \vdash \langle es, s_1 \rangle [\Rightarrow] \langle es', s_2 \rangle \rrbracket \\ \Longrightarrow P, E \vdash \langle e \# es, s_0 \rangle [\Rightarrow] &\langle \text{Val } v \# es', s_2 \rangle \end{aligned}$$

| *ConsThrow*:

$$\begin{aligned} P, E \vdash \langle e, s_0 \rangle &\Rightarrow \langle \text{throw } e', s_1 \rangle \Longrightarrow \\ P, E \vdash \langle e \# es, s_0 \rangle [\Rightarrow] &\langle \text{throw } e' \# es, s_1 \rangle \end{aligned}$$

lemmas *eval-evals-induct* = *eval-evals.induct* [*split-format (complete)*]
and *eval-evals-inducts* = *eval-evals.inducts* [*split-format (complete)*]

inductive-cases *eval-cases* [*cases set*]:

$$\begin{aligned} P, E \vdash \langle \text{new } C, s \rangle &\Rightarrow \langle e', s' \rangle \\ P, E \vdash \langle \text{Cast } C \ e, s \rangle &\Rightarrow \langle e', s' \rangle \\ P, E \vdash \langle (\!| C |) e, s \rangle &\Rightarrow \langle e', s' \rangle \\ P, E \vdash \langle \text{Val } v, s \rangle &\Rightarrow \langle e', s' \rangle \\ P, E \vdash \langle e_1 \ll \text{bop} \gg e_2, s \rangle &\Rightarrow \langle e', s' \rangle \\ P, E \vdash \langle \text{Var } V, s \rangle &\Rightarrow \langle e', s' \rangle \\ P, E \vdash \langle V := e, s \rangle &\Rightarrow \langle e', s' \rangle \\ P, E \vdash \langle e \cdot F \{ Cs \}, s \rangle &\Rightarrow \langle e', s' \rangle \\ P, E \vdash \langle e_1 \cdot F \{ Cs \} := e_2, s \rangle &\Rightarrow \langle e', s' \rangle \\ P, E \vdash \langle e \cdot M (es), s \rangle &\Rightarrow \langle e', s' \rangle \\ P, E \vdash \langle e \cdot (C ::) M (es), s \rangle &\Rightarrow \langle e', s' \rangle \\ P, E \vdash \langle \{ V : T; e_1 \}, s \rangle &\Rightarrow \langle e', s' \rangle \\ P, E \vdash \langle e_1 ;; e_2, s \rangle &\Rightarrow \langle e', s' \rangle \\ P, E \vdash \langle \text{if } (e) \ e_1 \ \text{else} \ e_2, s \rangle &\Rightarrow \langle e', s' \rangle \end{aligned}$$

$P, E \vdash \langle \text{while } (b) \ c, s \rangle \Rightarrow \langle e', s' \rangle$
 $P, E \vdash \langle \text{throw } e, s \rangle \Rightarrow \langle e', s' \rangle$

inductive-cases *evals-cases* [*cases set*]:

$P, E \vdash \langle [], s \rangle [\Rightarrow] \langle e', s' \rangle$
 $P, E \vdash \langle e \# es, s \rangle [\Rightarrow] \langle e', s' \rangle$

12.2 Final expressions

constdefs

$\text{final} :: \text{expr} \Rightarrow \text{bool}$
 $\text{final } e \equiv (\exists v. e = \text{Val } v) \vee (\exists r. e = \text{Throw } r)$
 $\text{finals} :: \text{expr list} \Rightarrow \text{bool}$
 $\text{finals } es \equiv (\exists vs. es = \text{map Val } vs) \vee (\exists vs \ r \ es'. es = \text{map Val } vs @ \text{Throw } r \# es')$

lemma [*simp*]: $\text{final}(\text{Val } v)$
 $\langle \text{proof} \rangle$

lemma [*simp*]: $\text{final}(\text{throw } e) = (\exists r. e = \text{ref } r)$
 $\langle \text{proof} \rangle$

lemma *finalE*: $\llbracket \text{final } e; \bigwedge v. e = \text{Val } v \Longrightarrow Q; \bigwedge r. e = \text{Throw } r \Longrightarrow Q \rrbracket \Longrightarrow Q$
 $\langle \text{proof} \rangle$

lemma [*iff*]: $\text{finals } []$
 $\langle \text{proof} \rangle$

lemma [*iff*]: $\text{finals } (\text{Val } v \# es) = \text{finals } es$
 $\langle \text{proof} \rangle$

lemma *finals-app-map*[*iff*]: $\text{finals } (\text{map Val } vs @ es) = \text{finals } es$
 $\langle \text{proof} \rangle$

lemma [*iff*]: $\text{finals } (\text{map Val } vs)$
 $\langle \text{proof} \rangle$

lemma [*iff*]: $\text{finals } (\text{throw } e \# es) = (\exists r. e = \text{ref } r)$
 $\langle \text{proof} \rangle$

lemma *not-finals-ConsI*: $\neg \text{final } e \Longrightarrow \neg \text{finals}(e \# es)$
 $\langle \text{proof} \rangle$

lemma *eval-final*: $P, E \vdash \langle e, s \rangle \Rightarrow \langle e', s' \rangle \Longrightarrow \text{final } e'$
and *evals-final*: $P, E \vdash \langle es, s \rangle [\Rightarrow] \langle es', s' \rangle \Longrightarrow \text{finals } es'$
 ⟨proof⟩

lemma *eval-lcl-incr*: $P, E \vdash \langle e, (h_0, l_0) \rangle \Rightarrow \langle e', (h_1, l_1) \rangle \Longrightarrow \text{dom } l_0 \subseteq \text{dom } l_1$
and *evals-lcl-incr*: $P, E \vdash \langle es, (h_0, l_0) \rangle [\Rightarrow] \langle es', (h_1, l_1) \rangle \Longrightarrow \text{dom } l_0 \subseteq \text{dom } l_1$
 ⟨proof⟩

Only used later, in the small to big translation, but is already a good sanity check:

lemma *eval-finalId*: $\text{final } e \Longrightarrow P, E \vdash \langle e, s \rangle \Rightarrow \langle e, s \rangle$
 ⟨proof⟩

lemma *eval-finalsId*:
assumes *finals*: $\text{finals } es$ **shows** $P, E \vdash \langle es, s \rangle [\Rightarrow] \langle es, s \rangle$
 ⟨proof⟩

lemma
eval-preserves-obj: $P, E \vdash \langle e, (h, l) \rangle \Rightarrow \langle e', (h', l') \rangle \Longrightarrow (\bigwedge S. h \ a = \text{Some}(D, S) \Longrightarrow \exists S'. h' \ a = \text{Some}(D, S'))$
and *evals-preserves-obj*: $P, E \vdash \langle es, (h, l) \rangle [\Rightarrow] \langle es', (h', l') \rangle \Longrightarrow (\bigwedge S. h \ a = \text{Some}(D, S) \Longrightarrow \exists S'. h' \ a = \text{Some}(D, S'))$
 ⟨proof⟩

end

13 Small Step Semantics

theory *SmallStep* **imports** *Syntax State* **begin**

13.1 Some pre-definitions

consts *blocks* :: *vname list* \times *ty list* \times *val list* \times *expr* \Rightarrow *expr*

reodef *blocks* *measure*($\lambda(Vs, Ts, vs, e). \text{size } Vs$)

blocks-Cons: *blocks*($V\#Vs, T\#Ts, v\#vs, e$) = $\{V:T := \text{Val } v; \text{blocks}(Vs, Ts, vs, e)\}$

blocks-Nil: *blocks*($[], [], [], e$) = *e*

lemma [*simp*]:

$\llbracket \text{size } vs = \text{size } Vs; \text{size } Ts = \text{size } Vs \rrbracket \Longrightarrow \text{fv}(\text{blocks}(Vs, Ts, vs, e)) = \text{fv } e - \text{set } Vs$

<proof>

constdefs

assigned :: *vname* \Rightarrow *expr* \Rightarrow *bool*

assigned *V e* $\equiv \exists v e'. e = (V := \text{Val } v;; e')$

13.2 The rules

inductive-set

red :: *prog* \Rightarrow (*env* \times (*expr* \times *state*)) \times (*expr* \times *state*) *set*

and *reds* :: *prog* \Rightarrow (*env* \times (*expr list* \times *state*)) \times (*expr list* \times *state*) *set*

and *red'* :: *prog* \Rightarrow *env* \Rightarrow *expr* \Rightarrow *state* \Rightarrow *expr* \Rightarrow *state* \Rightarrow *bool*

($-, \vdash ((1\langle -, / - \rangle) \rightarrow / (1\langle -, / - \rangle))$) [51,0,0,0,0] 81

and *reds'* :: *prog* \Rightarrow *env* \Rightarrow *expr list* \Rightarrow *state* \Rightarrow *expr list* \Rightarrow *state* \Rightarrow *bool*

($-, \vdash ((1\langle -, / - \rangle) [\rightarrow] / (1\langle -, / - \rangle))$) [51,0,0,0,0] 81

for *P* :: *prog*

where

$P, E \vdash \langle e, s \rangle \rightarrow \langle e', s' \rangle \equiv (E, (e, s), e', s') \in \text{red } P$

$| P, E \vdash \langle es, s \rangle [\rightarrow] \langle es', s' \rangle \equiv (E, (es, s), es', s') \in \text{reds } P$

| RedNew:

$\llbracket \text{new-Addr } h = \text{Some } a; h' = h(a \mapsto (C, \text{Collect } (\text{init-obj } P C))) \rrbracket$

$\Longrightarrow P, E \vdash \langle \text{new } C, (h, l) \rangle \rightarrow \langle \text{ref } (a, [C]), (h', l) \rangle$

| RedNewFail:

$\text{new-Addr } h = \text{None} \Longrightarrow$

$P, E \vdash \langle \text{new } C, (h, l) \rangle \rightarrow \langle \text{THROW OutOfMemory}, (h, l) \rangle$

| StaticCastRed:

$P, E \vdash \langle e, s \rangle \rightarrow \langle e', s' \rangle \Longrightarrow$

$P, E \vdash \langle \llbracket C \rrbracket e, s \rangle \rightarrow \langle \llbracket C \rrbracket e', s' \rangle$

| *RedStaticCastNull*:
 $P, E \vdash \langle \langle C \rangle \text{null}, s \rangle \rightarrow \langle \text{null}, s \rangle$

| *RedStaticUpCast*:
 $\llbracket P \vdash \text{Path last } Cs \text{ to } C \text{ via } Cs'; Ds = Cs@_p Cs' \rrbracket$
 $\implies P, E \vdash \langle \langle C \rangle (\text{ref } (a, Cs)), s \rangle \rightarrow \langle \text{ref } (a, Ds), s \rangle$

| *RedStaticDownCast*:
 $P, E \vdash \langle \langle C \rangle (\text{ref } (a, Cs@[C]@Cs')), s \rangle \rightarrow \langle \text{ref } (a, Cs@[C]), s \rangle$

| *RedStaticCastFail*:
 $\llbracket C \notin \text{set } Cs; \neg P \vdash (\text{last } Cs) \preceq^* C \rrbracket$
 $\implies P, E \vdash \langle \langle C \rangle (\text{ref } (a, Cs)), s \rangle \rightarrow \langle \text{THROW ClassCast}, s \rangle$

| *DynCastRed*:
 $P, E \vdash \langle e, s \rangle \rightarrow \langle e', s' \rangle \implies$
 $P, E \vdash \langle \text{Cast } C \ e, s \rangle \rightarrow \langle \text{Cast } C \ e', s' \rangle$

| *RedDynCastNull*:
 $P, E \vdash \langle \text{Cast } C \ \text{null}, s \rangle \rightarrow \langle \text{null}, s \rangle$

| *RedStaticUpDynCast*:
 $\llbracket P \vdash \text{Path last } Cs \text{ to } C \text{ unique}; P \vdash \text{Path last } Cs \text{ to } C \text{ via } Cs'; Ds = Cs@_p Cs' \rrbracket$
 $\implies P, E \vdash \langle \text{Cast } C (\text{ref } (a, Cs)), s \rangle \rightarrow \langle \text{ref } (a, Ds), s \rangle$

| *RedStaticDownDynCast*:
 $P, E \vdash \langle \text{Cast } C (\text{ref } (a, Cs@[C]@Cs')), s \rangle \rightarrow \langle \text{ref } (a, Cs@[C]), s \rangle$

| *RedDynCast*:
 $\llbracket hp \ s \ a = \text{Some}(D, S); P \vdash \text{Path } D \text{ to } C \text{ via } Cs';$
 $\quad P \vdash \text{Path } D \text{ to } C \text{ unique} \rrbracket$
 $\implies P, E \vdash \langle \text{Cast } C (\text{ref } (a, Cs)), s \rangle \rightarrow \langle \text{ref } (a, Cs'), s \rangle$

| *RedDynCastFail*:
 $\llbracket hp \ s \ a = \text{Some}(D, S); \neg P \vdash \text{Path } D \text{ to } C \text{ unique};$
 $\quad \neg P \vdash \text{Path last } Cs \text{ to } C \text{ unique}; C \notin \text{set } Cs \rrbracket$
 $\implies P, E \vdash \langle \text{Cast } C (\text{ref } (a, Cs)), s \rangle \rightarrow \langle \text{null}, s \rangle$

| *BinOpRed1*:
 $P, E \vdash \langle e, s \rangle \rightarrow \langle e', s' \rangle \implies$
 $P, E \vdash \langle e \ll \text{bop} \gg e_2, s \rangle \rightarrow \langle e' \ll \text{bop} \gg e_2, s' \rangle$

| *BinOpRed2*:
 $P, E \vdash \langle e, s \rangle \rightarrow \langle e', s' \rangle \implies$
 $P, E \vdash \langle (\text{Val } v_1) \ll \text{bop} \gg e, s \rangle \rightarrow \langle (\text{Val } v_1) \ll \text{bop} \gg e', s' \rangle$

| *RedBinOp*:
 $\text{binop}(\text{bop}, v_1, v_2) = \text{Some } v \implies$

$$P, E \vdash \langle (Val v_1) \ll bop \gg (Val v_2), s \rangle \rightarrow \langle Val v, s \rangle$$

| *RedVar*:

$$lcl\ s\ V = Some\ v \implies$$

$$P, E \vdash \langle Var\ V, s \rangle \rightarrow \langle Val\ v, s \rangle$$

| *LAssRed*:

$$P, E \vdash \langle e, s \rangle \rightarrow \langle e', s' \rangle \implies$$

$$P, E \vdash \langle V := e, s \rangle \rightarrow \langle V := e', s' \rangle$$

| *RedLAss*:

$$\llbracket E\ V = Some\ T; P \vdash T\ casts\ v\ to\ v' \rrbracket \implies$$

$$P, E \vdash \langle V := (Val\ v), (h, l) \rangle \rightarrow \langle Val\ v', (h, l(V \mapsto v')) \rangle$$

| *FAccRed*:

$$P, E \vdash \langle e, s \rangle \rightarrow \langle e', s' \rangle \implies$$

$$P, E \vdash \langle e \cdot F\{Cs\}, s \rangle \rightarrow \langle e' \cdot F\{Cs\}, s' \rangle$$

| *RedFAcc*:

$$\llbracket hp\ s\ a = Some(D, S); Ds = Cs' @_p Cs; (Ds, fs) \in S; fs\ F = Some\ v \rrbracket$$

$$\implies P, E \vdash \langle (ref\ (a, Cs')) \cdot F\{Cs\}, s \rangle \rightarrow \langle Val\ v, s \rangle$$

| *RedFAccNull*:

$$P, E \vdash \langle null \cdot F\{Cs\}, s \rangle \rightarrow \langle THROW\ NullPointer, s \rangle$$

| *FAssRed1*:

$$P, E \vdash \langle e, s \rangle \rightarrow \langle e', s' \rangle \implies$$

$$P, E \vdash \langle e \cdot F\{Cs\} := e_2, s \rangle \rightarrow \langle e' \cdot F\{Cs\} := e_2, s' \rangle$$

| *FAssRed2*:

$$P, E \vdash \langle e, s \rangle \rightarrow \langle e', s' \rangle \implies$$

$$P, E \vdash \langle Val\ v \cdot F\{Cs\} := e, s \rangle \rightarrow \langle Val\ v \cdot F\{Cs\} := e', s' \rangle$$

| *RedFAss*:

$$\llbracket h\ a = Some(D, S); P \vdash (last\ Cs')\ has\ least\ F:T\ via\ Cs;$$

$$P \vdash T\ casts\ v\ to\ v'; Ds = Cs' @_p Cs; (Ds, fs) \in S \rrbracket \implies$$

$$P, E \vdash \langle (ref\ (a, Cs')) \cdot F\{Cs\} := (Val\ v), (h, l) \rangle \rightarrow \langle Val\ v', (h(a \mapsto (D, insert$$

$$(Ds, fs(F \mapsto v'))\ (S - \{(Ds, fs)\}))), l) \rangle$$

| *RedFAssNull*:

$$P, E \vdash \langle null \cdot F\{Cs\} := Val\ v, s \rangle \rightarrow \langle THROW\ NullPointer, s \rangle$$

| *CallObj*:

$$P, E \vdash \langle e, s \rangle \rightarrow \langle e', s' \rangle \implies$$

$$P, E \vdash \langle Call\ e\ Copt\ M\ es, s \rangle \rightarrow \langle Call\ e'\ Copt\ M\ es, s' \rangle$$

| *CallParams*:

$$P, E \vdash \langle es, s \rangle [\rightarrow] \langle es', s' \rangle \implies$$

$$P, E \vdash \langle Call\ (Val\ v)\ Copt\ M\ es, s \rangle \rightarrow \langle Call\ (Val\ v)\ Copt\ M\ es', s' \rangle$$

| *RedCall*:

$\llbracket hp \ s \ a = \text{Some}(C, S); P \vdash \text{last } Cs \text{ has least } M = (Ts', T', pns', \text{body}') \text{ via } Ds;$
 $P \vdash (C, Cs @_p Ds) \text{ selects } M = (Ts, T, pns, \text{body}) \text{ via } Cs';$
 $\text{size } vs = \text{size } pns; \text{size } Ts = \text{size } pns;$
 $bs = \text{blocks}(\text{this} \# pns, \text{Class}(\text{last } Cs') \# Ts, \text{Ref}(a, Cs') \# vs, \text{body});$
 $\text{new-body} = (\text{case } T' \text{ of Class } D \Rightarrow \langle D \rangle bs \mid - \Rightarrow bs) \rrbracket$
 $\implies P, E \vdash \langle (\text{ref } (a, Cs)) \cdot M(\text{map Val } vs), s \rangle \rightarrow \langle \text{new-body}, s \rangle$

| *RedStaticCall*:

$\llbracket P \vdash \text{Path}(\text{last } Cs) \text{ to } C \text{ unique}; P \vdash \text{Path}(\text{last } Cs) \text{ to } C \text{ via } Cs'';$
 $P \vdash C \text{ has least } M = (Ts, T, pns, \text{body}) \text{ via } Cs'; Ds = (Cs @_p Cs'') @_p Cs';$
 $\text{size } vs = \text{size } pns; \text{size } Ts = \text{size } pns \rrbracket$
 $\implies P, E \vdash \langle (\text{ref } (a, Cs)) \cdot (C ::) M(\text{map Val } vs), s \rangle \rightarrow$
 $\langle \text{blocks}(\text{this} \# pns, \text{Class}(\text{last } Ds) \# Ts, \text{Ref}(a, Ds) \# vs, \text{body}), s \rangle$

| *RedCallNull*:

$P, E \vdash \langle \text{Call null Copt } M(\text{map Val } vs), s \rangle \rightarrow \langle \text{THROW NullPointer}, s \rangle$

| *BlockRedNone*:

$\llbracket P, E(V \mapsto T) \vdash \langle e, (h, l(V := \text{None})) \rangle \rightarrow \langle e', (h', l') \rangle; l' V = \text{None}; \neg \text{assigned}$
 $V e \rrbracket$
 $\implies P, E \vdash \langle \{V:T; e\}, (h, l) \rangle \rightarrow \langle \{V:T; e'\}, (h', l'(V := l V)) \rangle$

| *BlockRedSome*:

$\llbracket P, E(V \mapsto T) \vdash \langle e, (h, l(V := \text{None})) \rangle \rightarrow \langle e', (h', l') \rangle; l' V = \text{Some } v;$
 $\neg \text{assigned } V e \rrbracket$
 $\implies P, E \vdash \langle \{V:T; e\}, (h, l) \rangle \rightarrow \langle \{V:T := \text{Val } v; e'\}, (h', l'(V := l V)) \rangle$

| *InitBlockRed*:

$\llbracket P, E(V \mapsto T) \vdash \langle e, (h, l(V \mapsto v')) \rangle \rightarrow \langle e', (h', l') \rangle; l' V = \text{Some } v'';$
 $P \vdash T \text{ casts } v \text{ to } v' \rrbracket$
 $\implies P, E \vdash \langle \{V:T := \text{Val } v; e\}, (h, l) \rangle \rightarrow \langle \{V:T := \text{Val } v''; e'\}, (h', l'(V := l V)) \rangle$

| *RedBlock*:

$P, E \vdash \langle \{V:T; \text{Val } u\}, s \rangle \rightarrow \langle \text{Val } u, s \rangle$

| *RedInitBlock*:

$P \vdash T \text{ casts } v \text{ to } v' \implies P, E \vdash \langle \{V:T := \text{Val } v; \text{Val } u\}, s \rangle \rightarrow \langle \text{Val } u, s \rangle$

| *SeqRed*:

$P, E \vdash \langle e, s \rangle \rightarrow \langle e', s' \rangle \implies$
 $P, E \vdash \langle e;; e_2, s \rangle \rightarrow \langle e';; e_2, s' \rangle$

| *RedSeq*:

$P, E \vdash \langle (\text{Val } v);; e_2, s \rangle \rightarrow \langle e_2, s \rangle$

| *CondRed*:

$$\begin{aligned}
& P, E \vdash \langle e, s \rangle \rightarrow \langle e', s' \rangle \implies \\
& P, E \vdash \langle \text{if } (e) \ e_1 \ \text{else } e_2, s \rangle \rightarrow \langle \text{if } (e') \ e_1 \ \text{else } e_2, s' \rangle
\end{aligned}$$

$$\begin{aligned}
& | \text{RedCondT}: \\
& P, E \vdash \langle \text{if } (\text{true}) \ e_1 \ \text{else } e_2, s \rangle \rightarrow \langle e_1, s \rangle
\end{aligned}$$

$$\begin{aligned}
& | \text{RedCondF}: \\
& P, E \vdash \langle \text{if } (\text{false}) \ e_1 \ \text{else } e_2, s \rangle \rightarrow \langle e_2, s \rangle
\end{aligned}$$

$$\begin{aligned}
& | \text{RedWhile}: \\
& P, E \vdash \langle \text{while}(b) \ c, s \rangle \rightarrow \langle \text{if}(b) \ (c;;\text{while}(b) \ c) \ \text{else } \text{unit}, s \rangle
\end{aligned}$$

$$\begin{aligned}
& | \text{ThrowRed}: \\
& P, E \vdash \langle e, s \rangle \rightarrow \langle e', s' \rangle \implies \\
& P, E \vdash \langle \text{throw } e, s \rangle \rightarrow \langle \text{throw } e', s' \rangle
\end{aligned}$$

$$\begin{aligned}
& | \text{RedThrowNull}: \\
& P, E \vdash \langle \text{throw } \text{null}, s \rangle \rightarrow \langle \text{THROW } \text{NullPointer}, s \rangle
\end{aligned}$$

$$\begin{aligned}
& | \text{ListRed1}: \\
& P, E \vdash \langle e, s \rangle \rightarrow \langle e', s' \rangle \implies \\
& P, E \vdash \langle e \# es, s \rangle [\rightarrow] \langle e' \# es, s' \rangle
\end{aligned}$$

$$\begin{aligned}
& | \text{ListRed2}: \\
& P, E \vdash \langle es, s \rangle [\rightarrow] \langle es', s' \rangle \implies \\
& P, E \vdash \langle \text{Val } v \ \# \ es, s \rangle [\rightarrow] \langle \text{Val } v \ \# \ es', s' \rangle
\end{aligned}$$

— Exception propagation

$$\begin{aligned}
& | \text{DynCastThrow}: P, E \vdash \langle \text{Cast } C \ (\text{Throw } r), s \rangle \rightarrow \langle \text{Throw } r, s \rangle \\
& | \text{StaticCastThrow}: P, E \vdash \langle (\downarrow C)(\text{Throw } r), s \rangle \rightarrow \langle \text{Throw } r, s \rangle \\
& | \text{BinOpThrow1}: P, E \vdash \langle (\text{Throw } r) \ll \text{bop} \gg e_2, s \rangle \rightarrow \langle \text{Throw } r, s \rangle \\
& | \text{BinOpThrow2}: P, E \vdash \langle (\text{Val } v_1) \ll \text{bop} \gg (\text{Throw } r), s \rangle \rightarrow \langle \text{Throw } r, s \rangle \\
& | \text{LAssThrow}: P, E \vdash \langle V := (\text{Throw } r), s \rangle \rightarrow \langle \text{Throw } r, s \rangle \\
& | \text{FAccThrow}: P, E \vdash \langle (\text{Throw } r) \cdot F \{Cs\}, s \rangle \rightarrow \langle \text{Throw } r, s \rangle \\
& | \text{FAssThrow1}: P, E \vdash \langle (\text{Throw } r) \cdot F \{Cs\} := e_2, s \rangle \rightarrow \langle \text{Throw } r, s \rangle \\
& | \text{FAssThrow2}: P, E \vdash \langle \text{Val } v \cdot F \{Cs\} := (\text{Throw } r), s \rangle \rightarrow \langle \text{Throw } r, s \rangle \\
& | \text{CallThrowObj}: P, E \vdash \langle \text{Call } (\text{Throw } r) \ \text{Copt } M \ es, s \rangle \rightarrow \langle \text{Throw } r, s \rangle \\
& | \text{CallThrowParams}: \ll es = \text{map } \text{Val } vs \ @ \ \text{Throw } r \ \# \ es' \rrbracket \\
& \implies P, E \vdash \langle \text{Call } (\text{Val } v) \ \text{Copt } M \ es, s \rangle \rightarrow \langle \text{Throw } r, s \rangle \\
& | \text{BlockThrow}: P, E \vdash \langle \{V:T; \text{Throw } r\}, s \rangle \rightarrow \langle \text{Throw } r, s \rangle \\
& | \text{InitBlockThrow}: P \vdash T \ \text{casts } v \ \text{to } v' \\
& \implies P, E \vdash \langle \{V:T := \text{Val } v; \text{Throw } r\}, s \rangle \rightarrow \langle \text{Throw } r, s \rangle \\
& | \text{SeqThrow}: P, E \vdash \langle (\text{Throw } r); e_2, s \rangle \rightarrow \langle \text{Throw } r, s \rangle \\
& | \text{CondThrow}: P, E \vdash \langle \text{if } (\text{Throw } r) \ e_1 \ \text{else } e_2, s \rangle \rightarrow \langle \text{Throw } r, s \rangle \\
& | \text{ThrowThrow}: P, E \vdash \langle \text{throw}(\text{Throw } r), s \rangle \rightarrow \langle \text{Throw } r, s \rangle
\end{aligned}$$

lemmas *red-reds-induct* = *red-reds.induct* [*split-format* (*complete*)]

and *red-reds-inducts* = *red-reds.inducts* [*split-format* (*complete*)]

inductive-cases [*elim!*]:

$P, E \vdash \langle V := e, s \rangle \rightarrow \langle e', s' \rangle$
 $P, E \vdash \langle e1 ;; e2, s \rangle \rightarrow \langle e', s' \rangle$

declare *Cons-eq-map-conv* [*iff*]

lemma $P, E \vdash \langle e, s \rangle \rightarrow \langle e', s' \rangle \implies \text{True}$

and *reds-length*: $P, E \vdash \langle es, s \rangle [\rightarrow] \langle es', s' \rangle \implies \text{length } es = \text{length } es'$
<proof>

13.3 The reflexive transitive closure

consts

Red :: $\text{prog} \Rightarrow \text{env} \Rightarrow ((\text{expr} \times \text{state}) \times (\text{expr} \times \text{state})) \text{ set}$
Reds :: $\text{prog} \Rightarrow \text{env} \Rightarrow ((\text{expr list} \times \text{state}) \times (\text{expr list} \times \text{state})) \text{ set}$

defs

Red-def: $\text{Red } P E \equiv \{((e, s), e', s'). P, E \vdash \langle e, s \rangle \rightarrow \langle e', s' \rangle\}$
Reds-def: $\text{Reds } P E \equiv \{((es, s), es', s'). P, E \vdash \langle es, s \rangle [\rightarrow] \langle es', s' \rangle\}$

lemma[*simp*]: $((e, s), e', s') \in \text{Red } P E = P, E \vdash \langle e, s \rangle \rightarrow \langle e', s' \rangle$
<proof>

lemma[*simp*]: $((es, s), es', s') \in \text{Reds } P E = P, E \vdash \langle es, s \rangle [\rightarrow] \langle es', s' \rangle$
<proof>

abbreviation

Step :: $\text{prog} \Rightarrow \text{env} \Rightarrow \text{expr} \Rightarrow \text{state} \Rightarrow \text{expr} \Rightarrow \text{state} \Rightarrow \text{bool}$
 $(-, - \vdash ((1 \langle -, / - \rangle) \rightarrow^* / (1 \langle -, / - \rangle))) [51, 0, 0, 0, 0] 81$ **where**
 $P, E \vdash \langle e, s \rangle \rightarrow^* \langle e', s' \rangle \equiv ((e, s), e', s') \in (\text{Red } P E)^*$

abbreviation

Steps :: $\text{prog} \Rightarrow \text{env} \Rightarrow \text{expr list} \Rightarrow \text{state} \Rightarrow \text{expr list} \Rightarrow \text{state} \Rightarrow \text{bool}$
 $(-, - \vdash ((1 \langle -, / - \rangle) [\rightarrow]^* / (1 \langle -, / - \rangle))) [51, 0, 0, 0, 0] 81$ **where**
 $P, E \vdash \langle es, s \rangle [\rightarrow]^* \langle es', s' \rangle \equiv ((es, s), es', s') \in (\text{Reds } P E)^*$

lemma *converse-rtrancl-induct-red*[*consumes 1*]:

assumes $P, E \vdash \langle e, (h, l) \rangle \rightarrow^* \langle e', (h', l') \rangle$

and $\bigwedge e h l. R e h l e h l$

and $\bigwedge e_0 h_0 l_0 e_1 h_1 l_1 e' h' l'.$

$\llbracket P, E \vdash \langle e_0, (h_0, l_0) \rangle \rightarrow \langle e_1, (h_1, l_1) \rangle; R e_1 h_1 l_1 e' h' l' \rrbracket \implies R e_0 h_0 l_0 e'$
 $h' l'$

shows $R e h l e' h' l'$

$\langle proof \rangle$

lemma *steps-length*: $P, E \vdash \langle es, s \rangle [\rightarrow]^* \langle es', s' \rangle \implies \text{length } es = \text{length } es'$
 $\langle proof \rangle$

13.4 Some easy lemmas

lemma *[iff]*: $\neg P, E \vdash \langle [], s \rangle [\rightarrow] \langle es', s' \rangle$
 $\langle proof \rangle$

lemma *[iff]*: $\neg P, E \vdash \langle Val\ v, s \rangle \rightarrow \langle e', s' \rangle$
 $\langle proof \rangle$

lemma *[iff]*: $\neg P, E \vdash \langle Throw\ r, s \rangle \rightarrow \langle e', s' \rangle$
 $\langle proof \rangle$

lemma *red-lcl-incr*: $P, E \vdash \langle e, (h_0, l_0) \rangle \rightarrow \langle e', (h_1, l_1) \rangle \implies \text{dom } l_0 \subseteq \text{dom } l_1$
and $P, E \vdash \langle es, (h_0, l_0) \rangle [\rightarrow] \langle es', (h_1, l_1) \rangle \implies \text{dom } l_0 \subseteq \text{dom } l_1$
 $\langle proof \rangle$

lemma *red-lcl-add*: $P, E \vdash \langle e, (h, l) \rangle \rightarrow \langle e', (h', l') \rangle \implies (\bigwedge l_0. P, E \vdash \langle e, (h, l_0++l) \rangle \rightarrow \langle e', (h', l_0++l') \rangle)$
and $P, E \vdash \langle es, (h, l) \rangle [\rightarrow] \langle es', (h', l') \rangle \implies (\bigwedge l_0. P, E \vdash \langle es, (h, l_0++l) \rangle [\rightarrow] \langle es', (h', l_0++l') \rangle)$
 $\langle proof \rangle$

lemma *Red-lcl-add*:
assumes $P, E \vdash \langle e, (h, l) \rangle \rightarrow^* \langle e', (h', l') \rangle$ **shows** $P, E \vdash \langle e, (h, l_0++l) \rangle \rightarrow^* \langle e', (h', l_0++l') \rangle$

$\langle proof \rangle$

lemma
red-preserves-obj: $\llbracket P, E \vdash \langle e, (h, l) \rangle \rightarrow \langle e', (h', l') \rangle; h\ a = \text{Some}(D, S) \rrbracket$
 $\implies \exists S'. h'\ a = \text{Some}(D, S')$
and *reds-preserves-obj*: $\llbracket P, E \vdash \langle es, (h, l) \rangle [\rightarrow] \langle es', (h', l') \rangle; h\ a = \text{Some}(D, S) \rrbracket$
 $\implies \exists S'. h'\ a = \text{Some}(D, S')$
 $\langle proof \rangle$

end

14 System Classes

theory *SystemClasses* **imports** *Exceptions* **begin**

This theory provides definitions for the system exceptions.

constdefs

NullPointerC :: *cdecl*

NullPointerC \equiv (*NullPointer*, ([],[],[]))

ClassCastC :: *cdecl*

ClassCastC \equiv (*ClassCast*, ([],[],[]))

OutOfMemoryC :: *cdecl*

OutOfMemoryC \equiv (*OutOfMemory*, ([],[],[]))

SystemClasses :: *cdecl list*

SystemClasses \equiv [*NullPointerC*, *ClassCastC*, *OutOfMemoryC*]

end

15 The subtype relation

theory *TypeRel* **imports** *SubObj* **begin**

inductive

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

for *P* :: *prog*

where

widen-refl[*iff*]: $P \vdash T \leq T$

| *widen-subcls*: $P \vdash \text{Path } C \text{ to } D \text{ unique} \implies P \vdash \text{Class } C \leq \text{Class } D$

| *widen-null*[*iff*]: $P \vdash NT \leq \text{Class } C$

abbreviation (*xsymbols*)

widens :: *prog* \Rightarrow *ty list* \Rightarrow *ty list* \Rightarrow *bool*

(- \vdash - [\leq] - [71,71,71] 70) **where**

widens *P* *Ts* *Ts'* \equiv *list-all2* (*widen* *P*) *Ts* *Ts'*

lemma [*iff*]: $(P \vdash T \leq \text{Void}) = (T = \text{Void})$

\langle *proof* \rangle

lemma [*iff*]: $(P \vdash T \leq \text{Boolean}) = (T = \text{Boolean})$

\langle *proof* \rangle

lemma [*iff*]: $(P \vdash T \leq \text{Integer}) = (T = \text{Integer})$

\langle *proof* \rangle

lemma [*iff*]: $(P \vdash \text{Void} \leq T) = (T = \text{Void})$

\langle *proof* \rangle

lemma [*iff*]: $(P \vdash \text{Boolean} \leq T) = (T = \text{Boolean})$

\langle *proof* \rangle

lemma [*iff*]: $(P \vdash \text{Integer} \leq T) = (T = \text{Integer})$

\langle *proof* \rangle

lemma [*iff*]: $(P \vdash T \leq NT) = (T = NT)$

\langle *proof* \rangle

lemmas *widens-refl* [*iff*] = *list-all2-refl* [of *widen* *P*, OF *widen-refl*, *standard*]

lemmas *widens-Cons* [*iff*] = *list-all2-Cons1* [of *widen* *P*, *standard*]

end

16 Well-typedness of CoreC++ expressions

theory *WellType* **imports** *Syntax TypeRel* **begin**

16.1 The rules

inductive

$WT :: [prog, env, expr, ty] \Rightarrow bool$
 $(-, - \vdash - :: - [51, 51, 51] 50)$

and $WTs :: [prog, env, expr\ list, ty\ list] \Rightarrow bool$
 $(-, - \vdash - [::] - [51, 51, 51] 50)$

for $P :: prog$

where

WTNew:
 $is_class\ P\ C \Longrightarrow$
 $P, E \vdash new\ C :: Class\ C$

| *WTDynCast*:
 $[P, E \vdash e :: Class\ D; is_class\ P\ C;$
 $P \vdash Path\ D\ to\ C\ unique \vee (\forall Cs. \neg P \vdash Path\ D\ to\ C\ via\ Cs)]$
 $\Longrightarrow P, E \vdash Cast\ C\ e :: Class\ C$

| *WTStaticCast*:
 $[P, E \vdash e :: Class\ D; is_class\ P\ C;$
 $P \vdash Path\ D\ to\ C\ unique \vee$
 $(P \vdash C \preceq^* D \wedge (\forall Cs. P \vdash Path\ C\ to\ D\ via\ Cs \longrightarrow Subobjs_R\ P\ C\ Cs))]$
 $\Longrightarrow P, E \vdash \langle C \rangle e :: Class\ C$

| *WTVal*:
 $typeof\ v = Some\ T \Longrightarrow$
 $P, E \vdash Val\ v :: T$

| *WTVar*:
 $E\ V = Some\ T \Longrightarrow$
 $P, E \vdash Var\ V :: T$

| *WTBinOp*:
 $[P, E \vdash e_1 :: T_1; P, E \vdash e_2 :: T_2;$
 $case\ bop\ of\ Eq \Rightarrow T_1 = T_2 \wedge T = Boolean$
 $| Add \Rightarrow T_1 = Integer \wedge T_2 = Integer \wedge T = Integer]$
 $\Longrightarrow P, E \vdash e_1 \langle bop \rangle e_2 :: T$

| *WTLAss*:
 $[E\ V = Some\ T; P, E \vdash e :: T'; P \vdash T' \leq T]$
 $\Longrightarrow P, E \vdash V := e :: T$

| *WTFAcc*:
 $[P, E \vdash e :: Class\ C; P \vdash C\ has\ least\ F:T\ via\ Cs]$
 $\Longrightarrow P, E \vdash e \cdot F\{Cs\} :: T$

| *WTFAss*:
 $\llbracket P, E \vdash e_1 :: \text{Class } C; P \vdash C \text{ has least } F:T \text{ via } Cs;$
 $\quad P, E \vdash e_2 :: T'; P \vdash T' \leq T \rrbracket$
 $\implies P, E \vdash e_1 \cdot F\{Cs\} := e_2 :: T$

| *WTStaticCall*:
 $\llbracket P, E \vdash e :: \text{Class } C'; P \vdash \text{Path } C' \text{ to } C \text{ unique};$
 $\quad P \vdash C \text{ has least } M = (Ts, T, m) \text{ via } Cs; P, E \vdash es \llbracket :: Ts'; P \vdash Ts' \llbracket \leq Ts \rrbracket$
 $\implies P, E \vdash e \cdot (C::)M(es) :: T$

| *WTCall*:
 $\llbracket P, E \vdash e :: \text{Class } C; P \vdash C \text{ has least } M = (Ts, T, m) \text{ via } Cs;$
 $\quad P, E \vdash es \llbracket :: Ts'; P \vdash Ts' \llbracket \leq Ts \rrbracket$
 $\implies P, E \vdash e \cdot M(es) :: T$

| *WTBlock*:
 $\llbracket \text{is-type } P \ T; P, E(V \mapsto T) \vdash e :: T' \rrbracket$
 $\implies P, E \vdash \{V:T; e\} :: T'$

| *WTSeq*:
 $\llbracket P, E \vdash e_1 :: T_1; P, E \vdash e_2 :: T_2 \rrbracket$
 $\implies P, E \vdash e_1 ; e_2 :: T_2$

| *WTCond*:
 $\llbracket P, E \vdash e :: \text{Boolean}; P, E \vdash e_1 :: T; P, E \vdash e_2 :: T \rrbracket$
 $\implies P, E \vdash \text{if } (e) \ e_1 \ \text{else } e_2 :: T$

| *WTWhile*:
 $\llbracket P, E \vdash e :: \text{Boolean}; P, E \vdash c :: T \rrbracket$
 $\implies P, E \vdash \text{while } (e) \ c :: \text{Void}$

| *WTThrow*:
 $P, E \vdash e :: \text{Class } C \implies$
 $P, E \vdash \text{throw } e :: \text{Void}$

— well-typed expression lists

| *WTNil*:
 $P, E \vdash [] \llbracket :: [] \rrbracket$

| *WTCons*:
 $\llbracket P, E \vdash e :: T; P, E \vdash es \llbracket :: Ts \rrbracket$
 $\implies P, E \vdash e \# es \llbracket :: T \# Ts \rrbracket$

declare *WT-WTs.intros*[*intro!*] *WTNil*[*iff*]

lemmas $WT\text{-}WTs\text{-}induct = WT\text{-}WTs.induct$ [*split-format (complete)*]
and $WT\text{-}WTs\text{-}inducts = WT\text{-}WTs.inducts$ [*split-format (complete)*]

16.2 Easy consequences

lemma [*iff*]: $(P, E \vdash [] [::] Ts) = (Ts = [])$

<proof>

lemma [*iff*]: $(P, E \vdash e \# es [::] T \# Ts) = (P, E \vdash e :: T \wedge P, E \vdash es [::] Ts)$

<proof>

lemma [*iff*]: $(P, E \vdash (e \# es) [::] Ts) =$
 $(\exists U Us. Ts = U \# Us \wedge P, E \vdash e :: U \wedge P, E \vdash es [::] Us)$

<proof>

lemma [*iff*]: $\bigwedge Ts. (P, E \vdash es_1 @ es_2 [::] Ts) =$
 $(\exists Ts_1 Ts_2. Ts = Ts_1 @ Ts_2 \wedge P, E \vdash es_1 [::] Ts_1 \wedge P, E \vdash es_2 [::] Ts_2)$

<proof>

lemma [*iff*]: $P, E \vdash Val v :: T = (typeof v = Some T)$

<proof>

lemma [*iff*]: $P, E \vdash Var V :: T = (E V = Some T)$

<proof>

lemma [*iff*]: $P, E \vdash e_1;;e_2 :: T_2 = (\exists T_1. P, E \vdash e_1::T_1 \wedge P, E \vdash e_2::T_2)$

<proof>

lemma [*iff*]: $(P, E \vdash \{V:T; e\} :: T') = (is\text{-type } P T \wedge P, E(V \mapsto T) \vdash e :: T')$

<proof>

inductive-cases $WT\text{-}elim\text{-cases}[elim!]$:

$$\begin{aligned}
P, E &\vdash \text{new } C :: T \\
P, E &\vdash \text{Cast } C \ e :: T \\
P, E &\vdash \langle C \rangle e :: T \\
P, E &\vdash e_1 \ll \text{bop} \gg e_2 :: T \\
P, E &\vdash V := e :: T \\
P, E &\vdash e \cdot F \{Cs\} :: T \\
P, E &\vdash e \cdot F \{Cs\} := v :: T \\
P, E &\vdash e \cdot M(ps) :: T \\
P, E &\vdash e \cdot (C ::) M(ps) :: T \\
P, E &\vdash \text{if } (e) \ e_1 \ \text{else } e_2 :: T \\
P, E &\vdash \text{while } (e) \ c :: T \\
P, E &\vdash \text{throw } e :: T
\end{aligned}$$

lemma *wt-env-mono*:

$$\begin{aligned}
P, E \vdash e :: T &\implies (\bigwedge E'. E \subseteq_m E' \implies P, E' \vdash e :: T) \ \text{and} \\
P, E \vdash es \ [::] \ Ts &\implies (\bigwedge E'. E \subseteq_m E' \implies P, E' \vdash es \ [::] \ Ts)
\end{aligned}$$

\langle proof \rangle

lemma *WT-fv*: $P, E \vdash e :: T \implies \text{fv } e \subseteq \text{dom } E$
and $P, E \vdash es \ [::] \ Ts \implies \text{fvs } es \subseteq \text{dom } E$

\langle proof \rangle

end

17 Generic Well-formedness of programs

theory *WellForm* **imports** *SystemClasses TypeRel WellType* **begin**

This theory defines global well-formedness conditions for programs but does not look inside method bodies. Well-typing of expressions is defined elsewhere (in theory *WellType*).

CoreC++ allows covariant return types

types *wf-mdecl-test* = *prog* \Rightarrow *cname* \Rightarrow *mdecl* \Rightarrow *bool*

constdefs

wf-fdecl :: *prog* \Rightarrow *fdecl* \Rightarrow *bool*

wf-fdecl *P* \equiv $\lambda(F, T). \text{is-type } P \ T$

wf-mdecl :: *wf-mdecl-test* \Rightarrow *wf-mdecl-test*

wf-mdecl *wf-md* *P* *C* \equiv $\lambda(M, Ts, T, mb).$

$(\forall T \in \text{set } Ts. \text{is-type } P \ T) \wedge \text{is-type } P \ T \wedge T \neq NT \wedge \text{wf-md } P \ C \ (M, Ts, T, mb)$

wf-cdecl :: *wf-mdecl-test* \Rightarrow *prog* \Rightarrow *cdecl* \Rightarrow *bool*

wf-cdecl *wf-md* *P* \equiv $\lambda(C, (Bs, fs, ms)).$

$(\forall M \ \text{mthd } Cs. P \vdash C \ \text{has } M = \text{mthd via } Cs \longrightarrow$

$(\exists \text{mthd}' \ Cs'. P \vdash (C, Cs) \ \text{has overrider } M = \text{mthd}' \ \text{via } Cs')) \wedge$

$(\forall f \in \text{set } fs. \text{wf-fdecl } P \ f) \wedge \text{distinct-fst } fs \wedge$

$(\forall m \in \text{set } ms. \text{wf-mdecl } \text{wf-md } P \ C \ m) \wedge \text{distinct-fst } ms \wedge$

$(\forall D \in \text{baseClasses } Bs.$

$\text{is-class } P \ D \wedge \neg P \vdash D \leq^* C \wedge$

$(\forall (M, Ts, T, m) \in \text{set } ms.$

$\forall Ts' \ T' \ m' \ Cs. P \vdash D \ \text{has } M = (Ts', T', m') \ \text{via } Cs \longrightarrow$

$Ts' = Ts \wedge P \vdash T \leq T'))$

wf-syscls :: *prog* \Rightarrow *bool*

wf-syscls *P* $\equiv \text{sys-xcpts} \subseteq \text{set}(\text{map } \text{fst } P)$

wf-prog :: *wf-mdecl-test* \Rightarrow *prog* \Rightarrow *bool*

wf-prog *wf-md* *P* $\equiv \text{wf-syscls } P \wedge \text{distinct-fst } P \wedge$

$(\forall c \in \text{set } P. \text{wf-cdecl } \text{wf-md } P \ c)$

17.1 Well-formedness lemmas

lemma *class-wf*:

$\llbracket \text{class } P \ C = \text{Some } c; \text{wf-prog } \text{wf-md } P \rrbracket \Longrightarrow \text{wf-cdecl } \text{wf-md } P \ (C, c)$

$\langle \text{proof} \rangle$

lemma *is-class-xcpt*:

$\llbracket C \in \text{sys-xcpts}; \text{wf-prog } \text{wf-md } P \rrbracket \Longrightarrow \text{is-class } P \ C$

<proof>

lemma *is-type-pTs*:

assumes *wf-prog wf-md P* **and** $(C, S, fs, ms) \in \text{set } P$ **and** $(M, Ts, T, m) \in \text{set } ms$
shows $\text{set } Ts \subseteq \text{types } P$

<proof>

17.2 Well-formedness subclass lemmas

lemma *subcls1-wfD*:

$\llbracket P \vdash C \prec^1 D; \text{wf-prog wf-md } P \rrbracket \implies D \neq C \wedge (D, C) \notin (\text{subcls1 } P)^+$

<proof>

lemma *wf-cdecl-supD*:

$\llbracket \text{wf-cdecl wf-md } P (C, Bs, r); D \in \text{baseClasses } Bs \rrbracket \implies \text{is-class } P D$
<proof>

lemma *subcls1-asym*:

$\llbracket \text{wf-prog wf-md } P; (C, D) \in (\text{subcls1 } P)^+ \rrbracket \implies (D, C) \notin (\text{subcls1 } P)^+$

<proof>

lemma *subcls1-irrefl*:

$\llbracket \text{wf-prog wf-md } P; (C, D) \in (\text{subcls1 } P)^+ \rrbracket \implies C \neq D$

<proof>

lemma *subcls1-asym2*:

$\llbracket (C, D) \in (\text{subcls1 } P)^*; \text{wf-prog wf-md } P; (D, C) \in (\text{subcls1 } P)^* \rrbracket \implies C = D$

<proof>

lemma *acyclic-subcls1*:

$\text{wf-prog wf-md } P \implies \text{acyclic } (\text{subcls1 } P)$

<proof>

lemma *wf-subcls1*:

$wf\text{-prog } wf\text{-md } P \implies wf ((subcls1 P)^{-1})$

<proof>

lemma *subcls-induct*:

$\llbracket wf\text{-prog } wf\text{-md } P; \bigwedge C. \forall D. (C,D) \in (subcls1 P)^+ \longrightarrow Q D \implies Q C \rrbracket \implies Q C$

(is ?A \implies PROP ?P \implies -)

<proof>

17.3 Well-formedness leq_path lemmas

lemma *last-leq-path*:

assumes $leq:P, C \vdash Cs \sqsubset^1 Ds$ **and** $wf:wf\text{-prog } wf\text{-md } P$

shows $P \vdash last Cs \prec^1 last Ds$

<proof>

lemma *last-leq-paths*:

assumes $leq:(Cs,Ds) \in (leq\text{-path1 } P C)^+$ **and** $wf:wf\text{-prog } wf\text{-md } P$

shows $(last Cs, last Ds) \in (subcls1 P)^+$

<proof>

lemma *leq-path1-wfD*:

$\llbracket P, C \vdash Cs \sqsubset^1 Cs'; wf\text{-prog } wf\text{-md } P \rrbracket \implies Cs \neq Cs' \wedge (Cs', Cs) \notin (leq\text{-path1 } P C)^+$

<proof>

lemma *leq-path-asym*:

$\llbracket (Cs, Cs') \in (leq\text{-path1 } P C)^+; wf\text{-prog } wf\text{-md } P \rrbracket \implies (Cs', Cs) \notin (leq\text{-path1 } P C)^+$

<proof>

lemma *leq-path-asym2*: $\llbracket P, C \vdash Cs \sqsubseteq Cs'; P, C \vdash Cs' \sqsubseteq Cs; wf\text{-prog } wf\text{-md } P \rrbracket \implies Cs = Cs'$

<proof>

lemma *leq-path-Subobjs*:

$\llbracket P, C \vdash [C] \sqsubseteq Cs; is\text{-class } P \ C; wf\text{-prog } wf\text{-md } P \rrbracket \implies Subobjs \ P \ C \ Cs$
<proof>

17.4 Lemmas concerning Subobjs

lemma *Subobj-last-isClass*: $\llbracket wf\text{-prog } wf\text{-md } P; Subobjs \ P \ C \ Cs \rrbracket \implies is\text{-class } P \ (last \ Cs)$

<proof>

lemma *converse-SubobjsR-Rep*:

$\llbracket Subobjs_R \ P \ C \ Cs; P \vdash last \ Cs \prec_R \ C'; wf\text{-prog } wf\text{-md } P \rrbracket$
 $\implies Subobjs_R \ P \ C \ (Cs@[C'])$

<proof>

lemma *converse-Subobjs-Rep*:

$\llbracket Subobjs \ P \ C \ Cs; P \vdash last \ Cs \prec_R \ C'; wf\text{-prog } wf\text{-md } P \rrbracket$
 $\implies Subobjs \ P \ C \ (Cs@[C'])$
<proof>

lemma *isSubobj-Subobjs-rev*:

assumes *subo:is-subobj*($P, (C, C'\#rev \ Cs')$) **and** *wf:wf-prog wf-md P*
shows *Subobjs* $P \ C \ (C'\#rev \ Cs')$
<proof>

lemma *isSubobj-Subobjs*:

assumes *subo:is-subobj*($P, (C, Cs)$) **and** *wf:wf-prog wf-md P*
shows *Subobjs* $P \ C \ Cs$

<proof>

lemma *isSubobj-eq-Subobjs*:
 $wf\text{-prog } wf\text{-md } P \implies is\text{-subobj}(P, (C, Cs)) = (Subobjs P C Cs)$
 ⟨proof⟩

lemma *subo-trans-subcls*:
 assumes $subo:Subobjs P C (Cs@ C'\#rev Cs')$
 shows $\forall C'' \in set Cs'. (C', C'') \in (subcls1 P)^+$
 ⟨proof⟩

lemma *unique1*:
 assumes $subo:Subobjs P C (Cs@ C'\#Cs')$ and $wf:wf\text{-prog } wf\text{-md } P$
 shows $C' \notin set Cs'$
 ⟨proof⟩

lemma *subo-subcls-trans*:
 assumes $subo:Subobjs P C (Cs@ C'\#Cs')$
 shows $\forall C'' \in set Cs. (C'', C') \in (subcls1 P)^+$
 ⟨proof⟩

lemma *unique2*:
 assumes $subo:Subobjs P C (Cs@ C'\#Cs')$ and $wf:wf\text{-prog } wf\text{-md } P$
 shows $C' \notin set Cs$
 ⟨proof⟩

lemma *mdc-hd-path*:
 assumes $subo:Subobjs P C Cs$ and $set:C \in set Cs$ and $wf:wf\text{-prog } wf\text{-md } P$
 shows $C = hd Cs$
 ⟨proof⟩

lemma *mdc-eq-last*:

assumes *subo*:*Subobjs* *P C Cs* **and** *last*:*last Cs = C* **and** *wf*:*wf-prog wf-md P*
shows *Cs = [C]*

<proof>

lemma **assumes** *leq*:*P ⊢ C ≼* D* **and** *wf*:*wf-prog wf-md P*

shows *subcls-leq-path*: $\exists Cs. P, C ⊢ [C] \sqsubseteq Cs@[D]$

<proof>

lemma **assumes** *subo*:*Subobjs P C (rev Cs)* **and** *wf*:*wf-prog wf-md P*

shows *subobjs-rel-rev*: $P, C ⊢ [C] \sqsubseteq (rev Cs)$

<proof>

lemma *subobjs-rel*:

assumes *subo*:*Subobjs P C Cs* **and** *wf*:*wf-prog wf-md P*

shows $P, C ⊢ [C] \sqsubseteq Cs$

<proof>

lemma **assumes** *wf*:*wf-prog wf-md P*

shows *leq-path-last*: $\llbracket P, C ⊢ Cs \sqsubseteq Cs'; last Cs = last Cs' \rrbracket \implies Cs = Cs'$

<proof>

17.5 Well-formedness and appendPath

lemma *appendPath1*:

$\llbracket Subobjs P C Cs; Subobjs P (last Cs) Ds; last Cs \neq hd Ds \rrbracket$
 $\implies Subobjs P C Ds$

<proof>

lemma *appendPath2-rev*:

assumes *subo1*:*Subobjs P C Cs* **and** *subo2*:*Subobjs P (last Cs) (last Cs#rev Ds)*

and $wf:wf\text{-prog } wf\text{-md } P$
shows $Subobjs\ P\ C\ (Cs@(tl\ (last\ Cs\#\text{rev}\ Ds)))$
 $\langle proof \rangle$

lemma *appendPath2*:
assumes $subo1:Subobjs\ P\ C\ Cs$ **and** $subo2:Subobjs\ P\ (last\ Cs)\ Ds$
and $eq:last\ Cs = hd\ Ds$ **and** $wf:wf\text{-prog } wf\text{-md } P$
shows $Subobjs\ P\ C\ (Cs@(tl\ Ds))$

$\langle proof \rangle$

lemma *Subobjs-appendPath*:
 $\llbracket Subobjs\ P\ C\ Cs; Subobjs\ P\ (last\ Cs)\ Ds; wf\text{-prog } wf\text{-md } P \rrbracket$
 $\implies Subobjs\ P\ C\ (Cs@_p\ Ds)$
 $\langle proof \rangle$

17.6 Path and program size

lemma **assumes** $subo:Subobjs\ P\ C\ Cs$ **and** $wf:wf\text{-prog } wf\text{-md } P$
shows $path\text{-contains}\text{-classes}:\forall C' \in set\ Cs. is\text{-class } P\ C'$
 $\langle proof \rangle$

lemma *path-subset-classes*: $\llbracket Subobjs\ P\ C\ Cs; wf\text{-prog } wf\text{-md } P \rrbracket$
 $\implies set\ Cs \subseteq \{C. is\text{-class } P\ C\}$
 $\langle proof \rangle$

lemma **assumes** $subo:Subobjs\ P\ C\ (rev\ Cs)$ **and** $wf:wf\text{-prog } wf\text{-md } P$
shows $rev\text{-path}\text{-distinct}\text{-classes}:\text{distinct } Cs$
 $\langle proof \rangle$

lemma **assumes** $subo:Subobjs\ P\ C\ Cs$ **and** $wf:wf\text{-prog } wf\text{-md } P$
shows $path\text{-distinct}\text{-classes}:\text{distinct } Cs$

$\langle proof \rangle$

lemma **assumes** $wf:wf\text{-prog } wf\text{-md } P$
shows $prog\text{-length}:\text{length } P = card\ \{C. is\text{-class } P\ C\}$

$\langle proof \rangle$

lemma *assumes* $subo:Subobjs\ P\ C\ Cs$ **and** $wf:wf-prog\ wf-md\ P$
shows $path-length:length\ Cs \leq length\ P$

<proof>

lemma *empty-path-empty-set*: $\{Cs. Subobjs\ P\ C\ Cs \wedge length\ Cs \leq 0\} = \{\}$
<proof>

lemma *split-set-path-length*: $\{Cs. Subobjs\ P\ C\ Cs \wedge length\ Cs \leq Suc(n)\} =$
 $\{Cs. Subobjs\ P\ C\ Cs \wedge length\ Cs \leq n\} \cup \{Cs. Subobjs\ P\ C\ Cs \wedge length\ Cs =$
 $Suc(n)\}$
<proof>

lemma *empty-list-set*: $\{xs. set\ xs \subseteq F \wedge xs = []\} = \{[]\}$
<proof>

lemma *suc-n-union-of-union*: $\{xs. set\ xs \subseteq F \wedge length\ xs = Suc\ n\} = (UN\ x:F.$
 $UN\ xs : \{xs. set\ xs \subseteq F \wedge length\ xs = n\}. \{x\#xs\})$
<proof>

lemma *max-length-finite-set*: $finite\ F \implies finite\ \{xs. set\ xs \subseteq F \wedge length\ xs = n\}$
<proof>

lemma *path-length-n-finite-set*:
 $wf-prog\ wf-md\ P \implies finite\ \{Cs. Subobjs\ P\ C\ Cs \wedge length\ Cs = n\}$
<proof>

lemma *path-finite-leq*:
 $wf-prog\ wf-md\ P \implies finite\ \{Cs. Subobjs\ P\ C\ Cs \wedge length\ Cs \leq length\ P\}$
<proof>

lemma *path-finite*: $wf-prog\ wf-md\ P \implies finite\ \{Cs. Subobjs\ P\ C\ Cs\}$
<proof>

17.7 Well-formedness and Path

lemma *path-via-reverse*:
assumes $path-via:P \vdash Path\ C\ to\ D\ via\ Cs$ **and** $wf:wf-prog\ wf-md\ P$
shows $\forall Cs'. P \vdash Path\ D\ to\ C\ via\ Cs' \implies Cs = [C] \wedge Cs' = [C] \wedge C = D$
<proof>

lemma *path-hd-appendPath*:

assumes *path*: $P, C \vdash Cs \sqsubseteq Cs' @_p Cs$ **and** *last*: $last\ Cs' = hd\ Cs$
and *notemptyCs*: $Cs \neq []$ **and** *notemptyCs'*: $Cs' \neq []$ **and** *wf*:*wf-prog wf-md P*
shows $Cs' = [hd\ Cs]$

<proof>

lemma *path-via-C*: $\llbracket P \vdash Path\ C\ to\ C\ via\ Cs; wf-prog\ wf-md\ P \rrbracket \implies Cs = [C]$

<proof>

lemma *assumes wf:wf-prog wf-md P*

and *path-via*: $P \vdash Path\ last\ Cs\ to\ C\ via\ Cs'$
and *path-via'*: $P \vdash Path\ last\ Cs\ to\ C\ via\ Cs''$
and *appendPath*: $Cs = Cs @_p Cs'$

shows *appendPath-path-via*: $Cs = Cs @_p Cs''$

<proof>

lemma *subo-no-path*:

assumes *subo*:*Subobjs P C' (Cs @ C # Cs')* **and** *wf*:*wf-prog wf-md P*
and *notempty*: $Cs' \neq []$
shows $\neg P \vdash Path\ last\ Cs'\ to\ C\ via\ Cs$

<proof>

lemma *leq-implies-path*:

assumes *leq*: $P \vdash C \preceq^* D$ **and** *class*:*is-class P C*
and *wf*:*wf-prog wf-md P*
shows $\exists Cs. P \vdash Path\ C\ to\ D\ via\ Cs$

<proof>

lemma *least-method-implies-path-unique*:

assumes *least*: $P \vdash C\ has\ least\ M = (Ts, T, m)\ via\ Cs$ **and** *wf*:*wf-prog wf-md P*
shows $P \vdash Path\ C\ to\ (last\ Cs)\ unique$

<proof>

lemma *least-field-implies-path-unique*:

assumes *least*: $P \vdash C\ has\ least\ F:T\ via\ Cs$ **and** *wf*:*wf-prog wf-md P*

shows $P \vdash \text{Path } C \text{ to } (\text{hd } Cs) \text{ unique}$

$\langle \text{proof} \rangle$

lemma *least-field-implies-path-via-hd*:

$\llbracket P \vdash C \text{ has least } F:T \text{ via } Cs; \text{ wf-prog wf-md } P \rrbracket$
 $\implies P \vdash \text{Path } C \text{ to } (\text{hd } Cs) \text{ via } [\text{hd } Cs]$

$\langle \text{proof} \rangle$

lemma *path-C-to-C-unique*:

$\llbracket \text{wf-prog wf-md } P; \text{ is-class } P \ C \rrbracket \implies P \vdash \text{Path } C \text{ to } C \text{ unique}$

$\langle \text{proof} \rangle$

lemma *leqR-SubobjsR*: $\llbracket (C,D) \in (\text{subclsR } P)^*; \text{ is-class } P \ C; \text{ wf-prog wf-md } P \rrbracket$
 $\implies \exists Cs. \text{SubobjsR } P \ C \ (Cs@[D])$

$\langle \text{proof} \rangle$

lemma *assumes path-unique*: $P \vdash \text{Path } C \text{ to } D \text{ unique}$ **and** $\text{leq}: P \vdash C \preceq^* C'$
and $\text{leqR}: (C',D) \in (\text{subclsR } P)^*$ **and** $\text{wf}: \text{wf-prog wf-md } P$
shows $P \vdash \text{Path } C \text{ to } C' \text{ unique}$

$\langle \text{proof} \rangle$

17.8 Well-formedness and member lookup

lemma *has-path-has*:

$\llbracket P \vdash \text{Path } D \text{ to } C \text{ via } Ds; P \vdash C \text{ has } M = (Ts, T, m) \text{ via } Cs; \text{ wf-prog wf-md } P \rrbracket$
 $\implies P \vdash D \text{ has } M = (Ts, T, m) \text{ via } Ds@_p Cs$

$\langle \text{proof} \rangle$

lemma *has-least-wf-mdecl*:

$\llbracket \text{wf-prog wf-md } P; P \vdash C \text{ has least } M = m \text{ via } Cs \rrbracket$
 $\implies \text{wf-mdecl wf-md } P \ (\text{last } Cs) \ (M, m)$

$\langle \text{proof} \rangle$

lemma *has-overrider-wf-mdecl*:

$\llbracket \text{wf-prog wf-md } P; P \vdash (C, Cs) \text{ has overrider } M = m \text{ via } Cs' \rrbracket$

$\implies \text{wf-mdecl wf-md } P \text{ (last } Cs') (M,m)$
 $\langle \text{proof} \rangle$

lemma *select-method-wf-mdecl*:
[[*wf-prog wf-md* P ; $P \vdash (C, Cs)$ selects $M = m$ via Cs']]
 $\implies \text{wf-mdecl wf-md } P \text{ (last } Cs') (M,m)$
 $\langle \text{proof} \rangle$

lemma *wf-sees-method-fun*:
[[$P \vdash C$ has least $M = \text{mthd}$ via Cs ; $P \vdash C$ has least $M = \text{mthd}'$ via Cs' ;
wf-prog wf-md P]]
 $\implies \text{mthd} = \text{mthd}' \wedge Cs = Cs'$

$\langle \text{proof} \rangle$

lemma *wf-select-method-fun*:
assumes *wf:wf-prog wf-md* P
shows [[$P \vdash (C, Cs)$ selects $M = \text{mthd}$ via Cs' ; $P \vdash (C, Cs)$ selects $M = \text{mthd}'$
via Cs'']]
 $\implies \text{mthd} = \text{mthd}' \wedge Cs' = Cs''$
 $\langle \text{proof} \rangle$

lemma *least-field-is-type*:
assumes *field*: $P \vdash C$ has least $F:T$ via Cs **and** *wf:wf-prog wf-md* P
shows *is-type* $P T$

$\langle \text{proof} \rangle$

lemma *least-method-is-type*:
assumes *method*: $P \vdash C$ has least $M = (Ts, T, m)$ via Cs **and** *wf:wf-prog wf-md* P
shows *is-type* $P T$

$\langle \text{proof} \rangle$

lemma *least-overrider-is-type*:
assumes *method*: $P \vdash (C, Cs)$ has overrider $M = (Ts, T, m)$ via Cs'
and *wf:wf-prog wf-md* P
shows *is-type* $P T$

$\langle proof \rangle$

lemma *select-method-is-type*:

$\llbracket P \vdash (C, Cs) \text{ selects } M = (Ts, T, m) \text{ via } Cs'; \text{ wf-prog wf-md } P \rrbracket \implies \text{is-type } P T$
 $\langle proof \rangle$

lemma *base-subtype*:

$\llbracket \text{wf-cdecl wf-md } P (C, Bs, fs, ms); C' \in \text{baseClasses } Bs;$
 $P \vdash C' \text{ has } M = (Ts', T', m') \text{ via } Cs@_p[D]; (M, Ts, T, m) \in \text{set } ms \rrbracket$
 $\implies Ts' = Ts \wedge P \vdash T \leq T'$

$\langle proof \rangle$

lemma *subclsPlus-subtype*:

assumes $\text{classD: class } P D = \text{Some}(Bs', fs', ms')$
and $\text{mapMs': map-of } ms' M = \text{Some}(Ts', T', m')$
and $\text{leq: } (C, D) \in (\text{subcls1 } P)^+$ **and** $\text{wf: wf-prog wf-md } P$
shows $\forall Bs \ fs \ ms \ Ts \ T \ m. \text{ class } P C = \text{Some}(Bs, fs, ms) \wedge \text{map-of } ms M =$
 $\text{Some}(Ts, T, m)$
 $\longrightarrow Ts' = Ts \wedge P \vdash T \leq T'$

$\langle proof \rangle$

lemma *leq-method-subtypes*:

assumes $\text{leq: } P \vdash D \preceq^* C$ **and** $\text{least: } P \vdash D \text{ has least } M = (Ts', T', m') \text{ via } Ds$
and $\text{wf: wf-prog wf-md } P$
shows $\forall Ts \ T \ m \ Cs. P \vdash C \text{ has } M = (Ts, T, m) \text{ via } Cs \longrightarrow$
 $Ts = Ts' \wedge P \vdash T' \leq T$

$\langle proof \rangle$

lemma *leq-methods-subtypes*:

assumes $\text{leq: } P \vdash D \preceq^* C$ **and** $\text{least: } (Ds, (Ts', T', m')) \in \text{MinimalMethodDefs } P$
 $D M$
and $\text{wf: wf-prog wf-md } P$
shows $\forall Ts \ T \ m \ Cs \ Cs'. P \vdash \text{Path } D \text{ to } C \text{ via } Cs' \wedge P, D \vdash Ds \sqsubseteq Cs'@_p Cs \wedge$
 $Cs \neq [] \wedge$
 $P \vdash C \text{ has } M = (Ts, T, m) \text{ via } Cs$
 $\longrightarrow Ts = Ts' \wedge P \vdash T' \leq T$

<proof>

lemma *select-least-methods-subtypes*:

assumes *select-method*: $P \vdash (C, Cs@_pDs)$ selects $M = (Ts, T, pns, body)$ via Cs'

and *least-method*: $P \vdash$ last Cs has least $M = (Ts', T', pns', body')$ via Ds

and *path*: $P \vdash$ Path C to (last Cs) via Cs

and *wf*: *wf-prog wf-md P*

shows $Ts' = Ts \wedge P \vdash T \leq T'$

<proof>

lemma *wf-syscls*:

set SystemClasses \subseteq *set P* \implies *wf-syscls P*

<proof>

17.9 Well formedness and widen

lemma *Class-widen*: $\llbracket P \vdash \text{Class } C \leq T; \text{wf-prog wf-md } P; \text{is-class } P \ C \rrbracket$

$\implies \exists D. T = \text{Class } D \wedge P \vdash \text{Path } C \text{ to } D \text{ unique}$

<proof>

lemma *Class-widen-Class [iff]*: $\llbracket \text{wf-prog wf-md } P; \text{is-class } P \ C \rrbracket \implies$

$(P \vdash \text{Class } C \leq \text{Class } D) = (P \vdash \text{Path } C \text{ to } D \text{ unique})$

<proof>

lemma *widen-Class*: $\llbracket \text{wf-prog wf-md } P; \text{is-class } P \ C \rrbracket \implies$

$(P \vdash T \leq \text{Class } C) =$

$(T = NT \vee (\exists D. T = \text{Class } D \wedge P \vdash \text{Path } D \text{ to } C \text{ unique}))$

<proof>

17.10 Well formedness and well typing

lemma *assumes wf*: *wf-prog wf-md P*

shows *WT-determ*: $P, E \vdash e :: T \implies (\bigwedge T'. P, E \vdash e :: T' \implies T = T')$

and *WTs-determ*: $P, E \vdash es \llbracket :: \rrbracket Ts \implies (\bigwedge Ts'. P, E \vdash es \llbracket :: \rrbracket Ts' \implies Ts = Ts')$

<proof>

end

18 Weak well-formedness of CoreC++ programs

theory *WWellForm* **imports** *WellForm Expr* **begin**

constdefs

wf-mdecl :: *prog* \Rightarrow *cname* \Rightarrow *mdecl* \Rightarrow *bool*
wf-mdecl *P C* \equiv $\lambda(M, Ts, T, (pns, body)).$
length *Ts* = *length* *pns* \wedge *distinct* *pns* \wedge *this* \notin *set* *pns* \wedge *fv* *body* \subseteq $\{this\} \cup$ *set*
pns

lemma *wf-mdecl[simp]*:

wf-mdecl *P C* (*M, Ts, T, pns, body*) =
(*length* *Ts* = *length* *pns* \wedge *distinct* *pns* \wedge *this* \notin *set* *pns* \wedge *fv* *body* \subseteq $\{this\} \cup$ *set*
pns)
(*proof*)

abbreviation

wf-prog :: *prog* \Rightarrow *bool* **where**
wf-prog == *wf-prog* *wf-mdecl*

end

19 Equivalence of Big Step and Small Step Semantics

theory *Equivalence* **imports** *BigStep SmallStep WWellForm* **begin**

19.1 Some casts-lemmas

lemma *assumes* *wf:wf-prog wf-md P*

shows *casts-casts*:

$P \vdash T \text{ casts } v \text{ to } v' \implies P \vdash T \text{ casts } v' \text{ to } v'$

<proof>

lemma *casts-casts-eq*:

$\llbracket P \vdash T \text{ casts } v \text{ to } v; P \vdash T \text{ casts } v \text{ to } v'; \text{wf-prog wf-md } P \rrbracket \implies v = v'$

<proof>

lemma *assumes* *wf:wf-prog wf-md P*

shows *None-lcl-casts-values*:

$P, E \vdash \langle e, (h, l) \rangle \rightarrow \langle e', (h', l') \rangle \implies$

$(\bigwedge V. \llbracket l \ V = \text{None}; E \ V = \text{Some } T; l' \ V = \text{Some } v' \rrbracket$

$\implies P \vdash T \text{ casts } v' \text{ to } v')$

and $P, E \vdash \langle es, (h, l) \rangle [\rightarrow] \langle es', (h', l') \rangle \implies$

$(\bigwedge V. \llbracket l \ V = \text{None}; E \ V = \text{Some } T; l' \ V = \text{Some } v' \rrbracket$

$\implies P \vdash T \text{ casts } v' \text{ to } v')$

<proof>

lemma *assumes* *wf:wf-prog wf-md P*

shows *Some-lcl-casts-values*:

$P, E \vdash \langle e, (h, l) \rangle \rightarrow \langle e', (h', l') \rangle \implies$

$(\bigwedge V. \llbracket l \ V = \text{Some } v; E \ V = \text{Some } T;$

$P \vdash T \text{ casts } v'' \text{ to } v; l' \ V = \text{Some } v' \rrbracket$

$\implies P \vdash T \text{ casts } v' \text{ to } v')$

and $P, E \vdash \langle es, (h, l) \rangle [\rightarrow] \langle es', (h', l') \rangle \implies$

$(\bigwedge V. \llbracket l \ V = \text{Some } v; E \ V = \text{Some } T;$

$P \vdash T \text{ casts } v'' \text{ to } v; l' \ V = \text{Some } v' \rrbracket$

$\implies P \vdash T \text{ casts } v' \text{ to } v')$

<proof>

19.2 Small steps simulate big step

19.3 Cast

lemma *StaticCastReds*:

$$P, E \vdash \langle e, s \rangle \rightarrow^* \langle e', s' \rangle \implies P, E \vdash \langle \langle C \rangle e, s \rangle \rightarrow^* \langle \langle C \rangle e', s' \rangle$$

<proof>

lemma *StaticCastRedsNull*:

$$P, E \vdash \langle e, s \rangle \rightarrow^* \langle \text{null}, s' \rangle \implies P, E \vdash \langle \langle C \rangle e, s \rangle \rightarrow^* \langle \text{null}, s' \rangle$$

<proof>

lemma *StaticUpCastReds*:

$$\begin{aligned} & \llbracket P, E \vdash \langle e, s \rangle \rightarrow^* \langle \text{ref}(a, Cs), s' \rangle; P \vdash \text{Path last } Cs \text{ to } C \text{ via } Cs'; Ds = Cs @_p Cs' \\ & \rrbracket \\ & \implies P, E \vdash \langle \langle C \rangle e, s \rangle \rightarrow^* \langle \text{ref}(a, Ds), s' \rangle \end{aligned}$$

<proof>

lemma *StaticDownCastReds*:

$$\begin{aligned} & P, E \vdash \langle e, s \rangle \rightarrow^* \langle \text{ref}(a, Cs @ [C] @ Cs'), s' \rangle \\ & \implies P, E \vdash \langle \langle C \rangle e, s \rangle \rightarrow^* \langle \text{ref}(a, Cs @ [C]), s' \rangle \end{aligned}$$

<proof>

lemma *StaticCastRedsFail*:

$$\begin{aligned} & \llbracket P, E \vdash \langle e, s \rangle \rightarrow^* \langle \text{ref}(a, Cs), s' \rangle; C \notin \text{set } Cs; \neg P \vdash (\text{last } Cs) \preceq^* C \rrbracket \\ & \implies P, E \vdash \langle \langle C \rangle e, s \rangle \rightarrow^* \langle \text{THROW ClassCast}, s' \rangle \end{aligned}$$

<proof>

lemma *StaticCastRedsThrow*:

$$\llbracket P, E \vdash \langle e, s \rangle \rightarrow^* \langle \text{Throw } r, s' \rangle \rrbracket \implies P, E \vdash \langle \langle C \rangle e, s \rangle \rightarrow^* \langle \text{Throw } r, s' \rangle$$

<proof>

lemma *DynCastReds*:

$$P, E \vdash \langle e, s \rangle \rightarrow^* \langle e', s' \rangle \implies P, E \vdash \langle \text{Cast } C \ e, s \rangle \rightarrow^* \langle \text{Cast } C \ e', s' \rangle$$

<proof>

lemma *DynCastRedsNull*:

$$P, E \vdash \langle e, s \rangle \rightarrow^* \langle \text{null}, s' \rangle \implies P, E \vdash \langle \text{Cast } C \ e, s \rangle \rightarrow^* \langle \text{null}, s' \rangle$$

<proof>

lemma *DynCastRedsRef*:

$\llbracket P, E \vdash \langle e, s \rangle \rightarrow^* \langle \text{ref}(a, Cs), s' \rangle; \text{hp } s' \ a = \text{Some } (D, S); P \vdash \text{Path } D \text{ to } C \text{ via } Cs' \rrbracket$

$P \vdash \text{Path } D \text{ to } C \text{ unique } \rrbracket$

$$\implies P, E \vdash \langle \text{Cast } C \ e, s \rangle \rightarrow^* \langle \text{ref}(a, Cs'), s' \rangle$$

<proof>

lemma *StaticUpDynCastReds*:

$\llbracket P, E \vdash \langle e, s \rangle \rightarrow^* \langle \text{ref}(a, Cs), s' \rangle; P \vdash \text{Path last } Cs \text{ to } C \text{ unique};$

$P \vdash \text{Path last } Cs \text{ to } C \text{ via } Cs'; Ds = Cs @_p Cs' \rrbracket$

$$\implies P, E \vdash \langle \text{Cast } C \ e, s \rangle \rightarrow^* \langle \text{ref}(a, Ds), s' \rangle$$

<proof>

lemma *StaticDownDynCastReds*:

$P, E \vdash \langle e, s \rangle \rightarrow^* \langle \text{ref}(a, Cs @ [C] @ Cs'), s' \rangle$

$$\implies P, E \vdash \langle \text{Cast } C \ e, s \rangle \rightarrow^* \langle \text{ref}(a, Cs @ [C]), s' \rangle$$

<proof>

lemma *DynCastRedsFail*:

$\llbracket P, E \vdash \langle e, s \rangle \rightarrow^* \langle \text{ref}(a, Cs), s' \rangle; \text{hp } s' \ a = \text{Some } (D, S); \neg P \vdash \text{Path } D \text{ to } C \text{ unique};$

$\neg P \vdash \text{Path last } Cs \text{ to } C \text{ unique}; C \notin \text{set } Cs \rrbracket$

$$\implies P, E \vdash \langle \text{Cast } C \ e, s \rangle \rightarrow^* \langle \text{null}, s' \rangle$$

<proof>

lemma *DynCastRedsThrow*:

$$\llbracket P, E \vdash \langle e, s \rangle \rightarrow^* \langle \text{Throw } r, s' \rangle \rrbracket \implies P, E \vdash \langle \text{Cast } C \ e, s \rangle \rightarrow^* \langle \text{Throw } r, s' \rangle$$

<proof>

19.4 LAss

lemma *LAssReds*:

$$P, E \vdash \langle e, s \rangle \rightarrow^* \langle e', s' \rangle \implies P, E \vdash \langle V := e, s \rangle \rightarrow^* \langle V := e', s' \rangle$$

$\langle proof \rangle$

lemma *LAssRedsVal*:

$$\begin{aligned} & \llbracket P, E \vdash \langle e, s \rangle \rightarrow^* \langle Val\ v, (h', l') \rangle; E\ V = Some\ T; P \vdash T\ casts\ v\ to\ v' \rrbracket \\ & \implies P, E \vdash \langle V := e, s \rangle \rightarrow^* \langle Val\ v', (h', l'(V \mapsto v')) \rangle \end{aligned}$$

$\langle proof \rangle$

lemma *LAssRedsThrow*:

$$\llbracket P, E \vdash \langle e, s \rangle \rightarrow^* \langle Throw\ r, s' \rangle \rrbracket \implies P, E \vdash \langle V := e, s \rangle \rightarrow^* \langle Throw\ r, s' \rangle$$

$\langle proof \rangle$

19.5 BinOp

lemma *BinOp1Reds*:

$$P, E \vdash \langle e, s \rangle \rightarrow^* \langle e', s' \rangle \implies P, E \vdash \langle e \ll bop \gg e_2, s \rangle \rightarrow^* \langle e' \ll bop \gg e_2, s' \rangle$$

$\langle proof \rangle$

lemma *BinOp2Reds*:

$$P, E \vdash \langle e, s \rangle \rightarrow^* \langle e', s' \rangle \implies P, E \vdash \langle (Val\ v) \ll bop \gg e, s \rangle \rightarrow^* \langle (Val\ v) \ll bop \gg e', s' \rangle$$

$\langle proof \rangle$

lemma *BinOpRedsVal*:

$$\begin{aligned} & \llbracket P, E \vdash \langle e_1, s_0 \rangle \rightarrow^* \langle Val\ v_1, s_1 \rangle; P, E \vdash \langle e_2, s_1 \rangle \rightarrow^* \langle Val\ v_2, s_2 \rangle; \\ & \quad binop(bop, v_1, v_2) = Some\ v \rrbracket \\ & \implies P, E \vdash \langle e_1 \ll bop \gg e_2, s_0 \rangle \rightarrow^* \langle Val\ v, s_2 \rangle \end{aligned}$$

$\langle proof \rangle$

lemma *BinOpRedsThrow1*:

$$P, E \vdash \langle e, s \rangle \rightarrow^* \langle Throw\ r, s' \rangle \implies P, E \vdash \langle e \ll bop \gg e_2, s \rangle \rightarrow^* \langle Throw\ r, s' \rangle$$

$\langle proof \rangle$

lemma *BinOpRedsThrow2*:

$$\begin{aligned} & \llbracket P, E \vdash \langle e_1, s_0 \rangle \rightarrow^* \langle Val\ v_1, s_1 \rangle; P, E \vdash \langle e_2, s_1 \rangle \rightarrow^* \langle Throw\ r, s_2 \rangle \rrbracket \\ & \implies P, E \vdash \langle e_1 \ll bop \gg e_2, s_0 \rangle \rightarrow^* \langle Throw\ r, s_2 \rangle \end{aligned}$$

$\langle proof \rangle$

19.6 FAcc

lemma *FAccReds*:

$$P, E \vdash \langle e, s \rangle \rightarrow^* \langle e', s' \rangle \implies P, E \vdash \langle e \cdot F\{Cs\}, s \rangle \rightarrow^* \langle e' \cdot F\{Cs\}, s' \rangle$$

<proof>

lemma *FAccRedsVal*:

$$\begin{aligned} & \llbracket P, E \vdash \langle e, s \rangle \rightarrow^* \langle \text{ref}(a, Cs'), s' \rangle; \text{hp } s' a = \text{Some}(D, S); \\ & \quad Ds = Cs' @_p Cs; (Ds, fs) \in S; fs F = \text{Some } v \rrbracket \\ & \implies P, E \vdash \langle e \cdot F\{Cs\}, s \rangle \rightarrow^* \langle \text{Val } v, s' \rangle \end{aligned}$$

<proof>

lemma *FAccRedsNull*:

$$P, E \vdash \langle e, s \rangle \rightarrow^* \langle \text{null}, s' \rangle \implies P, E \vdash \langle e \cdot F\{Cs\}, s \rangle \rightarrow^* \langle \text{THROW NullPointer}, s' \rangle$$

<proof>

lemma *FAccRedsThrow*:

$$P, E \vdash \langle e, s \rangle \rightarrow^* \langle \text{Throw } r, s' \rangle \implies P, E \vdash \langle e \cdot F\{Cs\}, s \rangle \rightarrow^* \langle \text{Throw } r, s' \rangle$$

<proof>

19.7 FAss

lemma *FAssReds1*:

$$P, E \vdash \langle e, s \rangle \rightarrow^* \langle e', s' \rangle \implies P, E \vdash \langle e \cdot F\{Cs\} := e_2, s \rangle \rightarrow^* \langle e' \cdot F\{Cs\} := e_2, s' \rangle$$

<proof>

lemma *FAssReds2*:

$$P, E \vdash \langle e, s \rangle \rightarrow^* \langle e', s' \rangle \implies P, E \vdash \langle \text{Val } v \cdot F\{Cs\} := e, s \rangle \rightarrow^* \langle \text{Val } v \cdot F\{Cs\} := e', s' \rangle$$

<proof>

lemma *FAssRedsVal*:

$$\begin{aligned} & \llbracket P, E \vdash \langle e_1, s_0 \rangle \rightarrow^* \langle \text{ref}(a, Cs'), s_1 \rangle; P, E \vdash \langle e_2, s_1 \rangle \rightarrow^* \langle \text{Val } v, (h_2, l_2) \rangle; \\ & \quad h_2 a = \text{Some}(D, S); P \vdash (\text{last } Cs') \text{ has least } F:T \text{ via } Cs; P \vdash T \text{ casts } v \text{ to } v'; \\ & \quad Ds = Cs' @_p Cs; (Ds, fs) \in S \rrbracket \implies \\ & P, E \vdash \langle e_1 \cdot F\{Cs\} := e_2, s_0 \rangle \rightarrow^* \\ & \quad \langle \text{Val } v', (h_2(a \mapsto (D, \text{insert } (Ds, fs(F \mapsto v')) (S - \{(Ds, fs)\}))), l_2) \rangle \end{aligned}$$

<proof>

lemma *FAssRedsNull*:

$$\llbracket P, E \vdash \langle e_1, s_0 \rangle \rightarrow^* \langle \text{null}, s_1 \rangle; P, E \vdash \langle e_2, s_1 \rangle \rightarrow^* \langle \text{Val } v, s_2 \rangle \rrbracket \implies \\ P, E \vdash \langle e_1 \cdot F\{Cs\} := e_2, s_0 \rangle \rightarrow^* \langle \text{THROW NullPointer}, s_2 \rangle$$

<proof>

lemma *FAssRedsThrow1*:

$$P, E \vdash \langle e, s \rangle \rightarrow^* \langle \text{Throw } r, s' \rangle \implies P, E \vdash \langle e \cdot F\{Cs\} := e_2, s \rangle \rightarrow^* \langle \text{Throw } r, s' \rangle$$

<proof>

lemma *FAssRedsThrow2*:

$$\llbracket P, E \vdash \langle e_1, s_0 \rangle \rightarrow^* \langle \text{Val } v, s_1 \rangle; P, E \vdash \langle e_2, s_1 \rangle \rightarrow^* \langle \text{Throw } r, s_2 \rangle \rrbracket \\ \implies P, E \vdash \langle e_1 \cdot F\{Cs\} := e_2, s_0 \rangle \rightarrow^* \langle \text{Throw } r, s_2 \rangle$$

<proof>

19.8 ;;

lemma *SeqReds*:

$$P, E \vdash \langle e, s \rangle \rightarrow^* \langle e', s' \rangle \implies P, E \vdash \langle e;;e_2, s \rangle \rightarrow^* \langle e';;e_2, s' \rangle$$

<proof>

lemma *SeqRedsThrow*:

$$P, E \vdash \langle e, s \rangle \rightarrow^* \langle \text{Throw } r, s' \rangle \implies P, E \vdash \langle e;;e_2, s \rangle \rightarrow^* \langle \text{Throw } r, s' \rangle$$

<proof>

lemma *SeqReds2*:

$$\llbracket P, E \vdash \langle e_1, s_0 \rangle \rightarrow^* \langle \text{Val } v_1, s_1 \rangle; P, E \vdash \langle e_2, s_1 \rangle \rightarrow^* \langle e_2', s_2 \rangle \rrbracket \implies P, E \vdash \langle e_1;;e_2, s_0 \rangle \rightarrow^* \langle e_2', s_2 \rangle$$

<proof>

19.9 If

lemma *CondReds*:

$$P, E \vdash \langle e, s \rangle \rightarrow^* \langle e', s' \rangle \implies P, E \vdash \langle \text{if } (e) e_1 \text{ else } e_2, s \rangle \rightarrow^* \langle \text{if } (e') e_1 \text{ else } e_2, s' \rangle$$

<proof>

lemma *CondRedsThrow*:

$$P, E \vdash \langle e, s \rangle \rightarrow^* \langle \text{Throw } r, s \rangle \implies P, E \vdash \langle \text{if } (e) \ e_1 \ \text{else } e_2, s \rangle \rightarrow^* \langle \text{Throw } r, s \rangle$$

<proof>

lemma *CondReds2T*:

$$\llbracket P, E \vdash \langle e, s_0 \rangle \rightarrow^* \langle \text{true}, s_1 \rangle; P, E \vdash \langle e_1, s_1 \rangle \rightarrow^* \langle e', s_2 \rangle \rrbracket \implies P, E \vdash \langle \text{if } (e) \ e_1 \ \text{else } e_2, s_0 \rangle \rightarrow^* \langle e', s_2 \rangle$$

<proof>

lemma *CondReds2F*:

$$\llbracket P, E \vdash \langle e, s_0 \rangle \rightarrow^* \langle \text{false}, s_1 \rangle; P, E \vdash \langle e_2, s_1 \rangle \rightarrow^* \langle e', s_2 \rangle \rrbracket \implies P, E \vdash \langle \text{if } (e) \ e_1 \ \text{else } e_2, s_0 \rangle \rightarrow^* \langle e', s_2 \rangle$$

<proof>

19.10 While

lemma *WhileFReds*:

$$P, E \vdash \langle b, s \rangle \rightarrow^* \langle \text{false}, s \rangle \implies P, E \vdash \langle \text{while } (b) \ c, s \rangle \rightarrow^* \langle \text{unit}, s \rangle$$

<proof>

lemma *WhileRedsThrow*:

$$P, E \vdash \langle b, s \rangle \rightarrow^* \langle \text{Throw } r, s \rangle \implies P, E \vdash \langle \text{while } (b) \ c, s \rangle \rightarrow^* \langle \text{Throw } r, s \rangle$$

<proof>

lemma *WhileTReds*:

$$\begin{aligned} & \llbracket P, E \vdash \langle b, s_0 \rangle \rightarrow^* \langle \text{true}, s_1 \rangle; P, E \vdash \langle c, s_1 \rangle \rightarrow^* \langle \text{Val } v_1, s_2 \rangle; P, E \vdash \langle \text{while } (b) \ c, s_2 \rangle \rightarrow^* \langle e, s_3 \rangle \rrbracket \\ & \implies P, E \vdash \langle \text{while } (b) \ c, s_0 \rangle \rightarrow^* \langle e, s_3 \rangle \end{aligned}$$

<proof>

lemma *WhileTRedsThrow*:

$$\begin{aligned} & \llbracket P, E \vdash \langle b, s_0 \rangle \rightarrow^* \langle \text{true}, s_1 \rangle; P, E \vdash \langle c, s_1 \rangle \rightarrow^* \langle \text{Throw } r, s_2 \rangle \rrbracket \\ & \implies P, E \vdash \langle \text{while } (b) \ c, s_0 \rangle \rightarrow^* \langle \text{Throw } r, s_2 \rangle \end{aligned}$$

<proof>

19.11 Throw

lemma *ThrowReds*:

$$P, E \vdash \langle e, s \rangle \rightarrow^* \langle e', s \rangle \implies P, E \vdash \langle \text{throw } e, s \rangle \rightarrow^* \langle \text{throw } e', s \rangle$$

$\langle proof \rangle$

lemma *ThrowRedsNull*:

$$P, E \vdash \langle e, s \rangle \rightarrow^* \langle null, s^\wedge \rangle \implies P, E \vdash \langle throw\ e, s \rangle \rightarrow^* \langle THROW\ NullPointer, s^\wedge \rangle$$

$\langle proof \rangle$

lemma *ThrowRedsThrow*:

$$P, E \vdash \langle e, s \rangle \rightarrow^* \langle Throw\ r, s^\wedge \rangle \implies P, E \vdash \langle throw\ e, s \rangle \rightarrow^* \langle Throw\ r, s^\wedge \rangle$$

$\langle proof \rangle$

19.12 InitBlock

lemma *assumes* $wf:wf\text{-prog}\ wf\text{-md}\ P$

shows *InitBlockReds- aux* :

$$\begin{aligned} P, E(V \mapsto T) \vdash \langle e, s \rangle \rightarrow^* \langle e', s^\wedge \rangle \implies \\ \forall h\ l\ h'\ l'\ v\ v'.\ s = (h, l(V \mapsto v')) \longrightarrow \\ P \vdash T\ \text{casts}\ v\ \text{to}\ v' \longrightarrow s' = (h', l') \longrightarrow \\ (\exists v''\ w.\ P, E \vdash \langle \{V:T := Val\ v; e\}, (h, l) \rangle \rightarrow^* \\ \langle \{V:T := Val\ v''; e'\}, (h', l'(V := l\ V)) \rangle) \wedge \\ P \vdash T\ \text{casts}\ v''\ \text{to}\ w) \end{aligned}$$

$\langle proof \rangle$

lemma *InitBlockReds*:

$$\begin{aligned} \llbracket P, E(V \mapsto T) \vdash \langle e, (h, l(V \mapsto v')) \rangle \rightarrow^* \langle e', (h', l') \rangle; \\ P \vdash T\ \text{casts}\ v\ \text{to}\ v';\ wf\text{-prog}\ wf\text{-md}\ P \rrbracket \implies \\ \exists v''\ w.\ P, E \vdash \langle \{V:T := Val\ v; e\}, (h, l) \rangle \rightarrow^* \\ \langle \{V:T := Val\ v''; e'\}, (h', l'(V := l\ V)) \rangle) \wedge \\ P \vdash T\ \text{casts}\ v''\ \text{to}\ w \end{aligned}$$

$\langle proof \rangle$

lemma *InitBlockRedsFinal*:

$$\begin{aligned} \text{assumes}\ reds: P, E(V \mapsto T) \vdash \langle e, (h, l(V \mapsto v')) \rangle \rightarrow^* \langle e', (h', l') \rangle \\ \text{and}\ final: final\ e' \text{ and}\ casts: P \vdash T\ \text{casts}\ v\ \text{to}\ v' \\ \text{and}\ wf: wf\text{-prog}\ wf\text{-md}\ P \\ \text{shows}\ P, E \vdash \langle \{V:T := Val\ v; e\}, (h, l) \rangle \rightarrow^* \langle e', (h', l'(V := l\ V)) \rangle \end{aligned}$$

$\langle proof \rangle$

19.13 Block

lemma *BlockRedsFinal*:

$$\begin{aligned} \text{assumes}\ reds: P, E(V \mapsto T) \vdash \langle e_0, s_0 \rangle \rightarrow^* \langle e_2, (h_2, l_2) \rangle \text{ and}\ fin: final\ e_2 \\ \text{and}\ wf: wf\text{-prog}\ wf\text{-md}\ P \end{aligned}$$

shows $\bigwedge h_0 l_0. s_0 = (h_0, l_0(V := None)) \implies P, E \vdash \langle \{V:T; e_0\}, (h_0, l_0) \rangle \rightarrow^* \langle e_2, (h_2, l_2(V := l_0 V)) \rangle$

<proof>

19.14 List

lemma *ListReds1*:

$P, E \vdash \langle e, s \rangle \rightarrow^* \langle e', s' \rangle \implies P, E \vdash \langle e \# es, s \rangle [\rightarrow]^* \langle e' \# es, s' \rangle$

<proof>

lemma *ListReds2*:

$P, E \vdash \langle es, s \rangle [\rightarrow]^* \langle es', s' \rangle \implies P, E \vdash \langle Val v \# es, s \rangle [\rightarrow]^* \langle Val v \# es', s' \rangle$

<proof>

lemma *ListRedsVal*:

$\llbracket P, E \vdash \langle e, s_0 \rangle \rightarrow^* \langle Val v, s_1 \rangle; P, E \vdash \langle es, s_1 \rangle [\rightarrow]^* \langle es', s_2 \rangle \rrbracket$
 $\implies P, E \vdash \langle e \# es, s_0 \rangle [\rightarrow]^* \langle Val v \# es', s_2 \rangle$

<proof>

19.15 Call

First a few lemmas on what happens to free variables during redction.

lemma *assumes wf*: *wf-prog P*

shows *Red-fv*: $P, E \vdash \langle e, (h, l) \rangle \rightarrow \langle e', (h', l') \rangle \implies fv e' \subseteq fv e$

and $P, E \vdash \langle es, (h, l) \rangle [\rightarrow] \langle es', (h', l') \rangle \implies fvs es' \subseteq fvs es$

<proof>

lemma *Red-dom-lcl*:

$P, E \vdash \langle e, (h, l) \rangle \rightarrow \langle e', (h', l') \rangle \implies dom l' \subseteq dom l \cup fv e$ **and**

$P, E \vdash \langle es, (h, l) \rangle [\rightarrow] \langle es', (h', l') \rangle \implies dom l' \subseteq dom l \cup fvs es$

<proof>

lemma *Reds-dom-lcl*:

$\llbracket wf-prog P; P, E \vdash \langle e, (h, l) \rangle \rightarrow^* \langle e', (h', l') \rangle \rrbracket \implies dom l' \subseteq dom l \cup fv e$

<proof>

Now a few lemmas on the behaviour of blocks during reduction.

lemma *override-on-upd-lemma*:

$(\text{override-on } f (g(a \mapsto b)) A)(a := g a) = \text{override-on } f g (\text{insert } a A)$

$\langle \text{proof} \rangle$

declare *fun-upd-apply*[simp del] *map-upds-twist*[simp del]

lemma *assumes wf:wf-prog wf-md P*

shows *blocksReds*:

$\bigwedge l_0 E vs'. \llbracket \text{length } Vs = \text{length } Ts; \text{length } vs = \text{length } Ts;$
 $\text{distinct } Vs; (*\forall T \in \text{set } Ts. \text{is-type } P T;*) P \vdash Ts \text{ Casts } vs \text{ to } vs';$
 $P, E(Vs \mapsto Ts) \vdash \langle e, (h_0, l_0(Vs \mapsto vs')) \rangle \rightarrow^* \langle e', (h_1, l_1) \rangle \rrbracket$
 $\implies \exists vs''. P, E \vdash \langle \text{blocks}(Vs, Ts, vs, e), (h_0, l_0) \rangle \rightarrow^*$
 $\langle \text{blocks}(Vs, Ts, vs'', e'), (h_1, \text{override-on } l_1 l_0 (\text{set } Vs)) \rangle \wedge$
 $(\exists ws. P \vdash Ts \text{ Casts } vs'' \text{ to } ws) \wedge \text{length } vs = \text{length } vs''$

$\langle \text{proof} \rangle$

lemma *assumes wf:wf-prog wf-md P*

shows *blocksFinal*:

$\bigwedge E l vs'. \llbracket \text{length } Vs = \text{length } Ts; \text{length } vs = \text{length } Ts;$
 $(*\forall T \in \text{set } Ts. \text{is-type } P T;*) \text{final } e; P \vdash Ts \text{ Casts } vs \text{ to } vs' \rrbracket \implies$
 $P, E \vdash \langle \text{blocks}(Vs, Ts, vs, e), (h, l) \rangle \rightarrow^* \langle e, (h, l) \rangle$

$\langle \text{proof} \rangle$

lemma *assumes wfmd:wf-prog wf-md P*

and *wf*: $\text{length } Vs = \text{length } Ts \text{ length } vs = \text{length } Ts \text{ distinct } Vs$

and *casts*: $P \vdash Ts \text{ Casts } vs \text{ to } vs'$

and *reds*: $P, E(Vs \mapsto Ts) \vdash \langle e, (h_0, l_0(Vs \mapsto vs')) \rangle \rightarrow^* \langle e', (h_1, l_1) \rangle$

and *fin*: $\text{final } e'$ **and** *l2*: $l_2 = \text{override-on } l_1 l_0 (\text{set } Vs)$

shows *blocksRedsFinal*: $P, E \vdash \langle \text{blocks}(Vs, Ts, vs, e), (h_0, l_0) \rangle \rightarrow^* \langle e', (h_1, l_2) \rangle$

$\langle \text{proof} \rangle$

An now the actual method call reduction lemmas.

lemma *CallRedsObj*:

$P, E \vdash \langle e, s \rangle \rightarrow^* \langle e', s' \rangle \implies$

$P, E \vdash \langle \text{Call } e \text{ Copt } M \text{ es}, s \rangle \rightarrow^* \langle \text{Call } e' \text{ Copt } M \text{ es}, s' \rangle$

$\langle \text{proof} \rangle$

lemma *CallRedsParams*:

$$P, E \vdash \langle es, s \rangle [\rightarrow]^* \langle es', s' \rangle \implies \\ P, E \vdash \langle \text{Call } (\text{Val } v) \text{ Copt } M \text{ } es, s \rangle \rightarrow^* \langle \text{Call } (\text{Val } v) \text{ Copt } M \text{ } es', s' \rangle$$

<proof>

lemma *cast-lcl*:

$$P, E \vdash \langle \llbracket C \rrbracket (\text{Val } v), (h, l) \rangle \rightarrow \langle \text{Val } v', (h, l) \rangle \implies \\ P, E \vdash \langle \llbracket C \rrbracket (\text{Val } v), (h, l') \rangle \rightarrow \langle \text{Val } v', (h, l') \rangle$$

<proof>

lemma *cast-env*:

$$P, E \vdash \langle \llbracket C \rrbracket (\text{Val } v), (h, l) \rangle \rightarrow \langle \text{Val } v', (h, l) \rangle \implies \\ P, E' \vdash \langle \llbracket C \rrbracket (\text{Val } v), (h, l) \rangle \rightarrow \langle \text{Val } v', (h, l) \rangle$$

<proof>

lemma *Cast-step-Cast-or-fin*:

$$P, E \vdash \langle \llbracket C \rrbracket e, s \rangle \rightarrow^* \langle e', s' \rangle \implies \text{final } e' \vee (\exists e''. e' = \llbracket C \rrbracket e'')$$

<proof>

lemma *Cast-red*: $P, E \vdash \langle e, s \rangle \rightarrow^* \langle e', s' \rangle \implies$

$$(\bigwedge e_1. \llbracket e = \llbracket C \rrbracket e_0; e' = \llbracket C \rrbracket e_1 \rrbracket \implies P, E \vdash \langle e_0, s \rangle \rightarrow^* \langle e_1, s' \rangle)$$

<proof>

lemma *Cast-final*: $\llbracket P, E \vdash \langle \llbracket C \rrbracket e, s \rangle \rightarrow^* \langle e', s' \rangle; \text{final } e \rrbracket \implies$

$$\exists e'' s''. P, E \vdash \langle e, s \rangle \rightarrow^* \langle e'', s'' \rangle \wedge P, E \vdash \langle \llbracket C \rrbracket e'', s'' \rangle \rightarrow \langle e', s' \rangle \wedge \text{final } e''$$

<proof>

lemma *Cast-final-eq*:

assumes *red*: $P, E \vdash \langle \llbracket C \rrbracket e, (h, l) \rangle \rightarrow \langle e', (h, l) \rangle$

and *final*: *final* e **and** *final'*: *final* e'

shows $P, E' \vdash \langle \llbracket C \rrbracket e, (h, l') \rangle \rightarrow \langle e', (h, l') \rangle$

<proof>

lemma *CallRedsFinal*:

assumes *wwf*: *wwf-prog P*

and $P, E \vdash \langle e, s_0 \rangle \rightarrow^* \langle \text{ref}(a, Cs), s_1 \rangle$

$P, E \vdash \langle es, s_1 \rangle [\rightarrow]^* \langle \text{map Val } vs, (h_2, l_2) \rangle$

and *hp*: $h_2 a = \text{Some}(C, S)$

and *method*: $P \vdash \text{last } Cs \text{ has least } M = (Ts', T', pns', \text{body}') \text{ via } Ds$

and *select*: $P \vdash (C, Cs@_p Ds) \text{ selects } M = (Ts, T, pns, \text{body}) \text{ via } Cs'$

and *size*: $\text{size } vs = \text{size } pns$

and *casts*: $P \vdash Ts \text{ Casts } vs \text{ to } vs'$

and $l_2': l_2' = [\text{this} \mapsto \text{Ref}(a, Cs'), pns[\mapsto]vs']$

and *body-case*: $\text{new-body} = (\text{case } T' \text{ of Class } D \Rightarrow \langle D \rangle \text{body} \mid - \Rightarrow \text{body})$

and *body*: $P, E(\text{this} \mapsto \text{Class}(\text{last } Cs'), pns[\mapsto]Ts) \vdash \langle \text{new-body}, (h_2, l_2') \rangle \rightarrow^* \langle ef, (h_3, l_3) \rangle$

and *final*: *final ef*

shows $P, E \vdash \langle e \cdot M(es), s_0 \rangle \rightarrow^* \langle ef, (h_3, l_2) \rangle$

<proof>

lemma *StaticCallRedsFinal*:

assumes *wwf*: *wwf-prog P*

and $P, E \vdash \langle e, s_0 \rangle \rightarrow^* \langle \text{ref}(a, Cs), s_1 \rangle$

$P, E \vdash \langle es, s_1 \rangle [\rightarrow]^* \langle \text{map Val } vs, (h_2, l_2) \rangle$

and *path-unique*: $P \vdash \text{Path}(\text{last } Cs) \text{ to } C \text{ unique}$

and *path-via*: $P \vdash \text{Path}(\text{last } Cs) \text{ to } C \text{ via } Cs''$

and *Ds*: $Ds = (Cs@_p Cs'')@_p Cs'$

and *least*: $P \vdash C \text{ has least } M = (Ts, T, pns, \text{body}) \text{ via } Cs'$

and *size*: $\text{size } vs = \text{size } pns$

and *casts*: $P \vdash Ts \text{ Casts } vs \text{ to } vs'$

and $l_2': l_2' = [\text{this} \mapsto \text{Ref}(a, Ds), pns[\mapsto]vs']$

and *body*: $P, E(\text{this} \mapsto \text{Class}(\text{last } Ds), pns[\mapsto]Ts) \vdash \langle \text{body}, (h_2, l_2') \rangle \rightarrow^* \langle ef, (h_3, l_3) \rangle$

and *final*: *final ef*

shows $P, E \vdash \langle e \cdot (C::)M(es), s_0 \rangle \rightarrow^* \langle ef, (h_3, l_2) \rangle$

<proof>

lemma *CallRedsThrowParams*:

$\llbracket P, E \vdash \langle e, s_0 \rangle \rightarrow^* \langle \text{Val } v, s_1 \rangle;$

$P, E \vdash \langle es, s_1 \rangle [\rightarrow]^* \langle \text{map Val } vs_1 @ \text{Throw } ex \# es_2, s_2 \rangle \rrbracket$

$\implies P, E \vdash \langle \text{Call } e \text{ Copt } M \text{ } es, s_0 \rangle \rightarrow^* \langle \text{Throw } ex, s_2 \rangle$

<proof>

lemma *CallRedsThrowObj*:

$P, E \vdash \langle e, s_0 \rangle \rightarrow^* \langle \text{Throw } ex, s_1 \rangle \implies P, E \vdash \langle \text{Call } e \text{ Copt } M \text{ } es, s_0 \rangle \rightarrow^* \langle \text{Throw}$

$ex, s_1 \rangle$

$\langle proof \rangle$

lemma *CallRedsNull*:

$$\begin{aligned} & \llbracket P, E \vdash \langle e, s_0 \rangle \rightarrow^* \langle null, s_1 \rangle; P, E \vdash \langle es, s_1 \rangle [\rightarrow]^* \langle map\ Val\ vs, s_2 \rangle \rrbracket \\ & \implies P, E \vdash \langle Call\ e\ Copt\ M\ es, s_0 \rangle \rightarrow^* \langle THROW\ NullPointer, s_2 \rangle \end{aligned}$$

$\langle proof \rangle$

19.16 The main Theorem

lemma *assumes wwf*: *wwf-prog* P

shows *big-by-small*: $P, E \vdash \langle e, s \rangle \Rightarrow \langle e', s' \rangle \implies P, E \vdash \langle e, s \rangle \rightarrow^* \langle e', s' \rangle$

and *bigs-by-small*s: $P, E \vdash \langle es, s \rangle [\Rightarrow] \langle es', s' \rangle \implies P, E \vdash \langle es, s \rangle [\rightarrow]^* \langle es', s' \rangle$

$\langle proof \rangle$ **thm** *CallRedsNull* $\langle proof \rangle$

19.17 Big steps simulates small step

The big step equivalent of *RedWhile*:

lemma *unfold-while*:

$$P, E \vdash \langle while(b)\ c, s \rangle \Rightarrow \langle e', s' \rangle = P, E \vdash \langle if(b)\ (c;;while(b)\ c)\ else\ (unit), s \rangle \Rightarrow \langle e', s' \rangle$$

$\langle proof \rangle$

lemma *blocksEval*:

$$\begin{aligned} & \wedge Ts\ vs\ l\ l'\ E. \llbracket size\ ps = size\ Ts; size\ ps = size\ vs; \\ & \quad P, E \vdash \langle blocks(ps, Ts, vs, e), (h, l) \rangle \Rightarrow \langle e', (h', l') \rangle \rrbracket \\ & \implies \exists l''\ vs'. P, E(ps\ [\mapsto]\ Ts) \vdash \langle e, (h, l(ps\ [\mapsto]\ vs')) \rangle \Rightarrow \langle e', (h', l'') \rangle \wedge \\ & \quad P \vdash Ts\ Casts\ vs\ to\ vs' \wedge length\ vs' = length\ vs \end{aligned}$$

$\langle proof \rangle$

lemma *CastblocksEval*:

$$\begin{aligned} & \wedge Ts\ vs\ l\ l'\ E. \llbracket size\ ps = size\ Ts; size\ ps = size\ vs; \\ & \quad P, E \vdash \langle \llbracket C' \rrbracket (blocks(ps, Ts, vs, e), (h, l)) \rangle \Rightarrow \langle e', (h', l') \rangle \rrbracket \\ & \implies \exists l''\ vs'. P, E(ps\ [\mapsto]\ Ts) \vdash \langle \llbracket C' \rrbracket e, (h, l(ps\ [\mapsto]\ vs')) \rangle \Rightarrow \langle e', (h', l'') \rangle \wedge \\ & \quad P \vdash Ts\ Casts\ vs\ to\ vs' \wedge length\ vs' = length\ vs \end{aligned}$$

$\langle proof \rangle$

lemma

assumes $wf: wwf\text{-}prog\ P$

shows $eval\text{-}restrict\text{-}lcl:$

$P, E \vdash \langle e, (h, l) \rangle \Rightarrow \langle e', (h', l') \rangle \Longrightarrow (\bigwedge W. fv\ e \subseteq W \Longrightarrow P, E \vdash \langle e, (h, l |' W) \rangle \Rightarrow \langle e', (h', l' |' W) \rangle)$

and $P, E \vdash \langle es, (h, l) \rangle [\Rightarrow] \langle es', (h', l') \rangle \Longrightarrow (\bigwedge W. fvs\ es \subseteq W \Longrightarrow P, E \vdash \langle es, (h, l |' W) \rangle [\Rightarrow] \langle es', (h', l' |' W) \rangle)$

$\langle proof \rangle$

lemma $eval\text{-}notfree\text{-}unchanged:$

assumes $wf: wwf\text{-}prog\ P$

shows $P, E \vdash \langle e, (h, l) \rangle \Rightarrow \langle e', (h', l') \rangle \Longrightarrow (\bigwedge V. V \notin fv\ e \Longrightarrow l'\ V = l\ V)$

and $P, E \vdash \langle es, (h, l) \rangle [\Rightarrow] \langle es', (h', l') \rangle \Longrightarrow (\bigwedge V. V \notin fvs\ es \Longrightarrow l'\ V = l\ V)$

$\langle proof \rangle$

lemma $eval\text{-}closed\text{-}lcl\text{-}unchanged:$

assumes $wf: wwf\text{-}prog\ P$

and $eval: P, E \vdash \langle e, (h, l) \rangle \Rightarrow \langle e', (h', l') \rangle$

and $fv: fv\ e = \{\}$

shows $l' = l$

$\langle proof \rangle$

declare $split\text{-}paired\text{-}All$ [*simp del*] $split\text{-}paired\text{-}Ex$ [*simp del*]

$\langle ML \rangle$

lemma $list\text{-}eval\text{-}Throw:$

assumes $eval\text{-}e: P, E \vdash \langle throw\ x, s \rangle \Rightarrow \langle e', s' \rangle$

shows $P, E \vdash \langle map\ Val\ vs\ @\ throw\ x\ \# es', s \rangle [\Rightarrow] \langle map\ Val\ vs\ @\ e'\ \# es', s' \rangle$

$\langle proof \rangle$

The key lemma:

lemma

assumes $wf: wwf\text{-}prog\ P$

shows *extend-1-eval*:

$P, E \vdash \langle e, s \rangle \rightarrow \langle e'', s'' \rangle \implies (\bigwedge s' e'. P, E \vdash \langle e'', s'' \rangle \Rightarrow \langle e', s' \rangle \implies P, E \vdash \langle e, s \rangle \Rightarrow \langle e', s' \rangle)$

and *extend-1-evals*:

$P, E \vdash \langle es, t \rangle [\rightarrow] \langle es'', t'' \rangle \implies (\bigwedge t' es'. P, E \vdash \langle es'', t'' \rangle [\Rightarrow] \langle es', t' \rangle \implies P, E \vdash \langle es, t \rangle [\Rightarrow] \langle es', t' \rangle)$

<proof>

declare *split-paired-All* [*simp*] *split-paired-Ex* [*simp*]

<ML>

Its extension to $\rightarrow*$:

lemma *extend-eval*:

assumes *wf*: *wf-prog* *P*

and *reds*: $P, E \vdash \langle e, s \rangle \rightarrow* \langle e'', s'' \rangle$ **and** *eval-rest*: $P, E \vdash \langle e'', s'' \rangle \Rightarrow \langle e', s' \rangle$

shows $P, E \vdash \langle e, s \rangle \Rightarrow \langle e', s' \rangle$

<proof>

lemma *extend-evals*:

assumes *wf*: *wf-prog* *P*

and *reds*: $P, E \vdash \langle es, s \rangle [\rightarrow]* \langle es'', s'' \rangle$ **and** *eval-rest*: $P, E \vdash \langle es'', s'' \rangle [\Rightarrow] \langle es', s' \rangle$

shows $P, E \vdash \langle es, s \rangle [\Rightarrow] \langle es', s' \rangle$

<proof>

Finally, small step semantics can be simulated by big step semantics:

theorem

assumes *wf*: *wf-prog* *P*

shows *small-by-big*: $\llbracket P, E \vdash \langle e, s \rangle \rightarrow* \langle e', s' \rangle; \text{final } e' \rrbracket \implies P, E \vdash \langle e, s \rangle \Rightarrow \langle e', s' \rangle$

and $\llbracket P, E \vdash \langle es, s \rangle [\rightarrow]* \langle es', s' \rangle; \text{finals } es' \rrbracket \implies P, E \vdash \langle es, s \rangle [\Rightarrow] \langle es', s' \rangle$

<proof>

19.18 Equivalence

And now, the crowning achievement:

corollary *big-iff-small*:

wf-prog *P* \implies

$P, E \vdash \langle e, s \rangle \Rightarrow \langle e', s' \rangle = (P, E \vdash \langle e, s \rangle \rightarrow* \langle e', s' \rangle \wedge \text{final } e')$

<proof>

end

20 Definite assignment

theory *DefAss* **imports** *BigStep* **begin**

20.1 Hypersets

types *hyperset* = *vname set option*

constdefs

hyperUn :: *hyperset* \Rightarrow *hyperset* \Rightarrow *hyperset* (**infixl** \sqcup 65)
 $A \sqcup B \equiv$ *case* *A* of *None* \Rightarrow *None*
| $[A] \Rightarrow$ (*case* *B* of *None* \Rightarrow *None* | $[B] \Rightarrow [A \cup B]$)

hyperInt :: *hyperset* \Rightarrow *hyperset* \Rightarrow *hyperset* (**infixl** \sqcap 70)
 $A \sqcap B \equiv$ *case* *A* of *None* \Rightarrow *B*
| $[A] \Rightarrow$ (*case* *B* of *None* \Rightarrow $[A]$ | $[B] \Rightarrow [A \cap B]$)

hyperDiff1 :: *hyperset* \Rightarrow *vname* \Rightarrow *hyperset* (**infixl** \ominus 65)
 $A \ominus a \equiv$ *case* *A* of *None* \Rightarrow *None* | $[A] \Rightarrow [A - \{a\}]$

hyper-isin :: *vname* \Rightarrow *hyperset* \Rightarrow *bool* (**infix** $\in\in$ 50)
 $a \in\in A \equiv$ *case* *A* of *None* \Rightarrow *True* | $[A] \Rightarrow a \in A$

hyper-subset :: *hyperset* \Rightarrow *hyperset* \Rightarrow *bool* (**infix** \sqsubseteq 50)
 $A \sqsubseteq B \equiv$ *case* *B* of *None* \Rightarrow *True*
| $[B] \Rightarrow$ (*case* *A* of *None* \Rightarrow *False* | $[A] \Rightarrow A \subseteq B$)

lemmas *hyperset-defs* =

hyperUn-def hyperInt-def hyperDiff1-def hyper-isin-def hyper-subset-def

lemma [*simp*]: $[\{\}] \sqcup A = A \wedge A \sqcup [\{\}] = A$
<proof>

lemma [*simp*]: $[A] \sqcup [B] = [A \cup B] \wedge [A] \ominus a = [A - \{a\}]$
<proof>

lemma [*simp*]: $\text{None} \sqcup A = \text{None} \wedge A \sqcup \text{None} = \text{None}$
<proof>

lemma [*simp*]: $a \in\in \text{None} \wedge \text{None} \ominus a = \text{None}$
<proof>

lemma *hyperUn-assoc*: $(A \sqcup B) \sqcup C = A \sqcup (B \sqcup C)$
<proof>

lemma *hyper-insert-comm*: $A \sqcup [\{a\}] = [\{a\}] \sqcup A \wedge A \sqcup ([\{a\}] \sqcup B) = [\{a\}] \sqcup (A \sqcup B)$
<proof>

20.2 Definite assignment

consts

$\mathcal{A} :: \text{expr} \Rightarrow \text{hyperset}$
 $\mathcal{A}s :: \text{expr list} \Rightarrow \text{hyperset}$
 $\mathcal{D} :: \text{expr} \Rightarrow \text{hyperset} \Rightarrow \text{bool}$
 $\mathcal{D}s :: \text{expr list} \Rightarrow \text{hyperset} \Rightarrow \text{bool}$

primrec

$\mathcal{A} (\text{new } C) = [\{\}]$
 $\mathcal{A} (\text{Cast } C \ e) = \mathcal{A} \ e$
 $\mathcal{A} (\downarrow C) \ e) = \mathcal{A} \ e$
 $\mathcal{A} (\text{Val } v) = [\{\}]$
 $\mathcal{A} (e_1 \ll \text{bop} \gg e_2) = \mathcal{A} \ e_1 \sqcup \mathcal{A} \ e_2$
 $\mathcal{A} (\text{Var } V) = [\{\}]$
 $\mathcal{A} (\text{LAss } V \ e) = [\{V\}] \sqcup \mathcal{A} \ e$
 $\mathcal{A} (e \cdot F\{Cs\}) = \mathcal{A} \ e$
 $\mathcal{A} (e_1 \cdot F\{Cs\} := e_2) = \mathcal{A} \ e_1 \sqcup \mathcal{A} \ e_2$
 $\mathcal{A} (\text{Call } e \ \text{Copt } M \ es) = \mathcal{A} \ e \sqcup \mathcal{A} \ s \ es$
 $\mathcal{A} (\{V:T; e\}) = \mathcal{A} \ e \ominus V$
 $\mathcal{A} (e_1 ;; e_2) = \mathcal{A} \ e_1 \sqcup \mathcal{A} \ e_2$
 $\mathcal{A} (\text{if } (e) \ e_1 \ \text{else } e_2) = \mathcal{A} \ e \sqcup (\mathcal{A} \ e_1 \sqcap \mathcal{A} \ e_2)$
 $\mathcal{A} (\text{while } (b) \ e) = \mathcal{A} \ b$
 $\mathcal{A} (\text{throw } e) = \text{None}$

$\mathcal{A}s (\[]) = [\{\}]$
 $\mathcal{A}s (e \# es) = \mathcal{A} \ e \sqcup \mathcal{A} \ s \ es$

primrec

$\mathcal{D} (\text{new } C) \ A = \text{True}$
 $\mathcal{D} (\text{Cast } C \ e) \ A = \mathcal{D} \ e \ A$
 $\mathcal{D} (\downarrow C) \ e) \ A = \mathcal{D} \ e \ A$
 $\mathcal{D} (\text{Val } v) \ A = \text{True}$
 $\mathcal{D} (e_1 \ll \text{bop} \gg e_2) \ A = (\mathcal{D} \ e_1 \ A \wedge \mathcal{D} \ e_2 \ (A \sqcup \mathcal{A} \ e_1))$
 $\mathcal{D} (\text{Var } V) \ A = (V \in \in A)$
 $\mathcal{D} (\text{LAss } V \ e) \ A = \mathcal{D} \ e \ A$
 $\mathcal{D} (e \cdot F\{Cs\}) \ A = \mathcal{D} \ e \ A$
 $\mathcal{D} (e_1 \cdot F\{Cs\} := e_2) \ A = (\mathcal{D} \ e_1 \ A \wedge \mathcal{D} \ e_2 \ (A \sqcup \mathcal{A} \ e_1))$
 $\mathcal{D} (\text{Call } e \ \text{Copt } M \ es) \ A = (\mathcal{D} \ e \ A \wedge \mathcal{D} \ s \ es \ (A \sqcup \mathcal{A} \ e))$
 $\mathcal{D} (\{V:T; e\}) \ A = \mathcal{D} \ e \ (A \ominus V)$
 $\mathcal{D} (e_1 ;; e_2) \ A = (\mathcal{D} \ e_1 \ A \wedge \mathcal{D} \ e_2 \ (A \sqcup \mathcal{A} \ e_1))$
 $\mathcal{D} (\text{if } (e) \ e_1 \ \text{else } e_2) \ A =$
 $(\mathcal{D} \ e \ A \wedge \mathcal{D} \ e_1 \ (A \sqcup \mathcal{A} \ e) \wedge \mathcal{D} \ e_2 \ (A \sqcup \mathcal{A} \ e))$
 $\mathcal{D} (\text{while } (e) \ c) \ A = (\mathcal{D} \ e \ A \wedge \mathcal{D} \ c \ (A \sqcup \mathcal{A} \ e))$
 $\mathcal{D} (\text{throw } e) \ A = \mathcal{D} \ e \ A$

$\mathcal{D}s (\[]) \ A = \text{True}$
 $\mathcal{D}s (e \# es) \ A = (\mathcal{D} \ e \ A \wedge \mathcal{D} \ s \ es \ (A \sqcup \mathcal{A} \ e))$

lemma $\mathcal{A}s\text{-map-Val[simp]}$: $\mathcal{A}s (\text{map } \text{Val } vs) = [\{\}]$

<proof>

lemma *D-append[iff]*: $\bigwedge A. \mathcal{D}s (es @ es') A = (\mathcal{D}s es A \wedge \mathcal{D}s es' (A \sqcup \mathcal{A}s es))$
<proof>

lemma *A-fv*: $\bigwedge A. \mathcal{A} e = \lfloor A \rfloor \implies A \subseteq fv e$
and $\bigwedge A. \mathcal{A}s es = \lfloor A \rfloor \implies A \subseteq fvs es$

<proof>

lemma *sqUn-lem*: $A \sqsubseteq A' \implies A \sqcup B \sqsubseteq A' \sqcup B$
<proof>

lemma *diff-lem*: $A \sqsubseteq A' \implies A \ominus b \sqsubseteq A' \ominus b$
<proof>

lemma *D-mono*: $\bigwedge A A'. A \sqsubseteq A' \implies \mathcal{D} e A \implies \mathcal{D} (e::expr) A'$
and *Ds-mono*: $\bigwedge A A'. A \sqsubseteq A' \implies \mathcal{D}s es A \implies \mathcal{D}s (es::expr list) A'$

<proof>

lemma *D-mono'*: $\mathcal{D} e A \implies A \sqsubseteq A' \implies \mathcal{D} e A'$
and *Ds-mono'*: $\mathcal{D}s es A \implies A \sqsubseteq A' \implies \mathcal{D}s es A'$
<proof>

end

21 Runtime Well-typedness

theory *WellTypeRT* **imports** *WellType* **begin**

21.1 Run time types

consts

typeof-h :: *prog* \Rightarrow *heap* \Rightarrow *val* \Rightarrow *ty option* ($- \vdash \text{typeof } -$)

primrec

$P \vdash \text{typeof}_h \text{Unit} = \text{Some Void}$

$P \vdash \text{typeof}_h \text{Null} = \text{Some NT}$

$P \vdash \text{typeof}_h (\text{Bool } b) = \text{Some Boolean}$

$P \vdash \text{typeof}_h (\text{Intg } i) = \text{Some Integer}$

$P \vdash \text{typeof}_h (\text{Ref } r) = (\text{case } h \text{ (the-addr (Ref } r)) \text{ of None } \Rightarrow \text{None}$
 $\quad \mid \text{Some}(C,S) \Rightarrow (\text{if Subobjs } P \ C \text{ (the-path(Ref } r)) \text{ then}$
 $\quad \quad \text{Some(Class(last(the-path(Ref } r)))}$
 $\quad \quad \text{else None}))$

lemma *type-eq-type*: $\text{typeof } v = \text{Some } T \Longrightarrow P \vdash \text{typeof}_h v = \text{Some } T$
 $\langle \text{proof} \rangle$

lemma *typeof-Void* [*simp*]: $P \vdash \text{typeof}_h v = \text{Some Void} \Longrightarrow v = \text{Unit}$
 $\langle \text{proof} \rangle$

lemma *typeof-NT* [*simp*]: $P \vdash \text{typeof}_h v = \text{Some NT} \Longrightarrow v = \text{Null}$
 $\langle \text{proof} \rangle$

lemma *typeof-Boolean* [*simp*]: $P \vdash \text{typeof}_h v = \text{Some Boolean} \Longrightarrow \exists b. v = \text{Bool } b$
 $\langle \text{proof} \rangle$

lemma *typeof-Integer* [*simp*]: $P \vdash \text{typeof}_h v = \text{Some Integer} \Longrightarrow \exists i. v = \text{Intg } i$
 $\langle \text{proof} \rangle$

lemma *typeof-Class-Subo*:

$P \vdash \text{typeof}_h v = \text{Some (Class } C) \Longrightarrow$

$\exists a \ Cs \ D \ S. v = \text{Ref}(a,Cs) \wedge h \ a = \text{Some}(D,S) \wedge \text{Subobjs } P \ D \ Cs \wedge \text{last } Cs = C$
 $\langle \text{proof} \rangle$

21.2 The rules

inductive

$WTrt :: [\text{prog}, \text{env}, \text{heap}, \text{expr}, \quad \text{ty} \quad] \Rightarrow \text{bool}$
 $(-, -, - \vdash - : - \quad [51, 51, 51] 50)$

and $WTrts :: [\text{prog}, \text{env}, \text{heap}, \text{expr list}, \text{ty list}] \Rightarrow \text{bool}$
 $(-, -, - \vdash - [:] - \quad [51, 51, 51] 50)$

for $P :: \text{prog}$

where

$WTrtNew:$
 $is-class\ P\ C \implies$
 $P, E, h \vdash new\ C : Class\ C$

$| WTrtDynCast:$
 $\llbracket P, E, h \vdash e : T; is-refT\ T; is-class\ P\ C \rrbracket$
 $\implies P, E, h \vdash Cast\ C\ e : Class\ C$

$| WTrtStaticCast:$
 $\llbracket P, E, h \vdash e : T; is-refT\ T; is-class\ P\ C \rrbracket$
 $\implies P, E, h \vdash \langle C \rangle e : Class\ C$

$| WTrtVal:$
 $P \vdash typeof_h\ v = Some\ T \implies$
 $P, E, h \vdash Val\ v : T$

$| WTrtVar:$
 $E\ V = Some\ T \implies$
 $P, E, h \vdash Var\ V : T$

$| WTrtBinOp:$
 $\llbracket P, E, h \vdash e_1 : T_1; P, E, h \vdash e_2 : T_2;$
 $\quad case\ bop\ of\ Eq \Rightarrow T = Boolean$
 $\quad \quad | Add \Rightarrow T_1 = Integer \wedge T_2 = Integer \wedge T = Integer \rrbracket$
 $\implies P, E, h \vdash e_1 \ll bop \gg e_2 : T$

$| WTrtLAss:$
 $\llbracket E\ V = Some\ T; P, E, h \vdash e : T'; P \vdash T' \leq T \rrbracket$
 $\implies P, E, h \vdash V := e : T$

$| WTrtFAcc:$
 $\llbracket P, E, h \vdash e : Class\ C; Cs \neq []; P \vdash C\ has\ least\ F:T\ via\ Cs \rrbracket$
 $\implies P, E, h \vdash e \cdot F\{Cs\} : T$

$| WTrtFAccNT:$
 $P, E, h \vdash e : NT \implies P, E, h \vdash e \cdot F\{Cs\} : T$

$| WTrtFAss:$
 $\llbracket P, E, h \vdash e_1 : Class\ C; Cs \neq [];$
 $\quad P \vdash C\ has\ least\ F:T\ via\ Cs; P, E, h \vdash e_2 : T'; P \vdash T' \leq T \rrbracket$
 $\implies P, E, h \vdash e_1 \cdot F\{Cs\} := e_2 : T$

$| WTrtFAssNT:$
 $\llbracket P, E, h \vdash e_1 : NT; P, E, h \vdash e_2 : T'; P \vdash T' \leq T \rrbracket$
 $\implies P, E, h \vdash e_1 \cdot F\{Cs\} := e_2 : T$

$| WTrtCall:$
 $\llbracket P, E, h \vdash e : Class\ C; P \vdash C\ has\ least\ M = (Ts, T, m)\ via\ Cs;$

$$P, E, h \vdash es \text{ [:] } Ts'; P \vdash Ts' \text{ [}\leq\text{] } Ts \text{]} \\ \implies P, E, h \vdash e \cdot M(es) : T$$

| *WTrtStaticCall*:
 $\llbracket P, E, h \vdash e : \text{Class } C'; P \vdash \text{Path } C' \text{ to } C \text{ unique};$
 $P \vdash C \text{ has least } M = (Ts, T, m) \text{ via } Cs;$
 $P, E, h \vdash es \text{ [:] } Ts'; P \vdash Ts' \text{ [}\leq\text{] } Ts \text{]}$
 $\implies P, E, h \vdash e \cdot (C::)M(es) : T$

| *WTrtCallNT*:
 $\llbracket P, E, h \vdash e : NT; P, E, h \vdash es \text{ [:] } Ts \text{]} \implies P, E, h \vdash \text{Call } e \text{ Copt } M \text{ es} : T$

| *WTrtBlock*:
 $\llbracket P, E(V \mapsto T), h \vdash e : T'; \text{is-type } P \ T \text{]} \implies$
 $P, E, h \vdash \{ V:T; e \} : T'$

| *WTrtSeq*:
 $\llbracket P, E, h \vdash e_1 : T_1; P, E, h \vdash e_2 : T_2 \text{]} \implies P, E, h \vdash e_1;;e_2 : T_2$

| *WTrtCond*:
 $\llbracket P, E, h \vdash e : \text{Boolean}; P, E, h \vdash e_1 : T; P, E, h \vdash e_2 : T \text{]}$
 $\implies P, E, h \vdash \text{if } (e) \ e_1 \ \text{else } e_2 : T$

| *WTrtWhile*:
 $\llbracket P, E, h \vdash e : \text{Boolean}; P, E, h \vdash c : T \text{]}$
 $\implies P, E, h \vdash \text{while}(e) \ c : \text{Void}$

| *WTrtThrow*:
 $\llbracket P, E, h \vdash e : T'; \text{is-refT } T \text{]}$
 $\implies P, E, h \vdash \text{throw } e : T$

| *WTrtNil*:
 $P, E, h \vdash [] \text{ [:] } []$

| *WTrtCons*:
 $\llbracket P, E, h \vdash e : T; P, E, h \vdash es \text{ [:] } Ts \text{]} \implies P, E, h \vdash e \# es \text{ [:] } T \# Ts$

declare

WTrt-WTrts.intros[intro!]

WTrtNil[iff]

declare

WTrtFAcc[rule del] *WTrtFAccNT*[rule del]

WTrtFAss[rule del] *WTrtFAssNT*[rule del]

WTrtCall[rule del] *WTrtCallNT*[rule del]

lemmas $WTrt-induct = WTrt-WTrts.induct$ [*split-format (complete)*]
and $WTrt-inducts = WTrt-WTrts.inducts$ [*split-format (complete)*]

21.3 Easy consequences

lemma [*iff*]: $(P, E, h \vdash [] \[:] Ts) = (Ts = [])$

<proof>

lemma [*iff*]: $(P, E, h \vdash e \# es \[:] T \# Ts) = (P, E, h \vdash e : T \wedge P, E, h \vdash es \[:] Ts)$

<proof>

lemma [*iff*]: $(P, E, h \vdash (e \# es) \[:] Ts) =$
 $(\exists U Us. Ts = U \# Us \wedge P, E, h \vdash e : U \wedge P, E, h \vdash es \[:] Us)$

<proof>

lemma [*simp*]: $\forall Ts. (P, E, h \vdash es_1 @ es_2 \[:] Ts) =$
 $(\exists Ts_1 Ts_2. Ts = Ts_1 @ Ts_2 \wedge P, E, h \vdash es_1 \[:] Ts_1 \& P, E, h \vdash es_2 \[:] Ts_2)$

<proof>

lemma [*iff*]: $P, E, h \vdash Val v : T = (P \vdash typeof_h v = Some T)$

<proof>

lemma [*iff*]: $P, E, h \vdash Var V : T = (E V = Some T)$

<proof>

lemma [*iff*]: $P, E, h \vdash e_1 ;; e_2 : T_2 = (\exists T_1. P, E, h \vdash e_1 : T_1 \wedge P, E, h \vdash e_2 : T_2)$

<proof>

lemma [*iff*]: $P, E, h \vdash \{V : T; e\} : T' = (P, E(V \mapsto T), h \vdash e : T' \wedge is-type P T)$

<proof>

inductive-cases $WTrt-elim-cases[elim!]$:

$P, E, h \vdash \text{new } C : T$
 $P, E, h \vdash \text{Cast } C \ e : T$
 $P, E, h \vdash \langle C \rangle e : T$
 $P, E, h \vdash e_1 \ll \text{bop} \gg e_2 : T$
 $P, E, h \vdash V := e : T$
 $P, E, h \vdash e \cdot F\{Cs\} : T$
 $P, E, h \vdash e \cdot F\{Cs\} := v : T$
 $P, E, h \vdash e \cdot M(es) : T$
 $P, E, h \vdash e \cdot (C ::) M(es) : T$
 $P, E, h \vdash \text{if } (e) \ e_1 \ \text{else } e_2 : T$
 $P, E, h \vdash \text{while}(e) \ c : T$
 $P, E, h \vdash \text{throw } e : T$

21.4 Some interesting lemmas

lemma *WTrts-Val[simp]*:

$\bigwedge Ts. (P, E, h \vdash \text{map Val } vs \ [:] \ Ts) = (\text{map } (\lambda v. (P \vdash \text{typeof}_h) \ v) \ vs = \text{map Some } Ts)$

<proof>

lemma *WTrts-same-length*: $\bigwedge Ts. P, E, h \vdash es \ [:] \ Ts \implies \text{length } es = \text{length } Ts$

<proof>

lemma *WTrt-env-mono*:

$P, E, h \vdash e : T \implies (\bigwedge E'. E \subseteq_m E' \implies P, E', h \vdash e : T)$ **and**
 $P, E, h \vdash es \ [:] \ Ts \implies (\bigwedge E'. E \subseteq_m E' \implies P, E', h \vdash es \ [:] \ Ts)$

<proof>

lemma *WT-implies-WTrt*: $P, E \vdash e :: T \implies P, E, h \vdash e : T$

and *WTs-implies-WTrts*: $P, E \vdash es \ [::] \ Ts \implies P, E, h \vdash es \ [:] \ Ts$

<proof>

end

22 Conformance Relations for Proofs

theory *Conform* **imports** *Exceptions WellTypeRT* **begin**

consts

$conf :: prog \Rightarrow heap \Rightarrow val \Rightarrow ty \Rightarrow bool \quad (-, - \vdash - : \leq - \text{ [51,51,51,51] } 50)$

primrec

$P, h \vdash v : \leq Void \quad = (P \vdash \text{typeof}_h v = \text{Some } Void)$
 $P, h \vdash v : \leq Boolean \quad = (P \vdash \text{typeof}_h v = \text{Some } Boolean)$
 $P, h \vdash v : \leq Integer \quad = (P \vdash \text{typeof}_h v = \text{Some } Integer)$
 $P, h \vdash v : \leq NT \quad = (P \vdash \text{typeof}_h v = \text{Some } NT)$
 $P, h \vdash v : \leq (Class C) \quad = (P \vdash \text{typeof}_h v = \text{Some } (Class C) \vee P \vdash \text{typeof}_h v = \text{Some } NT)$

constdefs

$fconf :: prog \Rightarrow heap \Rightarrow ('a \rightarrow val) \Rightarrow ('a \rightarrow ty) \Rightarrow bool \quad (-, - \vdash - '(:\leq)' - \text{ [51,51,51,51] } 50)$

$P, h \vdash v_m (: \leq) T_m \equiv$

$\forall FD T. T_m FD = \text{Some } T \longrightarrow (\exists v. v_m FD = \text{Some } v \wedge P, h \vdash v : \leq T)$

$oconf :: prog \Rightarrow heap \Rightarrow obj \Rightarrow bool \quad (-, - \vdash - \surd \text{ [51,51,51] } 50)$

$P, h \vdash obj \surd \equiv \text{let } (C, S) = obj \text{ in}$

$(\forall Cs. \text{Subobjs } P C Cs \longrightarrow (\exists ! fs'. (Cs, fs') \in S)) \wedge$

$(\forall Cs fs'. (Cs, fs') \in S \longrightarrow \text{Subobjs } P C Cs \wedge$

$(\exists fs Bs ms. \text{class } P (\text{last } Cs) = \text{Some } (Bs, fs, ms) \wedge$
 $P, h \vdash fs' (: \leq) \text{map-of } fs))$

$hconf :: prog \Rightarrow heap \Rightarrow bool \quad (- \vdash - \surd \text{ [51,51] } 50)$

$P \vdash h \surd \equiv$

$(\forall a obj. h a = \text{Some } obj \longrightarrow P, h \vdash obj \surd) \wedge \text{preallocated } h$

$lconf :: prog \Rightarrow heap \Rightarrow ('a \rightarrow val) \Rightarrow ('a \rightarrow ty) \Rightarrow bool \quad (-, - \vdash - '(:\leq)_w - \text{ [51,51,51,51] } 50)$

$P, h \vdash v_m (: \leq)_w T_m \equiv$

$\forall V v. v_m V = \text{Some } v \longrightarrow (\exists T. T_m V = \text{Some } T \wedge P, h \vdash v : \leq T)$

abbreviation

$conf_s :: prog \Rightarrow heap \Rightarrow val \text{ list} \Rightarrow ty \text{ list} \Rightarrow bool$

$(-, - \vdash - [:\leq] - \text{ [51,51,51,51] } 50) \text{ where}$

$P, h \vdash vs [:\leq] Ts \equiv \text{list-all2 } (conf P h) \text{ vs } Ts$

22.1 Value conformance $: \leq$

lemma *conf-Null* [*simp*]: $P, h \vdash \text{Null} : \leq T = P \vdash NT \leq T$

<proof>

lemma *typeof-conf[simp]*: $P \vdash \text{typeof}_h v = \text{Some } T \implies P, h \vdash v : \leq T$
 ⟨proof⟩

lemma *typeof-lit-conf[simp]*: $\text{typeof } v = \text{Some } T \implies P, h \vdash v : \leq T$
 ⟨proof⟩

lemma *defval-conf[simp]*: $\text{is-type } P T \implies P, h \vdash \text{default-val } T : \leq T$
 ⟨proof⟩

lemma *typeof-notclass-heap*:
 $\forall C. T \neq \text{Class } C \implies (P \vdash \text{typeof}_h v = \text{Some } T) = (P \vdash \text{typeof}_{h'} v = \text{Some } T)$
 ⟨proof⟩

lemma *assumes* $h:h a = \text{Some}(C, S)$
shows *conf-upd-obj*: $(P, h(a \mapsto (C, S'))) \vdash v : \leq T) = (P, h \vdash v : \leq T)$
 ⟨proof⟩

lemma *conf-NT [iff]*: $P, h \vdash v : \leq NT = (v = \text{Null})$
 ⟨proof⟩

22.2 Value list conformance $[:\leq]$

lemma *confs-rev*: $P, h \vdash \text{rev } s [:\leq] t = (P, h \vdash s [:\leq] \text{rev } t)$
 ⟨proof⟩

lemma *confs-Cons2*: $P, h \vdash xs [:\leq] y \# ys = (\exists z zs. xs = z \# zs \wedge P, h \vdash z : \leq y \wedge P, h \vdash zs [:\leq] ys)$
 ⟨proof⟩

22.3 Field conformance $(:\leq)$

lemma *fconf-init-fields*:
 $\text{class } P C = \text{Some}(Bs, fs, ms) \implies P, h \vdash \text{init-class-fieldmap } P C (:\leq) \text{map-of } fs$
 ⟨proof⟩

22.4 Heap conformance

lemma *hconfD*: $\llbracket P \vdash h \checkmark; h a = \text{Some } \text{obj} \rrbracket \implies P, h \vdash \text{obj } \checkmark$
 ⟨proof⟩

lemma *hconf-Subobjs*:

$\llbracket h a = \text{Some}(C,S); (Cs, fs) \in S; P \vdash h \checkmark \rrbracket \implies \text{Subobjs } P \ C \ Cs$

$\langle \text{proof} \rangle$

22.5 Local variable conformance

lemma *lconf-upd*:

$\llbracket P, h \vdash l (\leq)_w E; P, h \vdash v \leq T; E \ V = \text{Some } T \rrbracket \implies P, h \vdash l(V \mapsto v) (\leq)_w E$

$\langle \text{proof} \rangle$

lemma *lconf-empty[iff]*: $P, h \vdash \text{empty} (\leq)_w E$

$\langle \text{proof} \rangle$

lemma *lconf-upd2*: $\llbracket P, h \vdash l (\leq)_w E; P, h \vdash v \leq T \rrbracket \implies P, h \vdash l(V \mapsto v) (\leq)_w E(V \mapsto T)$

$\langle \text{proof} \rangle$

22.6 Environment conformance

constdefs

$\text{envconf} :: \text{prog} \Rightarrow \text{env} \Rightarrow \text{bool} \ (- \vdash - \checkmark [51,51] 50)$

$P \vdash E \checkmark \equiv \forall V \ T. E \ V = \text{Some } T \longrightarrow \text{is-type } P \ T$

22.7 Type conformance

primrec

$\text{type-conf} :: \text{prog} \Rightarrow \text{env} \Rightarrow \text{heap} \Rightarrow \text{expr} \Rightarrow \text{ty} \Rightarrow \text{bool}$

$(-, -, - \vdash - :_{NT} - [51,51,51] 50)$

where

$\text{type-conf-Void}: P, E, h \vdash e :_{NT} \text{Void} \longleftrightarrow (P, E, h \vdash e : \text{Void})$
 $| \text{type-conf-Boolean}: P, E, h \vdash e :_{NT} \text{Boolean} \longleftrightarrow (P, E, h \vdash e : \text{Boolean})$
 $| \text{type-conf-Integer}: P, E, h \vdash e :_{NT} \text{Integer} \longleftrightarrow (P, E, h \vdash e : \text{Integer})$
 $| \text{type-conf-NT}: P, E, h \vdash e :_{NT} \text{NT} \longleftrightarrow (P, E, h \vdash e : \text{NT})$
 $| \text{type-conf-Class}: P, E, h \vdash e :_{NT} \text{Class } C \longleftrightarrow$
 $(P, E, h \vdash e : \text{Class } C \vee P, E, h \vdash e : \text{NT})$

fun

$\text{types-conf} :: \text{prog} \Rightarrow \text{env} \Rightarrow \text{heap} \Rightarrow \text{expr list} \Rightarrow \text{ty list} \Rightarrow \text{bool}$

$(-, -, - \vdash - [:]_{NT} - [51,51,51] 50)$

where

$P, E, h \vdash [] [:]_{NT} [] \longleftrightarrow \text{True}$
 $| P, E, h \vdash (e \# es) [:]_{NT} (T \# Ts) \longleftrightarrow$
 $(P, E, h \vdash e :_{NT} T \wedge P, E, h \vdash es [:]_{NT} Ts)$
 $| P, E, h \vdash es [:]_{NT} Ts \longleftrightarrow \text{False}$

lemma *wt-same-type-typeconf*:

$P, E, h \vdash e : T \implies P, E, h \vdash e :_{NT} T$
 $\langle proof \rangle$

lemma *wts-same-types-typesconf*:
 $P, E, h \vdash es \ [:] \ Ts \implies types-conf \ P \ E \ h \ es \ Ts$
 $\langle proof \rangle$

lemma *types-conf-smaller-types*:
 $\bigwedge es \ Ts. \ [length \ es = length \ Ts'; \ types-conf \ P \ E \ h \ es \ Ts'; \ P \vdash \ Ts' \ [\leq] \ Ts] \implies \exists \ Ts''. \ P, E, h \vdash es \ [:] \ Ts'' \wedge P \vdash Ts'' \ [\leq] \ Ts$

$\langle proof \rangle$

end

23 Progress of Small Step Semantics

theory *Progress* imports *Equivalence DefAss Conform* begin

23.1 Some pre-definitions

lemma *final-refE*:

$$\begin{aligned} & \llbracket P, E, h \vdash e : \text{Class } C; \text{final } e; \\ & \quad \bigwedge r. e = \text{ref } r \implies Q; \\ & \quad \bigwedge r. e = \text{Throw } r \implies Q \rrbracket \implies Q \\ \langle \text{proof} \rangle \end{aligned}$$

lemma *finalRefE*:

$$\begin{aligned} & \llbracket P, E, h \vdash e : T; \text{is-refT } T; \text{final } e; \\ & \quad e = \text{null} \implies Q; \\ & \quad \bigwedge r. e = \text{ref } r \implies Q; \\ & \quad \bigwedge r. e = \text{Throw } r \implies Q \rrbracket \implies Q \end{aligned}$$

$\langle \text{proof} \rangle$

lemma *subE*:

$$\begin{aligned} & \llbracket P \vdash T \leq T'; \text{is-type } P \ T'; \text{wf-prog wf-md } P; \\ & \quad \llbracket T = T'; \forall C. T \neq \text{Class } C \rrbracket \implies Q; \\ & \quad \bigwedge C \ D. \llbracket T = \text{Class } C; T' = \text{Class } D; P \vdash \text{Path } C \text{ to } D \text{ unique} \rrbracket \implies Q; \\ & \quad \bigwedge C. \llbracket T = \text{NT}; T' = \text{Class } C \rrbracket \implies Q \rrbracket \implies Q \end{aligned}$$

$\langle \text{proof} \rangle$

lemma *assumes wf:wf-prog wf-md P*

and *typeof: P ⊢ typeof_h v = Some T'*

and *type:is-type P T*

shows *sub-casts: P ⊢ T' ≤ T ⟹ ∃ v'. P ⊢ T casts v to v'*

$\langle \text{proof} \rangle$

Derivation of new induction scheme for well typing:

inductive

WTrt' :: $[prog, env, heap, expr, \quad ty \quad] \Rightarrow bool$
 $(-, -, - \vdash - : ' - [51, 51, 51] 50)$

and *WTrts'* :: $[prog, env, heap, expr \text{ list}, ty \text{ list}] \Rightarrow bool$
 $(-, -, - \vdash - [:'] - [51, 51, 51] 50)$

for *P* :: *prog*

where

is-class P C ⟹ $P, E, h \vdash \text{new } C : ' \text{Class } C$

| $\llbracket \text{is-class } P \ C; P, E, h \vdash e : ' T; \text{is-refT } T \rrbracket$

⟹ $P, E, h \vdash \text{Cast } C \ e : ' \text{Class } C$

| $\llbracket \text{is-class } P \ C; P, E, h \vdash e : ' T; \text{is-refT } T \rrbracket$

$$\begin{array}{l}
\implies P, E, h \vdash \langle C \rangle e : ' \text{Class } C \\
| P \vdash \text{typeof}_h v = \text{Some } T \implies P, E, h \vdash \text{Val } v : ' T \\
| E V = \text{Some } T \implies P, E, h \vdash \text{Var } V : ' T \\
| \llbracket P, E, h \vdash e_1 : ' T_1; P, E, h \vdash e_2 : ' T_2; \\
\quad \text{case bop of } Eq \Rightarrow T = \text{Boolean} \\
\quad | \text{Add} \Rightarrow T_1 = \text{Integer} \wedge T_2 = \text{Integer} \wedge T = \text{Integer} \rrbracket \\
\implies P, E, h \vdash e_1 \ll \text{bop} \gg e_2 : ' T \\
| \llbracket P, E, h \vdash \text{Var } V : ' T; P, E, h \vdash e : ' T' (* V \neq \text{This*}); P \vdash T' \leq T \rrbracket \\
\implies P, E, h \vdash V := e : ' T \\
| \llbracket P, E, h \vdash e : ' \text{Class } C; Cs \neq []; P \vdash C \text{ has least } F:T \text{ via } Cs \rrbracket \\
\implies P, E, h \vdash e \cdot F\{Cs\} : ' T \\
| P, E, h \vdash e : ' NT \implies P, E, h \vdash e \cdot F\{Cs\} : ' T \\
| \llbracket P, E, h \vdash e_1 : ' \text{Class } C; Cs \neq []; P \vdash C \text{ has least } F:T \text{ via } Cs; \\
P, E, h \vdash e_2 : ' T'; P \vdash T' \leq T \rrbracket \\
\implies P, E, h \vdash e_1 \cdot F\{Cs\} := e_2 : ' T \\
| \llbracket P, E, h \vdash e_1 : ' NT; P, E, h \vdash e_2 : ' T'; P \vdash T' \leq T \rrbracket \\
\implies P, E, h \vdash e_1 \cdot F\{Cs\} := e_2 : ' T \\
| \llbracket P, E, h \vdash e : ' \text{Class } C; P \vdash C \text{ has least } M = (Ts, T, m) \text{ via } Cs; \\
P, E, h \vdash es [:\uparrow] Ts'; P \vdash Ts' [\leq] Ts \rrbracket \\
\implies P, E, h \vdash e \cdot M(es) : ' T \\
| \llbracket P, E, h \vdash e : ' \text{Class } C'; P \vdash \text{Path } C' \text{ to } C \text{ unique}; \\
P \vdash C \text{ has least } M = (Ts, T, m) \text{ via } Cs; \\
P, E, h \vdash es [:\uparrow] Ts'; P \vdash Ts' [\leq] Ts \rrbracket \\
\implies P, E, h \vdash e \cdot (C::)M(es) : ' T \\
| \llbracket P, E, h \vdash e : ' NT; P, E, h \vdash es [:\uparrow] Ts \rrbracket \implies P, E, h \vdash \text{Call } e \text{ Copt } M \text{ es} : ' T \\
| \llbracket P \vdash \text{typeof}_h v = \text{Some } T'; P, E(V \mapsto T), h \vdash e_2 : ' T_2; P \vdash T' \leq T; \text{is-type } P \\
T \rrbracket \\
\implies P, E, h \vdash \{V:T := \text{Val } v; e_2\} : ' T_2 \\
| \llbracket P, E(V \mapsto T), h \vdash e : ' T'; \neg \text{assigned } V e; \text{is-type } P T \rrbracket \\
\implies P, E, h \vdash \{V:T; e\} : ' T' \\
| \llbracket P, E, h \vdash e_1 : ' T_1; P, E, h \vdash e_2 : ' T_2 \rrbracket \implies P, E, h \vdash e_1; e_2 : ' T_2 \\
| \llbracket P, E, h \vdash e : ' \text{Boolean}; P, E, h \vdash e_1 : ' T; P, E, h \vdash e_2 : ' T \rrbracket \\
\implies P, E, h \vdash \text{if } (e) e_1 \text{ else } e_2 : ' T \\
| \llbracket P, E, h \vdash e : ' \text{Boolean}; P, E, h \vdash c : ' T \rrbracket \\
\implies P, E, h \vdash \text{while}(e) c : ' \text{Void} \\
| \llbracket P, E, h \vdash e : ' T'; \text{is-refT } T \rrbracket \implies P, E, h \vdash \text{throw } e : ' T \\
\\
| P, E, h \vdash [] [:\uparrow] [] \\
| \llbracket P, E, h \vdash e : ' T; P, E, h \vdash es [:\uparrow] Ts \rrbracket \implies P, E, h \vdash e \# es [:\uparrow] T \# Ts
\end{array}$$

lemmas $WTrt'$ -induct = $WTrt'$ - $WTrts'$.induct [split-format (complete)]
and $WTrt'$ -inducts = $WTrt'$ - $WTrts'$.inducts [split-format (complete)]

inductive-cases $WTrt'$ -elim-cases[elim!]:
 $P, E, h \vdash V := e : ' T$

... and some easy consequences:

lemma [iff]: $P, E, h \vdash e_1;; e_2 : ' T_2 = (\exists T_1. P, E, h \vdash e_1 : ' T_1 \wedge P, E, h \vdash e_2 : ' T_2)$

$\langle proof \rangle$

lemma [iff]: $P, E, h \vdash Val\ v : ' T = (P \vdash typeof_h\ v = Some\ T)$

$\langle proof \rangle$

lemma [iff]: $P, E, h \vdash Var\ V : ' T = (E\ V = Some\ T)$

$\langle proof \rangle$

lemma $wt\text{-}wt'$: $P, E, h \vdash e : T \implies P, E, h \vdash e : ' T$
and $wts\text{-}wts'$: $P, E, h \vdash es\ [\cdot] Ts \implies P, E, h \vdash es\ [\cdot]' Ts$

$\langle proof \rangle$

lemma $wt'\text{-}wt$: $P, E, h \vdash e : ' T \implies P, E, h \vdash e : T$
and $wts'\text{-}wts$: $P, E, h \vdash es\ [\cdot]' Ts \implies P, E, h \vdash es\ [\cdot] Ts$

$\langle proof \rangle$

corollary $wt'\text{-}iff\text{-}wt$: $(P, E, h \vdash e : ' T) = (P, E, h \vdash e : T)$

$\langle proof \rangle$

corollary $wts'\text{-}iff\text{-}wts$: $(P, E, h \vdash es\ [\cdot]' Ts) = (P, E, h \vdash es\ [\cdot] Ts)$

$\langle proof \rangle$

lemmas $WTrt\text{-}inducts2 = WTrt'\text{-}inducts$ [unfolded $wt'\text{-}iff\text{-}wt$ $wts'\text{-}iff\text{-}wts$,
case-names $WTrtNew\ WTrtDynCast\ WTrtStaticCast\ WTrtVal\ WTrtVar\ WTrt\text{-}BinOp$
 $WTrtLAss\ WTrtFAcc\ WTrtFAccNT\ WTrtFAss\ WTrtFAssNT\ WTrtCall\ WTrt\text{-}StaticCall\ WTrtCallNT$
 $WTrtInitBlock\ WTrtBlock\ WTrtSeq\ WTrtCond\ WTrtWhile\ WTrtThrow$
 $WTrtNil\ WTrtCons$, consumes 1]

23.2 The theorem *progress*

lemma $mdc\text{-}leq\text{-}dyn\text{-}type$:

$P, E, h \vdash e : T \implies$

$\forall C\ a\ Cs\ D\ S. T = Class\ C \wedge e = ref(a, Cs) \wedge h\ a = Some(D, S) \implies P \vdash D$

$\preceq^* C$
and $P, E, h \vdash es \ [:] \ Ts \implies$
 $\forall T \ Ts' \ e \ es' \ C \ a \ Cs \ D \ S. \ Ts = T \# \ Ts' \wedge es = e \# \ es' \wedge$
 $T = \text{Class } C \wedge e = \text{ref}(a, Cs) \wedge h \ a = \text{Some}(D, S)$
 $\implies P \vdash D \preceq^* C$

$\langle \text{proof} \rangle$

lemma *appendPath-append-last*:
assumes *notempty*: $Ds \neq []$
shows $(Cs \ @_p \ Ds) \ @_p \ [\text{last } Ds] = (Cs \ @_p \ Ds)$

$\langle \text{proof} \rangle$

theorem **assumes** *wf*: *wuf-prog* P
shows *progress*: $P, E, h \vdash e : T \implies$
 $(\bigwedge l. \llbracket P \vdash h \ \checkmark; P \vdash E \ \checkmark; \mathcal{D} \ e \ [dom \ l]; \neg \text{final } e \rrbracket \implies \exists e' \ s'. P, E \vdash \langle e, (h, l) \rangle$
 $\rightarrow \langle e', s^\wedge \rangle)$
and $P, E, h \vdash es \ [:] \ Ts \implies$
 $(\bigwedge l. \llbracket P \vdash h \ \checkmark; P \vdash E \ \checkmark; \mathcal{D} \ es \ [dom \ l]; \neg \text{finals } es \rrbracket \implies \exists es' \ s'. P, E \vdash$
 $\langle es, (h, l) \rangle \ [\rightarrow] \langle es', s^\wedge \rangle)$

$\langle \text{proof} \rangle$

end

24 Heap Extension

theory *HeapExtension* **imports** *Progress* **begin**

24.1 The Heap Extension

constdefs

hext :: *heap* \Rightarrow *heap* \Rightarrow *bool* (- \leq - [51,51] 50)
 $h \leq h' \equiv \forall a C S. h a = \text{Some}(C,S) \longrightarrow (\exists S'. h' a = \text{Some}(C,S'))$

lemma *hextI*: $\forall a C S. h a = \text{Some}(C,S) \longrightarrow (\exists S'. h' a = \text{Some}(C,S')) \Longrightarrow h \leq h'$

<proof>

lemma *hext-objD*: $\llbracket h \leq h'; h a = \text{Some}(C,S) \rrbracket \Longrightarrow \exists S'. h' a = \text{Some}(C,S')$

<proof>

lemma *hext-refl [iff]*: $h \leq h$

<proof>

lemma *hext-new [simp]*: $h a = \text{None} \Longrightarrow h \leq h(a \mapsto x)$

<proof>

lemma *hext-trans*: $\llbracket h \leq h'; h' \leq h'' \rrbracket \Longrightarrow h \leq h''$

<proof>

lemma *hext-upd-obj*: $h a = \text{Some}(C,S) \Longrightarrow h \leq h(a \mapsto (C,S'))$

<proof>

24.2 \leq and preallocated

lemma *preallocated-hext*:

$\llbracket \text{preallocated } h; h \leq h' \rrbracket \Longrightarrow \text{preallocated } h'$
<proof>

lemmas *preallocated-upd-obj = preallocated-hext [OF - hext-upd-obj]*

lemmas *preallocated-new = preallocated-hext [OF - hext-new]*

24.3 \sqsubseteq in Small- and BigStep

lemma *red-hext-incr*: $P, E \vdash \langle e, (h, l) \rangle \rightarrow \langle e', (h', l') \rangle \implies h \sqsubseteq h'$
and *reds-hext-incr*: $P, E \vdash \langle es, (h, l) \rangle [\rightarrow] \langle es', (h', l') \rangle \implies h \sqsubseteq h'$

<proof>

lemma *step-hext-incr*: $P, E \vdash \langle e, s \rangle \rightarrow^* \langle e', s' \rangle \implies hp\ s \sqsubseteq hp\ s'$

<proof>

lemma *steps-hext-incr*: $P, E \vdash \langle es, s \rangle [\rightarrow]^* \langle es', s' \rangle \implies hp\ s \sqsubseteq hp\ s'$

<proof>

lemma *eval-hext*: $P, E \vdash \langle e, (h, l) \rangle \Rightarrow \langle e', (h', l') \rangle \implies h \sqsubseteq h'$
and *evals-hext*: $P, E \vdash \langle es, (h, l) \rangle [\Rightarrow] \langle es', (h', l') \rangle \implies h \sqsubseteq h'$

<proof>

24.4 \sqsubseteq and conformance

lemma *conf-hext*: $h \sqsubseteq h' \implies P, h \vdash v : \leq T \implies P, h' \vdash v : \leq T$
<proof>

lemma *confs-hext*: $P, h \vdash vs [:\leq] Ts \implies h \sqsubseteq h' \implies P, h' \vdash vs [:\leq] Ts$
<proof>

lemma *fconf-hext*: $\llbracket P, h \vdash fs (:\leq) E; h \sqsubseteq h' \rrbracket \implies P, h' \vdash fs (:\leq) E$

<proof>

lemmas *fconf-upd-obj* = *fconf-hext* [*OF* - *hext-upd-obj*]

lemmas *fconf-new* = *fconf-hext* [*OF* - *hext-new*]

lemma *oconf-hext*: $P, h \vdash obj\ \checkmark \implies h \sqsubseteq h' \implies P, h' \vdash obj\ \checkmark$

<proof>

lemmas *oconf-new* = *oconf-hext* [*OF* - *hext-new*]

lemmas *oconf-upd-obj* = *oconf-hext* [*OF* - *hext-upd-obj*]

lemma *hconf-new*: $\llbracket P \vdash h \checkmark; h a = \text{None}; P, h \vdash \text{obj } \checkmark \rrbracket \Longrightarrow P \vdash h(a \mapsto \text{obj}) \checkmark$
 $\langle \text{proof} \rangle$

lemma $\llbracket P \vdash h \checkmark; h' = h(a \mapsto (C, \text{Collect}(\text{init-obj } P C))); h a = \text{None}; \text{wf-prog wf-md } P \rrbracket$
 $\Longrightarrow P \vdash h' \checkmark$
 $\langle \text{proof} \rangle$ **thm** *fconf-new*
 $\langle \text{proof} \rangle$

lemma *hconf-upd-obj*:
 $\llbracket P \vdash h \checkmark; h a = \text{Some}(C, S); P, h \vdash (C, S') \checkmark \rrbracket \Longrightarrow P \vdash h(a \mapsto (C, S')) \checkmark$
 $\langle \text{proof} \rangle$

lemma *lconf-heat*: $\llbracket P, h \vdash l (: \leq)_w E; h \sqsubseteq h' \rrbracket \Longrightarrow P, h' \vdash l (: \leq)_w E$
 $\langle \text{proof} \rangle$

24.5 \sqsubseteq in the runtime type system

lemma *heat-typeof-mono*: $\llbracket h \sqsubseteq h'; P \vdash \text{typeof}_h v = \text{Some } T \rrbracket \Longrightarrow P \vdash \text{typeof}_{h'} v = \text{Some } T$

$\langle \text{proof} \rangle$

lemma *WTrt-heat-mono*: $P, E, h \vdash e : T \Longrightarrow (\bigwedge h'. h \sqsubseteq h' \Longrightarrow P, E, h' \vdash e : T)$
and *WTrts-heat-mono*: $P, E, h \vdash \text{es } [:] Ts \Longrightarrow (\bigwedge h'. h \sqsubseteq h' \Longrightarrow P, E, h' \vdash \text{es } [:] Ts)$

$\langle \text{proof} \rangle$

end

25 Well-formedness Constraints

theory *CWellForm* **imports** *WellForm WWellForm WellTypeRT DefAss* **begin**

constdefs

wf-C-mdecl :: *prog* \Rightarrow *cname* \Rightarrow *mdecl* \Rightarrow *bool*
wf-C-mdecl *P C* \equiv $\lambda(M, Ts, T, (pns, body)).$
 $length\ Ts = length\ pns \wedge$
 $distinct\ pns \wedge$
 $this \notin set\ pns \wedge$
 $P, [this \mapsto Class\ C, pns[\mapsto] Ts] \vdash body :: T \wedge$
 $\mathcal{D}\ body \subseteq \{this\} \cup set\ pns]$

lemma *wf-C-mdecl[simp]*:

wf-C-mdecl *P C* (*M, Ts, T, pns, body*) \equiv
 $(length\ Ts = length\ pns \wedge$
 $distinct\ pns \wedge$
 $this \notin set\ pns \wedge$
 $P, [this \mapsto Class\ C, pns[\mapsto] Ts] \vdash body :: T \wedge$
 $\mathcal{D}\ body \subseteq \{this\} \cup set\ pns])$
 $\langle proof \rangle$

abbreviation

wf-C-prog :: *prog* \Rightarrow *bool* **where**
wf-C-prog == *wf-prog wf-C-mdecl*

lemma *wf-C-prog-wf-C-mdecl*:

$\llbracket wf-C-prog\ P; (C, Bs, fs, ms) \in set\ P; m \in set\ ms \rrbracket$
 $\implies wf-C-mdecl\ P\ C\ m$

$\langle proof \rangle$

lemma *wf-mdecl-wwf-mdecl*: *wf-C-mdecl* *P C Md* $\implies wwf-mdecl\ P\ C\ Md$

$\langle proof \rangle$

lemma *wf-prog-wwf-prog*: *wf-C-prog* *P* $\implies wwf-prog\ P$

$\langle proof \rangle$

end

26 Type Safety Proof

theory *TypeSafe* **imports** *HeapExtension* *CWellForm* **begin**

26.1 Basic preservation lemmas

lemma **assumes** *wf:wvf-prog P* **and** *casts:P ⊢ T casts v to v'*
and *typeof:P ⊢ typeof_h v = Some T' and leq:P ⊢ T' ≤ T*
shows *casts-conf:P, h ⊢ v' :≤ T*

<proof>

theorem **assumes** *wf:wvf-prog P*

shows *red-preserves-hconf:*

$P, E ⊢ \langle e, (h, l) \rangle \rightarrow \langle e', (h', l') \rangle \implies (\bigwedge T. \llbracket P, E, h ⊢ e : T; P ⊢ h \checkmark \rrbracket \implies P ⊢ h' \checkmark)$

and *reds-preserves-hconf:*

$P, E ⊢ \langle es, (h, l) \rangle [\rightarrow] \langle es', (h', l') \rangle \implies (\bigwedge Ts. \llbracket P, E, h ⊢ es [:] Ts; P ⊢ h \checkmark \rrbracket \implies P ⊢ h' \checkmark)$

<proof>

theorem **assumes** *wf:wvf-prog P*

shows *red-preserves-lconf:*

$P, E ⊢ \langle e, (h, l) \rangle \rightarrow \langle e', (h', l') \rangle \implies (\bigwedge T. \llbracket P, E, h ⊢ e : T; P, h ⊢ l (:≤)_w E; P ⊢ E \checkmark \rrbracket \implies P, h' ⊢ l' (:≤)_w E)$

and *reds-preserves-lconf:*

$P, E ⊢ \langle es, (h, l) \rangle [\rightarrow] \langle es', (h', l') \rangle \implies (\bigwedge Ts. \llbracket P, E, h ⊢ es [:] Ts; P, h ⊢ l (:≤)_w E; P ⊢ E \checkmark \rrbracket \implies P, h' ⊢ l' (:≤)_w E)$

<proof>

Preservation of definite assignment more complex and requires a few lemmas first.

lemma *[iff]:* $\bigwedge A. \llbracket \text{length } Vs = \text{length } Ts; \text{length } vs = \text{length } Ts \rrbracket \implies \mathcal{D} (\text{blocks } (Vs, Ts, vs, e)) A = \mathcal{D} e (A \sqcup [\text{set } Vs])$

<proof>

lemma *red-lA-incr:* $P, E ⊢ \langle e, (h, l) \rangle \rightarrow \langle e', (h', l') \rangle \implies [\text{dom } l] \sqcup \mathcal{A} e \sqsubseteq [\text{dom } l'] \sqcup \mathcal{A} e'$

and *reds-lA-incr:* $P, E ⊢ \langle es, (h, l) \rangle [\rightarrow] \langle es', (h', l') \rangle \implies [\text{dom } l] \sqcup \mathcal{A} s es \sqsubseteq [\text{dom } l'] \sqcup \mathcal{A} s es'$

<proof>

Now preservation of definite assignment.

lemma assumes $wf: wf\text{-}C\text{-}prog\ P$

shows $red\text{-}preserves\text{-}defass:$

$P, E \vdash \langle e, (h, l) \rangle \rightarrow \langle e', (h', l') \rangle \implies \mathcal{D} e \ [dom\ l] \implies \mathcal{D} e' \ [dom\ l']$
and $P, E \vdash \langle es, (h, l) \rangle [\mapsto] \langle es', (h', l') \rangle \implies \mathcal{D}s\ es \ [dom\ l] \implies \mathcal{D}s\ es' \ [dom\ l']$

$\langle proof \rangle$

Combining conformance of heap and local variables:

constdefs

$sconf :: prog \Rightarrow env \Rightarrow state \Rightarrow bool \quad (-, - \vdash - \checkmark \quad [51, 51, 51] 50)$
 $P, E \vdash s \checkmark \equiv let\ (h, l) = s\ in\ P \vdash h \checkmark \wedge P, h \vdash l \ (:\leq)_w\ E \wedge P \vdash E \checkmark$

lemma $red\text{-}preserves\text{-}sconf:$

$\llbracket P, E \vdash \langle e, s \rangle \rightarrow \langle e', s' \rangle; P, E, hp\ s \vdash e : T; P, E \vdash s \checkmark; wwf\text{-}prog\ P \rrbracket$
 $\implies P, E \vdash s' \checkmark$

$\langle proof \rangle$

lemma $reds\text{-}preserves\text{-}sconf:$

$\llbracket P, E \vdash \langle es, s \rangle [\mapsto] \langle es', s' \rangle; P, E, hp\ s \vdash es \ [:\ Ts]; P, E \vdash s \checkmark; wwf\text{-}prog\ P \rrbracket$
 $\implies P, E \vdash s' \checkmark$

$\langle proof \rangle$

26.2 Subject reduction

lemma $wt\text{-}blocks:$

$\bigwedge E. \llbracket length\ Vs = length\ Ts; length\ vs = length\ Ts;$
 $\quad \forall T' \in set\ Ts. is\text{-}type\ P\ T' \rrbracket \implies$
 $(P, E, h \vdash blocks(Vs, Ts, vs, e) : T) =$
 $(P, E(Vs[\mapsto]Ts), h \vdash e : T \wedge$
 $(\exists Ts'. map\ (P \vdash typeof_h)\ vs = map\ Some\ Ts' \wedge P \vdash Ts' \ [:\leq] Ts))$

$\langle proof \rangle$

theorem assumes $wf: wf\text{-}C\text{-}prog\ P$

shows $subject\text{-}reduction2: P, E \vdash \langle e, (h, l) \rangle \rightarrow \langle e', (h', l') \rangle \implies$

$(\bigwedge T. \llbracket P, E \vdash (h, l) \checkmark; P, E, h \vdash e : T \rrbracket \implies P, E, h' \vdash e' :_{NT} T)$

and $subjects\text{-}reduction2: P, E \vdash \langle es, (h, l) \rangle [\mapsto] \langle es', (h', l') \rangle \implies$

$(\bigwedge Ts. \llbracket P, E \vdash (h, l) \checkmark; P, E, h \vdash es \ [:\ Ts] \rrbracket \implies types\text{-}conf\ P\ E\ h'\ es'\ Ts)$

$\langle proof \rangle$

corollary *subject-reduction*:

$\llbracket wf\text{-}C\text{-}prog\ P; P, E \vdash \langle e, s \rangle \rightarrow \langle e', s' \rangle; P, E \vdash s \checkmark; P, E, hp\ s \vdash e : T \rrbracket$
 $\implies P, E, (hp\ s') \vdash e' :_{NT} T$
 $\langle proof \rangle$

corollary *subjects-reduction*:

$\llbracket wf\text{-}C\text{-}prog\ P; P, E \vdash \langle es, s \rangle [\rightarrow] \langle es', s' \rangle; P, E \vdash s \checkmark; P, E, hp\ s \vdash es[:]\ Ts \rrbracket$
 $\implies types\text{-}conf\ P\ E\ (hp\ s')\ es'\ Ts$
 $\langle proof \rangle$

26.3 Lifting to \rightarrow^*

Now all these preservation lemmas are first lifted to the transitive closure

...

lemma *step-preserves-sconf*:

assumes $wf: wf\text{-}C\text{-}prog\ P$ **and** $step: P, E \vdash \langle e, s \rangle \rightarrow^* \langle e', s' \rangle$
shows $\bigwedge T. \llbracket P, E, hp\ s \vdash e : T; P, E \vdash s \checkmark \rrbracket \implies P, E \vdash s' \checkmark$

$\langle proof \rangle$

lemma *steps-preserves-sconf*:

assumes $wf: wf\text{-}C\text{-}prog\ P$ **and** $step: P, E \vdash \langle es, s \rangle [\rightarrow]^* \langle es', s' \rangle$
shows $\bigwedge Ts. \llbracket P, E, hp\ s \vdash es[:]\ Ts; P, E \vdash s \checkmark \rrbracket \implies P, E \vdash s' \checkmark$

$\langle proof \rangle$

lemma *step-preserves-defass*:

assumes $wf: wf\text{-}C\text{-}prog\ P$ **and** $step: P, E \vdash \langle e, s \rangle \rightarrow^* \langle e', s' \rangle$
shows $\mathcal{D}\ e\ [dom(lcl\ s)] \implies \mathcal{D}\ e'\ [dom(lcl\ s')]$

$\langle proof \rangle$

lemma *step-preserves-type*:

assumes $wf: wf\text{-}C\text{-}prog\ P$ **and** $step: P, E \vdash \langle e, s \rangle \rightarrow^* \langle e', s' \rangle$
shows $\bigwedge T. \llbracket P, E \vdash s \checkmark; P, E, hp\ s \vdash e : T \rrbracket$
 $\implies P, E, (hp\ s') \vdash e' :_{NT} T$

$\langle proof \rangle$

predicate to show the same lemma for lists

fun

$conformable :: ty\ list \Rightarrow ty\ list \Rightarrow bool$

where

$conformable\ []\ [] \longleftrightarrow True$

$| conformable\ (T''\#Ts'')\ (T'\#Ts') \longleftrightarrow (T'' = T')$

$\vee (\exists C. T'' = NT \wedge T' = \text{Class } C)) \wedge \text{conformable } Ts'' Ts'$
 $| \text{conformable} - - \longleftrightarrow \text{False}$

lemma *types-conf-conf-types-conf*:

$\llbracket \text{types-conf } P E h es Ts; \text{conformable } Ts Ts' \rrbracket \Longrightarrow \text{types-conf } P E h es Ts'$
 $\langle \text{proof} \rangle$

lemma *types-conf-Wtrt-conf*:

$\text{types-conf } P E h es Ts \Longrightarrow \exists Ts'. P, E, h \vdash es [:] Ts' \wedge \text{conformable } Ts' Ts$
 $\langle \text{proof} \rangle$

lemma *steps-preserves-types*:

assumes *wf*: *wf-C-prog* *P* **and** *steps*: $P, E \vdash \langle es, s \rangle [\rightarrow]^* \langle es', s' \rangle$

shows $\bigwedge Ts. \llbracket P, E \vdash s \checkmark; P, E, hp s \vdash es [:] Ts \rrbracket$

$\Longrightarrow \text{types-conf } P E (hp s') es' Ts$

$\langle \text{proof} \rangle$

26.4 Lifting to \Rightarrow

... and now to the big step semantics, just for fun.

lemma *eval-preserves-sconf*:

$\llbracket \text{wf-C-prog } P; P, E \vdash \langle e, s \rangle \Rightarrow \langle e', s' \rangle; P, E \vdash e :: T; P, E \vdash s \checkmark \rrbracket \Longrightarrow P, E \vdash s' \checkmark$

$\langle \text{proof} \rangle$

lemma *evals-preserves-sconf*:

$\llbracket \text{wf-C-prog } P; P, E \vdash \langle es, s \rangle [\Rightarrow] \langle es', s' \rangle; P, E \vdash es [::] Ts; P, E \vdash s \checkmark \rrbracket$

$\Longrightarrow P, E \vdash s' \checkmark$

$\langle \text{proof} \rangle$

lemma *eval-preserves-type*: **assumes** *wf*: *wf-C-prog* *P*

shows $\llbracket P, E \vdash \langle e, s \rangle \Rightarrow \langle e', s' \rangle; P, E \vdash s \checkmark; P, E \vdash e :: T \rrbracket$

$\Longrightarrow P, E, (hp s') \vdash e' :_{NT} T$

$\langle \text{proof} \rangle$

lemma *evals-preserves-types*: **assumes** *wf*: *wf-C-prog* *P*

shows $\llbracket P, E \vdash \langle es, s \rangle [\Rightarrow] \langle es', s' \rangle; P, E \vdash s \checkmark; P, E \vdash es [::] Ts \rrbracket$

$\Longrightarrow \text{types-conf } P E (hp s') es' Ts$

$\langle \text{proof} \rangle$

26.5 The final polish

The above preservation lemmas are now combined and packed nicely.

constdefs

$wf\text{-}config :: prog \Rightarrow env \Rightarrow state \Rightarrow expr \Rightarrow ty \Rightarrow bool \quad (-,-,- \vdash - : - \checkmark$
 $[51,0,0,0,0]50)$

$P,E,s \vdash e:T \checkmark \equiv P,E \vdash s \checkmark \wedge P,E,hp \ s \vdash e : T$

theorem *Subject-reduction: assumes* $wf: wf\text{-}C\text{-}prog \ P$

shows $P,E \vdash \langle e,s \rangle \rightarrow \langle e',s' \rangle \Longrightarrow P,E,s \vdash e : T \checkmark$

$\Longrightarrow P,E,(hp \ s') \vdash e' :_{NT} T$

$\langle proof \rangle$

theorem *Subject-reductions:*

assumes $wf: wf\text{-}C\text{-}prog \ P$ **and** $reds: P,E \vdash \langle e,s \rangle \rightarrow^* \langle e',s' \rangle$

shows $\bigwedge T. P,E,s \vdash e : T \checkmark \Longrightarrow P,E,(hp \ s') \vdash e' :_{NT} T$

$\langle proof \rangle$

corollary *Progress: assumes* $wf: wf\text{-}C\text{-}prog \ P$

shows $\llbracket P,E,s \vdash e : T \checkmark; \mathcal{D} \ e \ [dom(lcl \ s)]; \neg \ final \ e \rrbracket \Longrightarrow \exists e' \ s'. P,E \vdash \langle e,s \rangle$
 $\rightarrow \langle e',s' \rangle$

$\langle proof \rangle$

corollary *TypeSafety:*

fixes $s \ s' :: state$

assumes $wf: wf\text{-}C\text{-}prog \ P$ **and** $sconf: P,E \vdash s \checkmark$ **and** $wte: P,E \vdash e :: T$

and $D: \mathcal{D} \ e \ [dom(lcl \ s)]$ **and** $step: P,E \vdash \langle e,s \rangle \rightarrow^* \langle e',s' \rangle$

and $nored: \neg(\exists e'' \ s''. P,E \vdash \langle e',s' \rangle \rightarrow \langle e'',s'' \rangle)$

shows $(\exists v. e' = Val \ v \wedge P, hp \ s' \vdash v : \leq T) \vee$

$(\exists r. e' = Throw \ r \wedge the\text{-}addr \ (Ref \ r) \in dom(hp \ s'))$

$\langle proof \rangle$

end

27 Determinism Proof

theory *Determinism* **imports** *TypeSafe* **begin**

27.1 Some lemmas

lemma *maps-nth*:

$\llbracket (E(xs \mapsto ys)) \ x = \text{Some } y; \text{ length } xs = \text{length } ys; \text{ distinct } xs \rrbracket$
 $\implies \forall i. x = xs!i \wedge i < \text{length } xs \longrightarrow y = ys!i$
 <proof>

lemma *nth-maps*: $\llbracket \text{length } pns = \text{length } Ts; \text{ distinct } pns; i < \text{length } Ts \rrbracket$

$\implies (E(pns \mapsto Ts)) (pns!i) = \text{Some } (Ts!i)$
 <proof>

lemma *casts-casts-eq-result*:

fixes $s :: \text{state}$
assumes $\text{casts}: P \vdash T \text{ casts } v \text{ to } v'$ **and** $\text{casts}': P \vdash T \text{ casts } v \text{ to } w'$
and $\text{type}: \text{is-type } P \ T$ **and** $\text{wte}: P, E \vdash e :: T'$ **and** $\text{leq}: P \vdash T' \leq T$
and $\text{eval}: P, E \vdash \langle e, s \rangle \Rightarrow \langle \text{Val } v, (h, l) \rangle$ **and** $\text{sconf}: P, E \vdash s \checkmark$
and $\text{wf}: \text{wf-C-prog } P$
shows $v' = w'$
 <proof>

lemma *Casts-Casts-eq-result*:

assumes $\text{wf}: \text{wf-C-prog } P$
shows $\llbracket P \vdash Ts \text{ Casts } vs \text{ to } vs'; P \vdash Ts \text{ Casts } vs \text{ to } ws'; \forall T \in \text{set } Ts. \text{is-type } P \ T; \llbracket P, E \vdash es \llbracket :: Ts'; P \vdash Ts' \llbracket \leq Ts; P, E \vdash \langle es, s \rangle \llbracket \Rightarrow \langle \text{map Val } vs, (h, l) \rangle; P, E \vdash s \checkmark \rrbracket \implies vs' = ws' \rrbracket$
 <proof>

lemma *Casts-conf*: **assumes** $\text{wf}: \text{wf-C-prog } P$

shows $P \vdash Ts \text{ Casts } vs \text{ to } vs' \implies$
 $(\bigwedge es \ s \ Ts'. \llbracket P, E \vdash es \llbracket :: Ts'; P, E \vdash \langle es, s \rangle \llbracket \Rightarrow \langle \text{map Val } vs, (h, l) \rangle; P, E \vdash s \checkmark; P \vdash Ts' \llbracket \leq Ts \rrbracket \implies \forall i < \text{length } Ts. P, h \vdash vs!i \llbracket \leq Ts!i \rrbracket)$
 <proof>

lemma *map-Val-throw-False*: $\text{map Val } vs = \text{map Val } ws \ @ \ \text{throw } ex \ \# \ es \implies \text{False}$

<proof>

lemma *map-Val-throw-eq*: $\text{map Val } vs \ @ \ \text{throw } ex \ \# \ es = \text{map Val } ws \ @ \ \text{throw } ex' \ \# \ es'$

$\implies vs = ws \wedge ex = ex' \wedge es = es'$

<proof>

27.2 The proof

lemma *deterministic-big-step*:

assumes *wf:wf-C-prog P*

shows $P, E \vdash \langle e, s \rangle \Rightarrow \langle e_1, s_1 \rangle \Longrightarrow$

$(\bigwedge e_2 s_2 T. \llbracket P, E \vdash \langle e, s \rangle \Rightarrow \langle e_2, s_2 \rangle; P, E \vdash e :: T; P, E \vdash s \sqrt{} \rrbracket$
 $\Longrightarrow e_1 = e_2 \wedge s_1 = s_2)$

and $P, E \vdash \langle es, s \rangle [\Rightarrow] \langle es_1, s_1 \rangle \Longrightarrow$

$(\bigwedge es_2 s_2 Ts. \llbracket P, E \vdash \langle es, s \rangle [\Rightarrow] \langle es_2, s_2 \rangle; P, E \vdash es [::] Ts; P, E \vdash s \sqrt{} \rrbracket$
 $\Longrightarrow es_1 = es_2 \wedge s_1 = s_2)$

<proof>

end

28 Program annotation

theory *Annotate* **imports** *WellType* **begin**

abbreviation (output)

$unanFAcc :: expr \Rightarrow vname \Rightarrow expr \Rightarrow expr \ ((\dots) [10,10] 90)$ **where**
 $unanFAcc\ e\ F == FAcc\ e\ F \ []$

abbreviation (output)

$unanFAss :: expr \Rightarrow vname \Rightarrow expr \Rightarrow expr \Rightarrow expr \ ((\dots := -) [10,0,90] 90)$ **where**
 $unanFAss\ e\ F\ e' == FAss\ e\ F \ []\ e'$

inductive

$Anno :: [prog, env, expr \quad , expr] \Rightarrow bool$
 $(-, - \vdash - \rightsquigarrow - [51,0,0,51]50)$

and $Annos :: [prog, env, expr\ list, expr\ list] \Rightarrow bool$
 $(-, - \vdash - [\rightsquigarrow] - [51,0,0,51]50)$

for $P :: prog$

where

$AnnoNew: is-class\ P\ C \Longrightarrow P, E \vdash new\ C \rightsquigarrow new\ C$
 $| AnnoCast: P, E \vdash e \rightsquigarrow e' \Longrightarrow P, E \vdash Cast\ C\ e \rightsquigarrow Cast\ C\ e'$
 $| AnnoStatCast: P, E \vdash e \rightsquigarrow e' \Longrightarrow P, E \vdash StatCast\ C\ e \rightsquigarrow StatCast\ C\ e'$
 $| AnnoVal: P, E \vdash Val\ v \rightsquigarrow Val\ v$
 $| AnnoVarVar: E\ V = [T] \Longrightarrow P, E \vdash Var\ V \rightsquigarrow Var\ V$
 $| AnnoVarField: \llbracket E\ V = None; E\ this = [Class\ C]; P \vdash C\ has\ least\ V:T\ via\ Cs$
 \rrbracket
 $\Longrightarrow P, E \vdash Var\ V \rightsquigarrow Var\ this \cdot V \{Cs\}$
 $| AnnoBinOp:$
 $\llbracket P, E \vdash e1 \rightsquigarrow e1'; P, E \vdash e2 \rightsquigarrow e2' \rrbracket$
 $\Longrightarrow P, E \vdash e1 \ll bop \gg e2 \rightsquigarrow e1' \ll bop \gg e2'$
 $| AnnoLAss:$
 $P, E \vdash e \rightsquigarrow e' \Longrightarrow P, E \vdash V := e \rightsquigarrow V := e'$
 $| AnnoFAcc:$
 $\llbracket P, E \vdash e \rightsquigarrow e'; P, E \vdash e' :: Class\ C; P \vdash C\ has\ least\ F:T\ via\ Cs \rrbracket$
 $\Longrightarrow P, E \vdash e \cdot F \{ \} \rightsquigarrow e' \cdot F \{ Cs \}$
 $| AnnoFAss: \llbracket P, E \vdash e1 \rightsquigarrow e1'; P, E \vdash e2 \rightsquigarrow e2';$
 $P, E \vdash e1' :: Class\ C; P \vdash C\ has\ least\ F:T\ via\ Cs \rrbracket$
 $\Longrightarrow P, E \vdash e1 \cdot F \{ \} := e2 \rightsquigarrow e1' \cdot F \{ Cs \} := e2'$
 $| AnnoCall:$
 $\llbracket P, E \vdash e \rightsquigarrow e'; P, E \vdash es [\rightsquigarrow] es' \rrbracket$
 $\Longrightarrow P, E \vdash Call\ e\ Copt\ M\ es \rightsquigarrow Call\ e'\ Copt\ M\ es'$
 $| AnnoBlock:$
 $P, E(V \mapsto T) \vdash e \rightsquigarrow e' \Longrightarrow P, E \vdash \{V:T; e\} \rightsquigarrow \{V:T; e'\}$
 $| AnnoComp: \llbracket P, E \vdash e1 \rightsquigarrow e1'; P, E \vdash e2 \rightsquigarrow e2' \rrbracket$
 $\Longrightarrow P, E \vdash e1;;e2 \rightsquigarrow e1';;e2'$
 $| AnnoCond: \llbracket P, E \vdash e \rightsquigarrow e'; P, E \vdash e1 \rightsquigarrow e1'; P, E \vdash e2 \rightsquigarrow e2' \rrbracket$

$\implies P, E \vdash \text{if } (e) \ e1 \ \text{else } e2 \rightsquigarrow \text{if } (e') \ e1' \ \text{else } e2'$
| AnnoLoop: $\llbracket P, E \vdash e \rightsquigarrow e'; P, E \vdash c \rightsquigarrow c' \rrbracket$
 $\implies P, E \vdash \text{while } (e) \ c \rightsquigarrow \text{while } (e') \ c'$
| AnnoThrow: $P, E \vdash e \rightsquigarrow e' \implies P, E \vdash \text{throw } e \rightsquigarrow \text{throw } e'$

| AnnoNil: $P, E \vdash [] \rightsquigarrow []$
| AnnoCons: $\llbracket P, E \vdash e \rightsquigarrow e'; P, E \vdash es \rightsquigarrow es' \rrbracket$
 $\implies P, E \vdash e \# es \rightsquigarrow e' \# es'$

end

Relating (finite) sets and lists theory List-Set

imports Main

begin

28.1 Various additional list functions

definition *insert* :: 'a \Rightarrow 'a list \Rightarrow 'a list **where**
insert x xs = (if x \in set xs then xs else x # xs)

definition *remove-all* :: 'a \Rightarrow 'a list \Rightarrow 'a list **where**
remove-all x xs = filter (Not o op = x) xs

28.2 Various additional set functions

definition *is-empty* :: 'a set \Rightarrow bool **where**
is-empty A \longleftrightarrow A = {}

definition *remove* :: 'a \Rightarrow 'a set \Rightarrow 'a set **where**
remove x A = A - {x}

lemma *fun-left-comm-idem-remove*:
fun-left-comm-idem remove
<proof>

lemma *minus-fold-remove*:
assumes finite A
shows B - A = fold remove B A
<proof>

definition *project* :: ('a \Rightarrow bool) \Rightarrow 'a set \Rightarrow 'a set **where**
project P A = {a \in A. P a}

28.3 Basic set operations

lemma *is-empty-set*:
is-empty (set xs) \longleftrightarrow null xs
<proof>

lemma *ball-set*:

$(\forall x \in \text{set } xs. P x) \longleftrightarrow \text{list-all } P \text{ } xs$
 $\langle \text{proof} \rangle$

lemma *bex-set*:
 $(\exists x \in \text{set } xs. P x) \longleftrightarrow \text{list-ex } P \text{ } xs$
 $\langle \text{proof} \rangle$

lemma *empty-set*:
 $\{\} = \text{set } []$
 $\langle \text{proof} \rangle$

lemma *insert-set*:
 $\text{Set.insert } x (\text{set } xs) = \text{set } (\text{insert } x \text{ } xs)$
 $\langle \text{proof} \rangle$

lemma *insert-set-compl*:
 $\text{Set.insert } x (\text{-- set } xs) = \text{-- set } (\text{List-Set.remove-all } x \text{ } xs)$
 $\langle \text{proof} \rangle$

lemma *remove-set*:
 $\text{remove } x (\text{set } xs) = \text{set } (\text{remove-all } x \text{ } xs)$
 $\langle \text{proof} \rangle$

lemma *remove-set-compl*:
 $\text{List-Set.remove } x (\text{-- set } xs) = \text{-- set } (\text{List-Set.insert } x \text{ } xs)$
 $\langle \text{proof} \rangle$

lemma *image-set*:
 $\text{image } f (\text{set } xs) = \text{set } (\text{map } f \text{ } xs)$
 $\langle \text{proof} \rangle$

lemma *project-set*:
 $\text{project } P (\text{set } xs) = \text{set } (\text{filter } P \text{ } xs)$
 $\langle \text{proof} \rangle$

28.4 Functorial set operations

lemma *union-set*:
 $\text{set } xs \cup A = \text{foldl } (\lambda A x. \text{Set.insert } x \text{ } A) A \text{ } xs$
 $\langle \text{proof} \rangle$

lemma *minus-set*:
 $A - \text{set } xs = \text{foldl } (\lambda A x. \text{remove } x \text{ } A) A \text{ } xs$
 $\langle \text{proof} \rangle$

lemma *Inter-set*:
 $\text{Inter } (\text{set } As) = \text{foldl } (\text{op } \cap) \text{ } UNIV \text{ } As$
 $\langle \text{proof} \rangle$

lemma *Union-set*:

$Union (set As) = foldl (op \cup) \{\} As$
<proof>

lemma *INTER-set*:

$INTER (set As) f = foldl (\lambda B A. f A \cap B) UNIV As$
<proof>

lemma *UNION-set*:

$UNION (set As) f = foldl (\lambda B A. f A \cup B) \{\} As$
<proof>

28.5 Derived set operations

lemma *member*:

$a \in A \longleftrightarrow (\exists x \in A. a = x)$
<proof>

lemma *subset-eq*:

$A \subseteq B \longleftrightarrow (\forall x \in A. x \in B)$
<proof>

lemma *subset*:

$A \subset B \longleftrightarrow A \subseteq B \wedge \neg B \subseteq A$
<proof>

lemma *set-eq*:

$A = B \longleftrightarrow A \subseteq B \wedge B \subseteq A$
<proof>

lemma *inter*:

$A \cap B = project (\lambda x. x \in A) B$
<proof>

hide (open) *const insert*

end

Executable finite sets **theory** *Fset*

imports *List-Set*

begin

declare *mem-def* [*simp*]

28.6 Lifting

datatype *'a fset* = *Fset 'a set*

primrec *member* :: *'a fset* \Rightarrow *'a set* **where**

member (*Fset* *A*) = *A*

lemma *Fset-member* [*simp*]:

$Fset (member A) = A$

$\langle proof \rangle$

definition *Set* :: 'a list \Rightarrow 'a fset **where**

$Set\ xs = Fset (set\ xs)$

lemma *member-Set* [*simp*]:

$member (Set\ xs) = set\ xs$

$\langle proof \rangle$

definition *Coset* :: 'a list \Rightarrow 'a fset **where**

$Coset\ xs = Fset (-\ set\ xs)$

lemma *member-Coset* [*simp*]:

$member (Coset\ xs) = -\ set\ xs$

$\langle proof \rangle$

code-datatype *Set Coset*

lemma *member-code* [*code*]:

$member (Set\ xs)\ y \longleftrightarrow List.member\ y\ xs$

$member (Coset\ xs)\ y \longleftrightarrow \neg List.member\ y\ xs$

$\langle proof \rangle$

lemma *member-image-UNIV* [*simp*]:

$member\ 'UNIV = UNIV$

$\langle proof \rangle$

28.7 Basic operations

definition *is-empty* :: 'a fset \Rightarrow bool **where**

[*simp*]: $is_empty\ A \longleftrightarrow List-Set.is_empty (member\ A)$

lemma *is-empty-Set* [*code*]:

$is_empty (Set\ xs) \longleftrightarrow null\ xs$

$\langle proof \rangle$

definition *empty* :: 'a fset **where**

[*simp*]: $empty = Fset\ \{\}$

lemma *empty-Set* [*code*]:

$empty = Set\ []$

$\langle proof \rangle$

definition *insert* :: 'a \Rightarrow 'a fset \Rightarrow 'a fset **where**

[*simp*]: $insert\ x\ A = Fset (Set.insert\ x (member\ A))$

lemma *insert-Set* [code]:

$insert\ x\ (Set\ xs) = Set\ (List-Set.insert\ x\ xs)$
 $insert\ x\ (Coset\ xs) = Coset\ (remove-all\ x\ xs)$
(proof)

definition *remove* :: 'a \Rightarrow 'a fset \Rightarrow 'a fset **where**

[simp]: $remove\ x\ A = Fset\ (List-Set.remove\ x\ (member\ A))$

lemma *remove-Set* [code]:

$remove\ x\ (Set\ xs) = Set\ (remove-all\ x\ xs)$
 $remove\ x\ (Coset\ xs) = Coset\ (List-Set.insert\ x\ xs)$
(proof)

definition *map* :: ('a \Rightarrow 'b) \Rightarrow 'a fset \Rightarrow 'b fset **where**

[simp]: $map\ f\ A = Fset\ (image\ f\ (member\ A))$

lemma *map-Set* [code]:

$map\ f\ (Set\ xs) = Set\ (remdups\ (List.map\ f\ xs))$
(proof)

definition *filter* :: ('a \Rightarrow bool) \Rightarrow 'a fset \Rightarrow 'a fset **where**

[simp]: $filter\ P\ A = Fset\ (List-Set.project\ P\ (member\ A))$

lemma *filter-Set* [code]:

$filter\ P\ (Set\ xs) = Set\ (List.filter\ P\ xs)$
(proof)

definition *forall* :: ('a \Rightarrow bool) \Rightarrow 'a fset \Rightarrow bool **where**

[simp]: $forall\ P\ A \longleftrightarrow Ball\ (member\ A)\ P$

lemma *forall-Set* [code]:

$forall\ P\ (Set\ xs) \longleftrightarrow list-all\ P\ xs$
(proof)

definition *exists* :: ('a \Rightarrow bool) \Rightarrow 'a fset \Rightarrow bool **where**

[simp]: $exists\ P\ A \longleftrightarrow Bex\ (member\ A)\ P$

lemma *exists-Set* [code]:

$exists\ P\ (Set\ xs) \longleftrightarrow list-ex\ P\ xs$
(proof)

definition *card* :: 'a fset \Rightarrow nat **where**

[simp]: $card\ A = Finite-Set.card\ (member\ A)$

lemma *card-Set* [code]:

$card\ (Set\ xs) = length\ (remdups\ xs)$
(proof)

28.8 Derived operations

definition *subfset-eq* :: 'a fset \Rightarrow 'a fset \Rightarrow bool **where**
[simp]: *subfset-eq* A B \longleftrightarrow member A \subseteq member B

lemma *subfset-eq-forall* [code]:
subfset-eq A B \longleftrightarrow forall (member B) A
(proof)

definition *subfset* :: 'a fset \Rightarrow 'a fset \Rightarrow bool **where**
[simp]: *subfset* A B \longleftrightarrow member A \subset member B

lemma *subfset-subfset-eq* [code]:
subfset A B \longleftrightarrow *subfset-eq* A B \wedge \neg *subfset-eq* B A
(proof)

lemma *eq-fset-subfset-eq* [code]:
eq-class.eq A B \longleftrightarrow *subfset-eq* A B \wedge *subfset-eq* B A
(proof)

28.9 Functorial operations

definition *inter* :: 'a fset \Rightarrow 'a fset \Rightarrow 'a fset **where**
[simp]: *inter* A B = Fset (member A \cap member B)

lemma *inter-project* [code]:
inter A (Set xs) = Set (List.filter (member A) xs)
inter A (Coset xs) = foldl (λ A x. remove x A) A xs
(proof)

definition *subtract* :: 'a fset \Rightarrow 'a fset \Rightarrow 'a fset **where**
[simp]: *subtract* A B = Fset (member B $-$ member A)

lemma *subtract-remove* [code]:
subtract (Set xs) A = foldl (λ A x. remove x A) A xs
subtract (Coset xs) A = Set (List.filter (member A) xs)
(proof)

definition *union* :: 'a fset \Rightarrow 'a fset \Rightarrow 'a fset **where**
[simp]: *union* A B = Fset (member A \cup member B)

lemma *union-insert* [code]:
union (Set xs) A = foldl (λ A x. insert x A) A xs
union (Coset xs) A = Coset (List.filter (Not \circ member A) xs)
(proof)

definition *Inter* :: 'a fset fset \Rightarrow 'a fset **where**
[simp]: *Inter* A = Fset (Complete-Lattice.Inter (member \circ member A))

lemma *Inter-inter* [code]:

$Inter (Set As) = foldl inter (Coset []) As$
 $Inter (Coset []) = empty$
 <proof>

definition *Union* :: 'a fset fset \Rightarrow 'a fset **where**
 [simp]: $Union A = Fset (Complete-Lattice.Union (member ' member A))$

lemma *Union-union* [code]:
 $Union (Set As) = foldl union empty As$
 $Union (Coset []) = Coset []$
 <proof>

28.10 Misc operations

lemma *size-fset* [code]:
 $fset-size f A = 0$
 $size A = 0$
 <proof>

lemma *fset-case-code* [code]:
 $fset-case f A = f (member A)$
 <proof>

lemma *fset-rec-code* [code]:
 $fset-rec f A = f (member A)$
 <proof>

28.11 Simplified simprules

lemma *is-empty-simp* [simp]:
 $is-empty A \longleftrightarrow member A = \{\}$
 <proof>

declare *is-empty-def* [simp del]

lemma *remove-simp* [simp]:
 $remove x A = Fset (member A - \{x\})$
 <proof>

declare *remove-def* [simp del]

lemma *filter-simp* [simp]:
 $filter P A = Fset \{x \in member A. P x\}$
 <proof>

declare *filter-def* [simp del]

declare *mem-def* [simp del]

hide (open) *const is-empty empty insert remove map filter forall exists card
 subset-eq subset inter union subtract Inter Union*

end

Install quickcheck of SML code generator **theory** *SML-Quickcheck*
imports *Main*
begin

<ML>

end

Implementation of finite sets by lists **theory** *Executable-Set*
imports *Main Fset SML-Quickcheck*
begin

28.12 Preprocessor setup

declare *member* [*code*]

definition *empty* :: 'a set **where**
empty = {}

declare *empty-def* [*symmetric, code-unfold*]

definition *inter* :: 'a set \Rightarrow 'a set \Rightarrow 'a set **where**
inter = *op* \cap

declare *inter-def* [*symmetric, code-unfold*]

definition *union* :: 'a set \Rightarrow 'a set \Rightarrow 'a set **where**
union = *op* \cup

declare *union-def* [*symmetric, code-unfold*]

definition *subset* :: 'a set \Rightarrow 'a set \Rightarrow bool **where**
subset = *op* \leq

declare *subset-def* [*symmetric, code-unfold*]

lemma [*code*]:
subset *A B* \longleftrightarrow ($\forall x \in A. x \in B$)
<proof>

definition *eq-set* :: 'a set \Rightarrow 'a set \Rightarrow bool **where**
[*code del*]: *eq-set* = *op* =

lemma [*code*]:
eq-set *A B* \longleftrightarrow $A \subseteq B \wedge B \subseteq A$

```

    <proof>

declare inter [code]

declare List-Set.project-def [symmetric, code-unfold]

definition Inter :: 'a set set  $\Rightarrow$  'a set where
    Inter = Complete-Lattice.Inter

declare Inter-def [symmetric, code-unfold]

definition Union :: 'a set set  $\Rightarrow$  'a set where
    Union = Complete-Lattice.Union

declare Union-def [symmetric, code-unfold]

```

28.13 Code generator setup

<ML>

```

definition flip :: ('a  $\Rightarrow$  'b  $\Rightarrow$  'c)  $\Rightarrow$  'b  $\Rightarrow$  'a  $\Rightarrow$  'c where
    flip f a b = f b a

```

types-code

```

    fset ((-/ <module>fset))
attach <<
    datatype 'a fset = Set of 'a list | Coset of 'a list;
    >>

```

consts-code

```

    Set (<module>Set)
    Coset (<module>Coset)

```

consts-code

```

    empty          ( {*Fset.empty*} )
    List-Set.is-empty ( {*Fset.is-empty*} )
    Set.insert      ( {*Fset.insert*} )
    List-Set.remove ( {*Fset.remove*} )
    Set.image       ( {*Fset.map*} )
    List-Set.project ( {*Fset.filter*} )
    Ball           ( {*flip Fset.forall*} )
    Bex           ( {*flip Fset.exists*} )
    union          ( {*Fset.union*} )
    inter          ( {*Fset.inter*} )
    op - :: 'a set  $\Rightarrow$  'a set  $\Rightarrow$  'a set ( {*flip Fset.subtract*} )
    Union         ( {*Fset.Union*} )
    Inter         ( {*Fset.Inter*} )
    card          ( {*Fset.card*} )
    fold          ( {*foldl o flip*} )

```

hide (open) *const empty inter union subset eq-set Inter Union flip*
end

29 Code generation for Semantics and Type System

```

theory Execute
imports BigStep WellType Executable-Set Efficient-Nat
begin

```

29.1 General redefinitions

```

inductive app :: 'a list  $\Rightarrow$  'a list  $\Rightarrow$  'a list  $\Rightarrow$  bool
where
  app [] ys ys
| app xs ys zs  $\Longrightarrow$  app (x # xs) ys (x # zs)

```

```

theorem app-eq1:  $\bigwedge$  ys zs. zs = xs @ ys  $\Longrightarrow$  app xs ys zs
  <proof>

```

```

theorem app-eq2: app xs ys zs  $\Longrightarrow$  zs = xs @ ys
  <proof>

```

```

theorem app-eq: app xs ys zs = (zs = xs @ ys)
  <proof>

```

```

inductive
  casts-aux :: prog  $\Rightarrow$  ty  $\Rightarrow$  val  $\Rightarrow$  val  $\Rightarrow$  bool
  for P :: prog
where
  (case T of Class C  $\Rightarrow$  False | -  $\Rightarrow$  True)  $\Longrightarrow$  casts-aux P T v v
| casts-aux P (Class C) Null Null
| [[Subobjs P (last Cs) Cs'; last Cs' = C;
   last Cs = hd Cs'; Cs @ tl Cs' = Ds]]
   $\Longrightarrow$  casts-aux P (Class C) (Ref(a,Cs)) (Ref(a,Ds))
| [[Subobjs P (last Cs) Cs'; last Cs' = C; last Cs  $\neq$  hd Cs']]
   $\Longrightarrow$  casts-aux P (Class C) (Ref(a,Cs)) (Ref(a,Cs'))

```

```

lemma casts-aux-eq:
  (P  $\vdash$  T casts v to v') = casts-aux P T v v'
  <proof>

```

```

inductive
  Casts-aux :: prog  $\Rightarrow$  ty list  $\Rightarrow$  val list  $\Rightarrow$  val list  $\Rightarrow$  bool
  for P :: prog
where
  Casts-aux P [] [] []
| [[casts-aux P T v v'; Casts-aux P Ts vs vs']]
   $\Longrightarrow$  Casts-aux P (T # Ts) (v # vs) (v' # vs')

```

```

lemma Casts-aux-eq:

```

$(P \vdash Ts \text{ Casts } vs \text{ to } vs') = \text{Casts-aux } P \ Ts \ vs \ vs'$
 $\langle \text{proof} \rangle$

Redefine map Val vs

inductive $\text{map-val} :: \text{expr list} \Rightarrow \text{val list} \Rightarrow \text{bool}$

where

$\text{Nil}: \text{map-val} [] []$

$| \text{Cons}: \text{map-val } xs \ ys \Longrightarrow \text{map-val } (\text{Val } y \ \# \ xs) \ (y \ \# \ ys)$

inductive $\text{map-val2} :: \text{expr list} \Rightarrow \text{val list} \Rightarrow \text{expr list} \Rightarrow \text{bool}$

where

$\text{Nil}: \text{map-val2} [] [] []$

$| \text{Cons}: \text{map-val2 } xs \ ys \ zs \Longrightarrow \text{map-val2 } (\text{Val } y \ \# \ xs) \ (y \ \# \ ys) \ zs$

$| \text{Throw}: \text{map-val2 } (\text{throw } e \ \# \ xs) [] (\text{throw } e \ \# \ xs)$

theorem $\text{map-val-conv}: (xs = \text{map Val } ys) = \text{map-val } xs \ ys \langle \text{proof} \rangle$

theorem $\text{map-val2-conv}:$

$(xs = \text{map Val } ys \ @ \ \text{throw } e \ \# \ zs) = \text{map-val2 } xs \ ys \ (\text{throw } e \ \# \ zs) \langle \text{proof} \rangle$

Redefine MethodDefs and FieldDecls

constdefs

$\text{MethodDefs}' :: \text{prog} \Rightarrow \text{cname} \Rightarrow \text{mname} \Rightarrow \text{path} \Rightarrow \text{method} \Rightarrow \text{bool}$

$\text{MethodDefs}' \ P \ C \ M \ Cs \ \text{mthd} \equiv (Cs, \text{mthd}) \in \text{MethodDefs } P \ C \ M$

$\text{FieldDecls}' :: \text{prog} \Rightarrow \text{cname} \Rightarrow \text{vname} \Rightarrow \text{path} \Rightarrow \text{ty} \Rightarrow \text{bool}$

$\text{FieldDecls}' \ P \ C \ F \ Cs \ T \equiv (Cs, T) \in \text{FieldDecls } P \ C \ F$

$\text{MinimalMethodDefs}' :: \text{prog} \Rightarrow \text{cname} \Rightarrow \text{mname} \Rightarrow \text{path} \Rightarrow \text{method} \Rightarrow \text{bool}$

$\text{MinimalMethodDefs}' \ P \ C \ M \ Cs \ \text{mthd} \equiv (Cs, \text{mthd}) \in \text{MinimalMethodDefs } P \ C \ M$

lemma $[\text{code-ind params: } \mathcal{I}]:$

$\text{Subobjs } P \ C \ Cs \Longrightarrow \text{class } P \ (\text{last } Cs) = \llbracket (Bs, fs, ms) \rrbracket \Longrightarrow \text{map-of } ms \ M =$
 $\llbracket \text{mthd} \rrbracket \Longrightarrow$

$\text{MethodDefs}' \ P \ C \ M \ Cs \ \text{mthd}$

$\langle \text{proof} \rangle$

lemma $[\text{code-ind params: } \mathcal{I}]:$

$\text{Subobjs } P \ C \ Cs \Longrightarrow \text{class } P \ (\text{last } Cs) = \llbracket (Bs, fs, ms) \rrbracket \Longrightarrow \text{map-of } fs \ F = \llbracket T \rrbracket$
 \Longrightarrow

$\text{FieldDecls}' \ P \ C \ F \ Cs \ T$

$\langle \text{proof} \rangle$

lemma $[\text{code-ind params: } \mathcal{I}]:$

$\text{MethodDefs}' \ P \ C \ M \ Cs \ \text{mthd} \Longrightarrow$

$\forall (Cs', \text{mthd}) \in \{(Cs', \text{mthd}). \text{MethodDefs}' \ P \ C \ M \ Cs' \ \text{mthd}\}. P, C \vdash Cs' \sqsubseteq Cs$
 $\longrightarrow Cs' = Cs \Longrightarrow$

$\text{MinimalMethodDefs}' \ P \ C \ M \ Cs \ \text{mthd}$

$\langle \text{proof} \rangle$

lemma *ForallMethodDefs-eq*:

$(\forall (Cs, mthd) \in \text{MethodDefs } P \ C \ M. \ Q \ Cs) = (\forall (Cs, mthd) \in \{(Cs, mthd). \text{MethodDefs}' \ P \ C \ M \ Cs \ mthd\}. \ Q \ Cs)$
 $\langle \text{proof} \rangle$

lemma *ForallFieldDecls-eq*:

$(\forall (Cs, T) \in \text{FieldDecls } P \ C \ F. \ Q \ Cs) = (\forall (Cs, T) \in \{(Cs, T). \text{FieldDecls}' \ P \ C \ F \ Cs \ T\}. \ Q \ Cs)$
 $\langle \text{proof} \rangle$

constdefs

$\text{OverrideMethodDefs}' :: \text{prog} \Rightarrow \text{subobj} \Rightarrow \text{mname} \Rightarrow \text{path} \Rightarrow \text{method} \Rightarrow \text{bool}$
 $\text{OverrideMethodDefs}' \ P \ R \ M \ Cs \ mthd \equiv (Cs, mthd) \in \text{OverrideMethodDefs } P \ R \ M$

lemma *OverrideMethodDefs-card-eq*:

$\text{card } (\text{OverrideMethodDefs } P \ R \ M) = \text{card } \{(Cs, mthd). \text{OverrideMethodDefs}' \ P \ R \ M \ Cs \ mthd\}$
 $\langle \text{proof} \rangle$

lemmas $\text{codegen-simps} = \text{MethodDefs}'\text{-def } [\text{symmetric}] \text{ForallMethodDefs-eq}$
 $\text{FieldDecls}'\text{-def } [\text{symmetric}] \text{ForallFieldDecls-eq } \text{OverrideMethodDefs}'\text{-def } [\text{symmetric}]$
 $\text{OverrideMethodDefs-card-eq}$

29.2 Rewriting lemmas for Semantic rules

Cast

lemma *StaticUpCast-new1*:

$\llbracket P, E \vdash \langle e, s_0 \rangle \Rightarrow \langle \text{ref } (a, Cs), (h, l) \rangle;$
 $\text{Subobjs } P \ (\text{last } Cs) \ Cs'; \text{last } Cs' = C;$
 $\text{last } Cs = \text{hd } Cs'; \text{Cs } @ \ \text{tl } Cs' = \text{Ds} \rrbracket$
 $\Rightarrow P, E \vdash \langle \llbracket C \rrbracket e, s_0 \rangle \Rightarrow \langle \text{ref } (a, \text{Ds}), (h, l) \rangle$
 $\langle \text{proof} \rangle$

lemma *StaticUpCast-new2*:

$\llbracket P, E \vdash \langle e, s_0 \rangle \Rightarrow \langle \text{ref } (a, Cs), (h, l) \rangle;$
 $\text{Subobjs } P \ (\text{last } Cs) \ Cs'; \text{last } Cs' = C;$
 $\text{last } Cs \neq \text{hd } Cs' \rrbracket$
 $\Rightarrow P, E \vdash \langle \llbracket C \rrbracket e, s_0 \rangle \Rightarrow \langle \text{ref } (a, Cs'), (h, l) \rangle$
 $\langle \text{proof} \rangle$

lemma *StaticDownCast-new*:

$\llbracket P, E \vdash \langle e, s_0 \rangle \Rightarrow \langle \text{ref } (a, \text{Ds}), s_1 \rangle; \text{app } Cs \ [C] \ \text{Ds}'; \text{app } \text{Ds}' \ Cs' \ \text{Ds} \rrbracket$
 $\Rightarrow P, E \vdash \langle \llbracket C \rrbracket e, s_0 \rangle \Rightarrow \langle \text{ref } (a, \text{Cs}@[C]), s_1 \rangle$
 $\langle \text{proof} \rangle$

lemma *StaticCastFail-new*:

$\llbracket P, E \vdash \langle e, s_0 \rangle \Rightarrow \langle \text{ref } (a, Cs), (h, l) \rangle; \neg P \vdash (\text{last } Cs) \preceq^* C; C \notin \text{set } Cs \rrbracket$

$\Rightarrow P, E \vdash \langle (C) e, s_0 \rangle \Rightarrow \langle \text{THROW } \text{ClassCast}, (h, l) \rangle$
 <proof>

lemma *StaticUpDynCast-new1*:

$\llbracket P, E \vdash \langle e, s_0 \rangle \Rightarrow \langle \text{ref } (a, Cs), (h, l) \rangle;$
 $\text{Subobjs } P \text{ (last } Cs) \text{ } Cs'; \text{ last } Cs' = C;$
 $\forall Cs'' \in \{Cs''. \text{Subobjs } P \text{ (last } Cs) \text{ } Cs''\}. \text{last } Cs'' = C \longrightarrow Cs' = Cs'';$
 $\text{last } Cs = \text{hd } Cs'; Cs @ \text{tl } Cs' = Ds \rrbracket$
 $\Rightarrow P, E \vdash \langle \text{Cast } C \text{ } e, s_0 \rangle \Rightarrow \langle \text{ref } (a, Ds), (h, l) \rangle$
 <proof>

lemma *StaticUpDynCast-new2*:

$\llbracket P, E \vdash \langle e, s_0 \rangle \Rightarrow \langle \text{ref } (a, Cs), (h, l) \rangle;$
 $\text{Subobjs } P \text{ (last } Cs) \text{ } Cs'; \text{ last } Cs' = C;$
 $\forall Cs'' \in \{Cs''. \text{Subobjs } P \text{ (last } Cs) \text{ } Cs''\}. \text{last } Cs'' = C \longrightarrow Cs' = Cs'';$
 $\text{last } Cs \neq \text{hd } Cs' \rrbracket$
 $\Rightarrow P, E \vdash \langle \text{Cast } C \text{ } e, s_0 \rangle \Rightarrow \langle \text{ref } (a, Cs'), (h, l) \rangle$
 <proof>

lemma *StaticDownDynCast-new*:

$\llbracket P, E \vdash \langle e, s_0 \rangle \Rightarrow \langle \text{ref } (a, Ds), s_1 \rangle; \text{app } Cs [C] \text{ } Ds'; \text{app } Ds' \text{ } Cs' \text{ } Ds \rrbracket$
 $\Rightarrow P, E \vdash \langle \text{Cast } C \text{ } e, s_0 \rangle \Rightarrow \langle \text{ref } (a, Cs@[C]), s_1 \rangle$
 <proof>

lemma *DynCast-new*:

$\llbracket P, E \vdash \langle e, s_0 \rangle \Rightarrow \langle \text{ref } (a, Cs), (h, l) \rangle; h \text{ } a = \text{Some}(D, S);$
 $\text{Subobjs } P \text{ } D \text{ } Cs'; \text{last } Cs' = C;$
 $\forall Cs'' \in \{Cs''. \text{Subobjs } P \text{ } D \text{ } Cs''\}. \text{last } Cs'' = C \longrightarrow Cs' = Cs'' \rrbracket$
 $\Rightarrow P, E \vdash \langle \text{Cast } C \text{ } e, s_0 \rangle \Rightarrow \langle \text{ref } (a, Cs'), (h, l) \rangle$
 <proof>

lemma *DynCastFail-new*:

$\llbracket P, E \vdash \langle e, s_0 \rangle \Rightarrow \langle \text{ref } (a, Cs), (h, l) \rangle; h \text{ } a = \text{Some}(D, S);$
 $\forall Cs' \in \{Cs'. \text{Subobjs } P \text{ } D \text{ } Cs'\}. \text{last } Cs' = C \longrightarrow$
 $(\exists Cs'' \in \{Cs''. \text{Subobjs } P \text{ } D \text{ } Cs''\}. \text{last } Cs'' = C \wedge Cs' \neq Cs'');$
 $\forall Cs' \in \{Cs'. \text{Subobjs } P \text{ (last } Cs) \text{ } Cs'\}. \text{last } Cs' = C \longrightarrow$
 $(\exists Cs'' \in \{Cs''. \text{Subobjs } P \text{ (last } Cs) \text{ } Cs''\}. \text{last } Cs'' = C \wedge Cs' \neq Cs'');$
 $C \notin \text{set } Cs \rrbracket$
 $\Rightarrow P, E \vdash \langle \text{Cast } C \text{ } e, s_0 \rangle \Rightarrow \langle \text{null}, (h, l) \rangle$
 <proof>

Assignment

lemma *LAss-new*:

$\llbracket P, E \vdash \langle e, s_0 \rangle \Rightarrow \langle \text{Val } v, (h, l) \rangle; E \text{ } V = \lfloor T \rfloor;$
 $\text{casts-aux } P \text{ } T \text{ } v \text{ } v'; l' = l(V \mapsto v') \rrbracket$
 $\Rightarrow P, E \vdash \langle V := e, s_0 \rangle \Rightarrow \langle \text{Val } v', (h, l') \rangle$
 <proof>

Fields

lemma *FAcc-new1*:

$$\begin{aligned} & \llbracket P, E \vdash \langle e, s_0 \rangle \Rightarrow \langle \text{ref } (a, Cs'), (h, l) \rangle; h a = \text{Some}(D, S); \\ & \quad \text{last } Cs' = \text{hd } Cs; Cs' @ \text{tl } Cs = Ds; (Ds, fs) \in S; fs F = \text{Some } v \rrbracket \\ & \Rightarrow P, E \vdash \langle e \cdot F \{Cs\}, s_0 \rangle \Rightarrow \langle \text{Val } v, (h, l) \rangle \\ & \langle \text{proof} \rangle \end{aligned}$$

lemma *F_{Acc}-new2*:

$$\begin{aligned} & \llbracket P, E \vdash \langle e, s_0 \rangle \Rightarrow \langle \text{ref } (a, Cs'), (h, l) \rangle; h a = \text{Some}(D, S); \\ & \quad \text{last } Cs' \neq \text{hd } Cs; (Cs, fs) \in S; fs F = \text{Some } v \rrbracket \\ & \Rightarrow P, E \vdash \langle e \cdot F \{Cs\}, s_0 \rangle \Rightarrow \langle \text{Val } v, (h, l) \rangle \\ & \langle \text{proof} \rangle \end{aligned}$$

lemma *F_{Ass}-new1*:

$$\begin{aligned} & \llbracket P, E \vdash \langle e_1, s_0 \rangle \Rightarrow \langle \text{ref } (a, Cs'), s_1 \rangle; P, E \vdash \langle e_2, s_1 \rangle \Rightarrow \langle \text{Val } v, (h_2, l_2) \rangle; \\ & \quad h_2 a = \text{Some}(D, S); P \vdash (\text{last } Cs' \text{ has least } F:T \text{ via } Cs); \\ & \quad \text{casts-aux } P T v v'; \text{last } Cs' = \text{hd } Cs; Cs' @ \text{tl } Cs = Ds; \\ & \quad (Ds, fs) \in S; fs' = fs(F \mapsto v'); \\ & \quad S' = S - \{(Ds, fs)\} \cup \{(Ds, fs')\}; h_2' = h_2(a \mapsto (D, S')) \rrbracket \\ & \Rightarrow P, E \vdash \langle e_1 \cdot F \{Cs\} := e_2, s_0 \rangle \Rightarrow \langle \text{Val } v', (h_2', l_2) \rangle \\ & \langle \text{proof} \rangle \end{aligned}$$

lemma *F_{Ass}-new2*:

$$\begin{aligned} & \llbracket P, E \vdash \langle e_1, s_0 \rangle \Rightarrow \langle \text{ref } (a, Cs'), s_1 \rangle; P, E \vdash \langle e_2, s_1 \rangle \Rightarrow \langle \text{Val } v, (h_2, l_2) \rangle; \\ & \quad h_2 a = \text{Some}(D, S); P \vdash (\text{last } Cs' \text{ has least } F:T \text{ via } Cs); \\ & \quad \text{casts-aux } P T v v'; \text{last } Cs' \neq \text{hd } Cs; (Cs, fs) \in S; fs' = fs(F \mapsto v'); \\ & \quad S' = S - \{(Cs, fs)\} \cup \{(Cs, fs')\}; h_2' = h_2(a \mapsto (D, S')) \rrbracket \\ & \Rightarrow P, E \vdash \langle e_1 \cdot F \{Cs\} := e_2, s_0 \rangle \Rightarrow \langle \text{Val } v', (h_2', l_2) \rangle \\ & \langle \text{proof} \rangle \end{aligned}$$

Call

lemma *CallParamsThrow-new*:

$$\begin{aligned} & \llbracket P, E \vdash \langle e, s_0 \rangle \Rightarrow \langle \text{Val } v, s_1 \rangle; P, E \vdash \langle es, s_1 \rangle [\Rightarrow] \langle evs, s_2 \rangle; \\ & \quad \text{map-val2 } evs vs (\text{throw } ex \# es') \rrbracket \\ & \Rightarrow P, E \vdash \langle \text{Call } e \text{ Copt } M \text{ } es, s_0 \rangle \Rightarrow \langle \text{throw } ex, s_2 \rangle \\ & \langle \text{proof} \rangle \end{aligned}$$

lemma *CallNull-new*:

$$\begin{aligned} & \llbracket P, E \vdash \langle e, s_0 \rangle \Rightarrow \langle \text{null}, s_1 \rangle; P, E \vdash \langle es, s_1 \rangle [\Rightarrow] \langle evs, s_2 \rangle; \text{map-val } evs vs \rrbracket \\ & \Rightarrow P, E \vdash \langle \text{Call } e \text{ Copt } M \text{ } es, s_0 \rangle \Rightarrow \langle \text{THROW } \text{NullPointer}, s_2 \rangle \\ & \langle \text{proof} \rangle \end{aligned}$$

lemma *Call-new*:

$$\begin{aligned} & \llbracket P, E \vdash \langle e, s_0 \rangle \Rightarrow \langle \text{ref } (a, Cs), s_1 \rangle; P, E \vdash \langle ps, s_1 \rangle [\Rightarrow] \langle evs, (h_2, l_2) \rangle; \\ & \quad \text{map-val } evs vs; h_2 a = \text{Some}(C, S); \\ & \quad P \vdash \text{last } Cs \text{ has least } M = (Ts', T', pns', \text{body}') \text{ via } Ds; \\ & \quad P \vdash C \text{ has least } M = (Ts, T, pns, \text{body}) \text{ via } Cs'; \text{length } vs = \text{length } pns; \\ & \quad \text{Casts-aux } P Ts vs vs'; l_2' = [\text{this} \mapsto \text{Ref } (a, Cs'), pns[\mapsto] vs']; \\ & \quad \text{new-body} = (\text{case } T' \text{ of Class } D \Rightarrow ([D])\text{body} \quad | _ \Rightarrow \text{body}); \\ & \quad P, E(\text{this} \mapsto \text{Class}(\text{last } Cs'), pns[\mapsto] Ts) \vdash \langle \text{new-body}, (h_2, l_2') \rangle \Rightarrow \langle e', (h_3, l_3) \rangle \rrbracket \\ & \Rightarrow P, E \vdash \langle e \cdot M(ps), s_0 \rangle \Rightarrow \langle e', (h_3, l_2) \rangle \end{aligned}$$

$\langle \text{proof} \rangle$

lemma *Overrider1*:

$P \vdash (\text{ldc } R) \text{ has least } M = \text{mthd}' \text{ via } Cs' \implies$
 $\text{MinimalMethodDefs}' P (\text{mdc } R) M Cs \text{ mthd} \implies$
 $\text{last } (\text{snd } R) = \text{hd } Cs' \implies P, \text{mdc } R \vdash Cs \sqsubseteq (\text{snd } R)@tl Cs' \implies$
 $\text{OverriderMethodDefs}' P R M Cs \text{ mthd}$

$\langle \text{proof} \rangle$

lemma *Overrider2*:

$P \vdash (\text{ldc } R) \text{ has least } M = \text{mthd}' \text{ via } Cs' \implies$
 $\text{MinimalMethodDefs}' P (\text{mdc } R) M Cs \text{ mthd} \implies$
 $\text{last } (\text{snd } R) \neq \text{hd } Cs' \implies P, \text{mdc } R \vdash Cs \sqsubseteq Cs' \implies$
 $\text{OverriderMethodDefs}' P R M Cs \text{ mthd}$

$\langle \text{proof} \rangle$

lemma *ambiguous*: $(\neg P \vdash C \text{ has least } M = \text{mthd} \text{ via } Cs') =$

$(\text{MethodDefs}' P C M Cs' \text{ mthd} \longrightarrow (\exists (Cs'', \text{mthd}') \in \{(Cs'', \text{mthd}') . \text{MethodDefs}'$
 $P C M Cs'' \text{ mthd}'\}. \neg P, C \vdash Cs' \sqsubseteq Cs''))$

$\langle \text{proof} \rangle$

lemma *CallOverrider-new1*:

$\llbracket P, E \vdash \langle e, s_0 \rangle \Rightarrow \langle \text{ref } (a, Cs), s_1 \rangle; P, E \vdash \langle ps, s_1 \rangle [\Rightarrow] \langle \text{evs}, (h_2, l_2) \rangle;$
 $\text{map-val evs vs}; h_2 a = \text{Some}(C, S);$
 $P \vdash \text{last } Cs \text{ has least } M = (Ts', T', pns', \text{body}') \text{ via } Ds;$
 $\forall (Cs', \text{mthd}) \in \{(Cs', \text{mthd}) . \text{MethodDefs}' P (\text{last } Cs) M Cs' \text{ mthd}\}. P, \text{last } Cs$
 $\vdash Ds \sqsubseteq Cs';$
 $\forall (Cs', \text{mthd}) \in \{(Cs', \text{mthd}) . \text{MethodDefs}' P C M Cs' \text{ mthd}\}. \exists (Cs'', \text{mthd}') \in \{(Cs'', \text{mthd}') .$
 $\text{MethodDefs}' P C M Cs'' \text{ mthd}'\}. \neg P, C \vdash Cs' \sqsubseteq Cs'';$
 $\text{last } Cs = \text{hd } Ds; P \vdash (C, Cs@tl Ds) \text{ has overrider } M = (Ts, T, pns, \text{body}) \text{ via}$
 $Cs';$
 $\text{length vs} = \text{length pns}; \text{Casts-aux } P Ts vs vs';$
 $l_2' = [\text{this} \mapsto \text{Ref } (a, Cs'), pns \mapsto vs];$
 $\text{new-body} = (\text{case } T' \text{ of Class } D \Rightarrow \langle D \rangle \text{body} \mid - \Rightarrow \text{body});$
 $P, E(\text{this} \mapsto \text{Class}(\text{last } Cs'), pns \mapsto Ts) \vdash \langle \text{new-body}, (h_2, l_2') \rangle \Rightarrow \langle e', (h_3, l_3) \rangle \rrbracket$
 $\implies P, E \vdash \langle e \cdot M(ps), s_0 \rangle \Rightarrow \langle e', (h_3, l_2) \rangle$

$\langle \text{proof} \rangle$

lemma *CallOverrider-new2*:

$\llbracket P, E \vdash \langle e, s_0 \rangle \Rightarrow \langle \text{ref } (a, Cs), s_1 \rangle; P, E \vdash \langle ps, s_1 \rangle [\Rightarrow] \langle \text{evs}, (h_2, l_2) \rangle;$
 $\text{map-val evs vs}; h_2 a = \text{Some}(C, S);$
 $P \vdash \text{last } Cs \text{ has least } M = (Ts', T', pns', \text{body}') \text{ via } Ds;$
 $\forall (Cs', \text{mthd}) \in \{(Cs', \text{mthd}) . \text{MethodDefs}' P (\text{last } Cs) M Cs' \text{ mthd}\}. P, \text{last } Cs$
 $\vdash Ds \sqsubseteq Cs';$
 $\forall (Cs', \text{mthd}) \in \{(Cs', \text{mthd}) . \text{MethodDefs}' P C M Cs' \text{ mthd}\}. \exists (Cs'', \text{mthd}') \in \{(Cs'', \text{mthd}') .$
 $\text{MethodDefs}' P C M Cs'' \text{ mthd}'\}. \neg P, C \vdash Cs' \sqsubseteq Cs'';$
 $\text{last } Cs \neq \text{hd } Ds; P \vdash (C, Ds) \text{ has overrider } M = (Ts, T, pns, \text{body}) \text{ via } Cs';$

$length\ vs = length\ pns;$ *Casts-aux* $P\ Ts\ vs\ vs'$;
 $l_2' = [this \mapsto Ref\ (a, Cs'), pns[\mapsto]vs']$;
 $new-body = (case\ T'\ of\ Class\ D \Rightarrow ([D])body\ \mid - \Rightarrow body)$;
 $P, E(this \mapsto Class(last\ Cs'), pns[\mapsto]Ts) \vdash \langle new-body, (h_2, l_2') \rangle \Rightarrow \langle e', (h_3, l_3) \rangle$]
 $\Rightarrow P, E \vdash \langle e \cdot M(ps), s_0 \rangle \Rightarrow \langle e', (h_3, l_2) \rangle$
<proof>

lemma *StaticCall-new1*:

assumes $evals: P, E \vdash \langle ps, s_1 \rangle [\Rightarrow] \langle evs, (h_2, l_2) \rangle$ **and** $map: map-val\ evs\ vs$
and $path: Subobjs\ P\ (last\ Cs)\ Cs''\ last\ Cs'' = C$
and $unique: \forall Xs \in \{Xs.\ Subobjs\ P\ (last\ Cs)\ Xs\}.\ last\ Xs = C \longrightarrow Cs'' = Xs$
and $eq1: last\ Cs = hd\ Cs''$ **and** $eq2: last\ Cs'' = hd\ Cs'$
and $append: Ds = (Cs @tl\ Cs'') @tl\ Cs'$ **and** $casts: Casts-aux\ P\ Ts\ vs\ vs'$
and $rest: P, E \vdash \langle e, s_0 \rangle \Rightarrow \langle ref\ (a, Cs), s_1 \rangle$ $length\ vs = length\ pns$
 $l_2' = [this \mapsto Ref\ (a, Ds), pns[\mapsto]vs']$
 $P \vdash C\ has\ least\ M = (Ts, T, pns, body)\ via\ Cs'$
 $P, E(this \mapsto Class(last\ Ds), pns[\mapsto]Ts) \vdash \langle body, (h_2, l_2') \rangle \Rightarrow \langle e', (h_3, l_3) \rangle$
shows $P, E \vdash \langle e \cdot (C::)M(ps), s_0 \rangle \Rightarrow \langle e', (h_3, l_2) \rangle$
<proof>

lemma *StaticCall-new2*:

assumes $evals: P, E \vdash \langle ps, s_1 \rangle [\Rightarrow] \langle evs, (h_2, l_2) \rangle$ **and** $map: map-val\ evs\ vs$
and $path: Subobjs\ P\ (last\ Cs)\ Cs''\ last\ Cs'' = C$
and $unique: \forall Xs \in \{Xs.\ Subobjs\ P\ (last\ Cs)\ Xs\}.\ last\ Xs = C \longrightarrow Cs'' = Xs$
and $eq1: last\ Cs = hd\ Cs''$ **and** $eq2: last\ Cs'' \neq hd\ Cs'$
and $append: Ds = Cs'$ **and** $casts: Casts-aux\ P\ Ts\ vs\ vs'$
and $rest: P, E \vdash \langle e, s_0 \rangle \Rightarrow \langle ref\ (a, Cs), s_1 \rangle$ $length\ vs = length\ pns$
 $l_2' = [this \mapsto Ref\ (a, Ds), pns[\mapsto]vs']$
 $P \vdash C\ has\ least\ M = (Ts, T, pns, body)\ via\ Cs'$
 $P, E(this \mapsto Class(last\ Ds), pns[\mapsto]Ts) \vdash \langle body, (h_2, l_2') \rangle \Rightarrow \langle e', (h_3, l_3) \rangle$
shows $P, E \vdash \langle e \cdot (C::)M(ps), s_0 \rangle \Rightarrow \langle e', (h_3, l_2) \rangle$
<proof>

lemma *StaticCall-new3*:

assumes $evals: P, E \vdash \langle ps, s_1 \rangle [\Rightarrow] \langle evs, (h_2, l_2) \rangle$ **and** $map: map-val\ evs\ vs$
and $path: Subobjs\ P\ (last\ Cs)\ Cs''\ last\ Cs'' = C$
and $unique: \forall Xs \in \{Xs.\ Subobjs\ P\ (last\ Cs)\ Xs\}.\ last\ Xs = C \longrightarrow Cs'' = Xs$
and $eq1: last\ Cs \neq hd\ Cs''$ **and** $eq2: last\ Cs'' = hd\ Cs'$
and $append: Ds = Cs'' @tl\ Cs'$ **and** $casts: Casts-aux\ P\ Ts\ vs\ vs'$
and $rest: P, E \vdash \langle e, s_0 \rangle \Rightarrow \langle ref\ (a, Cs), s_1 \rangle$ $length\ vs = length\ pns$
 $l_2' = [this \mapsto Ref\ (a, Ds), pns[\mapsto]vs']$
 $P \vdash C\ has\ least\ M = (Ts, T, pns, body)\ via\ Cs'$
 $P, E(this \mapsto Class(last\ Ds), pns[\mapsto]Ts) \vdash \langle body, (h_2, l_2') \rangle \Rightarrow \langle e', (h_3, l_3) \rangle$
shows $P, E \vdash \langle e \cdot (C::)M(ps), s_0 \rangle \Rightarrow \langle e', (h_3, l_2) \rangle$
<proof>

lemma *StaticCall-new4*:

assumes $evals: P, E \vdash \langle ps, s_1 \rangle [\Rightarrow] \langle evs, (h_2, l_2) \rangle$ **and** $map: map-val\ evs\ vs$
and $path: Subobjs\ P\ (last\ Cs)\ Cs''\ last\ Cs'' = C$

and *unique*: $\forall Xs \in \{Xs. \text{Subobjs } P \text{ (last } Cs) Xs\}. \text{last } Xs = C \longrightarrow Cs'' = Xs$
and *eq1*: $\text{last } Cs \neq \text{hd } Cs''$ **and** *eq2*: $\text{last } Cs'' \neq \text{hd } Cs'$
and *append*: $Ds = Cs'$ **and** *casts*: $\text{Casts-aux } P \text{ } Ts \text{ } vs \text{ } vs'$
and *rest*: $P, E \vdash \langle e, s_0 \rangle \Rightarrow \langle \text{ref } (a, Cs), s_1 \rangle \text{ length } vs = \text{length } pns$
 $l_2' = [\text{this} \mapsto \text{Ref } (a, Ds), pns[\mapsto] vs']$
 $P \vdash C \text{ has least } M = (Ts, T, pns, \text{body}) \text{ via } Cs'$
 $P, E(\text{this} \mapsto \text{Class}(\text{last } Ds), pns[\mapsto] Ts) \vdash \langle \text{body}, (h_2, l_2') \rangle \Rightarrow \langle e', (h_3, l_3) \rangle$
shows $P, E \vdash \langle e \cdot (C::)M(ps), s_0 \rangle \Rightarrow \langle e', (h_3, l_2) \rangle$
 $\langle \text{proof} \rangle$

29.3 Rewriting lemmas for Type rules

lemma *WTDynCast-new1*:

$\llbracket P, E \vdash e :: \text{Class } D; \text{is-class } P \text{ } C;$
 $\text{Subobjs } P \text{ } D \text{ } Cs'; \text{last } Cs' = C;$
 $\forall Cs'' \in \{Cs''. \text{Subobjs } P \text{ } D \text{ } Cs''\}. \text{last } Cs'' = C \longrightarrow Cs' = Cs'' \rrbracket$
 $\Longrightarrow P, E \vdash \text{Cast } C \text{ } e :: \text{Class } C$
 $\langle \text{proof} \rangle$

lemma *WTDynCast-new2*:

$\llbracket P, E \vdash e :: \text{Class } D; \text{is-class } P \text{ } C;$
 $\forall Cs'' \in \{Cs''. \text{Subobjs } P \text{ } D \text{ } Cs''\}. \text{last } Cs'' = C \longrightarrow \text{False} \rrbracket$
 $\Longrightarrow P, E \vdash \text{Cast } C \text{ } e :: \text{Class } C$
 $\langle \text{proof} \rangle$

lemma *WTStaticCast-new1*:

$\llbracket P, E \vdash e :: \text{Class } D; \text{is-class } P \text{ } C;$
 $\text{Subobjs } P \text{ } D \text{ } Cs'; \text{last } Cs' = C;$
 $\forall Cs'' \in \{Cs''. \text{Subobjs } P \text{ } D \text{ } Cs''\}. \text{last } Cs'' = C \longrightarrow Cs' = Cs'' \rrbracket$
 $\Longrightarrow P, E \vdash \langle C \rangle e :: \text{Class } C$
 $\langle \text{proof} \rangle$

lemma *WTStaticCast-new2*:

$\llbracket P, E \vdash e :: \text{Class } D; \text{is-class } P \text{ } C; P \vdash C \preceq^* D;$
 $\forall Cs \in \{Cs. \text{Subobjs } P \text{ } C \text{ } Cs\}. \text{last } Cs = D \longrightarrow \text{Subobjs}_R \text{ } P \text{ } C \text{ } Cs \rrbracket$
 $\Longrightarrow P, E \vdash \langle C \rangle e :: \text{Class } C$
 $\langle \text{proof} \rangle$

lemma *WTBinOp1*: $\llbracket P, E \vdash e_1 :: T; P, E \vdash e_2 :: T \rrbracket$

$\Longrightarrow P, E \vdash e_1 \ll \text{Eq} \gg e_2 :: \text{Boolean}$
 $\langle \text{proof} \rangle$

lemma *WTBinOp2*: $\llbracket P, E \vdash e_1 :: \text{Integer}; P, E \vdash e_2 :: \text{Integer} \rrbracket$

$\Longrightarrow P, E \vdash e_1 \ll \text{Add} \gg e_2 :: \text{Integer}$
 $\langle \text{proof} \rangle$

lemma *WTStaticCall-new*:

$\llbracket P, E \vdash e :: \text{Class } C'; \text{Subobjs } P \text{ } C' \text{ } Cs'; \text{last } Cs' = C;$
 $\forall Cs'' \in \{Cs''. \text{Subobjs } P \text{ } C' \text{ } Cs''\}. \text{last } Cs'' = C \longrightarrow Cs' = Cs'' \rrbracket$

$P \vdash C$ has least $M = (Ts, T, m)$ via Cs ; $P, E \vdash es [::] Ts'$; $P \vdash Ts' [\leq] Ts$]
 $\implies P, E \vdash e.(C::)M(es) [::] T$
 <proof>

lemma [code-ind]:
 $\llbracket \text{Subobjs } P \ C \ Cs'; \text{ last } Cs' = D;$
 $\forall Cs'' \in \{Cs''. \text{Subobjs } P \ C \ Cs''\}. \text{ last } Cs'' = D \implies Cs' = Cs'' \rrbracket$
 $\implies P \vdash \text{Class } C \leq \text{Class } D$
 <proof>

lemmas [code-ind] = widen-refl widen-null

29.4 Code generation

lemmas [code-ind] =
Overrider1 [simplified LeastMethodDef-def codegen-simps, OF conjI]
Overrider2 [simplified LeastMethodDef-def codegen-simps, OF conjI]

eval-vals.New eval-vals.NewFail
StaticUpCast-new1 StaticUpCast-new2 StaticDownCast-new
eval-vals.StaticCastNull StaticCastFail-new eval-vals.StaticCastThrow
StaticUpDynCast-new1 StaticUpDynCast-new2 StaticDownDynCast-new
DynCast-new eval-vals.DynCastNull
DynCastFail-new eval-vals.DynCastThrow
eval-vals.Val eval-vals.Var
eval-vals.BinOp eval-vals.BinOpThrow1 eval-vals.BinOpThrow2
LAss-new eval-vals.LAssThrow FAcc-new1 FAcc-new2
FAss-new1 [simplified LeastFieldDecl-def codegen-simps, OF - - - conjI]
FAss-new2 [simplified LeastFieldDecl-def codegen-simps, OF - - - conjI]
eval-vals.FAssNull eval-vals.FAssThrow1 eval-vals.FAssThrow2
eval-vals.CallObjThrow CallNull-new CallParamsThrow-new
Call-new [simplified LeastMethodDef-def codegen-simps, OF - - - conjI conjI]
CallOverrider-new1 [simplified FinalOverriderMethodDef-def LeastMethodDef-def
 codegen-simps,
 OF - - - - conjI - - - conjI]
CallOverrider-new2 [simplified FinalOverriderMethodDef-def LeastMethodDef-def
 codegen-simps,
 OF - - - - conjI - - - conjI]
StaticCall-new1 [simplified FinalOverriderMethodDef-def LeastMethodDef-def codegen-simps,
 OF - - - - - - - - - conjI]
StaticCall-new2 [simplified FinalOverriderMethodDef-def LeastMethodDef-def codegen-simps,
 OF - - - - - - - - - conjI]
StaticCall-new3 [simplified FinalOverriderMethodDef-def LeastMethodDef-def codegen-simps,
 OF - - - - - - - - - conjI]
StaticCall-new4 [simplified FinalOverriderMethodDef-def LeastMethodDef-def codegen-simps,
 OF - - - - - - - - - conjI]
eval-vals.Block eval-vals.Seq eval-vals.SeqThrow
eval-vals.CondT eval-vals.CondF eval-vals.CondThrow

eval-vals.WhileF eval-vals.WhileT
eval-vals.WhileCondThrow eval-vals.WhileBodyThrow
eval-vals.Throw eval-vals.ThrowNull eval-vals.ThrowThrow
eval-vals.Nil eval-vals.Cons eval-vals.ConsThrow

WT-WTs.WTNew WTDynCast-new1 WTDynCast-new2
WTStaticCast-new1 WTStaticCast-new2
WT-WTs.WTVal WT-WTs.WTVar WTBinOp1 WTBinOp2 WT-WTs.WTLAss
WT-WTs.WTFAcc[unfolded LeastFieldDecl-def codegen-simps, OF - conjI]
WT-WTs.WTFAss[unfolded LeastFieldDecl-def codegen-simps, OF - conjI]
WT-WTs.WTCall[unfolded LeastMethodDef-def codegen-simps, OF - conjI]
WTStaticCall-new[unfolded LeastMethodDef-def codegen-simps, OF - - - - conjI]
WT-WTs.WTBlock WT-WTs.WTSeq WT-WTs.WTCond WT-WTs.WTWhile WT-WTs.WTThrow
WT-WTs.WTNil WT-WTs.WTCons

lemmas [code-ind-set] = *rtrancl.rtrancl-refl converse-rtrancl-into-rtrancl*

consts-code

insert :: ('a × ('b ⇒ 'c)) ⇒ ('a × ('b ⇒ 'c)) set ⇒ ('a × ('b ⇒ 'c)) set
*((fn x => fn { *Set* } xs => { *Set* } (Library.insert (eq'-fst (op =)) x xs)))*
Executable-Set.union :: ('a × ('b ⇒ 'c)) set ⇒ ('a × ('b ⇒ 'c)) set => ('a ×
('b ⇒ 'c)) set
*((fn { *Set* } xs => fn { *Set* } ys => { *Set* } (Library.union (eq'-fst (op =))*
xs ys)))
minus :: ('a × ('b ⇒ 'c)) set ⇒ ('a × ('b ⇒ 'c)) set ⇒ ('a × ('b ⇒ 'c)) set
*((fn { *Set* } xs => fn { *Set* } ys => { *Set* } (Library.subtract (eq'-fst (op*
=)) ys xs)))

consts-code

new-Addr
*(⟨module⟩new'-addr { * 0::nat * } { * Suc * }*
*{ * %x. case x of None => True | Some y => False * } { * Some * }*)

attach ⟨⟨

fun new-addr z s alloc some hp =
let fun nr i = if alloc (hp i) then some i else nr (s i);
in nr z end;

⟩⟩

undefined ((raise ERROR undefined))

Definition of program examples

lemma [code-unfold]: *x mem xs = ListMem x xs*
⟨proof⟩

... and off we go

code-module *NoProg*

contains

```
test0 = [], empty ⊢ ⟨{"V":Integer; "V" := Val(Intg 5);; Var "V"},(empty,empty)⟩
⇒ ⟨-, -⟩
test1 = [], empty ⊢ ⟨Val(Intg 5),(empty,empty)⟩ ⇒ ⟨-, -⟩
test2 = [], empty ⊢ ⟨(Val(Intg 5)) «Add» (Val(Intg 6)),(empty,empty)⟩ ⇒ ⟨-, -⟩
test3 = [], ["V ↦ Integer] ⊢ ⟨(Var "V") «Add» (Val(Intg 6)),
    (empty,["V ↦ Intg 77])⟩ ⇒ ⟨-, -⟩
test4 = [], ["V ↦ Integer] ⊢ ⟨"V" := Val(Intg 6),(empty,empty)⟩ ⇒ ⟨-, -⟩
testWhile = [], ["V ↦ Integer, "a" ↦ Integer, "b" ↦ Integer, "mult" ↦ Integer]
⊢ ⟨("a" := Val(Intg 3));("b" := Val(Intg 4));("mult" := Val(Intg 0));
    ("V" := Val(Intg 1));;
    while (Var "V" «Eq» Val(Intg 1))(("mult" := Var "mult" «Add» Var "b");;
    ("a" := Var "a" «Add» Val(Intg -1));;
    ("V" := (if (Var "a" «Eq» Val(Intg 0)) Val(Intg 0) else Val(Intg 1))))),
    (empty,empty)⟩ ⇒ ⟨-, -⟩

testIf = [], ["a" ↦ Integer, "b" ↦ Integer, "c" ↦ Integer, "cond" ↦ Boolean] ⊢
⟨"a" := Val(Intg 17);; "b" := Val(Intg 13);; "c" := Val(Intg 42);; "cond" :=
true;; if (Var "cond") (Var "a" «Add» Var "b") else (Var "a" «Add» Var
"c"),(empty,empty)⟩ ⇒ ⟨-, -⟩
```

```
V = "V"
mult = "mult"
```

⟨ML⟩

constdefs

— Overrider example

```
classBottom :: cdecl
classBottom == ("Bottom", [Repeats "Left", Repeats "Right"],
    [{"x", Integer}], [])

classLeft :: cdecl
classLeft == ("Left", [Repeats "Top"], [{"f", [Class "Top", Integer], Integer,
["V", "W"], Var this · "x" {"Left", "Top"} «Add» Val (Intg 5)]])

classRight :: cdecl
classRight == ("Right", [Shares "Right2"], [{"f", [Class "Top", Integer], Integer, ["V", "W"], Var this · "x" {"Right2", "Top"}
«Add» Val (Intg 7)], ["g", [], Class "Left", [], new "Left"]])

classRight2 :: cdecl
classRight2 == ("Right2", [Repeats "Top"], [{"f", [Class "Top", Integer], Integer, ["V", "W"], Var this · "x" {"Right2", "Top"}]
```

«Add» Val (Intg 9)),("g",[],Class "Top",[],new "Top'))

classTop :: cdecl
classTop == ("Top", [], [("x",Integer)],[])

progOverrider :: cdecl list
progOverrider == [classBottom, classLeft, classRight, classRight2, classTop]

code-module ProgOverrider

contains

dynCastSide = progOverrider,["V"↦Class "Right"] ⊢
⟨"V" := new "Bottom" ;; Cast "Left" (Var "V"),(empty,empty)⟩ ⇒ ⟨-, -⟩

dynCastViaSh = progOverrider,["V"↦Class "Right2"] ⊢
⟨"V" := new "Right" ;; Cast "Right" (Var "V"),(empty,empty)⟩ ⇒ ⟨-, -⟩

block = progOverrider,["V"↦Integer] ⊢ ⟨"V" := Val(Intg 42) ;; {"V":Class
"Left"; "V" := new "Bottom"} ;; Var "V", (empty,empty)⟩ ⇒ ⟨-, -⟩

staticCall = progOverrider,["V"↦Class "Right", "W"↦Class "Bottom"] ⊢
⟨"V" := new "Bottom" ;; "W" := new "Bottom" ;;
((Cast "Left" (Var "W"))·"x"{"Left", "Top"} := Val(Intg 3));;
(Var "W".("Left":)·"f"([Var "V", Val(Intg 2)])), (empty,empty)⟩ ⇒ ⟨-, -⟩

call = progOverrider,["V"↦Class "Right2", "W"↦Class "Left"] ⊢
⟨"V" := new "Right" ;; "W" := new "Left" ;; (Var "V"."f"([Var "W", Val(Intg
42)])) «Add» (Var "W"."f"([Var "V", Val(Intg 13)])), (empty,empty)⟩ ⇒ ⟨-, -⟩

callOverrider = progOverrider,["V"↦Class "Right2", "W"↦Class "Left"] ⊢
⟨"V" := new "Bottom" ;; (Var "V" · "x" {"Right2", "Top"} := Val(Intg 6));;
"W" := new "Left" ;; Var "V"."f"([Var "W", Val(Intg 42)]), (empty,empty)⟩ ⇒
⟨-, -⟩

callClass = progOverrider,["V"↦Class "Right2"] ⊢
⟨"V" := new "Right" ;; Var "V"."g"([], (empty,empty))⟩ ⇒ ⟨-, -⟩

fieldAss = progOverrider,["V"↦Class "Right2"] ⊢ ⟨"V" := new "Right" ;;
(Var "V"."x"{"Right2", "Top"} := (Val(Intg 42))) ;;
(Var "V"."x"{"Right2", "Top"}), (empty,empty)⟩ ⇒ ⟨-, -⟩

typeNew = progOverrider, empty ⊢ new "Bottom" :: -
typeDynCast = progOverrider, empty ⊢ Cast "Left" (new "Bottom") :: -
typeStaticCast = progOverrider, empty ⊢ (!"Left") (new "Bottom") :: -
typeVal = [], empty ⊢ Val(Intg 17) :: -
typeVar = [], ["V" ↦ Integer] ⊢ Var "V" :: -
typeBinOp = [], empty ⊢ (Val(Intg 5)) «Eq» (Val(Intg 6)) :: -
typeLAss = progOverrider, ["V" ↦ Class "Top"] ⊢ "V" := (new "Left") :: -
typeFAcc = progOverrider, empty ⊢ (new "Right")·"x"{"Right2", "Top"} :: -

```

typeFAss = progOverride,empty ⊢
  (new "Right")."x"{"Right2","Top"} := (Val(Intg 17)) :: -
  typeStaticCall = progOverride,["V"↦Class "Left" ⊢ "V" := new "Left" ;;
  Var "V"."Left"::"f"([new "Top", Val(Intg 13)])] :: -
  typeCall = progOverride,["V"↦Class "Right2" ⊢ "V" := new "Right" ;; Var
  "V"."g"([]) :: -
  typeBlock = progOverride,empty ⊢ {"V":Class "Top"; "V" := new "Left"} :: -
-
typeCond = [],empty ⊢ if (true) Val(Intg 6) else Val(Intg 9) :: -
typeWhile = [],empty ⊢ while (false) Val(Intg 17) :: -
typeThrow = progOverride,empty ⊢ throw (new "Bottom") :: -

typeBig = progOverride,["V"↦Class "Right2","W"↦Class "Left" ⊢
  "V" := new "Right" ;; "W" := new "Left" ;; (Var "V"."f"([Var "W",
  Val(Intg 7)])] «Add» (Var "W"."f"([Var "V", Val(Intg 13)])) :: -

Bottom = "Bottom"
Left = "Left"
Right = "Right"
Top = "Top"

```

⟨ML⟩

constdefs

— Diamond class-name DAG

```

classDiamondBottom :: cdecl
classDiamondBottom == ("Bottom", [Repeats "Left", Repeats "Right"],[("x",Integer)],
  [("g", [],Integer, [], Var this · "x" {"Bottom"} «Add» Val (Intg 5)])]

classDiamondLeft :: cdecl
classDiamondLeft == ("Left", [Repeats "TopRep",Shares "TopSh"],[],[])

classDiamondRight :: cdecl
classDiamondRight == ("Right", [Repeats "TopRep",Shares "TopSh"],[],
  [("f", [Integer], Boolean,["i"], Var "i" «Eq» Val (Intg 7)])]

classDiamondTopRep :: cdecl
classDiamondTopRep == ("TopRep", [], [("x",Integer)],
  [("g", [],Integer, [], Var this · "x" {"TopRep"} «Add» Val (Intg 10)])]

classDiamondTopSh :: cdecl
classDiamondTopSh == ("TopSh", [], [],
  [("f", [Integer], Boolean,["i"], Var "i" «Eq» Val (Intg 3)])]

```

```

progDiamond :: cdecl list
progDiamond == [classDiamondBottom, classDiamondLeft, classDiamondRight,
classDiamondTopRep, classDiamondTopSh]

```

code-module *ProgDiamond*

contains

```

cast1 = progDiamond,[V ↦ Class "Left"] ⊢ ⟨"V" := new "Bottom",
      (empty,empty)⟩ ⇒ ⟨-, -⟩
cast2 = progDiamond,[V ↦ Class "TopSh"] ⊢ ⟨"V" := new "Bottom",
      (empty,empty)⟩ ⇒ ⟨-, -⟩
cast3 = progDiamond,[V ↦ Class "TopRep"] ⊢ ⟨"V" := new "Bottom",
      (empty,empty)⟩ ⇒ ⟨-, -⟩
typeCast3 = progDiamond,[V ↦ Class "TopRep"] ⊢ "V" := new "Bottom" ::
-

fieldAss = progDiamond,[V ↦ Class "Bottom"] ⊢ ⟨"V" := new "Bottom" ;;
      ((Var "V")."x"{"Bottom"} := (Val(Intg 17))) ;;
      ((Var "V")."x"{"Bottom"}),(empty,empty)⟩ ⇒ ⟨-, -⟩

dynCastNull = progDiamond,empty ⊢ ⟨Cast "Right" null,(empty,empty)⟩ ⇒
⟨-, -⟩

dynCastViaSh = progDiamond,[V ↦ Class "TopSh"] ⊢
  ⟨"V" := new "Right" ;; Cast "Right" (Var "V"),(empty,empty)⟩ ⇒ ⟨-, -⟩

dynCastFail = progDiamond,[V ↦ Class "TopRep"] ⊢
  ⟨"V" := new "Right" ;; Cast "Bottom" (Var "V"),(empty,empty)⟩ ⇒ ⟨-, -⟩

dynCastSide = progDiamond,[V ↦ Class "Right"] ⊢
  ⟨"V" := new "Bottom" ;; Cast "Left" (Var "V"),(empty,empty)⟩ ⇒ ⟨-, -⟩

Bottom = "Bottom"
Left = "Left"
TopSh = "TopSh"
TopRep = "TopRep"

```

⟨ML⟩

constdefs

— failing example

```
classD :: cdecl
classD == ("D", [Shares "A", Shares "B", Repeats "C"],[],[])
```

```
classC :: cdecl
classC == ("C", [Shares "A", Shares "B"],[],
  [("f",[],Integer,[],Val(Intg 42))])
```

```
classB :: cdecl
classB == ("B", [],[],
  [("f",[],Integer,[],Val(Intg 17))])
```

```
classA :: cdecl
classA == ("A", [],[],
  [("f",[],Integer,[],Val(Intg 13))])
```

```
ProgFailing :: cdecl list
ProgFailing == [classA,classB,classC,classD]
```

code-module Fail**contains**

```
callFailGplusplus = ProgFailing,empty ⊢
  {{"V":Class "D"; "V" := new "D";; Var "V"."f"([])},(empty,empty)}
  ⇒ ⟨-,⟩
```

⟨ML⟩

end

References

- [1] Daniel Wasserrab, Tobias Nipkow, Gregor Snelting, and Frank Tip. An operational semantics and type safety proof for multiple inheritance in C++. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming languages, systems, and applications*, pages 345–362. ACM Press, 2006.