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/

February 23, 2021

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

theory *Auxiliary*
imports *Complex-Main HOL−Library.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*]:
$(Suc(n + max\ i\ j) \leq m) = (Suc(n + i) \leq m \wedge Suc(n + j) \leq m)$
⟨*proof*⟩

notation *Some* $((\lfloor\text{-}\rfloor))$

lemma *butlast-tail*:
$butlast\ (Xs@[X, Y]) = Xs@[X]$
⟨*proof*⟩

lemma *butlast-noteq*:$Cs \neq [] \Longrightarrow butlast\ Cs \neq Cs$
⟨*proof*⟩

lemma *app-hd-tl*:$[\![Cs \neq []; Cs = Cs' @ tl\ Cs]\!] \Longrightarrow Cs' = [hd\ Cs]$

⟨*proof*⟩

lemma *only-one-append*:$[\![C' \notin set\ Cs; C' \notin set\ Cs'; Ds@\ C'\#Ds' = Cs@\ C'\#Cs']\!]$

$\Longrightarrow Cs = Ds \wedge Cs' = Ds'$

⟨*proof*⟩

definition *pick* :: *'a set* ⇒ *'a* **where**
$pick\ A \equiv SOME\ x.\ x \in A$

lemma *pick-is-element*:$x \in A \Longrightarrow pick\ A \in A$
⟨*proof*⟩

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

lemma *card-pick*:⟦*finite A*; $A \neq \{\}$⟧ ⟹ $Suc(card(A-\{pick(A)\})) = card\ A$
⟨*proof*⟩

lemma *set2list-prop*:⟦*finite A*; $A \neq \{\}$⟧ ⟹
$\exists xs.$ *while* $(\lambda(Es,S).\ S \neq \{\})$
$(\lambda(Es,S).$ *let* $x =$ *pick* S *in* $(x\#Es,S-\{x\}))$
$([],A) = (xs,\{\}) \wedge (set\ xs \cup \{\} = A)$

⟨*proof*⟩

lemma *set2list-correct*:⟦*finite A*; $A \neq \{\}$; *set2list* $A = xs$⟧ ⟹ *set* $xs = A$
⟨*proof*⟩

1.1 *distinct-fst*

definition *distinct-fst* :: $('a \times 'b)$ *list* ⇒ *bool* **where**
distinct-fst ≡ *distinct* ∘ *map fst*

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

⟨*proof*⟩

lemma *distinct-fst-Cons* [*simp*]:
distinct-fst $((k,x)\#kxs) =$ (*distinct-fst kxs* $\wedge$ $(\forall y.\ (k,y) \notin set\ kxs)$)

⟨*proof*⟩

lemma *map-of-SomeI*:
⟦ *distinct-fst kxs*; $(k,x) \in set\ kxs$ ⟧ ⟹ *map-of kxs k* = *Some x*
⟨*proof*⟩

1.2 Using *list-all2* for relations

definition *fun-of* :: $('a \times 'b)$ *set* ⇒ $'a$ ⇒ $'b$ ⇒ *bool* **where**
fun-of $S \equiv \lambda x\ y.\ (x,y) \in S$

Convenience lemmas

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

lemma *rel-list-all2-Cons* [*iff*]:
list-all2 (*fun-of S*) $(x\#xs)$ $(y\#ys)$ =
$((x,y) \in S \wedge$ *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** *Auxiliary* **begin**

type-synonym *cname* = *string* — class names

type-synonym *mname* = *string* — method name
type-synonym *vname* = *string* — names for local/field variables

definition *this* :: *vname* **where**
  this ≡ ''*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

primrec *getbase* :: *base* ⇒ *cname* **where**
  getbase (*Repeats C*) = *C*
| *getbase* (*Shares C*) = *C*

primrec *isRepBase* :: *base* ⇒ *bool* **where**
  isRepBase (*Repeats C*) = *True*
| *isRepBase* (*Shares C*) = *False*

primrec *isShBase* :: *base* ⇒ *bool* **where**
  isShBase(*Repeats C*) = *False*
| *isShBase*(*Shares C*) = *True*

definition *is-refT* :: *ty* ⇒ *bool* **where**
  is-refT T ≡ *T* = *NT* ∨ (∃ *C*. *T* = *Class C*)

lemma [*iff*]: *is-refT NT*
⟨*proof*⟩

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

lemma *refTE*:
  ⟦*is-refT T*; *T* = *NT* ⟹ *Q*; ⋀*C*. *T* = *Class C* ⟹ *Q* ⟧ ⟹ *Q*
⟨*proof*⟩

lemma *not-refTE*:
  ⟦ ¬*is-refT T*; *T* = *Void* ∨ *T* = *Boolean* ∨ *T* = *Integer* ⟹ *Q* ⟧ ⟹ *Q*
⟨*proof*⟩

type-synonym
  env = *vname* ⇀ *ty*

end

3 CoreC++ values

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

type-synonym *addr* = *nat*
type-synonym *path* = *cname list* — Path-component in subobjects
type-synonym *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

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

primrec *the-addr* :: *val* ⇒ *addr* **where**
the-addr (*Ref r*) = *fst r*

primrec *the-path* :: *val* ⇒ *path* **where**
the-path (*Ref r*) = *snd r*

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

primrec *typeof* :: *val* ⇒ *ty option* **where**
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*) = (∃ *b*. *v* = *Bool b*)
⟨*proof*⟩

lemma [*simp*]: (*typeof v* = *Some Integer*) = (∃ *i*. *v* = *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
(⦇-⦈- [*80*,*81*] *80*)
| *Val val* — value
| *BinOp expr bop expr* (- «-» - [*80*,*0*,*81*] *80*)
— binary operation
| *Var vname* — local variable
| *LAss vname expr* (-:=- [*70*,*70*] *70*)
— local assignment
| *FAcc expr vname path* (-•-{-} [*10*,*90*,*99*] *90*)
— field access
| *FAss expr vname path expr* (-•-{-} := - [*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* ⇒ *mname* ⇒ *expr list* ⇒ *expr* (-•-′(-′) [*90*,*99*,*0*] *90*) **where**
e•*M*(*es*) == *Call e None M es*

abbreviation (*input*)
StaticCall :: *expr* ⇒ *cname* ⇒ *mname* ⇒ *expr list* ⇒ *expr*
(-•′(-::′)-′(-′) [*90*,*99*,*99*,*0*] *90*) **where**
e•(*C*::)*M*(*es*) == *Call e* (*Some C*) *M es*

The semantics of binary operators:

fun *binop* :: *bop* × *val* × *val* ⇒ *val option* **where**
  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*]:
  (*binop*(*Add*,v_1,v_2) = *Some* *v*) = (∃ i_1 i_2. v_1 = *Intg* i_1 ∧ v_2 = *Intg* i_2 ∧ *v* = *Intg*(i_1+i_2))
⟨*proof*⟩

lemma *binop-not-ref*[*simp*]:
  binop(*bop*,v_1,v_2) = *Some* (*Ref* *r*) ⟹ *False*
⟨*proof*⟩

4.2 Free Variables

primrec
  fv :: *expr* ⇒ *vname set*
  and *fvs* :: *expr list* ⇒ *vname set* **where**
  fv(*new* *C*) = {}
| *fv*(*Cast* *C* *e*) = *fv* *e*
| *fv*(⦇*C*⦈*e*) = *fv* *e*
| *fv*(*Val* *v*) = {}
| *fv*(e_1 «*bop*» e_2) = *fv* e_1 ∪ *fv* e_2
| *fv*(*Var* *V*) = {*V*}
| *fv*(*V* := *e*) = {*V*} ∪ *fv* *e*
| *fv*(*e*·*F*{*Cs*}) = *fv* *e*
| *fv*(e_1·*F*{*Cs*}:=e_2) = *fv* e_1 ∪ *fv* e_2
| *fv*(*Call* *e* *Copt* *M* *es*) = *fv* *e* ∪ *fvs* *es*
| *fv*({*V*:*T*; *e*}) = *fv* *e* − {*V*}
| *fv*(e_1;;e_2) = *fv* e_1 ∪ *fv* e_2
| *fv*(*if* (*b*) e_1 *else* e_2) = *fv* *b* ∪ *fv* e_1 ∪ *fv* e_2
| *fv*(*while* (*b*) *e*) = *fv* *b* ∪ *fv* *e*
| *fv*(*throw* *e*) = *fv* *e*

| *fvs*([]) = {}
| *fvs*(*e*#*es*) = *fv* *e* ∪ *fvs* *es*

lemma [*simp*]: *fvs*(es_1 @ es_2) = *fvs* es_1 ∪ *fvs* es_2
⟨*proof*⟩

lemma [*simp*]: *fvs*(*map* *Val* *vs*) = {}
⟨*proof*⟩

end

5 Class Declarations and Programs

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

type-synonym
fdecl = *vname* × *ty* — field declaration
type-synonym
method = *ty list* × *ty* × (*vname list* × *expr*) — arg. types, return type, params, body
type-synonym
mdecl = *mname* × *method* — method declaration
type-synonym
class = *base list* × *fdecl list* × *mdecl list* — class = superclasses, fields, methods
type-synonym
cdecl = *cname* × *class* — classa declaration
type-synonym
prog = *cdecl list* — program

translations
(*type*) *fdecl* <= (*type*) *vname* × *ty*
(*type*) *mdecl* <= (*type*) *mname* × *ty list* × *ty* × (*vname list* × *expr*)
(*type*) *class* <= (*type*) *cname* × *fdecl list* × *mdecl list*
(*type*) *cdecl* <= (*type*) *cname* × *class*
(*type*) *prog* <= (*type*) *cdecl list*

definition *class* :: *prog* ⇒ *cname* ⇀ *class* **where**
class ≡ *map-of*

definition *is-class* :: *prog* ⇒ *cname* ⇒ *bool* **where**
is-class P C ≡ *class P C* ≠ *None*

definition *baseClasses* :: *base list* ⇒ *cname set* **where**
baseClasses Bs ≡ *set* ((*map getbase*) *Bs*)

definition *RepBases* :: *base list* ⇒ *cname set* **where**
RepBases Bs ≡ *set* ((*map getbase*) (*filter isRepBase Bs*))

definition *SharedBases* :: *base list* ⇒ *cname set* **where**
SharedBases Bs ≡ *set* ((*map getbase*) (*filter isShBase Bs*))

lemma *not-getbase-repeats*:
D ∉ *set* (*map getbase xs*) ⟹ *Repeats D* ∉ *set xs*
⟨*proof*⟩

lemma *not-getbase-shares*:

$D \notin set\ (map\ getbase\ xs) \Longrightarrow Shares\ D \notin set\ xs$
⟨*proof*⟩

lemma *RepBaseclass-isBaseclass*:
$[\![class\ P\ C = Some(Bs,fs,ms);\ Repeats\ D \in set\ Bs]\!]$
$\Longrightarrow D \in baseClasses\ Bs$
⟨*proof*⟩

lemma *ShBaseclass-isBaseclass*:
$[\![class\ P\ C = Some(Bs,fs,ms);\ Shares\ D \in set\ Bs]\!]$
$\Longrightarrow D \in baseClasses\ Bs$
⟨*proof*⟩

lemma *base-repeats-or-shares*:
$[\![B \in set\ Bs;\ D = getbase\ B]\!]$
$\Longrightarrow Repeats\ D \in set\ Bs \vee Shares\ D \in set\ Bs$
⟨*proof*⟩

lemma *baseClasses-repeats-or-shares*:
$D \in baseClasses\ Bs \Longrightarrow Repeats\ D \in set\ Bs \vee Shares\ D \in set\ Bs$
⟨*proof*⟩

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

⟨*proof*⟩

lemma *finite-baseClasses*:
$class\ P\ C = Some(Bs,fs,ms) \Longrightarrow finite\ (baseClasses\ Bs)$

⟨*proof*⟩

definition *is-type* :: *prog* ⇒ *ty* ⇒ *bool* **where**
is-type P T ≡
(*case T of Void* ⇒ *True* | *Boolean* ⇒ *True* | *Integer* ⇒ *True* | *NT* ⇒ *True*
| *Class C* ⇒ *is-class P C*)

lemma *is-type-simps* [*simp*]:
is-type P Void ∧ *is-type P Boolean* ∧ *is-type P Integer* ∧
is-type P NT ∧ *is-type P* (*Class C*) = *is-class P C*
⟨*proof*⟩

abbreviation
types P == *Collect* (*CONST is-type P*)

lemma *typeof-lit-is-type*:
typeof v = Some T ⟹ is-type P T
⟨*proof*⟩

end

6 The subclass relation

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

— direct repeated subclass
inductive-set
subclsR :: prog ⇒ (cname × cname) set
and *subclsR′ :: prog ⇒ [cname, cname] ⇒ bool (- ⊢ - ≺R - [71,71,71] 70)*
for *P :: prog*
where
P ⊢ C ≺R D ≡ (C,D) ∈ subclsR P
| *subclsRI*: ⟦*class P C = Some (Bs,rest)*; *Repeats(D) ∈ set Bs*⟧ ⟹ *P ⊢ C ≺R D*

— direct shared subclass
inductive-set
subclsS :: prog ⇒ (cname × cname) set
and *subclsS′ :: prog ⇒ [cname, cname] ⇒ bool (- ⊢ - ≺S - [71,71,71] 70)*
for *P :: prog*
where
P ⊢ C ≺S D ≡ (C,D) ∈ subclsS P
| *subclsSI*: ⟦*class P C = Some (Bs,rest)*; *Shares(D) ∈ set Bs*⟧ ⟹ *P ⊢ C ≺S D*

— direct subclass
inductive-set
subcls1 :: prog ⇒ (cname × cname) set
and *subcls1′ :: prog ⇒ [cname, cname] ⇒ bool (- ⊢ - ≺1 - [71,71,71] 70)*
for *P :: prog*
where
P ⊢ C ≺1 D ≡ (C,D) ∈ subcls1 P
| *subcls1I*: ⟦*class P C = Some (Bs,rest)*; *D ∈ baseClasses Bs*⟧ ⟹ *P ⊢ C ≺1 D*

abbreviation
subcls :: prog ⇒ [cname, cname] ⇒ bool (- ⊢ - ≼ - [71,71,71] 70)* **where**
P ⊢ C ≼ D ≡ (C,D) ∈ (subcls1 P)**

lemma *subclsRD*:
P ⊢ C ≺R D ⟹ ∃fs ms Bs. (class P C = Some (Bs,fs,ms)) ∧ (Repeats(D) ∈ set Bs)
⟨*proof*⟩

lemma *subclsSD*:
$P \vdash C \prec_S D \Longrightarrow \exists fs\ ms\ Bs.\ (class\ P\ C = Some\ (Bs,fs,ms)) \wedge (Shares(D) \in set\ Bs)$
⟨*proof*⟩

lemma *subcls1D*:
$P \vdash C \prec^1 D \Longrightarrow \exists fs\ ms\ Bs.\ (class\ P\ C = Some\ (Bs,fs,ms)) \wedge (D \in baseClasses\ Bs)$
⟨*proof*⟩

lemma *subclsR-subcls1*:
$P \vdash C \prec_R D \Longrightarrow P \vdash C \prec^1 D$
⟨*proof*⟩

lemma *subclsS-subcls1*:
$P \vdash C \prec_S D \Longrightarrow P \vdash C \prec^1 D$
⟨*proof*⟩

lemma *subcls1-subclsR-or-subclsS*:
$P \vdash C \prec^1 D \Longrightarrow P \vdash C \prec_R D \vee P \vdash C \prec_S D$
⟨*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*:
$P \vdash C \prec^1 D \Longrightarrow is\text{-}class\ P\ C$
⟨*proof*⟩

lemma *subcls-is-class*:
$[\![P \vdash D \preceq^* C;\ is\text{-}class\ P\ C]\!] \Longrightarrow is\text{-}class\ P\ D$
⟨*proof*⟩

end

7 Definition of Subobjects

theory *SubObj*
imports *ClassRel*
begin

7.1 General definitions

type-synonym
subobj = *cname* × *path*

definition *mdc* :: *subobj* ⇒ *cname* **where**
mdc S = *fst S*

definition *ldc* :: *subobj* ⇒ *cname* **where**
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

fun *is-subobj* :: *prog* ⇒ *subobj* ⇒ *bool* — legal subobject to class hierarchie **where**
is-subobj P (*C*, []) ⟷ *False*
| *is-subobj P* (*C*, [*D*]) ⟷ (*is-class P C* ∧ *C* = *D*)
∨ (∃ *X*. *P* ⊢ *C* ≼* *X* ∧ *P* ⊢ *X* ≺$_S$ *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*) ∧ *P* ⊢ *X* ≺$_R$ *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 $\Longrightarrow$ *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-subobj P* ((*C*,*Xs*@[*X*, *Y*]))

⟨*proof*⟩

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

⟨*proof*⟩

Auxiliary lemmas

lemma *build-rec-isSubobj-rev*:
assumes *1*:*is-subobj P* ((*D*,*D*#*rev Cs*)) **and** *2*: $P \vdash C \prec_R D$
shows *is-subobj P* ((*C*,*C*#*D*#*rev Cs*))
⟨*proof*⟩

lemma *build-rec-isSubobj*:
assumes *1*:*is-subobj P* ((*D*,*D*#*Cs*)) **and** *2*: $P \vdash C \prec_R D$
shows *is-subobj P* ((*C*,*C*#*D*#*Cs*))

⟨*proof*⟩

lemma *isSubobj-isSubobj-isSubobj-rev*:
assumes *1*:*is-subobj P* ((*C*,[*D*])) **and** *2*:*is-subobj P* ((*D*,*D*#(*rev Cs*)))
shows *is-subobj P* ((*C*,*D*#(*rev Cs*)))
⟨*proof*⟩

lemma *isSubobj-isSubobj-isSubobj*:
assumes *1*:*is-subobj P* ((*C*,[*D*])) **and** *2*:*is-subobj P* ((*D*,*D*#*Cs*))
shows *is-subobj P* ((*C*,*D*#*Cs*))

⟨*proof*⟩

7.3 Subobject handling and lemmas

Subobjects consisting of repeated inheritance relations only:

inductive $Subobjs_R$ :: *prog* ⇒ *cname* ⇒ *path* ⇒ *bool* **for** *P* :: *prog*
where
  SubobjsR-Base: *is-class P C* ⟹ $Subobjs_R$ *P C* [*C*]
| *SubobjsR-Rep*: ⟦$P \vdash C \prec_R D$; $Subobjs_R$ *P D Cs*⟧ ⟹ $Subobjs_R$ *P C* (*C* # *Cs*)

All subobjects:

inductive *Subobjs* :: *prog* ⇒ *cname* ⇒ *path* ⇒ *bool* **for** *P* :: *prog*
where
  Subobjs-Rep: $Subobjs_R$ *P C Cs* ⟹ *Subobjs P C Cs*
| *Subobjs-Sh*: ⟦$P \vdash C \preceq^* C'$; $P \vdash C' \prec_S D$; $Subobjs_R$ *P D Cs*⟧
        ⟹ *Subobjs P C Cs*

lemma *Subobjs-Base*:*is-class P C* ⟹ *Subobjs P C* [*C*]

⟨*proof*⟩

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

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

lemma *hd-SubobjsR*:
$Subobjs_R\ P\ C\ Cs \Longrightarrow \exists\, Cs'.\ Cs = C\#Cs'$
⟨*proof*⟩

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

⟨*proof*⟩

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

⟨*proof*⟩

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

⟨*proof*⟩

lemma *Subobjs-notSubobjsR*:
$[\![Subobjs\ P\ C\ Cs;\ \neg\ Subobjs_R\ P\ C\ Cs]\!]$
$\Longrightarrow \exists\, C'\ D.\ P \vdash C \preceq^* C' \wedge P \vdash C' \prec_S D \wedge Subobjs_R\ P\ D\ Cs$
⟨*proof*⟩

lemma assumes *subo*:$Subobjs_R\ P\ (hd\ (Cs@\ C'\#Cs'))\ (Cs@\ C'\#Cs')$
shows *SubobjsR-Subobjs*:$Subobjs\ P\ C'\ (C'\#Cs')$
⟨*proof*⟩

lemma *Subobjs-Subobjs*:$Subobjs\ P\ C\ (Cs@\ C'\#Cs') \Longrightarrow Subobjs\ P\ C'\ (C'\#Cs')$

⟨*proof*⟩

lemma *SubobjsR-isClass*:
assumes *subo*:$Subobjs_R\ P\ C\ Cs$
shows *is-class* $P\ C$

⟨*proof*⟩

lemma *Subobjs-isClass*:
assumes *subo*:*Subobjs* $P\ C\ Cs$
shows *is-class* $P\ C$

⟨*proof*⟩

lemma *Subobjs-subclsR*:
assumes *subo*:*Subobjs* $P\ C\ (Cs@[D,D']@Cs')$
shows $P \vdash D \prec_R D'$

⟨*proof*⟩

lemma assumes *subo*:$Subobjs_R\ P\ (hd\ Cs)\ (Cs@[D])$ **and** *notempty*:$Cs \neq []$
shows *butlast-Subobjs-Rep*:$Subobjs_R\ P\ (hd\ Cs)\ Cs$
⟨*proof*⟩

lemma assumes *subo*:*Subobjs* $P\ C\ (Cs@[D])$ **and** *notempty*:$Cs \neq []$
shows *butlast-Subobjs*:*Subobjs* $P\ C\ Cs$

⟨*proof*⟩

lemma assumes *subo*:*Subobjs* $P\ C\ (Cs@(rev\ Cs'))$ **and** *notempty*:$Cs \neq []$
shows *rev-appendSubobj*:*Subobjs* $P\ C\ Cs$
⟨*proof*⟩

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* $\Longrightarrow$ *is-subobj P* ((*C*,*Cs*))
⟨*proof*⟩

lemma *leq-SubobjsR-isSubobj*:
⟦*P* ⊢ *C* $\preceq^*$ *C*′; *P* ⊢ *C*′ $\prec_S$ *D*; *Subobjs*$_R$ *P D Cs*⟧
$\Longrightarrow$ *is-subobj P* ((*C*,*Cs*))

⟨*proof*⟩

lemma *Subobjs-isSubobj*:
Subobjs P C Cs $\Longrightarrow$ *is-subobj P* ((*C*,*Cs*))
⟨*proof*⟩

7.4 Paths

7.5 Appending paths

Avoided name clash by calling one path Path.

definition *path-via* :: *prog* ⇒ *cname* ⇒ *cname* ⇒ *path* ⇒ *bool* (- ⊢ *Path* - *to* - *via* - [*51*,*51*,*51*,*51*] *50*) **where**
P ⊢ *Path C to D via Cs* ≡ *Subobjs P C Cs* ∧ *last Cs* = *D*

definition *path-unique* :: *prog* ⇒ *cname* ⇒ *cname* ⇒ *bool* (- ⊢ *Path* - *to* - *unique* [*51*,*51*,*51*] *50*) **where**
P ⊢ *Path C to D unique* ≡ ∃!*Cs*. *Subobjs P C Cs* ∧ *last Cs* = *D*

definition *appendPath* :: *path* ⇒ *path* ⇒ *path* (**infixr** @$_p$ *65*) **where**
Cs @$_p$ *Cs*′ ≡ *if* (*last Cs* = *hd Cs*′) *then Cs* @ (*tl Cs*′) *else Cs*′

lemma *appendPath-last*: *Cs* ≠ [] $\Longrightarrow$ *last Cs* = *last* (*Cs*′@$_p$*Cs*)
⟨*proof*⟩

inductive
casts-to :: *prog* ⇒ *ty* ⇒ *val* ⇒ *val* ⇒ *bool*
(- ⊢ - *casts* - *to* - [*51*,*51*,*51*,*51*] *50*)
for *P* :: *prog*
where

casts-prim: ∀ *C*. *T* ≠ *Class C* $\Longrightarrow$ *P* ⊢ *T casts v to v*

| *casts-null*: *P* ⊢ *Class C casts Null to Null*

| *casts-ref*: ⟦ $P \vdash Path\ last\ Cs\ to\ C\ via\ Cs'$; $Ds = Cs@_p Cs'$ ⟧
  ⟹ $P \vdash Class\ C\ casts\ Ref(a,Cs)\ to\ Ref(a,Ds)$

inductive
  Casts-to :: *prog* ⇒ *ty list* ⇒ *val list* ⇒ *val list* ⇒ *bool*
    (- ⊢ - *Casts* - *to* - [51,51,51,51] 50)
  for P :: *prog*
where

  Casts-Nil: $P \vdash [] \ Casts\ [] \ to\ []$

| *Casts-Cons*: ⟦ $P \vdash T\ casts\ v\ to\ v'$; $P \vdash Ts\ Casts\ vs\ to\ vs'$ ⟧
  ⟹ $P \vdash (T\#Ts)\ Casts\ (v\#vs)\ to\ (v'\#vs')$

lemma *length-Casts-vs*:
  $P \vdash Ts\ Casts\ vs\ to\ vs' \Longrightarrow length\ Ts = length\ vs$
⟨*proof*⟩

lemma *length-Casts-vs'*:
  $P \vdash Ts\ Casts\ vs\ to\ vs' \Longrightarrow length\ Ts = length\ vs'$
⟨*proof*⟩

7.6 The relation on paths

inductive-set
  leq-path1 :: *prog* ⇒ *cname* ⇒ (*path* × *path*) *set*
  and *leq-path1'* :: *prog* ⇒ *cname* ⇒ [*path*, *path*] ⇒ *bool* (-,- ⊢ - $\sqsubset^1$ - [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-pathRep*: ⟦ *Subobjs P C Cs*; *Subobjs P C Ds*; $Cs = butlast\ Ds$⟧
  ⟹ $P,C \vdash Cs \sqsubset^1 Ds$
| *leq-pathSh*: ⟦ *Subobjs P C Cs*; $P \vdash last\ Cs \prec_S D$ ⟧
  ⟹ $P,C \vdash Cs \sqsubset^1 [D]$

abbreviation
  leq-path :: *prog* ⇒ *cname* ⇒ [*path*, *path*] ⇒ *bool* (-,- ⊢ - $\sqsubseteq$ - [71,71,71] 70) **where**
  $P,C \vdash Cs \sqsubseteq Ds \equiv (Cs,Ds) \in (leq\text{-}path1\ P\ C)^*$

lemma *leq-path-rep*:
  ⟦ *Subobjs P C* ($Cs@[C']$); *Subobjs P C* ($Cs@[C',C'']$)⟧
⟹ $P,C \vdash (Cs@[C']) \sqsubset^1 (Cs@[C',C''])$
⟨*proof*⟩

lemma *leq-path-sh*:
$[\![\mathit{Subobjs}\ P\ C\ (Cs@[C']);\ P \vdash C' \prec_S C'']\!]$
$\Longrightarrow P,C \vdash (Cs@[C']) \sqsubset^1 [C'']$
⟨*proof*⟩

7.7 Member lookups

definition *FieldDecls* :: *prog* ⇒ *cname* ⇒ *vname* ⇒ (*path* × *ty*) *set* **where**
FieldDecls P C F ≡
{(*Cs*,*T*). *Subobjs P C Cs* ∧ (∃ *Bs fs ms*. *class P* (*last Cs*) = *Some*(*Bs*,*fs*,*ms*)
∧ *map-of fs F* = *Some T*)}

definition *LeastFieldDecl* :: *prog* ⇒ *cname* ⇒ *vname* ⇒ *ty* ⇒ *path* ⇒ *bool*
(- ⊢ - *has least* -:- *via* - [51,0,0,0,51] 50) **where**
P ⊢ *C has least F*:*T via Cs* ≡
(*Cs*,*T*) ∈ *FieldDecls P C F* ∧
(∀ (*Cs'*,*T'*) ∈ *FieldDecls P C F*. *P*,*C* ⊢ *Cs* ⊑ *Cs'*)

definition *MethodDefs* :: *prog* ⇒ *cname* ⇒ *mname* ⇒ (*path* × *method*)*set* **where**
MethodDefs P C M ≡
{(*Cs*,*mthd*). *Subobjs P C Cs* ∧ (∃ *Bs fs ms*. *class P* (*last Cs*) = *Some*(*Bs*,*fs*,*ms*)
∧ *map-of ms M* = *Some mthd*)}

— needed for well formed criterion
definition *HasMethodDef* :: *prog* ⇒ *cname* ⇒ *mname* ⇒ *method* ⇒ *path* ⇒ *bool*
(- ⊢ - *has* - = - *via* - [51,0,0,0,51] 50) **where**
P ⊢ *C has M* = *mthd via Cs* ≡ (*Cs*,*mthd*) ∈ *MethodDefs P C M*

definition *LeastMethodDef* :: *prog* ⇒ *cname* ⇒ *mname* ⇒ *method* ⇒ *path* ⇒ *bool*
(- ⊢ - *has least* - = - *via* - [51,0,0,0,51] 50) **where**
P ⊢ *C has least M* = *mthd via Cs* ≡
(*Cs*,*mthd*) ∈ *MethodDefs P C M* ∧
(∀ (*Cs'*,*mthd'*) ∈ *MethodDefs P C M*. *P*,*C* ⊢ *Cs* ⊑ *Cs'*)

definition *MinimalMethodDefs* :: *prog* ⇒ *cname* ⇒ *mname* ⇒ (*path* × *method*)*set*
where
MinimalMethodDefs P C M ≡
{(*Cs*,*mthd*). (*Cs*,*mthd*) ∈ *MethodDefs P C M* ∧
(∀ (*Cs'*,*mthd'*)∈ *MethodDefs P C M*. *P*,*C* ⊢ *Cs'* ⊑ *Cs* ⟶ *Cs'* = *Cs*)}

definition *OverriderMethodDefs* :: *prog* ⇒ *subobj* ⇒ *mname* ⇒ (*path* × *method*)*set*
where
OverriderMethodDefs P R M ≡
{(*Cs*,*mthd*). ∃ *Cs' mthd'*. *P* ⊢ (*ldc R*) *has least M* = *mthd' via Cs'* ∧
(*Cs*,*mthd*) ∈ *MinimalMethodDefs P* (*mdc R*) *M* ∧
P,*mdc R* ⊢ *Cs* ⊑ (*snd R*)@$_p$*Cs'*}

definition *FinalOverriderMethodDef* :: *prog* ⇒ *subobj* ⇒ *mname* ⇒ *method* ⇒
path ⇒ *bool*

(- ⊢ - *has overrider* - = - *via* - [51,0,0,0,51] 50) **where**
$P \vdash R$ *has overrider* $M = mthd$ *via* $Cs \equiv$
$(Cs,mthd) \in OverriderMethodDefs\ P\ R\ M \wedge$
$card(OverriderMethodDefs\ P\ R\ M) = 1$

inductive
SelectMethodDef :: *prog* ⇒ *cname* ⇒ *path* ⇒ *mname* ⇒ *method* ⇒ *path* ⇒ *bool*
(- ⊢ '(-,-') *selects* - = - *via* - [51,0,0,0,0,51] 50)
for P :: *prog*
where

dyn-unique:
$P \vdash C$ *has least* $M = mthd$ *via* $Cs' \Longrightarrow P \vdash (C,Cs)$ *selects* $M = mthd$ *via* Cs'

| *dyn-ambiguous*:
⟦∀ *mthd* Cs'. ¬ $P \vdash C$ *has least* $M = mthd$ *via* Cs';
$P \vdash (C,Cs)$ *has overrider* $M = mthd$ *via* Cs'⟧
⟹ $P \vdash (C,Cs)$ *selects* $M = mthd$ *via* Cs'

lemma *sees-fields-fun*:
$(Cs,T) \in FieldDecls\ P\ C\ F \Longrightarrow (Cs,T') \in FieldDecls\ P\ C\ F \Longrightarrow T = T'$
⟨*proof*⟩

lemma *sees-field-fun*:
⟦$P \vdash C$ *has least* $F{:}T$ *via* Cs; $P \vdash C$ *has least* $F{:}T'$ *via* Cs⟧
⟹ $T = T'$
⟨*proof*⟩

lemma *has-least-method-has-method*:
$P \vdash C$ *has least* $M = mthd$ *via* $Cs \Longrightarrow P \vdash C$ *has* $M = mthd$ *via* Cs
⟨*proof*⟩

lemma *visible-methods-exist*:
$(Cs,mthd) \in MethodDefs\ P\ C\ M \Longrightarrow$
(∃ *Bs fs ms*. *class* P (*last* Cs) = *Some*(*Bs*,*fs*,*ms*) ∧ *map-of ms* M = *Some mthd*)
⟨*proof*⟩

lemma *sees-methods-fun*:
$(Cs,mthd) \in MethodDefs\ P\ C\ M \Longrightarrow (Cs,mthd') \in MethodDefs\ P\ C\ M \Longrightarrow mthd = mthd'$
⟨*proof*⟩

lemma *sees-method-fun*:
$[\![P \vdash C \text{ has least } M = mthd \text{ via } Cs;\ P \vdash C \text{ has least } M = mthd' \text{ via } Cs]\!]$
$\Longrightarrow mthd = mthd'$
⟨*proof*⟩

lemma *overrider-method-fun*:
assumes *overrider*:$P \vdash (C,Cs)$ *has overrider* $M = mthd$ *via* Cs'
and *overrider'*:$P \vdash (C,Cs)$ *has overrider* $M = mthd'$ *via* Cs''
shows $mthd = mthd' \wedge Cs' = Cs''$
⟨*proof*⟩

end

8 Objects and the Heap

theory *Objects* **imports** *SubObj* **begin**

8.1 Objects

type-synonym
subo = (*path* × (*vname* ⇀ *val*)) — subobjects realized on the heap
type-synonym
obj = *cname* × *subo set* — mdc and subobject

definition *init-class-fieldmap* :: *prog* ⇒ *cname* ⇒ (*vname* ⇀ *val*) **where**
init-class-fieldmap P C ≡
map-of (*map* (λ(*F*,*T*).(*F*,*default-val T*)) (*fst*(*snd*(*the*(*class P C*)))))

inductive
init-obj :: *prog* ⇒ *cname* ⇒ (*path* × (*vname* ⇀ *val*)) ⇒ *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* ≠ []
⟨*proof*⟩

lemma *init-obj-no-Ref*:
$[\![$*init-obj P C* (*Cs*,*fs*); *fs F* = *Some*(*Ref*(*a'*,*Cs'*))$]\!] \Longrightarrow$ *False*
⟨*proof*⟩

lemma *SubobjsSet-init-objSet*:
{*Cs*. *Subobjs P C Cs*} = {*Cs*. ∃ *vmap*. *init-obj P C* (*Cs*,*vmap*)}
⟨*proof*⟩

definition *obj-ty* :: *obj* ⇒ *ty* **where**
obj-ty obj ≡ *Class* (*fst obj*)

— a new, blank object with default values in all fields:
definition *blank* :: *prog* ⇒ *cname* ⇒ *obj* **where**
blank P C ≡ (*C*, *Collect* (*init-obj P C*))

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

8.2 Heap

type-synonym *heap* = *addr* ⇀ *obj*

abbreviation
cname-of :: *heap* ⇒ *addr* ⇒ *cname* **where**
cname-of hp a == *fst* (*the* (*hp a*))

definition *new-Addr* :: *heap* ⇒ *addr option* **where**
new-Addr h ≡ *if* ∃ *a*. *h a* = *None then Some*(*SOME a*. *h a* = *None*) *else None*

lemma *new-Addr-SomeD*:
new-Addr h = *Some a* ⟹ *h a* = *None*
⟨*proof*⟩

end

9 Exceptions

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

9.1 Exceptions

definition *NullPointer* :: *cname* **where**
NullPointer ≡ ''*NullPointer*''

definition *ClassCast* :: *cname* **where**
ClassCast ≡ ''*ClassCast*''

definition *OutOfMemory* :: *cname* **where**
OutOfMemory ≡ ''*OutOfMemory*''

definition *sys-xcpts* :: *cname set* **where**
sys-xcpts ≡ {*NullPointer*, *ClassCast*, *OutOfMemory*}

definition *addr-of-sys-xcpt* :: *cname* ⇒ *addr* **where**
addr-of-sys-xcpt s ≡ *if s* = *NullPointer then 0 else*
if s = *ClassCast then 1 else*
if s = *OutOfMemory then 2 else undefined*

definition *start-heap* :: *prog* ⇒ *heap* **where**
start-heap P ≡ *Map.empty* (*addr-of-sys-xcpt NullPointer* ↦ *blank P NullPointer*)
(*addr-of-sys-xcpt ClassCast* ↦ *blank P ClassCast*)
(*addr-of-sys-xcpt OutOfMemory* ↦ *blank P OutOfMemory*)

definition *preallocated* :: *heap* ⇒ *bool* **where**
preallocated h ≡ ∀ *C* ∈ *sys-xcpts*. ∃ *S*. *h* (*addr-of-sys-xcpt C*) = *Some* (*C*,*S*)

9.2 System exceptions

lemma [*simp*]:
NullPointer ∈ *sys-xcpts* ∧ *OutOfMemory* ∈ *sys-xcpts* ∧ *ClassCast* ∈ *sys-xcpts*
⟨*proof*⟩

lemma *sys-xcpts-cases* [*consumes 1*, *cases set*]:
⟦ *C* ∈ *sys-xcpts*; *P NullPointer*; *P OutOfMemory*; *P ClassCast*⟧ ⟹ *P C*
⟨*proof*⟩

9.3 *preallocated*

lemma *preallocated-dom* [*simp*]:
⟦ *preallocated h*; *C* ∈ *sys-xcpts* ⟧ ⟹ *addr-of-sys-xcpt C* ∈ *dom h*
⟨*proof*⟩

lemma *preallocatedD*:
⟦ *preallocated h*; *C* ∈ *sys-xcpts* ⟧ ⟹ ∃ *S*. *h* (*addr-of-sys-xcpt C*) = *Some* (*C*,*S*)
⟨*proof*⟩

lemma *preallocatedE* [*elim?*]:
⟦ *preallocated h*; *C* ∈ *sys-xcpts*; ⋀*S*. *h* (*addr-of-sys-xcpt C*) = *Some*(*C*,*S*) ⟹ *P h C*⟧
⟹ *P h C*
⟨*proof*⟩

lemma *cname-of-xcp* [*simp*]:
⟦ *preallocated h*; *C* ∈ *sys-xcpts* ⟧ ⟹ *cname-of h* (*addr-of-sys-xcpt C*) = *C*
⟨*proof*⟩

lemma *preallocated-start*:
preallocated (*start-heap P*)

⟨*proof*⟩

9.4 *start-heap*

lemma *start-Subobj*:
⟦*start-heap P a* = *Some*(*C*, *S*); (*Cs*,*fs*) ∈ *S*⟧ ⟹ *Subobjs P C Cs*
⟨*proof*⟩

lemma *start-SuboSet*:
⟦*start-heap P a* = *Some*(*C*, *S*); *Subobjs P C Cs*⟧ ⟹ ∃ *fs*. (*Cs*,*fs*) ∈ *S*
⟨*proof*⟩

lemma *start-init-obj*: *start-heap P a* = *Some*(*C*,*S*) ⟹ *S* = *Collect* (*init-obj P C*)
⟨*proof*⟩

lemma *start-subobj*:
⟦*start-heap P a* = *Some*(*C*, *S*); ∃ *fs*. (*Cs*, *fs*) ∈ *S*⟧ ⟹ *Subobjs P C Cs*
⟨*proof*⟩

end

10 Syntax

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

Syntactic sugar

abbreviation (*input*)
InitBlock :: *vname* ⇒ *ty* ⇒ *expr* ⇒ *expr* ⇒ *expr* (((*1*′{-:- := -;/ -})) **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* ⇒ *expr* **where**
Throw r == *throw*(*ref r*)

abbreviation (*input*)
THROW :: *cname* ⇒ *expr* **where**
THROW xc == *Throw*(*addr-of-sys-xcpt xc*,[*xc*])

end

11 Program State

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

type-synonym
locals = *vname* ⇀ *val* — local vars, incl. params and "this"
type-synonym
state = *heap* × *locals*

definition *hp* :: *state* ⇒ *heap* **where**
hp ≡ *fst*

definition *lcl* :: *state* ⇒ *locals* **where**
lcl ≡ *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* ⇒ *env* ⇒ *expr* ⇒ *state* ⇒ *expr* ⇒ *state* ⇒ *bool*
(-,- ⊢ ((1⟨-,/-⟩) ⇒/ (1⟨-,/-⟩))) [51,0,0,0,0] 81)
and *evals* :: *prog* ⇒ *env* ⇒ *expr list* ⇒ *state* ⇒ *expr list* ⇒ *state* ⇒ *bool*
(-,- ⊢ ((1⟨-,/-⟩) [⇒]/ (1⟨-,/-⟩))) [51,0,0,0,0] 81)
for *P* :: *prog*
where

New:
⟦ *new-Addr h* = *Some a*; *h'* = *h*(*a*↦(*C*,*Collect* (*init-obj P C*))) ⟧
⟹ *P*,*E* ⊢ ⟨*new C*,(*h*,*l*)⟩ ⇒ ⟨*ref* (*a*,[*C*]),(*h'*,*l*)⟩

| *NewFail*:
new-Addr h = *None* ⟹
P,*E* ⊢ ⟨*new C*, (*h*,*l*)⟩ ⇒ ⟨*THROW OutOfMemory*,(*h*,*l*)⟩

| *StaticUpCast*:
⟦ *P*,*E* ⊢ ⟨*e*,*s*$_0$⟩ ⇒ ⟨*ref* (*a*,*Cs*),*s*$_1$⟩; *P* ⊢ *Path last Cs to C via Cs'*; *Ds* = *Cs*@$_p$*Cs'* ⟧
⟹ *P*,*E* ⊢ ⟨(⦇*C*⦈)*e*,*s*$_0$⟩ ⇒ ⟨*ref* (*a*,*Ds*),*s*$_1$⟩

| *StaticDownCast*:
P,*E* ⊢ ⟨*e*,*s*$_0$⟩ ⇒ ⟨*ref* (*a*,*Cs*@[*C*]@*Cs'*),*s*$_1$⟩
⟹ *P*,*E* ⊢ ⟨(⦇*C*⦈)*e*,*s*$_0$⟩ ⇒ ⟨*ref* (*a*,*Cs*@[*C*]),*s*$_1$⟩

| *StaticCastNull*:

$P,E \vdash \langle e,s_0\rangle \Rightarrow \langle null,s_1\rangle \Longrightarrow$
$P,E \vdash \langle (\!| C |\!) e,s_0\rangle \Rightarrow \langle null,s_1\rangle$

| *StaticCastFail*:
$[\![P,E \vdash \langle e,s_0\rangle \Rightarrow \langle ref\ (a,Cs),s_1\rangle;\ \neg\ P \vdash (last\ Cs) \preceq^*\ C;\ C \notin set\ Cs\ ]\!]$
$\Longrightarrow P,E \vdash \langle (\!| C |\!) e,s_0\rangle \Rightarrow \langle THROW\ ClassCast,s_1\rangle$

| *StaticCastThrow*:
$P,E \vdash \langle e,s_0\rangle \Rightarrow \langle throw\ e',s_1\rangle \Longrightarrow$
$P,E \vdash \langle (\!| C |\!) e,s_0\rangle \Rightarrow \langle throw\ e',s_1\rangle$

| *StaticUpDynCast*:
$[\![P,E \vdash \langle e,s_0\rangle \Rightarrow \langle ref(a,Cs),s_1\rangle;\ P \vdash Path\ last\ Cs\ to\ C\ unique;$
$P \vdash Path\ last\ Cs\ to\ C\ via\ Cs';\ Ds = Cs@_p Cs'\ ]\!]$
$\Longrightarrow P,E \vdash \langle Cast\ C\ e,s_0\rangle \Rightarrow \langle ref(a,Ds),s_1\rangle$

| *StaticDownDynCast*:
$P,E \vdash \langle e,s_0\rangle \Rightarrow \langle ref\ (a,Cs@[C]@Cs'),s_1\rangle$
$\Longrightarrow P,E \vdash \langle Cast\ C\ e,s_0\rangle \Rightarrow \langle ref\ (a,Cs@[C]),s_1\rangle$

| *DynCast*:
$[\![P,E \vdash \langle e,s_0\rangle \Rightarrow \langle ref\ (a,Cs),(h,l)\rangle;\ h\ a = Some(D,S);$
$P \vdash Path\ D\ to\ C\ via\ Cs';\ P \vdash Path\ D\ to\ C\ unique\ ]\!]$
$\Longrightarrow P,E \vdash \langle Cast\ C\ e,s_0\rangle \Rightarrow \langle ref\ (a,Cs'),(h,l)\rangle$

| *DynCastNull*:
$P,E \vdash \langle e,s_0\rangle \Rightarrow \langle null,s_1\rangle \Longrightarrow$
$P,E \vdash \langle Cast\ C\ e,s_0\rangle \Rightarrow \langle null,s_1\rangle$

| *DynCastFail*:
$[\![P,E \vdash \langle e,s_0\rangle \Rightarrow \langle ref\ (a,Cs),(h,l)\rangle;\ h\ a = Some(D,S);\ \neg\ P \vdash Path\ D\ to\ C\ unique;$
$\neg\ P \vdash Path\ last\ Cs\ to\ C\ unique;\ C \notin set\ Cs\ ]\!]$
$\Longrightarrow P,E \vdash \langle Cast\ C\ e,s_0\rangle \Rightarrow \langle null,(h,l)\rangle$

| *DynCastThrow*:
$P,E \vdash \langle e,s_0\rangle \Rightarrow \langle throw\ e',s_1\rangle \Longrightarrow$
$P,E \vdash \langle Cast\ C\ e,s_0\rangle \Rightarrow \langle throw\ e',s_1\rangle$

| *Val*:
$P,E \vdash \langle Val\ v,s\rangle \Rightarrow \langle Val\ v,s\rangle$

| *BinOp*:
$[\![P,E \vdash \langle e_1,s_0\rangle \Rightarrow \langle Val\ v_1,s_1\rangle;\ P,E \vdash \langle e_2,s_1\rangle \Rightarrow \langle Val\ v_2,s_2\rangle;$
$binop(bop,v_1,v_2) = Some\ v\ ]\!]$
$\Longrightarrow P,E \vdash \langle e_1\ «bop»\ e_2,s_0\rangle \Rightarrow \langle Val\ v,s_2\rangle$

| *BinOpThrow1*:
$P,E \vdash \langle e_1,s_0\rangle \Rightarrow \langle throw\ e,s_1\rangle \Longrightarrow$
$P,E \vdash \langle e_1\ «bop»\ e_2,\ s_0\rangle \Rightarrow \langle throw\ e,s_1\rangle$

| *BinOpThrow2*:
$[\![\ P,E \vdash \langle e_1,s_0\rangle \Rightarrow \langle Val\ v_1,s_1\rangle;\ P,E \vdash \langle e_2,s_1\rangle \Rightarrow \langle throw\ e,s_2\rangle\ ]\!]$
$\Longrightarrow P,E \vdash \langle e_1\ «bop»\ e_2,s_0\rangle \Rightarrow \langle throw\ e,s_2\rangle$

| *Var*:
$l\ V = Some\ v \Longrightarrow$
$P,E \vdash \langle Var\ V,(h,l)\rangle \Rightarrow \langle Val\ v,(h,l)\rangle$

| *LAss*:
$[\![\ P,E \vdash \langle e,s_0\rangle \Rightarrow \langle Val\ v,(h,l)\rangle;\ E\ V = Some\ T;$
$P \vdash T\ casts\ v\ to\ v';\ l' = l(V \mapsto v')\ ]\!]$
$\Longrightarrow P,E \vdash \langle V:=e,s_0\rangle \Rightarrow \langle Val\ v',(h,l')\rangle$

| *LAssThrow*:
$P,E \vdash \langle e,s_0\rangle \Rightarrow \langle throw\ e',s_1\rangle \Longrightarrow$
$P,E \vdash \langle V:=e,s_0\rangle \Rightarrow \langle throw\ e',s_1\rangle$

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

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

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

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

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

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

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

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

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

| *Call*:
$[\![\ P,E \vdash \langle e,s_0\rangle \Rightarrow \langle \mathit{ref}\ (a,Cs),s_1\rangle;\ \ P,E \vdash \langle ps,s_1\rangle\ [\Rightarrow]\ \langle \mathit{map}\ \mathit{Val}\ vs,(h_2,l_2)\rangle;$
$h_2\ a = \mathit{Some}(C,S);\ \ P \vdash \mathit{last}\ Cs\ \mathit{has\ least}\ M = (Ts',T',pns',body')\ \mathit{via}\ Ds;$
$P \vdash (C,Cs@_pDs)\ \mathit{selects}\ M = (Ts,T,pns,body)\ \mathit{via}\ Cs';\ \mathit{length}\ vs = \mathit{length}\ pns;$
$P \vdash Ts\ \mathit{Casts}\ vs\ \mathit{to}\ vs';\ l_2' = [this{\mapsto}\mathit{Ref}\ (a,Cs'),\ pns[{\mapsto}]vs'];$
$\mathit{new\text{-}body} = (\mathit{case}\ T'\ \mathit{of}\ \mathit{Class}\ D \Rightarrow (\!|D|\!)body\ \ |\ \text{-}\ \Rightarrow body);$
$P,E(this{\mapsto}\mathit{Class}(\mathit{last}\ Cs'),\ pns[{\mapsto}]Ts) \vdash \langle \mathit{new\text{-}body},(h_2,l_2')\rangle \Rightarrow \langle e',(h_3,l_3)\rangle\ ]\!]$
$\Longrightarrow P,E \vdash \langle e{\cdot}M(ps),s_0\rangle \Rightarrow \langle e',(h_3,l_2)\rangle$

| *StaticCall*:
$[\![\ P,E \vdash \langle e,s_0\rangle \Rightarrow \langle \mathit{ref}\ (a,Cs),s_1\rangle;\ \ P,E \vdash \langle ps,s_1\rangle\ [\Rightarrow]\ \langle \mathit{map}\ \mathit{Val}\ vs,(h_2,l_2)\rangle;$
$P \vdash Path\ (\mathit{last}\ Cs)\ \mathit{to}\ C\ \mathit{unique};\ P \vdash Path\ (\mathit{last}\ Cs)\ \mathit{to}\ C\ \mathit{via}\ Cs'';$
$P \vdash C\ \mathit{has\ least}\ M = (Ts,T,pns,body)\ \mathit{via}\ Cs';\ Ds = (Cs@_pCs'')@_pCs';$
$\mathit{length}\ vs = \mathit{length}\ pns;\ P \vdash Ts\ \mathit{Casts}\ vs\ \mathit{to}\ vs';$
$l_2' = [this{\mapsto}\mathit{Ref}\ (a,Ds),\ pns[{\mapsto}]vs'];$
$P,E(this{\mapsto}\mathit{Class}(\mathit{last}\ Ds),\ pns[{\mapsto}]Ts) \vdash \langle body,(h_2,l_2')\rangle \Rightarrow \langle e',(h_3,l_3)\rangle\ ]\!]$
$\Longrightarrow P,E \vdash \langle e{\cdot}(C{::})M(ps),s_0\rangle \Rightarrow \langle e',(h_3,l_2)\rangle$

| *CallNull*:
$[\![\ P,E \vdash \langle e,s_0\rangle \Rightarrow \langle null,s_1\rangle;\ \ P,E \vdash \langle es,s_1\rangle\ [\Rightarrow]\ \langle \mathit{map}\ \mathit{Val}\ vs,s_2\rangle\ ]\!]$
$\Longrightarrow P,E \vdash \langle \mathit{Call}\ e\ \mathit{Copt}\ M\ es,s_0\rangle \Rightarrow \langle \mathit{THROW}\ \mathit{NullPointer},s_2\rangle$

| *Block*:
$[\![P,E(V\ {\mapsto}\ T) \vdash \langle e_0,(h_0,l_0(V{:=}None))\rangle \Rightarrow \langle e_1,(h_1,l_1)\rangle\ ]\!] \Longrightarrow$
$P,E \vdash \langle\{V{:}T;\ e_0\},(h_0,l_0)\rangle \Rightarrow \langle e_1,(h_1,l_1(V{:=}l_0\ V))\rangle$

| *Seq*:
$[\![\ P,E \vdash \langle e_0,s_0\rangle \Rightarrow \langle \mathit{Val}\ v,s_1\rangle;\ P,E \vdash \langle e_1,s_1\rangle \Rightarrow \langle e_2,s_2\rangle\ ]\!]$
$\Longrightarrow P,E \vdash \langle e_0;;e_1,s_0\rangle \Rightarrow \langle e_2,s_2\rangle$

| *SeqThrow*:
$P,E \vdash \langle e_0,s_0\rangle \Rightarrow \langle \mathit{throw}\ e,s_1\rangle \Longrightarrow$
$P,E \vdash \langle e_0;;e_1,s_0\rangle{\Rightarrow}\langle \mathit{throw}\ e,s_1\rangle$

| *CondT*:
$[\![\ P,E \vdash \langle e,s_0\rangle \Rightarrow \langle true,s_1\rangle;\ P,E \vdash \langle e_1,s_1\rangle \Rightarrow \langle e',s_2\rangle\ ]\!]$
$\Longrightarrow P,E \vdash \langle \mathit{if}\ (e)\ e_1\ \mathit{else}\ e_2,s_0\rangle \Rightarrow \langle e',s_2\rangle$

| *CondF*:

$[\![P,E \vdash \langle e,s_0\rangle \Rightarrow \langle false,s_1\rangle;\ P,E \vdash \langle e_2,s_1\rangle \Rightarrow \langle e',s_2\rangle\ ]\!]$
$\Longrightarrow P,E \vdash \langle if\ (e)\ e_1\ else\ e_2,s_0\rangle \Rightarrow \langle e',s_2\rangle$

| *CondThrow*:
$P,E \vdash \langle e,s_0\rangle \Rightarrow \langle throw\ e',s_1\rangle \Longrightarrow$
$P,E \vdash \langle if\ (e)\ e_1\ else\ e_2,\ s_0\rangle \Rightarrow \langle throw\ e',s_1\rangle$

| *WhileF*:
$P,E \vdash \langle e,s_0\rangle \Rightarrow \langle false,s_1\rangle \Longrightarrow$
$P,E \vdash \langle while\ (e)\ c,s_0\rangle \Rightarrow \langle unit,s_1\rangle$

| *WhileT*:
$[\![P,E \vdash \langle e,s_0\rangle \Rightarrow \langle true,s_1\rangle;\ P,E \vdash \langle c,s_1\rangle \Rightarrow \langle Val\ v_1,s_2\rangle;$
$P,E \vdash \langle while\ (e)\ c,s_2\rangle \Rightarrow \langle e_3,s_3\rangle\ ]\!]$
$\Longrightarrow P,E \vdash \langle while\ (e)\ c,s_0\rangle \Rightarrow \langle e_3,s_3\rangle$

| *WhileCondThrow*:
$P,E \vdash \langle e,s_0\rangle \Rightarrow \langle\ throw\ e',s_1\rangle \Longrightarrow$
$P,E \vdash \langle while\ (e)\ c,s_0\rangle \Rightarrow \langle throw\ e',s_1\rangle$

| *WhileBodyThrow*:
$[\![P,E \vdash \langle e,s_0\rangle \Rightarrow \langle true,s_1\rangle;\ P,E \vdash \langle c,s_1\rangle \Rightarrow \langle throw\ e',s_2\rangle]\!]$
$\Longrightarrow P,E \vdash \langle while\ (e)\ c,s_0\rangle \Rightarrow \langle throw\ e',s_2\rangle$

| *Throw*:
$P,E \vdash \langle e,s_0\rangle \Rightarrow \langle ref\ r,s_1\rangle \Longrightarrow$
$P,E \vdash \langle throw\ e,s_0\rangle \Rightarrow \langle Throw\ r,s_1\rangle$

| *ThrowNull*:
$P,E \vdash \langle e,s_0\rangle \Rightarrow \langle null,s_1\rangle \Longrightarrow$
$P,E \vdash \langle throw\ e,s_0\rangle \Rightarrow \langle THROW\ NullPointer,s_1\rangle$

| *ThrowThrow*:
$P,E \vdash \langle e,s_0\rangle \Rightarrow \langle throw\ e',s_1\rangle \Longrightarrow$
$P,E \vdash \langle throw\ e,s_0\rangle \Rightarrow \langle throw\ e',s_1\rangle$

| *Nil*:
$P,E \vdash \langle[],s\rangle\ [\Rightarrow]\ \langle[],s\rangle$

| *Cons*:
$[\![P,E \vdash \langle e,s_0\rangle \Rightarrow \langle Val\ v,s_1\rangle;\ P,E \vdash \langle es,s_1\rangle\ [\Rightarrow]\ \langle es',s_2\rangle\ ]\!]$
$\Longrightarrow P,E \vdash \langle e\#es,s_0\rangle\ [\Rightarrow]\ \langle Val\ v\ \#\ es',s_2\rangle$

| *ConsThrow*:
$P,E \vdash \langle e,\ s_0\rangle \Rightarrow \langle throw\ e',\ s_1\rangle \Longrightarrow$
$P,E \vdash \langle e\#es,\ s_0\rangle\ [\Rightarrow]\ \langle throw\ e'\ \#\ es,\ s_1\rangle$

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*]:
$P,E \vdash \langle new\ C,s\rangle \Rightarrow \langle e',s'\rangle$
$P,E \vdash \langle Cast\ C\ e,s\rangle \Rightarrow \langle e',s'\rangle$
$P,E \vdash \langle (\!|\, C\,|\!)e,s\rangle \Rightarrow \langle e',s'\rangle$
$P,E \vdash \langle Val\ v,s\rangle \Rightarrow \langle e',s'\rangle$
$P,E \vdash \langle e_1\ «bop»\ e_2,s\rangle \Rightarrow \langle e',s'\rangle$
$P,E \vdash \langle Var\ V,s\rangle \Rightarrow \langle e',s'\rangle$
$P,E \vdash \langle V:=e,s\rangle \Rightarrow \langle e',s'\rangle$
$P,E \vdash \langle e{\cdot}F\{Cs\},s\rangle \Rightarrow \langle e',s'\rangle$
$P,E \vdash \langle e_1{\cdot}F\{Cs\}:=e_2,s\rangle \Rightarrow \langle e',s'\rangle$
$P,E \vdash \langle e{\cdot}M(es),s\rangle \Rightarrow \langle e',s'\rangle$
$P,E \vdash \langle e{\cdot}(C::)M(es),s\rangle \Rightarrow \langle e',s'\rangle$
$P,E \vdash \langle \{V:T;e_1\},s\rangle \Rightarrow \langle e',s'\rangle$
$P,E \vdash \langle e_1;;e_2,s\rangle \Rightarrow \langle e',s'\rangle$
$P,E \vdash \langle if\ (e)\ e_1\ else\ e_2,s\rangle \Rightarrow \langle e',s'\rangle$
$P,E \vdash \langle while\ (b)\ c,s\rangle \Rightarrow \langle e',s'\rangle$
$P,E \vdash \langle throw\ e,s\rangle \Rightarrow \langle e',s'\rangle$

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

definition *final* :: *expr* ⇒ *bool* **where**
final e ≡ (∃ *v*. *e* = *Val v*) ∨ (∃ *r*. *e* = *Throw r*)

definition *finals*:: *expr list* ⇒ *bool* **where**
finals es ≡ (∃ *vs*. *es* = *map Val vs*) ∨ (∃ *vs r es'*. *es* = *map Val vs* @ *Throw r* # *es'*)

lemma [*simp*]: *final*(*Val v*)
⟨*proof*⟩

lemma [*simp*]: *final*(*throw e*) = (∃ *r*. *e* = *ref r*)
⟨*proof*⟩

lemma *finalE*: ⟦ *final e*; ⋀*v*. *e* = *Val v* ⟹ *Q*; ⋀*r*. *e* = *Throw r* ⟹ *Q* ⟧ ⟹ *Q*
⟨*proof*⟩

lemma [*iff*]: *finals* []
⟨*proof*⟩

lemma [*iff*]: *finals* (*Val v* # *es*) = *finals es*

⟨*proof*⟩

lemma *finals-app-map*[*iff*]: *finals* (*map Val vs* @ *es*) = *finals es*
⟨*proof*⟩

lemma [*iff*]: *finals* (*map Val vs*)
⟨*proof*⟩

lemma [*iff*]: *finals* (*throw e* # *es*) = (∃ *r*. *e* = *ref r*)

⟨*proof*⟩

lemma *not-finals-ConsI*: ¬ *final e* ⟹ ¬ *finals*(*e*#*es*)

⟨*proof*⟩

lemma *eval-final*: *P*,*E* ⊢ ⟨*e*,*s*⟩ ⇒ ⟨*e*′,*s*′⟩ ⟹ *final e*′
and *evals-final*: *P*,*E* ⊢ ⟨*es*,*s*⟩ [⇒] ⟨*es*′,*s*′⟩ ⟹ *finals es*′
⟨*proof*⟩

lemma *eval-lcl-incr*: *P*,*E* ⊢ ⟨*e*,(*h*$_0$,*l*$_0$)⟩ ⇒ ⟨*e*′,(*h*$_1$,*l*$_1$)⟩ ⟹ *dom l*$_0$ ⊆ *dom l*$_1$
and *evals-lcl-incr*: *P*,*E* ⊢ ⟨*es*,(*h*$_0$,*l*$_0$)⟩ [⇒] ⟨*es*′,(*h*$_1$,*l*$_1$)⟩ ⟹ *dom l*$_0$ ⊆ *dom l*$_1$
⟨*proof*⟩

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

lemma *eval-finalId*: *final e* ⟹ *P*,*E* ⊢ ⟨*e*,*s*⟩ ⇒ ⟨*e*,*s*⟩
⟨*proof*⟩

lemma *eval-finalsId*:
assumes *finals*: *finals es* **shows** *P*,*E* ⊢ ⟨*es*,*s*⟩ [⇒] ⟨*es*,*s*⟩

⟨*proof*⟩

lemma
eval-preserves-obj:*P*,*E* ⊢ ⟨*e*,(*h*,*l*)⟩ ⇒ ⟨*e*′,(*h*′,*l*′)⟩ ⟹ (⋀*S*. *h a* = *Some*(*D*,*S*)
⟹ ∃ *S*′. *h*′ *a* = *Some*(*D*,*S*′))
and *evals-preserves-obj*:*P*,*E* ⊢ ⟨*es*,(*h*,*l*)⟩ [⇒] ⟨*es*′,(*h*′,*l*′)⟩
⟹ (⋀*S*. *h a* = *Some*(*D*,*S*) ⟹ ∃ *S*′. *h*′ *a* = *Some*(*D*,*S*′))
⟨*proof*⟩

end

13 Small Step Semantics

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

13.1 Some pre-definitions

fun *blocks* :: *vname list* × *ty list* × *val list* × *expr* ⇒ *expr*
where
blocks-Cons:*blocks*(*V*#*Vs*, *T*#*Ts*, *v*#*vs*, *e*) = {*V*:*T* := *Val v*; *blocks*(*Vs*,*Ts*,*vs*,*e*)}
|
blocks-Nil: *blocks*([],[],[],*e*) = *e*

lemma *blocks-old-induct*:
fixes *P* :: *vname list* ⇒ *ty list* ⇒ *val list* ⇒ *expr* ⇒ *bool*
shows
⟦⋀*aj ak al*. *P* [] [] (*aj* # *ak*) *al*; ⋀*ad ae a b*. *P* [] (*ad* # *ae*) *a b*;
⋀*V Vs a b*. *P* (*V* # *Vs*) [] *a b*; ⋀*V Vs T Ts aw*. *P* (*V* # *Vs*) (*T* # *Ts*) [] *aw*;
⋀*V Vs T Ts v vs e*. *P Vs Ts vs e* ⟹ *P* (*V* # *Vs*) (*T* # *Ts*) (*v* # *vs*) *e*; ⋀*e*. *P* [] [] [] *e*⟧
⟹ *P u v w x*
⟨*proof*⟩

lemma [*simp*]:
⟦ *size vs* = *size Vs*; *size Ts* = *size Vs* ⟧ ⟹ *fv*(*blocks*(*Vs*,*Ts*,*vs*,*e*)) = *fv e* − *set Vs*

⟨*proof*⟩

definition *assigned* :: *vname* ⇒ *expr* ⇒ *bool* **where**
assigned V e ≡ ∃*v e*′. *e* = (*V*:= *Val v*;; *e*′)

13.2 The rules

inductive-set
red :: *prog* ⇒ (*env* × (*expr* × *state*) × (*expr* × *state*)) *set*
and *reds* :: *prog* ⇒ (*env* × (*expr list* × *state*) × (*expr list* × *state*)) *set*
and *red*′ :: *prog* ⇒ *env* ⇒ *expr* ⇒ *state* ⇒ *expr* ⇒ *state* ⇒ *bool*
(-,- ⊢ ((*1*⟨-,/-⟩) →/ (*1*⟨-,/-⟩))) [*51*,*0*,*0*,*0*,*0*] *81*)
and *reds*′ :: *prog* ⇒ *env* ⇒ *expr list* ⇒ *state* ⇒ *expr list* ⇒ *state* ⇒ *bool*
(-,- ⊢ ((*1*⟨-,/-⟩) [→]/ (*1*⟨-,/-⟩))) [*51*,*0*,*0*,*0*,*0*] *81*)
for *P* :: *prog*
where

P,*E* ⊢ ⟨*e*,*s*⟩ → ⟨*e*′,*s*′⟩ ≡ (*E*,(*e*,*s*), *e*′,*s*′) ∈ *red P*
| *P*,*E* ⊢ ⟨*es*,*s*⟩ [→] ⟨*es*′,*s*′⟩ ≡ (*E*,(*es*,*s*), *es*′,*s*′) ∈ *reds P*

| *RedNew*:
⟦ *new-Addr h* = *Some a*; *h*′ = *h*(*a*↦(*C*,*Collect* (*init-obj P C*))) ⟧
⟹ *P*,*E* ⊢ ⟨*new C*, (*h*,*l*)⟩ → ⟨*ref* (*a*,[*C*]), (*h*′,*l*)⟩

| *RedNewFail*:
new-Addr h = *None* ⟹
P,*E* ⊢ ⟨*new C*, (*h*,*l*)⟩ → ⟨*THROW OutOfMemory*, (*h*,*l*)⟩

| *StaticCastRed*:
$P,E \vdash \langle e,s\rangle \to \langle e',s'\rangle \Longrightarrow$
$P,E \vdash \langle (\!|\, C \,|\!)\, e,\ s\rangle \to \langle (\!|\, C \,|\!)\, e',\ s'\rangle$

| *RedStaticCastNull*:
$P,E \vdash \langle (\!|\, C \,|\!)\, null,\ s\rangle \to \langle null,s\rangle$

| *RedStaticUpCast*:
$[\![\ P \vdash Path\ last\ Cs\ to\ C\ via\ Cs';\ Ds = Cs @_p Cs'\ ]\!]$
$\Longrightarrow P,E \vdash \langle (\!|\, C \,|\!)(ref\ (a,Cs)),\ s\rangle \to \langle ref\ (a,Ds),\ s\rangle$

| *RedStaticDownCast*:
$P,E \vdash \langle (\!|\, C \,|\!)(ref\ (a,Cs@[C]@Cs')),\ s\rangle \to \langle ref\ (a,Cs@[C]),\ s\rangle$

| *RedStaticCastFail*:
$[\![C \notin set\ Cs;\ \neg\ P \vdash (last\ Cs) \preceq^*\ C]\!]$
$\Longrightarrow P,E \vdash \langle (\!|\, C \,|\!)(ref\ (a,Cs)),\ s\rangle \to \langle THROW\ ClassCast,\ s\rangle$

| *DynCastRed*:
$P,E \vdash \langle e,s\rangle \to \langle e',s'\rangle \Longrightarrow$
$P,E \vdash \langle Cast\ C\ e,\ s\rangle \to \langle Cast\ C\ e',\ s'\rangle$

| *RedDynCastNull*:
$P,E \vdash \langle Cast\ C\ null,\ s\rangle \to \langle null,s\rangle$

| *RedStaticUpDynCast*:
$[\![\ P \vdash Path\ last\ Cs\ to\ C\ unique;\ P \vdash Path\ last\ Cs\ to\ C\ via\ Cs';\ Ds = Cs @_p Cs'\ ]\!]$
$\Longrightarrow P,E \vdash \langle Cast\ C(ref(a,Cs)),s\rangle \to \langle ref(a,Ds),s\rangle$

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

| *RedDynCast*:
$[\![\ hp\ s\ a = Some(D,S);\ P \vdash Path\ D\ to\ C\ via\ Cs';$
$\quad P \vdash Path\ D\ to\ C\ unique\ ]\!]$
$\Longrightarrow P,E \vdash \langle Cast\ C\ (ref\ (a,Cs)),\ s\rangle \to \langle ref\ (a,Cs'),\ s\rangle$

| *RedDynCastFail*:
$[\![hp\ s\ a = Some(D,S);\ \neg\ P \vdash Path\ D\ to\ C\ unique;$
$\quad \neg\ P \vdash Path\ last\ Cs\ to\ C\ unique;\ C \notin set\ Cs\ ]\!]$
$\Longrightarrow P,E \vdash \langle Cast\ C\ (ref\ (a,Cs)),\ s\rangle \to \langle null,\ s\rangle$

| *BinOpRed1*:
$P,E \vdash \langle e,s\rangle \to \langle e',s'\rangle \Longrightarrow$
$P,E \vdash \langle e\ «bop»\ e_2,\ s\rangle \to \langle e'\ «bop»\ e_2,\ s'\rangle$

| *BinOpRed2*:
$P,E \vdash \langle e,s\rangle \to \langle e',s'\rangle \Longrightarrow$

$P,E \vdash \langle(Val\ v_1)\ «bop»\ e,\ s\rangle \to \langle(Val\ v_1)\ «bop»\ e',\ s'\rangle$

| *RedBinOp*:
$binop(bop,v_1,v_2) = Some\ v \Longrightarrow$
$P,E \vdash \langle(Val\ v_1)\ «bop»\ (Val\ v_2),\ s\rangle \to \langle Val\ v,s\rangle$

| *RedVar*:
$lcl\ s\ V = Some\ v \Longrightarrow$
$P,E \vdash \langle Var\ V,s\rangle \to \langle Val\ v,s\rangle$

| *LAssRed*:
$P,E \vdash \langle e,s\rangle \to \langle e',s'\rangle \Longrightarrow$
$P,E \vdash \langle V{:=}e,s\rangle \to \langle V{:=}e',s'\rangle$

| *RedLAss*:
$[\![E\ V = Some\ T;\ P \vdash T\ casts\ v\ to\ v']\!] \Longrightarrow$
$P,E \vdash \langle V{:=}(Val\ v),(h,l)\rangle \to \langle Val\ v',(h,l(V{\mapsto}v'))\rangle$

| *FAccRed*:
$P,E \vdash \langle e,s\rangle \to \langle e',s'\rangle \Longrightarrow$
$P,E \vdash \langle e{\cdot}F\{Cs\},\ s\rangle \to \langle e'{\cdot}F\{Cs\},\ s'\rangle$

| *RedFAcc*:
$[\![\ hp\ s\ a = Some(D,S);\ Ds = Cs'@_pCs;\ (Ds,fs) \in S;\ fs\ F = Some\ v\ ]\!]$
$\Longrightarrow P,E \vdash \langle(ref\ (a,Cs')){\cdot}F\{Cs\},\ s\rangle \to \langle Val\ v,s\rangle$

| *RedFAccNull*:
$P,E \vdash \langle null{\cdot}F\{Cs\},\ s\rangle \to \langle THROW\ NullPointer,\ s\rangle$

| *FAssRed1*:
$P,E \vdash \langle e,s\rangle \to \langle e',s'\rangle \Longrightarrow$
$P,E \vdash \langle e{\cdot}F\{Cs\}{:=}e_2,\ s\rangle \to \langle e'{\cdot}F\{Cs\}{:=}e_2,\ s'\rangle$

| *FAssRed2*:
$P,E \vdash \langle e,s\rangle \to \langle e',s'\rangle \Longrightarrow$
$P,E \vdash \langle Val\ v{\cdot}F\{Cs\}{:=}e,\ s\rangle \to \langle Val\ v{\cdot}F\{Cs\}{:=}e',\ s'\rangle$

| *RedFAss*:
$[\![h\ a = Some(D,S);\ P \vdash (last\ Cs')\ has\ least\ F{:}T\ via\ Cs;$
$P \vdash T\ casts\ v\ to\ v';\ Ds = Cs'@_pCs;\ (Ds,fs) \in S]\!] \Longrightarrow$
$P,E \vdash \langle(ref\ (a,Cs')){\cdot}F\{Cs\}{:=}(Val\ v),\ (h,l)\rangle \to \langle Val\ v',\ (h(a \mapsto (D,insert\ (Ds,fs(F{\mapsto}v'))$
$(S - \{(Ds,fs)\}))),l)\rangle$

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

| *CallObj*:
$P,E \vdash \langle e,s\rangle \to \langle e',s'\rangle \Longrightarrow$
$P,E \vdash \langle Call\ e\ Copt\ M\ es,s\rangle \to \langle Call\ e'\ Copt\ M\ es,s'\rangle$

| *CallParams*:
$P,E \vdash \langle es,s\rangle\ [\to]\ \langle es',s'\rangle \Longrightarrow$
$P,E \vdash \langle Call\ (Val\ v)\ Copt\ M\ es,s\rangle \to \langle Call\ (Val\ v)\ Copt\ M\ es',s'\rangle$

| *RedCall*:
$[\![\ hp\ s\ a = Some(C,S);\ P \vdash last\ Cs\ has\ least\ M = (Ts',T',pns',body')\ via\ Ds;$
$P \vdash (C,Cs@_pDs)\ selects\ M = (Ts,T,pns,body)\ via\ Cs';$
$size\ vs = size\ pns;\ size\ Ts = size\ pns;$
$bs = blocks(this\#pns,Class(last\ Cs')\#Ts,Ref(a,Cs')\#vs,body);$
$new\text{-}body = (case\ T'\ of\ Class\ D \Rightarrow (\!|D|\!)bs \mid \text{-} \Rightarrow bs)\ ]\!]$
$\Longrightarrow P,E \vdash \langle (ref\ (a,Cs))\cdot M(map\ Val\ vs),\ s\rangle \to \langle new\text{-}body,\ s\rangle$

| *RedStaticCall*:
$[\![\ P \vdash Path\ (last\ Cs)\ to\ C\ unique;\ P \vdash Path\ (last\ Cs)\ to\ C\ via\ Cs'';$
$P \vdash C\ has\ least\ M = (Ts,T,pns,body)\ via\ Cs';\ Ds = (Cs@_pCs'')@_pCs';$
$size\ vs = size\ pns;\ size\ Ts = size\ pns\ ]\!]$
$\Longrightarrow P,E \vdash \langle (ref\ (a,Cs))\cdot(C::)M(map\ Val\ vs),\ s\rangle \to$
$\langle blocks(this\#pns,Class(last\ Ds)\#Ts,Ref(a,Ds)\#vs,body),\ s\rangle$

| *RedCallNull*:
$P,E \vdash \langle Call\ null\ Copt\ M\ (map\ Val\ vs),s\rangle \to \langle THROW\ NullPointer,s\rangle$

| *BlockRedNone*:
$[\![\ P,E(V \mapsto T) \vdash \langle e,\ (h,l(V{:=}None))\rangle \to \langle e',\ (h',l')\rangle;\ l'\ V = None;\ \neg\ assigned\ V\ e\ ]\!]$
$\Longrightarrow P,E \vdash \langle\{V{:}T;\ e\},\ (h,l)\rangle \to \langle\{V{:}T;\ e'\},\ (h',l'(V := l\ V))\rangle$

| *BlockRedSome*:
$[\![\ P,E(V \mapsto T) \vdash \langle e,\ (h,l(V{:=}None))\rangle \to \langle e',\ (h',l')\rangle;\ l'\ V = Some\ v;$
$\neg\ assigned\ V\ e\ ]\!]$
$\Longrightarrow P,E \vdash \langle\{V{:}T;\ e\},\ (h,l)\rangle \to \langle\{V{:}T := Val\ v;\ e'\},\ (h',l'(V := l\ V))\rangle$

| *InitBlockRed*:
$[\![\ P,E(V \mapsto T) \vdash \langle e,\ (h,l(V{\mapsto}v'))\rangle \to \langle e',\ (h',l')\rangle;\ l'\ V = Some\ v'';$
$P \vdash T\ casts\ v\ to\ v'\ ]\!]$
$\Longrightarrow P,E \vdash \langle\{V{:}T := Val\ v;\ e\},\ (h,l)\rangle \to \langle\{V{:}T := Val\ v'';\ e'\},\ (h',l'(V := l\ V))\rangle$

| *RedBlock*:
$P,E \vdash \langle\{V{:}T;\ Val\ u\},\ s\rangle \to \langle Val\ u,\ s\rangle$

| *RedInitBlock*:
$P \vdash T\ casts\ v\ to\ v' \Longrightarrow P,E \vdash \langle\{V{:}T := Val\ v;\ Val\ u\},\ s\rangle \to \langle Val\ u,\ s\rangle$

| *SeqRed*:
$P,E \vdash \langle e,s\rangle \to \langle e',s'\rangle \Longrightarrow$
$P,E \vdash \langle e;;e_2,\ s\rangle \to \langle e';;e_2,\ s'\rangle$

| *RedSeq*:

$P,E \vdash \langle (Val\ v);;e_2,\ s\rangle \to \langle e_2,\ s\rangle$

| *CondRed*:
$P,E \vdash \langle e,s\rangle \to \langle e',s'\rangle \Longrightarrow$
$P,E \vdash \langle if\ (e)\ e_1\ else\ e_2,\ s\rangle \to \langle if\ (e')\ e_1\ else\ e_2,\ s'\rangle$

| *RedCondT*:
$P,E \vdash \langle if\ (true)\ e_1\ else\ e_2,\ s\rangle \to \langle e_1,\ s\rangle$

| *RedCondF*:
$P,E \vdash \langle if\ (false)\ e_1\ else\ e_2,\ s\rangle \to \langle e_2,\ s\rangle$

| *RedWhile*:
$P,E \vdash \langle while(b)\ c,\ s\rangle \to \langle if(b)\ (c;;while(b)\ c)\ else\ unit,\ s\rangle$

| *ThrowRed*:
$P,E \vdash \langle e,s\rangle \to \langle e',s'\rangle \Longrightarrow$
$P,E \vdash \langle throw\ e,\ s\rangle \to \langle throw\ e',\ s'\rangle$

| *RedThrowNull*:
$P,E \vdash \langle throw\ null,\ s\rangle \to \langle THROW\ NullPointer,\ s\rangle$

| *ListRed1*:
$P,E \vdash \langle e,s\rangle \to \langle e',s'\rangle \Longrightarrow$
$P,E \vdash \langle e\#es,s\rangle\ [\to]\ \langle e'\#es,s'\rangle$

| *ListRed2*:
$P,E \vdash \langle es,s\rangle\ [\to]\ \langle es',s'\rangle \Longrightarrow$
$P,E \vdash \langle Val\ v\ \#\ es,s\rangle\ [\to]\ \langle Val\ v\ \#\ es',s'\rangle$

— Exception propagation

| *DynCastThrow*: $P,E \vdash \langle Cast\ C\ (Throw\ r),\ s\rangle \to \langle Throw\ r,\ s\rangle$
| *StaticCastThrow*: $P,E \vdash \langle (\!|C|\!)(Throw\ r),\ s\rangle \to \langle Throw\ r,\ s\rangle$
| *BinOpThrow1*: $P,E \vdash \langle (Throw\ r)\ «bop»\ e_2,\ s\rangle \to \langle Throw\ r,\ s\rangle$
| *BinOpThrow2*: $P,E \vdash \langle (Val\ v_1)\ «bop»\ (Throw\ r),\ s\rangle \to \langle Throw\ r,\ s\rangle$
| *LAssThrow*: $P,E \vdash \langle V:=(Throw\ r),\ s\rangle \to \langle Throw\ r,\ s\rangle$
| *FAccThrow*: $P,E \vdash \langle (Throw\ r)\cdot F\{Cs\},\ s\rangle \to \langle Throw\ r,\ s\rangle$
| *FAssThrow1*: $P,E \vdash \langle (Throw\ r)\cdot F\{Cs\}:=e_2,\ s\rangle \to \langle Throw\ r,s\rangle$
| *FAssThrow2*: $P,E \vdash \langle Val\ v\cdot F\{Cs\}:=(Throw\ r),\ s\rangle \to \langle Throw\ r,\ s\rangle$
| *CallThrowObj*: $P,E \vdash \langle Call\ (Throw\ r)\ Copt\ M\ es,\ s\rangle \to \langle Throw\ r,\ s\rangle$
| *CallThrowParams*: $[\![\ es = map\ Val\ vs\ @\ Throw\ r\ \#\ es'\ ]\!]$
$\Longrightarrow P,E \vdash \langle Call\ (Val\ v)\ Copt\ M\ es,\ s\rangle \to \langle Throw\ r,\ s\rangle$
| *BlockThrow*: $P,E \vdash \langle\{V{:}T;\ Throw\ r\},\ s\rangle \to \langle Throw\ r,\ s\rangle$
| *InitBlockThrow*: $P \vdash T\ casts\ v\ to\ v'$
$\Longrightarrow P,E \vdash \langle\{V{:}T := Val\ v;\ Throw\ r\},\ s\rangle \to \langle Throw\ r,\ s\rangle$
| *SeqThrow*: $P,E \vdash \langle (Throw\ r);;e_2,\ s\rangle \to \langle Throw\ r,\ s\rangle$
| *CondThrow*: $P,E \vdash \langle if\ (Throw\ r)\ e_1\ else\ e_2,\ s\rangle \to \langle Throw\ r,\ s\rangle$
| *ThrowThrow*: $P,E \vdash \langle throw(Throw\ r),\ s\rangle \to \langle Throw\ r,\ s\rangle$

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

inductive-cases [*elim*!]:
$P,E \vdash \langle V:=e,s\rangle \to \langle e',s'\rangle$
$P,E \vdash \langle e1;;e2,s\rangle \to \langle e',s'\rangle$

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

lemma $P,E \vdash \langle e,s\rangle \to \langle e',s'\rangle \Longrightarrow True$
and *reds-length*:$P,E \vdash \langle es,s\rangle\ [\to]\ \langle es',s'\rangle \Longrightarrow length\ es = length\ es'$
⟨*proof*⟩

13.3 The reflexive transitive closure

definition *Red* :: *prog* ⇒ *env* ⇒ ((*expr* × *state*) × (*expr* × *state*)) *set*
where $Red\ P\ E = \{((e,s),e',s').\ P,E \vdash \langle e,s\rangle \to \langle e',s'\rangle\}$

definition *Reds* :: *prog* ⇒ *env* ⇒ ((*expr list* × *state*) × (*expr list* × *state*)) *set*
where $Reds\ P\ E = \{((es,s),es',s').\ P,E \vdash \langle es,s\rangle\ [\to]\ \langle es',s'\rangle\}$

lemma[*simp*]: $((e,s),e',s') \in Red\ P\ E = P,E \vdash \langle e,s\rangle \to \langle e',s'\rangle$
⟨*proof*⟩

lemma[*simp*]: $((es,s),es',s') \in Reds\ P\ E = P,E \vdash \langle es,s\rangle\ [\to]\ \langle es',s'\rangle$
⟨*proof*⟩

abbreviation
Step :: *prog* ⇒ *env* ⇒ *expr* ⇒ *state* ⇒ *expr* ⇒ *state* ⇒ *bool*
(-,- ⊢ ((1⟨-,/-⟩) →∗/ (1⟨-,/-⟩))) [51,0,0,0,0] 81) **where**
$P,E \vdash \langle e,s\rangle \to* \langle e',s'\rangle \equiv ((e,s),\ e',s') \in (Red\ P\ E)^*$

abbreviation
Steps :: *prog* ⇒ *env* ⇒ *expr list* ⇒ *state* ⇒ *expr list* ⇒ *state* ⇒ *bool*
(-,- ⊢ ((1⟨-,/-⟩) [→]∗/ (1⟨-,/-⟩))) [51,0,0,0,0] 81) **where**
$P,E \vdash \langle es,s\rangle\ [\to]* \langle es',s'\rangle \equiv ((es,s),\ es',s') \in (Reds\ P\ E)^*$

lemma *converse-rtrancl-induct-red*[*consumes 1*]:
assumes $P,E \vdash \langle e,(h,l)\rangle \to* \langle e',(h',l')\rangle$
and $\bigwedge e\ h\ l.\ R\ e\ h\ l\ e\ h\ l$
and $\bigwedge e_0\ h_0\ l_0\ e_1\ h_1\ l_1\ e'\ h'\ l'.$
$[\![\ P,E \vdash \langle e_0,(h_0,l_0)\rangle \to \langle e_1,(h_1,l_1)\rangle;\ R\ e_1\ h_1\ l_1\ e'\ h'\ l'\ ]\!] \Longrightarrow R\ e_0\ h_0\ l_0\ e'\ h'\ l'$
shows $R\ e\ h\ l\ e'\ h'\ l'$

⟨*proof*⟩

lemma *steps-length*:$P,E \vdash \langle es,s\rangle\ [\rightarrow]*\ \langle es',s'\rangle \Longrightarrow length\ es = length\ es'$
⟨*proof*⟩

13.4 Some easy lemmas

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

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

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

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

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

⟨*proof*⟩

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

lemma
red-preserves-obj:$[\![P,E \vdash \langle e,(h,l)\rangle \rightarrow \langle e',(h',l')\rangle;\ h\ a = Some(D,S)]\!]$
$\Longrightarrow \exists S'.\ h'\ a = Some(D,S')$
and *reds-preserves-obj*:$[\![P,E \vdash \langle es,(h,l)\rangle\ [\rightarrow]\ \langle es',(h',l')\rangle;\ h\ a = Some(D,S)]\!]$
$\Longrightarrow \exists S'.\ h'\ a = Some(D,S')$
⟨*proof*⟩

end

14 System Classes

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

This theory provides definitions for the system exceptions.

definition *NullPointerC* :: *cdecl* **where**
NullPointerC ≡ (*NullPointer*, ([],[],[]))

definition *ClassCastC* :: *cdecl* **where**
ClassCastC ≡ (*ClassCast*, ([],[],[]))

definition *OutOfMemoryC* :: *cdecl* **where**
OutOfMemoryC ≡ (*OutOfMemory*, ([],[],[]))

definition *SystemClasses* :: *cdecl list* **where**
SystemClasses ≡ [*NullPointerC*, *ClassCastC*, *OutOfMemoryC*]

end

15 The subtype relation

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

inductive
widen :: *prog* ⇒ *ty* ⇒ *ty* ⇒ *bool* (- ⊢ - ≤ - [71,71,71] 70)
for *P* :: *prog*
where
widen-refl[*iff*]: *P* ⊢ *T* ≤ *T*
| *widen-subcls*: *P* ⊢ *Path C to D unique* ⟹ *P* ⊢ *Class C* ≤ *Class D*
| *widen-null*[*iff*]: *P* ⊢ *NT* ≤ *Class C*

abbreviation
widens :: *prog* ⇒ *ty list* ⇒ *ty list* ⇒ *bool*
(- ⊢ - [≤] - [71,71,71] 70) **where**
widens P Ts Ts' ≡ *list-all2* (*widen P*) *Ts Ts'*

inductive-simps [*iff*]:
P ⊢ *T* ≤ *Void*
P ⊢ *T* ≤ *Boolean*
P ⊢ *T* ≤ *Integer*
P ⊢ *Void* ≤ *T*
P ⊢ *Boolean* ≤ *T*
P ⊢ *Integer* ≤ *T*
P ⊢ *T* ≤ *NT*

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

end

16 Well-typedness of CoreC++ expressions

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

16.1 The rules

inductive
  $WT :: [prog,env,expr\ \ \ ,ty\ \ \ ] \Rightarrow bool$
    $(\text{-,-} \vdash \text{-} :: \text{-}\ \ [51,51,51]50)$
  and $WTs :: [prog,env,expr\ list,ty\ list] \Rightarrow bool$
    $(\text{-,-} \vdash \text{-} [::] \text{-}\ [51,51,51]50)$
  for $P :: prog$
where

  WTNew:
  $is\text{-}class\ P\ C \Longrightarrow$
  $P,E \vdash new\ C :: Class\ C$

| *WTDynCast*:
  $[\![P,E \vdash e :: Class\ D;\ is\text{-}class\ P\ C;$
    $P \vdash Path\ D\ to\ C\ unique \vee (\forall\, Cs.\ \neg\ P \vdash Path\ D\ to\ C\ via\ Cs)]\!]$
  $\Longrightarrow P,E \vdash Cast\ C\ e :: Class\ C$

| *WTStaticCast*:
  $[\![P,E \vdash e :: Class\ D;\ is\text{-}class\ P\ C;$
    $P \vdash Path\ D\ to\ C\ unique \vee$
   $(P \vdash C \preceq^* D \wedge (\forall\, Cs.\ P \vdash Path\ C\ to\ D\ via\ Cs \longrightarrow Subobjs_R\ P\ C\ Cs))\ ]\!]$
  $\Longrightarrow P,E \vdash (\!|C|\!)e :: Class\ C$

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

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

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

| *WTFAss*:
$[\![\ P,E \vdash e_1 :: \mathit{Class}\ C;\ \ P \vdash C\ \mathit{has\ least}\ F{:}T\ \mathit{via}\ Cs;$
$\quad P,E \vdash e_2 :: T';\ P \vdash T' \leq T\ ]\!]$
$\Longrightarrow P,E \vdash e_1{\cdot}F\{Cs\}{:=}e_2 :: T$

| *WTStaticCall*:
$[\![\ P,E \vdash e :: \mathit{Class}\ C';\ P \vdash \mathit{Path}\ C'\ \mathit{to}\ C\ \mathit{unique};$
$\quad P \vdash C\ \mathit{has\ least}\ M = (Ts,T,m)\ \mathit{via}\ Cs;\ P,E \vdash es\ [::]\ Ts';\ P \vdash Ts'\ [\leq]\ Ts\ ]\!]$
$\Longrightarrow P,E \vdash e{\cdot}(C{::})M(es) :: T$

| *WTCall*:
$[\![\ P,E \vdash e :: \mathit{Class}\ C;\ \ P \vdash C\ \mathit{has\ least}\ M = (Ts,T,m)\ \mathit{via}\ Cs;$
$\quad P,E \vdash es\ [::]\ Ts';\ P \vdash Ts'\ [\leq]\ Ts\ ]\!]$
$\Longrightarrow P,E \vdash e{\cdot}M(es) :: T$

| *WTBlock*:
$[\![\ \mathit{is\text{-}type}\ P\ T;\ \ P,E(V \mapsto T) \vdash e :: T'\ ]\!]$
$\Longrightarrow\ P,E \vdash \{V{:}T;\ e\} :: T'$

| *WTSeq*:
$[\![\ P,E \vdash e_1{::}T_1;\ \ P,E \vdash e_2{::}T_2\ ]\!]$
$\Longrightarrow\ P,E \vdash e_1;;e_2 :: T_2$

| *WTCond*:
$[\![\ P,E \vdash e :: \mathit{Boolean};\ \ P,E \vdash e_1{::}T;\ \ P,E \vdash e_2{::}T\ ]\!]$
$\Longrightarrow P,E \vdash \mathit{if}\ (e)\ e_1\ \mathit{else}\ e_2 :: T$

| *WTWhile*:
$[\![\ P,E \vdash e :: \mathit{Boolean};\ \ P,E \vdash c{::}T\ ]\!]$
$\Longrightarrow P,E \vdash \mathit{while}\ (e)\ c :: \mathit{Void}$

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

— well-typed expression lists

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

| *WTCons*:
$[\![\ P,E \vdash e :: T;\ \ P,E \vdash es\ [::]\ Ts\ ]\!]$
$\Longrightarrow\ P,E \vdash e\#es\ [::]\ T\#Ts$

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

lemmas *WT-WTs-induct* = *WT-WTs.induct* [*split-format* (*complete*)]
and *WT-WTs-inducts* = *WT-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-elim-cases*[*elim*!]:
$P,E \vdash new\ C :: T$
$P,E \vdash Cast\ C\ e :: T$
$P,E \vdash (\!|C|\!)e :: T$
$P,E \vdash e_1\ «bop»\ e_2 :: T$
$P,E \vdash V := e :: T$
$P,E \vdash e{\cdot}F\{Cs\} :: T$
$P,E \vdash e{\cdot}F\{Cs\} := v :: T$
$P,E \vdash e{\cdot}M(ps) :: T$
$P,E \vdash e{\cdot}(C::)M(ps) :: T$
$P,E \vdash if\ (e)\ e_1\ else\ e_2 :: T$
$P,E \vdash while\ (e)\ c :: T$
$P,E \vdash throw\ e :: T$

lemma *wt-env-mono*:
$P,E \vdash e :: T \Longrightarrow (\bigwedge E'.\ E \subseteq_m E' \Longrightarrow P,E' \vdash e :: T)$ **and**
$P,E \vdash es\ [::]\ Ts \Longrightarrow (\bigwedge E'.\ E \subseteq_m E' \Longrightarrow P,E' \vdash es\ [::]\ Ts)$

⟨*proof*⟩

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

⟨*proof*⟩

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

type-synonym *wf-mdecl-test* = *prog* ⇒ *cname* ⇒ *mdecl* ⇒ *bool*

definition *wf-fdecl* :: *prog* ⇒ *fdecl* ⇒ *bool* **where**
wf-fdecl $P \equiv \lambda(F,T).$ *is-type* $P\ T$

definition *wf-mdecl* :: *wf-mdecl-test* ⇒ *wf-mdecl-test* **where**
wf-mdecl wf-md P C ≡ λ(*M*,*Ts*,*T*,*mb*).
(∀ *T*∈*set Ts*. *is-type P T*) ∧ *is-type P T* ∧ *T* ≠ *NT* ∧ *wf-md P C* (*M*,*Ts*,*T*,*mb*)

definition *wf-cdecl* :: *wf-mdecl-test* ⇒ *prog* ⇒ *cdecl* ⇒ *bool* **where**
wf-cdecl wf-md P ≡ λ(*C*,(*Bs*,*fs*,*ms*)).
(∀ *M mthd Cs*. *P* ⊢ *C has M* = *mthd via Cs* ⟶
(∃ *mthd*′ *Cs*′. *P* ⊢ (*C*,*Cs*) *has overrider M* = *mthd*′ *via Cs*′)) ∧
(∀ *f*∈*set fs*. *wf-fdecl P f*) ∧ *distinct-fst fs* ∧
(∀ *m*∈*set ms*. *wf-mdecl wf-md P C m*) ∧ *distinct-fst ms* ∧
(∀ *D* ∈ *baseClasses Bs*.
is-class P D ∧ ¬ *P* ⊢ *D* ≼* *C* ∧
(∀ (*M*,*Ts*,*T*,*m*)∈*set ms*.
∀ *Ts*′ *T*′ *m*′ *Cs*. *P* ⊢ *D has M* = (*Ts*′,*T*′,*m*′) *via Cs* ⟶
Ts′ = *Ts* ∧ *P* ⊢ *T* ≤ *T*′))

definition *wf-syscls* :: *prog* ⇒ *bool* **where**
wf-syscls P ≡ *sys-xcpts* ⊆ *set*(*map fst P*)

definition *wf-prog* :: *wf-mdecl-test* ⇒ *prog* ⇒ *bool* **where**
wf-prog wf-md P ≡ *wf-syscls P* ∧ *distinct-fst P* ∧
(∀ *c* ∈ *set P*. *wf-cdecl wf-md P c*)

17.1 Well-formedness lemmas

lemma *class-wf*:
⟦*class P C* = *Some c*; *wf-prog wf-md P*⟧ ⟹ *wf-cdecl wf-md P* (*C*,*c*)

⟨*proof*⟩

lemma *is-class-xcpt*:
⟦ *C* ∈ *sys-xcpts*; *wf-prog wf-md P* ⟧ ⟹ *is-class P C*

⟨*proof*⟩

lemma *is-type-pTs*:
assumes *wf-prog wf-md P* **and** (*C*,*S*,*fs*,*ms*) ∈ *set P* **and** (*M*,*Ts*,*T*,*m*) ∈ *set ms*
shows *set Ts* ⊆ *types P*
⟨*proof*⟩

17.2 Well-formedness subclass lemmas

lemma *subcls1-wfD*:
⟦ *P* ⊢ *C* ≺¹ *D*; *wf-prog wf-md P* ⟧ ⟹ *D* ≠ *C* ∧ (*D*,*C*) ∉ (*subcls1 P*)⁺

⟨*proof*⟩

lemma *wf-cdecl-supD*:
⟦*wf-cdecl wf-md P* (*C*,*Bs*,*r*); *D* ∈ *baseClasses Bs*⟧ ⟹ *is-class P D*
⟨*proof*⟩

lemma *subcls-asym*:
⟦ *wf-prog wf-md P*; (*C*,*D*) ∈ (*subcls1 P*)$^+$ ⟧ ⟹ (*D*,*C*) ∉ (*subcls1 P*)$^+$

⟨*proof*⟩

lemma *subcls-irrefl*:
⟦ *wf-prog wf-md P*; (*C*,*D*) ∈ (*subcls1 P*)$^+$ ⟧ ⟹ *C* ≠ *D*

⟨*proof*⟩

lemma *subcls-asym2*:
⟦ (*C*,*D*) ∈ (*subcls1 P*)*; *wf-prog wf-md P*; (*D*,*C*) ∈ (*subcls1 P*)* ⟧ ⟹ *C* = *D*

⟨*proof*⟩

lemma *acyclic-subcls1*:
wf-prog wf-md P ⟹ *acyclic* (*subcls1 P*)

⟨*proof*⟩

lemma *wf-subcls1*:
wf-prog wf-md P ⟹ *wf* ((*subcls1 P*)$^{-1}$)

⟨*proof*⟩

lemma *subcls-induct*:
⟦ *wf-prog wf-md P*; ⋀*C*. ∀ *D*. (*C*,*D*) ∈ (*subcls1 P*)$^+$ ⟶ *Q D* ⟹ *Q C* ⟧ ⟹ *Q C*

(**is** *?A* ⟹ *PROP ?P* ⟹ -)

⟨*proof*⟩

17.3 Well-formedness leq_path lemmas

lemma *last-leq-path*:
assumes *leq*:$P,C \vdash Cs \sqsubset^1 Ds$ **and** *wf*:*wf-prog wf-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-prog wf-md P*
shows $(last\ Cs,last\ Ds) \in (subcls1\ P)^+$

⟨*proof*⟩

lemma *leq-path1-wfD*:
$[\![\ P,C \vdash Cs \sqsubset^1 Cs';\ wf\text{-}prog\ wf\text{-}md\ P\ ]\!] \Longrightarrow Cs \neq Cs' \wedge (Cs',Cs) \notin (leq\text{-}path1\ P\ C)^+$

⟨*proof*⟩

lemma *leq-path-asym*:
$[\![(Cs,Cs') \in (leq\text{-}path1\ P\ C)^+;\ wf\text{-}prog\ wf\text{-}md\ P]\!] \Longrightarrow (Cs',Cs) \notin (leq\text{-}path1\ P\ C)^+$

⟨*proof*⟩

lemma *leq-path-asym2*:$[\![P,C \vdash Cs \sqsubseteq Cs';\ P,C \vdash Cs' \sqsubseteq Cs;\ wf\text{-}prog\ wf\text{-}md\ P]\!] \Longrightarrow Cs = Cs'$

⟨*proof*⟩

lemma *leq-path-Subobjs*:
$[\![P,C \vdash [C] \sqsubseteq Cs;\ is\text{-}class\ P\ C;\ wf\text{-}prog\ wf\text{-}md\ P]\!] \Longrightarrow Subobjs\ P\ C\ Cs$
⟨*proof*⟩

17.4 Lemmas concerning Subobjs

lemma *Subobj-last-isClass*:$[\![wf\text{-}prog\ wf\text{-}md\ P;\ Subobjs\ P\ C\ Cs]\!] \Longrightarrow is\text{-}class\ P\ (last\ Cs)$

⟨*proof*⟩

lemma *converse-SubobjsR-Rep*:
$[\![Subobjs_R\ P\ C\ Cs;\ P \vdash last\ Cs \prec_R C';\ wf\text{-}prog\ wf\text{-}md\ P]\!]$
$\Longrightarrow Subobjs_R\ P\ C\ (Cs@[C'])$

⟨*proof*⟩

lemma *converse-Subobjs-Rep*:
$[\![Subobjs\ P\ C\ Cs;\ P \vdash last\ Cs \prec_R C';\ \ wf\text{-}prog\ wf\text{-}md\ P]\!]$
$\Longrightarrow Subobjs\ P\ C\ (Cs@[C'])$
⟨*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-prog wf-md P $\Longrightarrow$ *is-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-prog wf-md P*

shows $C' \notin set\ Cs'$

⟨*proof*⟩

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

⟨*proof*⟩

lemma *unique2*:
assumes *subo*:*Subobjs P C* (*Cs*@ *C'*#*Cs'*) **and** *wf*:*wf-prog wf-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-prog wf-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*:$P \vdash C \preceq^* D$ **and** *wf*:*wf-prog wf-md P*
shows *subcls-leq-path*:$\exists\, Cs.\ P,C \vdash [C] \sqsubseteq Cs@[D]$

⟨*proof*⟩

lemma assumes *subo*:*Subobjs P C* (*rev Cs*) **and** *wf*:*wf-prog wf-md P*
shows *subobjs-rel-rev*:$P,C \vdash [C] \sqsubseteq (rev\ Cs)$

⟨*proof*⟩

lemma *subobjs-rel*:
assumes *subo*:*Subobjs P C Cs* **and** *wf*:*wf-prog wf-md P*
shows $P,C \vdash [C] \sqsubseteq Cs$

⟨*proof*⟩

lemma assumes *wf*:*wf-prog wf-md P*
shows *leq-path-last*:$[\![P,C \vdash Cs \sqsubseteq Cs'; last\ Cs = last\ Cs']\!] \Longrightarrow Cs = Cs'$

⟨*proof*⟩

17.5 Well-formedness and appendPath

lemma *appendPath1*:
$[\![Subobjs\ P\ C\ Cs; Subobjs\ P\ (last\ Cs)\ Ds; last\ Cs \neq hd\ Ds]\!]$
$\Longrightarrow Subobjs\ P\ C\ Ds$

⟨*proof*⟩

lemma *appendPath2-rev*:
assumes *subo1*:*Subobjs P C Cs* **and** *subo2*:*Subobjs P* (*last Cs*) (*last Cs*#*rev Ds*)
and *wf*:*wf-prog wf-md P*
shows *Subobjs P C* (*Cs*@(*tl* (*last Cs*#*rev Ds*)))
⟨*proof*⟩

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

⟨*proof*⟩

lemma *Subobjs-appendPath*:
$[\![Subobjs\ P\ C\ Cs; Subobjs\ P\ (last\ Cs)\ Ds; wf\text{-}prog\ wf\text{-}md\ P]\!]$
$\Longrightarrow Subobjs\ P\ C\ (Cs@_p Ds)$
⟨*proof*⟩

17.6 Path and program size

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

lemma *path-subset-classes*:$[\![Subobjs\ P\ C\ Cs;\ wf\text{-}prog\ wf\text{-}md\ P]\!]$
$\Longrightarrow set\ Cs \subseteq \{C.\ is\text{-}class\ P\ C\}$
⟨*proof*⟩

lemma assumes *subo*:*Subobjs P C* (*rev Cs*) **and** *wf*:*wf-prog wf-md P*
shows *rev-path-distinct-classes*:*distinct Cs*
⟨*proof*⟩

lemma assumes *subo*:*Subobjs P C Cs* **and** *wf*:*wf-prog wf-md P*
shows *path-distinct-classes*:*distinct Cs*

⟨*proof*⟩

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

⟨*proof*⟩

lemma assumes *subo*:*Subobjs P C Cs* **and** *wf*:*wf-prog wf-md P*
shows *path-length*:*length Cs* ≤ *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*:$\{xs.\ set\ xs \subseteq F \wedge length\ xs = Suc\ n\} = (UN\ x{:}F.\ UN\ xs : \{xs.\ set\ xs \leq F \wedge length\ xs = n\}.\ \{x\#xs\})$
⟨*proof*⟩

lemma *max-length-finite-set*:*finite F* $\Longrightarrow$ *finite*$\{xs.\ set\ xs <= F \wedge length\ xs = n\}$
⟨*proof*⟩

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

lemma *path-finite-leq*:
wf-prog wf-md P $\Longrightarrow$ *finite*$\{Cs.\ Subobjs\ P\ C\ Cs \wedge length\ Cs \leq length\ P\}$
⟨*proof*⟩

lemma *path-finite*:*wf-prog wf-md P* $\Longrightarrow$ *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 $\forall\ Cs'.\ P \vdash Path\ D\ to\ C\ via\ Cs' \longrightarrow Cs = [C] \wedge Cs' = [C] \wedge C = D$
⟨*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*: $[\![P \vdash Path\ C\ to\ C\ via\ Cs;\ wf\text{-}prog\ wf\text{-}md\ P]\!] \Longrightarrow Cs = [C]$
⟨*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′* ≠ []
shows ¬ *P* ⊢ *Path last Cs′ to C via Ds*

⟨*proof*⟩

lemma *leq-implies-path*:
assumes *leq*:*P* ⊢ *C* ≼* *D* **and** *class*: *is-class P C*
and *wf*:*wf-prog wf-md P*
shows ∃ *Cs*. *P* ⊢ *Path C to D via Cs*

⟨*proof*⟩

lemma *least-method-implies-path-unique*:
assumes *least*:*P* ⊢ *C has least M* = (*Ts*,*T*,*m*) *via Cs* **and** *wf*:*wf-prog wf-md P*
shows *P* ⊢ *Path C to* (*last Cs*) *unique*

⟨*proof*⟩

lemma *least-field-implies-path-unique*:
assumes *least*:*P* ⊢ *C has least F*:*T via Cs* **and** *wf*:*wf-prog wf-md P*
shows *P* ⊢ *Path C to* (*hd Cs*) *unique*

⟨*proof*⟩

lemma *least-field-implies-path-via-hd*:
⟦*P* ⊢ *C has least F*:*T via Cs*; *wf-prog wf-md P*⟧
⟹ *P* ⊢ *Path C to* (*hd Cs*) *via* [*hd Cs*]

⟨*proof*⟩

lemma *path-C-to-C-unique*:
⟦*wf-prog wf-md P*; *is-class P C*⟧ ⟹ *P* ⊢ *Path C to C unique*

⟨*proof*⟩

lemma *leqR-SubobjsR*:⟦(*C*,*D*) ∈ (*subclsR P*)*; *is-class P C*; *wf-prog wf-md P*⟧
⟹ ∃ *Cs*. *Subobjs*$_R$ *P C* (*Cs*@[*D*])

⟨*proof*⟩

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

⟨*proof*⟩

17.8 Well-formedness and member lookup

lemma *has-path-has*:
$[\![P \vdash Path\ D\ to\ C\ via\ Ds;\ P \vdash C\ has\ M = (Ts,T,m)\ via\ Cs;\ wf\text{-}prog\ wf\text{-}md\ P]\!]$
$\Longrightarrow P \vdash D\ has\ M = (Ts,T,m)\ via\ Ds@_p Cs$
⟨*proof*⟩

lemma *has-least-wf-mdecl*:
$[\![\ wf\text{-}prog\ wf\text{-}md\ P;\ P \vdash C\ has\ least\ M = m\ via\ Cs\ ]\!]$
$\Longrightarrow wf\text{-}mdecl\ wf\text{-}md\ P\ (last\ Cs)\ (M,m)$
⟨*proof*⟩

lemma *has-overrider-wf-mdecl*:
$[\![\ wf\text{-}prog\ wf\text{-}md\ P;\ P \vdash (C,Cs)\ has\ overrider\ M = m\ via\ Cs'\ ]\!]$
$\Longrightarrow wf\text{-}mdecl\ wf\text{-}md\ P\ (last\ Cs')\ (M,m)$
⟨*proof*⟩

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

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

⟨*proof*⟩

lemma *wf-select-method-fun*:
assumes *wf*:*wf-prog wf-md P*
shows $[\![P \vdash (C,Cs)\ selects\ M = mthd\ via\ Cs';\ P \vdash (C,Cs)\ selects\ M = mthd'$
$via\ Cs']\!]$

$\Longrightarrow$ *mthd* = *mthd'* $\wedge$ *Cs'* = *Cs''*
⟨*proof*⟩

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*

⟨*proof*⟩

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*

⟨*proof*⟩

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*

⟨*proof*⟩

lemma *select-method-is-type*:
⟦ *P* $\vdash$ (*C*,*Cs*) *selects M* = (*Ts*,*T*,*m*) *via Cs'*; *wf-prog wf-md P*⟧ $\Longrightarrow$ *is-type P T*
⟨*proof*⟩

lemma *base-subtype*:
⟦*wf-cdecl wf-md P* (*C*,*Bs*,*fs*,*ms*); *C'* $\in$ *baseClasses Bs*;
P $\vdash$ *C' has M* = (*Ts'*,*T'*,*m'*) *via Cs*@$_p$[*D*]; (*M*,*Ts*,*T*,*m*)$\in$*set ms*⟧
$\Longrightarrow$ *Ts'* = *Ts* $\wedge$ *P* $\vdash$ *T* $\leq$ *T'*

⟨*proof*⟩

lemma *subclsPlus-subtype*:
assumes *classD*:*class P D* = *Some*(*Bs'*,*fs'*,*ms'*)
and *mapMs'*:*map-of ms' M* = *Some*(*Ts'*,*T'*,*m'*)
and *leq*:(*C*,*D*) $\in$ (*subcls1 P*)$^+$ **and** *wf*:*wf-prog wf-md P*

shows $\forall\, Bs\ fs\ ms\ Ts\ T\ m.\ class\ P\ C = Some(Bs,fs,ms) \wedge map\text{-}of\ ms\ M = Some(Ts,T,m)$
$\longrightarrow Ts' = Ts \wedge P \vdash T \leq T'$

⟨*proof*⟩

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

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

lemma *select-least-methods-subtypes*:
assumes *select-method*:$P \vdash (C,Cs@_p Ds)$ *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*:
$set\ SystemClasses \subseteq set\ P \Longrightarrow wf\text{-}syscls\ P$
⟨*proof*⟩

17.9 Well formedness and widen

lemma *Class-widen*: $[\![P \vdash Class\ C \leq T;\ wf\text{-}prog\ wf\text{-}md\ P;\ is\text{-}class\ P\ C]\!]$
$\Longrightarrow \exists\, D.\ T = Class\ D \wedge P \vdash Path\ C\ to\ D\ unique$

⟨*proof*⟩

lemma *Class-widen-Class* [*iff*]: ⟦*wf-prog wf-md P*; *is-class P C*⟧ ⟹
(*P* ⊢ *Class C* ≤ *Class D*) = (*P* ⊢ *Path C to D unique*)

⟨*proof*⟩

lemma *widen-Class*: ⟦*wf-prog wf-md P*; *is-class P C*⟧ ⟹
(*P* ⊢ *T* ≤ *Class C*) =
(*T* = *NT* ∨ (∃ *D*. *T* = *Class D* ∧ *P* ⊢ *Path D to C unique*))

⟨*proof*⟩

17.10 Well formedness and well typing

lemma assumes *wf*:*wf-prog wf-md P*
shows *WT-determ*: *P*,*E* ⊢ *e* :: *T* ⟹ (⋀*T'*. *P*,*E* ⊢ *e* :: *T'* ⟹ *T* = *T'*)
and *WTs-determ*: *P*,*E* ⊢ *es* [::] *Ts* ⟹ (⋀*Ts'*. *P*,*E* ⊢ *es* [::] *Ts'* ⟹ *Ts* = *Ts'*)

⟨*proof*⟩

end

18 Weak well-formedness of CoreC++ programs

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

definition *wwf-mdecl* :: *prog* ⇒ *cname* ⇒ *mdecl* ⇒ *bool* **where**
wwf-mdecl P C ≡ λ(*M*,*Ts*,*T*,(*pns*,*body*)).
length Ts = *length pns* ∧ *distinct pns* ∧ *this* ∉ *set pns* ∧ *fv body* ⊆ {*this*} ∪ *set pns*

lemma *wwf-mdecl*[*simp*]:
wwf-mdecl P C (*M*,*Ts*,*T*,*pns*,*body*) =
(*length Ts* = *length pns* ∧ *distinct pns* ∧ *this* ∉ *set pns* ∧ *fv body* ⊆ {*this*} ∪ *set pns*)
⟨*proof*⟩

abbreviation
wwf-prog :: *prog* ⇒ *bool* **where**
wwf-prog == *wf-prog wwf-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*:
$P \vdash T$ *casts* v *to* $v' \Longrightarrow P \vdash T$ *casts* v' *to* v'

⟨*proof*⟩

lemma *casts-casts-eq*:
$[\![P \vdash T$ *casts* v *to* v; $P \vdash T$ *casts* v *to* v'; *wf-prog wf-md* $P]\!] \Longrightarrow v = v'$

⟨*proof*⟩

lemma assumes *wf*:*wf-prog wf-md P*
shows *None-lcl-casts-values*:
$P,E \vdash \langle e,(h,l)\rangle \to \langle e',(h',l')\rangle \Longrightarrow$
$(\bigwedge V.\ [\![l\ V = None;\ E\ V = Some\ T;\ l'\ V = Some\ v']\!]$
$\Longrightarrow P \vdash T$ *casts* v' *to* $v')$
and $P,E \vdash \langle es,(h,l)\rangle\ [\to]\ \langle es',(h',l')\rangle \Longrightarrow$
$(\bigwedge V.\ [\![l\ V = None;\ E\ V = Some\ T;\ l'\ V = Some\ v']\!]$
$\Longrightarrow P \vdash T$ *casts* v' *to* $v')$

⟨*proof*⟩

lemma assumes *wf*:*wf-prog wf-md P*
shows *Some-lcl-casts-values*:
$P,E \vdash \langle e,(h,l)\rangle \to \langle e',(h',l')\rangle \Longrightarrow$
$(\bigwedge V.\ [\![l\ V = Some\ v;\ E\ V = Some\ T;$
$P \vdash T$ *casts* v'' *to* $v;\ l'\ V = Some\ v']\!]$
$\Longrightarrow P \vdash T$ *casts* v' *to* $v')$
and $P,E \vdash \langle es,(h,l)\rangle\ [\to]\ \langle es',(h',l')\rangle \Longrightarrow$
$(\bigwedge V.\ [\![l\ V = Some\ v;\ E\ V = Some\ T;$
$P \vdash T$ *casts* v'' *to* $v;\ l'\ V = Some\ v']\!]$
$\Longrightarrow P \vdash T$ *casts* v' *to* $v')$

⟨*proof*⟩

19.2 Small steps simulate big step

19.3 Cast

lemma *StaticCastReds*:
$P,E \vdash \langle e,s\rangle \to* \langle e',s'\rangle \Longrightarrow P,E \vdash \langle (\!| C |\!)\, e,s\rangle \to* \langle (\!| C |\!)\, e',s'\rangle$

⟨*proof*⟩

lemma *StaticCastRedsNull*:
$P,E \vdash \langle e,s\rangle \to* \langle null,s'\rangle \Longrightarrow P,E \vdash \langle (\!| C |\!)\, e,s\rangle \to* \langle null,s'\rangle$

⟨*proof*⟩

lemma *StaticUpCastReds*:
$[\![\ P,E \vdash \langle e,s\rangle \to* \langle ref(a,Cs),s'\rangle;\ P \vdash Path\ last\ Cs\ to\ C\ via\ Cs';\ Ds = Cs@_p Cs'\ ]\!]$
$\Longrightarrow P,E \vdash \langle (\!| C |\!)\, e,s\rangle \to* \langle ref(a,Ds),s'\rangle$

⟨*proof*⟩

lemma *StaticDownCastReds*:
$P,E \vdash \langle e,s\rangle \to* \langle ref(a,Cs@[C]@Cs'),s'\rangle$
$\Longrightarrow\ P,E \vdash \langle (\!| C |\!)\, e,s\rangle \to* \langle ref(a,Cs@[C]),s'\rangle$

⟨*proof*⟩

lemma *StaticCastRedsFail*:
$[\![\ P,E \vdash \langle e,s\rangle \to* \langle ref(a,Cs),s'\rangle;\ C \notin set\ Cs;\ \neg\ P \vdash (last\ Cs) \preceq^*\ C\ ]\!]$
$\Longrightarrow P,E \vdash \langle (\!| C |\!)\, e,s\rangle \to* \langle THROW\ ClassCast,s'\rangle$

⟨*proof*⟩

lemma *StaticCastRedsThrow*:
$[\![\ P,E \vdash \langle e,s\rangle \to* \langle Throw\ r,s'\rangle\ ]\!] \Longrightarrow P,E \vdash \langle (\!| C |\!)\, e,s\rangle \to* \langle Throw\ r,s'\rangle$

⟨*proof*⟩

lemma *DynCastReds*:
$P,E \vdash \langle e,s\rangle \to* \langle e',s'\rangle \Longrightarrow P,E \vdash \langle Cast\ C\ e,s\rangle \to* \langle Cast\ C\ e',s'\rangle$

⟨*proof*⟩

lemma *DynCastRedsNull*:

$P,E \vdash \langle e,s\rangle \to* \langle null,s'\rangle \Longrightarrow P,E \vdash \langle Cast\ C\ e,s\rangle \to* \langle null,s'\rangle$

⟨*proof*⟩

lemma *DynCastRedsRef*:
$[\![P,E \vdash \langle e,s\rangle \to* \langle ref(a,Cs),s'\rangle;\ hp\ s'\ a = Some\ (D,S);\ P \vdash Path\ D\ to\ C\ via\ Cs';$
$P \vdash Path\ D\ to\ C\ unique\ ]\!]$
$\Longrightarrow P,E \vdash \langle Cast\ C\ e,s\rangle \to* \langle ref(a,Cs'),s'\rangle$

⟨*proof*⟩

lemma *StaticUpDynCastReds*:
$[\![P,E \vdash \langle e,s\rangle \to* \langle ref(a,Cs),s'\rangle;\ P \vdash Path\ last\ Cs\ to\ C\ unique;$
$P \vdash Path\ last\ Cs\ to\ C\ via\ Cs';\ Ds = Cs@_p Cs'\ ]\!]$
$\Longrightarrow P,E \vdash \langle Cast\ C\ e,s\rangle \to* \langle ref(a,Ds),s'\rangle$

⟨*proof*⟩

lemma *StaticDownDynCastReds*:
$P,E \vdash \langle e,s\rangle \to* \langle ref(a,Cs@[C]@Cs'),s'\rangle$
$\Longrightarrow\ P,E \vdash \langle Cast\ C\ e,s\rangle \to* \langle ref(a,Cs@[C]),s'\rangle$

⟨*proof*⟩

lemma *DynCastRedsFail*:
$[\![P,E \vdash \langle e,s\rangle \to* \langle ref(a,Cs),s'\rangle;\ hp\ s'\ a = Some\ (D,S);\ \neg\ P \vdash Path\ D\ to\ C$
unique;
$\neg\ P \vdash Path\ last\ Cs\ to\ C\ unique;\ C \notin set\ Cs\ ]\!]$
$\Longrightarrow P,E \vdash \langle Cast\ C\ e,s\rangle \to* \langle null,s'\rangle$

⟨*proof*⟩

lemma *DynCastRedsThrow*:
$[\![P,E \vdash \langle e,s\rangle \to* \langle Throw\ r,s'\rangle\ ]\!] \Longrightarrow P,E \vdash \langle Cast\ C\ e,s\rangle \to* \langle Throw\ r,s'\rangle$

⟨*proof*⟩

19.4 LAss

lemma *LAssReds*:
$P,E \vdash \langle e,s\rangle \to* \langle e',s'\rangle \Longrightarrow P,E \vdash \langle V{:=}e,s\rangle \to* \langle V{:=}e',s'\rangle$

⟨*proof*⟩

lemma *LAssRedsVal*:
$[\![P,E \vdash \langle e,s\rangle \to* \langle Val\ v,(h',l')\rangle;\ E\ V = Some\ T;\ P \vdash T\ casts\ v\ to\ v']\!]$
$\Longrightarrow P,E \vdash \langle\ V:=e,s\rangle \to* \langle Val\ v',(h',l'(V\mapsto v'))\rangle$

⟨*proof*⟩

lemma *LAssRedsThrow*:
$[\![P,E \vdash \langle e,s\rangle \to* \langle Throw\ r,s'\rangle\ ]\!] \Longrightarrow P,E \vdash \langle\ V:=e,s\rangle \to* \langle Throw\ r,s'\rangle$

⟨*proof*⟩

19.5 BinOp

lemma *BinOp1Reds*:
$P,E \vdash \langle e,s\rangle \to* \langle e',s'\rangle \Longrightarrow P,E \vdash \langle\ e\ «bop»\ e_2,\ s\rangle \to* \langle e'\ «bop»\ e_2,\ s'\rangle$

⟨*proof*⟩

lemma *BinOp2Reds*:
$P,E \vdash \langle e,s\rangle \to* \langle e',s'\rangle \Longrightarrow P,E \vdash \langle(Val\ v)\ «bop»\ e,\ s\rangle \to* \langle(Val\ v)\ «bop»\ e',\ s'\rangle$

⟨*proof*⟩

lemma *BinOpRedsVal*:
$[\![P,E \vdash \langle e_1,s_0\rangle \to* \langle Val\ v_1,s_1\rangle;\ P,E \vdash \langle e_2,s_1\rangle \to* \langle Val\ v_2,s_2\rangle;$
$binop(bop,v_1,v_2) = Some\ v\ ]\!]$
$\Longrightarrow P,E \vdash \langle e_1\ «bop»\ e_2,\ s_0\rangle \to* \langle Val\ v,s_2\rangle$

⟨*proof*⟩

lemma *BinOpRedsThrow1*:
$P,E \vdash \langle e,s\rangle \to* \langle Throw\ r,s'\rangle \Longrightarrow P,E \vdash \langle e\ «bop»\ e_2,\ s\rangle \to* \langle Throw\ r,\ s'\rangle$

⟨*proof*⟩

lemma *BinOpRedsThrow2*:
$[\![P,E \vdash \langle e_1,s_0\rangle \to* \langle Val\ v_1,s_1\rangle;\ P,E \vdash \langle e_2,s_1\rangle \to* \langle Throw\ r,s_2\rangle]\!]$
$\Longrightarrow P,E \vdash \langle e_1\ «bop»\ e_2,\ s_0\rangle \to* \langle Throw\ r,s_2\rangle$

⟨*proof*⟩

19.6 FAcc

lemma *FAccReds*:

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

⟨*proof*⟩

lemma *FAccRedsVal*:
$[\![P,E \vdash \langle e,s\rangle \to* \langle ref(a,Cs'),s'\rangle;\ hp\ s'\ a = Some(D,S);$
$Ds = Cs'@_p Cs;\ (Ds,fs) \in S;\ fs\ F = Some\ v\ ]\!]$
$\Longrightarrow P,E \vdash \langle e\cdot F\{Cs\},s\rangle \to* \langle Val\ v,s'\rangle$

⟨*proof*⟩

lemma *FAccRedsNull*:
$P,E \vdash \langle e,s\rangle \to* \langle null,s'\rangle \Longrightarrow P,E \vdash \langle e\cdot F\{Cs\},s\rangle \to* \langle THROW\ NullPointer,s'\rangle$

⟨*proof*⟩

lemma *FAccRedsThrow*:
$P,E \vdash \langle e,s\rangle \to* \langle Throw\ r,s'\rangle \Longrightarrow P,E \vdash \langle e\cdot F\{Cs\},s\rangle \to* \langle Throw\ r,s'\rangle$

⟨*proof*⟩

19.7 FAss

lemma *FAssReds1*:
$P,E \vdash \langle e,s\rangle \to* \langle e',s'\rangle \Longrightarrow P,E \vdash \langle e\cdot F\{Cs\}:=e_2,\ s\rangle \to* \langle e'\cdot F\{Cs\}:=e_2,\ s'\rangle$

⟨*proof*⟩

lemma *FAssReds2*:
$P,E \vdash \langle e,s\rangle \to* \langle e',s'\rangle \Longrightarrow P,E \vdash \langle Val\ v\cdot F\{Cs\}:=e,\ s\rangle \to* \langle Val\ v\cdot F\{Cs\}:=e',\ s'\rangle$

⟨*proof*⟩

lemma *FAssRedsVal*:
$[\![\ P,E \vdash \langle e_1,s_0\rangle \to* \langle ref(a,Cs'),s_1\rangle;\ P,E \vdash \langle e_2,s_1\rangle \to* \langle Val\ v,(h_2,l_2)\rangle;$
$h_2\ a = Some(D,S);\ P \vdash (last\ Cs')\ has\ least\ F{:}T\ via\ Cs;\ P \vdash T\ casts\ v\ to\ v';$
$Ds = Cs'@_p Cs;\ (Ds,fs) \in S\ ]\!] \Longrightarrow$
$P,E \vdash \langle e_1\cdot F\{Cs\}:=e_2,\ s_0\rangle \to*$
$\langle Val\ v',(h_2(a\mapsto(D,insert\ (Ds,fs(F\mapsto v'))\ (S - \{(Ds,fs)\}))),l_2)\rangle$

⟨*proof*⟩

lemma *FAssRedsNull*:

$[\![P,E \vdash \langle e_1,s_0\rangle \to* \langle null,s_1\rangle;\ P,E \vdash \langle e_2,s_1\rangle \to* \langle Val\ v,s_2\rangle\ ]\!] \Longrightarrow$
$P,E \vdash \langle e_1 \cdot F\{Cs\}:=e_2,\ s_0\rangle \to* \langle THROW\ NullPointer,\ s_2\rangle$

⟨*proof*⟩

lemma *FAssRedsThrow1*:
$P,E \vdash \langle e,s\rangle \to* \langle Throw\ r,s'\rangle \Longrightarrow P,E \vdash \langle e \cdot F\{Cs\}:=e_2,\ s\rangle \to* \langle Throw\ r,\ s'\rangle$

⟨*proof*⟩

lemma *FAssRedsThrow2*:
$[\![P,E \vdash \langle e_1,s_0\rangle \to* \langle Val\ v,s_1\rangle;\ P,E \vdash \langle e_2,s_1\rangle \to* \langle Throw\ r,s_2\rangle\ ]\!]$
$\Longrightarrow P,E \vdash \langle e_1 \cdot F\{Cs\}:=e_2,s_0\rangle \to* \langle Throw\ r,s_2\rangle$

⟨*proof*⟩

19.8 ;;

lemma *SeqReds*:
$P,E \vdash \langle e,s\rangle \to* \langle e',s'\rangle \Longrightarrow P,E \vdash \langle e;;e_2,\ s\rangle \to* \langle e';;e_2,\ s'\rangle$

⟨*proof*⟩

lemma *SeqRedsThrow*:
$P,E \vdash \langle e,s\rangle \to* \langle Throw\ r,s'\rangle \Longrightarrow P,E \vdash \langle e;;e_2,\ s\rangle \to* \langle Throw\ r,\ s'\rangle$

⟨*proof*⟩

lemma *SeqReds2*:
$[\![P,E \vdash \langle e_1,s_0\rangle \to* \langle Val\ v_1,s_1\rangle;\ P,E \vdash \langle e_2,s_1\rangle \to* \langle e_2{}',s_2\rangle\ ]\!] \Longrightarrow P,E \vdash \langle e_1;;e_2,$
$s_0\rangle \to* \langle e_2{}',s_2\rangle$

⟨*proof*⟩

19.9 If

lemma *CondReds*:
$P,E \vdash \langle e,s\rangle \to* \langle e',s'\rangle \Longrightarrow P,E \vdash \langle if\ (e)\ e_1\ else\ e_2,s\rangle \to* \langle if\ (e')\ e_1\ else\ e_2,s'\rangle$

⟨*proof*⟩

lemma *CondRedsThrow*:
$P,E \vdash \langle e,s\rangle \to* \langle Throw\ r,s'\rangle \Longrightarrow P,E \vdash \langle if\ (e)\ e_1\ else\ e_2,\ s\rangle \to* \langle Throw\ r,s'\rangle$

⟨*proof*⟩

lemma *CondReds2T*:
$[\![P,E \vdash \langle e,s_0\rangle \to* \langle true,s_1\rangle;\ P,E \vdash \langle e_1,\ s_1\rangle \to* \langle e',s_2\rangle\ ]\!] \Longrightarrow P,E \vdash \langle if\ (e)\ e_1\ else\ e_2,\ s_0\rangle \to* \langle e',s_2\rangle$

⟨*proof*⟩

lemma *CondReds2F*:
$[\![P,E \vdash \langle e,s_0\rangle \to* \langle false,s_1\rangle;\ P,E \vdash \langle e_2,\ s_1\rangle \to* \langle e',s_2\rangle\ ]\!] \Longrightarrow P,E \vdash \langle if\ (e)\ e_1\ else\ e_2,\ s_0\rangle \to* \langle e',s_2\rangle$

⟨*proof*⟩

19.10 While

lemma *WhileFReds*:
$P,E \vdash \langle b,s\rangle \to* \langle false,s'\rangle \Longrightarrow P,E \vdash \langle while\ (b)\ c,s\rangle \to* \langle unit,s'\rangle$

⟨*proof*⟩

lemma *WhileRedsThrow*:
$P,E \vdash \langle b,s\rangle \to* \langle Throw\ r,s'\rangle \Longrightarrow P,E \vdash \langle while\ (b)\ c,s\rangle \to* \langle Throw\ r,s'\rangle$

⟨*proof*⟩

lemma *WhileTReds*:
$[\![\ P,E \vdash \langle b,s_0\rangle \to* \langle true,s_1\rangle;\ P,E \vdash \langle c,s_1\rangle \to* \langle Val\ v_1,s_2\rangle;\ P,E \vdash \langle while\ (b)\ c,s_2\rangle \to* \langle e,s_3\rangle\ ]\!]$
$\Longrightarrow P,E \vdash \langle while\ (b)\ c,s_0\rangle \to* \langle e,s_3\rangle$

⟨*proof*⟩

lemma *WhileTRedsThrow*:
$[\![\ P,E \vdash \langle b,s_0\rangle \to* \langle true,s_1\rangle;\ P,E \vdash \langle c,s_1\rangle \to* \langle Throw\ r,s_2\rangle\ ]\!]$
$\Longrightarrow P,E \vdash \langle while\ (b)\ c,s_0\rangle \to* \langle Throw\ r,s_2\rangle$

⟨*proof*⟩

19.11 Throw

lemma *ThrowReds*:
$P,E \vdash \langle e,s\rangle \to* \langle e',s'\rangle \Longrightarrow P,E \vdash \langle throw\ e,s\rangle \to* \langle throw\ e',s'\rangle$

⟨*proof*⟩

lemma *ThrowRedsNull*:
$P,E \vdash \langle e,s\rangle \to* \langle null,s'\rangle \Longrightarrow P,E \vdash \langle throw\ e,s\rangle \to* \langle THROW\ NullPointer,s'\rangle$

⟨*proof*⟩

lemma *ThrowRedsThrow*:
$P,E \vdash \langle e,s\rangle \to* \langle Throw\ r,s'\rangle \Longrightarrow P,E \vdash \langle throw\ e,s\rangle \to* \langle Throw\ r,s'\rangle$

⟨*proof*⟩

19.12 InitBlock

lemma assumes *wf*:*wf-prog wf-md P*
shows *InitBlockReds-aux*:
$P,E(V \mapsto T) \vdash \langle e,s\rangle \to* \langle e',s'\rangle \Longrightarrow$
$\forall h\ l\ h'\ l'\ v\ v'.\ s = (h,l(V \mapsto v')) \longrightarrow$
$P \vdash T\ casts\ v\ to\ v' \longrightarrow s' = (h',l') \longrightarrow$
$(\exists v''\ w.\ P,E \vdash \langle\{V:T := Val\ v;\ e\},(h,l)\rangle \to*$
$\langle\{V:T := Val\ v'';\ e'\},(h',l'(V:=(l\ V)))\rangle \wedge$
$P \vdash T\ casts\ v''\ to\ w)$

⟨*proof*⟩

lemma *InitBlockReds*:
$[\![P,E(V \mapsto T) \vdash \langle e,\ (h,l(V \mapsto v'))\rangle \to* \langle e',\ (h',l')\rangle;$
$P \vdash T\ casts\ v\ to\ v';\ wf\text{-}prog\ wf\text{-}md\ P\ ]\!] \Longrightarrow$
$\exists v''\ w.\ P,E \vdash \langle\{V:T := Val\ v;\ e\},\ (h,l)\rangle \to*$
$\langle\{V:T := Val\ v'';\ e'\},\ (h',l'(V:=(l\ V)))\rangle \wedge$
$P \vdash T\ casts\ v''\ to\ w$

⟨*proof*⟩

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

⟨*proof*⟩

19.13 Block

lemma *BlockRedsFinal*:
assumes *reds*: $P,E(V \mapsto T) \vdash \langle e_0,s_0\rangle \to* \langle e_2,(h_2,l_2)\rangle$ **and** *fin*: *final* e_2
and *wf*:*wf-prog wf-md P*
shows $\bigwedge h_0\ l_0.\ s_0 = (h_0,l_0(V:=None)) \Longrightarrow P,E \vdash \langle\{V:T;\ e_0\},(h_0,l_0)\rangle \to* \langle e_2,(h_2,l_2(V:=l_0\ V))\rangle$

⟨*proof*⟩

19.14 List

lemma *ListReds1*:
$P,E \vdash \langle e,s\rangle \to* \langle e',s'\rangle \Longrightarrow P,E \vdash \langle e\#es,s\rangle\ [\to]*\ \langle e'\ \#\ es,s'\rangle$

⟨*proof*⟩

lemma *ListReds2*:
$P,E \vdash \langle es,s\rangle\ [\to]*\ \langle es',s'\rangle \Longrightarrow P,E \vdash \langle Val\ v\ \#\ es,s\rangle\ [\to]*\ \langle Val\ v\ \#\ es',s'\rangle$

⟨*proof*⟩

lemma *ListRedsVal*:
$[\![\ P,E \vdash \langle e,s_0\rangle \to* \langle Val\ v,s_1\rangle;\ P,E \vdash \langle es,s_1\rangle\ [\to]*\ \langle es',s_2\rangle\ ]\!]$
$\Longrightarrow P,E \vdash \langle e\#es,s_0\rangle\ [\to]*\ \langle Val\ v\ \#\ es',s_2\rangle$

⟨*proof*⟩

19.15 Call

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

lemma assumes *wf*: *wwf-prog P*
shows *Red-fv*: $P,E \vdash \langle e,(h,l)\rangle \to \langle e',(h',l')\rangle \Longrightarrow fv\ e' \subseteq fv\ e$
and $P,E \vdash \langle es,(h,l)\rangle\ [\to]\ \langle es',(h',l')\rangle \Longrightarrow fvs\ es' \subseteq fvs\ es$
⟨*proof*⟩

lemma *Red-dom-lcl*:
$P,E \vdash \langle e,(h,l)\rangle \to \langle e',(h',l')\rangle \Longrightarrow dom\ l' \subseteq dom\ l \cup fv\ e$ **and**
$P,E \vdash \langle es,(h,l)\rangle\ [\to]\ \langle es',(h',l')\rangle \Longrightarrow dom\ l' \subseteq dom\ l \cup fvs\ es$

⟨*proof*⟩

lemma *Reds-dom-lcl*:
$[\![\ wwf\text{-}prog\ P;\ P,E \vdash \langle e,(h,l)\rangle \to* \langle e',(h',l')\rangle\ ]\!] \Longrightarrow dom\ l' \subseteq dom\ l \cup fv\ e$

⟨*proof*⟩

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

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

⟨*proof*⟩

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

lemma assumes *wf*:*wf-prog wf-md P*
shows *blocksReds*:
$\bigwedge l_0\ E\ vs'.\ [\![\ length\ Vs = length\ Ts;\ length\ vs = length\ Ts;$
$distinct\ Vs;\ \#\!\#T\!\#\!\#set\#\!\#Ts\#\!\#is\#\!\#type\#\!\#P\#\!\#T;\ P \vdash Ts\ Casts\ vs\ to\ vs';$
$P,E(Vs\ [\mapsto]\ Ts) \vdash \langle e, (h_0,l_0(Vs\ [\mapsto]\ vs'))\rangle \to* \langle e', (h_1,l_1)\rangle\ ]\!]$
$\Longrightarrow \exists\, vs''.\ P,E \vdash \langle blocks(Vs,Ts,vs,e), (h_0,l_0)\rangle \to*$
$\langle blocks(Vs,Ts,vs'',e'), (h_1,override\text{-}on\ l_1\ l_0\ (set\ Vs))\rangle \wedge$
$(\exists\, ws.\ P \vdash Ts\ Casts\ vs''\ to\ ws) \wedge length\ vs = length\ vs''$

⟨*proof*⟩

lemma assumes *wf*:*wf-prog wf-md P*
shows *blocksFinal*:
$\bigwedge E\ l\ vs'.\ [\![\ length\ Vs = length\ Ts;\ length\ vs = length\ Ts;$
$\#\!\#T\#\!\#set\#\!\#Ts\#\!\#is\#\!\#type\#\!\#P\#\!\#T;\ final\ e;\ P \vdash Ts\ Casts\ vs\ to\ vs'\ ]\!] \Longrightarrow$
$P,E \vdash \langle blocks(Vs,Ts,vs,e), (h,l)\rangle \to* \langle e, (h,l)\rangle$

⟨*proof*⟩

lemma assumes *wfmd*:*wf-prog wf-md P*
and *wf*: *length Vs* = *length Ts length vs* = *length Ts distinct Vs*
and *casts*:$P \vdash Ts\ Casts\ vs\ to\ vs'$
and *reds*: $P,E(Vs\ [\mapsto]\ Ts) \vdash \langle e, (h_0,\ l_0(Vs\ [\mapsto]\ vs'))\rangle \to* \langle e', (h_1,\ l_1)\rangle$
and *fin*: *final e'* **and** *l2*: $l_2 = override\text{-}on\ l_1\ l_0\ (set\ Vs)$
shows *blocksRedsFinal*: $P,E \vdash \langle blocks(Vs,Ts,vs,e), (h_0,\ l_0)\rangle \to* \langle e', (h_1,l_2)\rangle$

⟨*proof*⟩

An now the actual method call reduction lemmas.

lemma *CallRedsObj*:
$P,E \vdash \langle e,s\rangle \to* \langle e',s'\rangle \Longrightarrow$
$P,E \vdash \langle Call\ e\ Copt\ M\ es,s\rangle \to* \langle Call\ e'\ Copt\ M\ es,s'\rangle$

⟨*proof*⟩

lemma *CallRedsParams*:
$P,E \vdash \langle es,s\rangle\ [\to]*\ \langle es',s'\rangle \Longrightarrow$
$P,E \vdash \langle Call\ (Val\ v)\ Copt\ M\ es,s\rangle \to* \langle Call\ (Val\ v)\ Copt\ M\ es',s'\rangle$

⟨*proof*⟩

lemma *cast-lcl*:
$P,E \vdash \langle (\!| C |\!)(Val\ v),(h,l)\rangle \to \langle Val\ v',(h,l)\rangle \Longrightarrow$
$P,E \vdash \langle (\!| C |\!)(Val\ v),(h,l')\rangle \to \langle Val\ v',(h,l')\rangle$

⟨*proof*⟩

lemma *cast-env*:
$P,E \vdash \langle (\!| C |\!)(Val\ v),(h,l)\rangle \to \langle Val\ v',(h,l)\rangle \Longrightarrow$
$P,E' \vdash \langle (\!| C |\!)(Val\ v),(h,l)\rangle \to \langle Val\ v',(h,l)\rangle$

⟨*proof*⟩

lemma *Cast-step-Cast-or-fin*:
$P,E \vdash \langle (\!| C |\!)e,s\rangle \to* \langle e',s'\rangle \Longrightarrow final\ e' \vee (\exists\, e''.\ e' = (\!| C |\!)e'')$
⟨*proof*⟩

lemma *Cast-red*:$P,E \vdash \langle e,s\rangle \to* \langle e',s'\rangle \Longrightarrow$
$(\bigwedge e_1.\ [\![e = (\!| C |\!)e_0;\ e' = (\!| C |\!)e_1]\!] \Longrightarrow P,E \vdash \langle e_0,s\rangle \to* \langle e_1,s'\rangle)$

⟨*proof*⟩

lemma *Cast-final*:$[\![P,E \vdash \langle (\!| C |\!)e,s\rangle \to* \langle e',s'\rangle;\ final\ e']\!] \Longrightarrow$
$\exists\, e''\ s''.\ P,E \vdash \langle e,s\rangle \to* \langle e'',s''\rangle \wedge P,E \vdash \langle (\!| C |\!)e'',s''\rangle \to \langle e',s'\rangle \wedge final\ e''$

⟨*proof*⟩

lemma *Cast-final-eq*:
assumes *red*:$P,E \vdash \langle (\!| C |\!)e,(h,l)\rangle \to \langle e',(h,l)\rangle$
and *final*:*final e* **and** *final'*:*final e'*
shows $P,E' \vdash \langle (\!| C |\!)e,(h,l')\rangle \to \langle e',(h,l')\rangle$

⟨*proof*⟩

lemma *CallRedsFinal*:
assumes *wwf*: *wwf-prog P*
and $P,E \vdash \langle e,s_0\rangle \to* \langle ref(a,Cs),s_1\rangle$
$P,E \vdash \langle es,s_1\rangle\ [\to]*\ \langle map\ Val\ vs,(h_2,l_2)\rangle$
and *hp*: $h_2\ a = Some(C,S)$

and *method*: $P \vdash$ *last Cs has least M* = (*Ts'*,*T'*,*pns'*,*body'*) *via Ds*
and *select*: $P \vdash (C,Cs@_pDs)$ *selects M* = (*Ts*,*T*,*pns*,*body*) *via Cs'*
and *size*: *size vs* = *size pns*
and *casts*: $P \vdash$ *Ts Casts vs to vs'*
and l_2': $l_2' = [this \mapsto Ref(a,Cs'),\ pns[\mapsto]vs']$
and *body-case*:*new-body* = (*case T' of Class D* $\Rightarrow$ $(\!|D|\!)$*body* | - $\Rightarrow$ *body*)
and *body*: $P,E(this \mapsto Class\ (last\ Cs'),\ pns\ [\mapsto]\ Ts) \vdash \langle new\text{-}body,(h_2,l_2')\rangle \to* \langle ef,(h_3,l_3)\rangle$
and *final*:*final ef*
shows $P,E \vdash \langle e{\cdot}M(es),\ s_0\rangle \to* \langle ef,(h_3,l_2)\rangle$
⟨*proof*⟩

lemma *StaticCallRedsFinal*:
assumes *wwf*: *wwf-prog P*
and $P,E \vdash \langle e,s_0\rangle \to* \langle ref(a,Cs),s_1\rangle$
$P,E \vdash \langle es,s_1\rangle\ [\to]* \langle map\ Val\ vs,(h_2,l_2)\rangle$
and *path-unique*: $P \vdash$ *Path* (*last Cs*) *to C unique*
and *path-via*: $P \vdash$ *Path* (*last Cs*) *to C via Cs''*
and *Ds*: $Ds = (Cs@_pCs'')@_pCs'$
and *least*: $P \vdash$ *C has least M* = (*Ts*,*T*,*pns*,*body*) *via Cs'*
and *size*: *size vs* = *size pns*
and *casts*: $P \vdash$ *Ts Casts vs to vs'*
and l_2': $l_2' = [this \mapsto Ref(a,Ds),\ pns[\mapsto]vs']$
and *body*: $P,E(this{\mapsto}Class(last\ Ds),\ pns[\mapsto]\,Ts) \vdash \langle body,(h_2,l_2')\rangle \to* \langle ef,(h_3,l_3)\rangle$
and *final*:*final ef*
shows $P,E \vdash \langle e{\cdot}(C::)M(es),\ s_0\rangle \to* \langle ef,(h_3,l_2)\rangle$
⟨*proof*⟩

lemma *CallRedsThrowParams*:
⟦ $P,E \vdash \langle e,s_0\rangle \to* \langle Val\ v,s_1\rangle$;
$P,E \vdash \langle es,s_1\rangle\ [\to]* \langle map\ Val\ vs_1\ @\ Throw\ ex\ \#\ es_2,s_2\rangle$ ⟧
$\Longrightarrow P,E \vdash \langle Call\ e\ Copt\ M\ es,s_0\rangle \to* \langle Throw\ ex,s_2\rangle$

⟨*proof*⟩

lemma *CallRedsThrowObj*:
$P,E \vdash \langle e,s_0\rangle \to* \langle Throw\ ex,s_1\rangle \Longrightarrow P,E \vdash \langle Call\ e\ Copt\ M\ es,s_0\rangle \to* \langle Throw\ ex,s_1\rangle$

⟨*proof*⟩

lemma *CallRedsNull*:

$[\![P,E \vdash \langle e,s_0\rangle \to* \langle null,s_1\rangle;\ P,E \vdash \langle es,s_1\rangle\ [\to]*\ \langle map\ Val\ vs,s_2\rangle\ ]\!]$
$\Longrightarrow P,E \vdash \langle Call\ e\ Copt\ M\ es,s_0\rangle \to* \langle THROW\ NullPointer,s_2\rangle$

⟨*proof*⟩

19.16 The main Theorem

lemma assumes *wwf*: *wwf-prog P*
shows *big-by-small*: $P,E \vdash \langle e,s\rangle \Rightarrow \langle e',s'\rangle \Longrightarrow P,E \vdash \langle e,s\rangle \to* \langle e',s'\rangle$
and *bigs-by-smalls*: $P,E \vdash \langle es,s\rangle\ [\Rightarrow]\ \langle es',s'\rangle \Longrightarrow P,E \vdash \langle es,s\rangle\ [\to]*\ \langle es',s'\rangle$

⟨*proof*⟩

19.17 Big steps simulates small step

The big step equivalent of *RedWhile*:

lemma *unfold-while*:
$P,E \vdash \langle while(b)\ c,s\rangle \Rightarrow \langle e',s'\rangle\ =\ P,E \vdash \langle if(b)\ (c;;while(b)\ c)\ else\ (unit),s\rangle \Rightarrow \langle e',s'\rangle$

⟨*proof*⟩

lemma *blocksEval*:
$\bigwedge Ts\ vs\ l\ l'\ E.\ [\![size\ ps = size\ Ts;\ size\ ps = size\ vs;$
$P,E \vdash \langle blocks(ps,Ts,vs,e),(h,l)\rangle \Rightarrow \langle e',(h',l')\rangle\ ]\!]$
$\Longrightarrow \exists\ l''\ vs'.\ P,E(ps\ [\mapsto]\ Ts) \vdash \langle e,(h,l(ps[\mapsto]vs'))\rangle \Rightarrow \langle e',(h',l'')\rangle \wedge$
$P \vdash Ts\ Casts\ vs\ to\ vs' \wedge length\ vs' = length\ vs$

⟨*proof*⟩

lemma *CastblocksEval*:
$\bigwedge Ts\ vs\ l\ l'\ E.\ [\![size\ ps = size\ Ts;\ size\ ps = size\ vs;$
$P,E \vdash \langle (\!| C' |\!)(blocks(ps,Ts,vs,e)),(h,l)\rangle \Rightarrow \langle e',(h',l')\rangle\ ]\!]$
$\Longrightarrow \exists\ l''\ vs'.\ P,E(ps\ [\mapsto]\ Ts) \vdash \langle (\!| C' |\!)e,(h,l(ps[\mapsto]vs'))\rangle \Rightarrow \langle e',(h',l'')\rangle \wedge$
$P \vdash Ts\ Casts\ vs\ to\ vs' \wedge length\ vs' = length\ vs$

⟨*proof*⟩

lemma
assumes *wf*: *wwf-prog P*
shows *eval-restrict-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 $P,E \vdash \langle es,(h,l)\rangle\ [\Rightarrow]\ \langle es',(h',l')\rangle \Longrightarrow (\bigwedge W.\ fvs\ es \subseteq W \Longrightarrow P,E \vdash \langle es,(h,l|`W)\rangle\ [\Rightarrow]\ \langle es',(h',l'|`W)\rangle)$

⟨*proof*⟩

lemma *eval-notfree-unchanged*:
assumes *wf*:*wwf-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 $P,E \vdash \langle es,(h,l)\rangle\ [\Rightarrow]\ \langle es',(h',l')\rangle \Longrightarrow (\bigwedge V.\ V \notin fvs\ es \Longrightarrow l'\ V = l\ V)$

⟨*proof*⟩

lemma *eval-closed-lcl-unchanged*:
assumes *wf*:*wwf-prog P*
and *eval*:$P,E \vdash \langle e,(h,l)\rangle \Rightarrow \langle e',(h',l')\rangle$
and *fv*:$fv\ e = \{\}$
shows $l' = l$

⟨*proof*⟩

declare *split-paired-All* [*simp del*] *split-paired-Ex* [*simp del*]

⟨*ML*⟩

lemma *list-eval-Throw*:
assumes *eval-e*: $P,E \vdash \langle throw\ x,s\rangle \Rightarrow \langle e',s'\rangle$
shows $P,E \vdash \langle map\ Val\ vs\ @\ throw\ x\ \#\ es',s\rangle\ [\Rightarrow]\ \langle map\ Val\ vs\ @\ e'\ \#\ es',s'\rangle$

⟨*proof*⟩

The key lemma:

lemma
assumes *wf*: *wwf-prog P*
shows *extend-1-eval*:
$P,E \vdash \langle e,s\rangle \rightarrow \langle e'',s''\rangle \Longrightarrow\ (\bigwedge s'\ e'.\ P,E \vdash \langle e'',s''\rangle \Rightarrow \langle e',s'\rangle \Longrightarrow P,E \vdash \langle e,s\rangle \Rightarrow \langle e',s'\rangle)$
and *extend-1-evals*:
$P,E \vdash \langle es,t\rangle\ [\rightarrow]\ \langle es'',t''\rangle \Longrightarrow (\bigwedge t'\ es'.\ P,E \vdash \langle es'',t''\rangle\ [\Rightarrow]\ \langle es',t'\rangle \Longrightarrow P,E \vdash \langle es,t\rangle\ [\Rightarrow]\ \langle es',t'\rangle)$

⟨*proof*⟩

declare *split-paired-All* [*simp*] *split-paired-Ex* [*simp*]
⟨*ML*⟩

Its extension to $\rightarrow*$:

lemma *extend-eval*:
assumes *wf*: *wwf-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*: *wwf-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*: *wwf-prog P*
shows *small-by-big*: $[\![P,E \vdash \langle e,s\rangle \rightarrow* \langle e',s'\rangle; \mathit{final}\ e']\!] \Longrightarrow P,E \vdash \langle e,s\rangle \Rightarrow \langle e',s'\rangle$
and $[\![P,E \vdash \langle es,s\rangle\ [\rightarrow]* \langle es',s'\rangle; \mathit{finals}\ es']\!] \Longrightarrow P,E \vdash \langle es,s\rangle\ [\Rightarrow]\ \langle es',s'\rangle$

⟨*proof*⟩

19.18 Equivalence

And now, the crowning achievement:

corollary *big-iff-small*:
$\mathit{wwf\text{-}prog}\ P \Longrightarrow$
$P,E \vdash \langle e,s\rangle \Rightarrow \langle e',s'\rangle \ = \ (P,E \vdash \langle e,s\rangle \rightarrow* \langle e',s'\rangle \wedge \mathit{final}\ e')$
⟨*proof*⟩

end

20 Definite assignment

theory *DefAss*
imports *BigStep*
begin

20.1 Hypersets

type-synonym *hyperset* = *vname set option*

definition *hyperUn* :: *hyperset* ⇒ *hyperset* ⇒ *hyperset* (**infixl** ⊔ *65*) **where**
A ⊔ *B* ≡ *case A of None* ⇒ *None*
| ⌊*A*⌋ ⇒ (*case B of None* ⇒ *None* | ⌊*B*⌋ ⇒ ⌊*A* ∪ *B*⌋)

definition *hyperInt* :: *hyperset* ⇒ *hyperset* ⇒ *hyperset* (**infixl** ⊓ *70*) **where**
A ⊓ *B* ≡ *case A of None* ⇒ *B*
| ⌊*A*⌋ ⇒ (*case B of None* ⇒ ⌊*A*⌋ | ⌊*B*⌋ ⇒ ⌊*A* ∩ *B*⌋)

definition *hyperDiff1* :: *hyperset* ⇒ *vname* ⇒ *hyperset* (**infixl** ⊖ *65*) **where**
A ⊖ *a* ≡ *case A of None* ⇒ *None* | ⌊*A*⌋ ⇒ ⌊*A* − {*a*}⌋

definition *hyper-isin* :: *vname* ⇒ *hyperset* ⇒ *bool* (**infix** ∈∈ *50*) **where**
a ∈∈ *A* ≡ *case A of None* ⇒ *True* | ⌊*A*⌋ ⇒ *a* ∈ *A*

definition *hyper-subset* :: *hyperset* ⇒ *hyperset* ⇒ *bool* (**infix** ⊑ *50*) **where**
A ⊑ *B* ≡ *case B of None* ⇒ *True*
| ⌊*B*⌋ ⇒ (*case A of None* ⇒ *False* | ⌊*A*⌋ ⇒ *A* ⊆ *B*)

lemmas *hyperset-defs* =
hyperUn-def hyperInt-def hyperDiff1-def hyper-isin-def hyper-subset-def

lemma [*simp*]: ⌊{}⌋ ⊔ *A* = *A* ∧ *A* ⊔ ⌊{}⌋ = *A*
⟨*proof*⟩

lemma [*simp*]: ⌊*A*⌋ ⊔ ⌊*B*⌋ = ⌊*A* ∪ *B*⌋ ∧ ⌊*A*⌋ ⊖ *a* = ⌊*A* − {*a*}⌋
⟨*proof*⟩

lemma [*simp*]: *None* ⊔ *A* = *None* ∧ *A* ⊔ *None* = *None*
⟨*proof*⟩

lemma [*simp*]: *a* ∈∈ *None* ∧ *None* ⊖ *a* = *None*
⟨*proof*⟩

lemma *hyperUn-assoc*: (*A* ⊔ *B*) ⊔ *C* = *A* ⊔ (*B* ⊔ *C*)
⟨*proof*⟩

lemma *hyper-insert-comm*: *A* ⊔ ⌊{*a*}⌋ = ⌊{*a*}⌋ ⊔ *A* ∧ *A* ⊔ (⌊{*a*}⌋ ⊔ *B*) = ⌊{*a*}⌋
⊔ (*A* ⊔ *B*)
⟨*proof*⟩

20.2 Definite assignment

primrec $\mathcal{A}$ :: *expr* ⇒ *hyperset* **and** $\mathcal{A}s$ :: *expr list* ⇒ *hyperset* **where**
$\mathcal{A}$ (*new C*) = ⌊{}⌋ |
$\mathcal{A}$ (*Cast C e*) = $\mathcal{A}$ *e* |
$\mathcal{A}$ (⦇*C*⦈*e*) = $\mathcal{A}$ *e* |

$\mathcal{A}\ (Val\ v) = \lfloor\{\}\rfloor$ |
$\mathcal{A}\ (e_1\ «bop»\ e_2) = \mathcal{A}\ e_1 \sqcup \mathcal{A}\ e_2$ |
$\mathcal{A}\ (Var\ V) = \lfloor\{\}\rfloor$ |
$\mathcal{A}\ (LAss\ V\ e) = \lfloor\{V\}\rfloor \sqcup \mathcal{A}\ e$ |
$\mathcal{A}\ (e{\cdot}F\{Cs\}) = \mathcal{A}\ e$ |
$\mathcal{A}\ (e_1{\cdot}F\{Cs\}{:=}e_2) = \mathcal{A}\ e_1 \sqcup \mathcal{A}\ e_2$ |
$\mathcal{A}\ (Call\ e\ Copt\ M\ es) = \mathcal{A}\ e \sqcup \mathcal{A}s\ es$ |
$\mathcal{A}\ (\{V{:}T;\ e\}) = \mathcal{A}\ e \ominus V$ |
$\mathcal{A}\ (e_1;;e_2) = \mathcal{A}\ e_1 \sqcup \mathcal{A}\ e_2$ |
$\mathcal{A}\ (if\ (e)\ e_1\ else\ e_2) = \mathcal{A}\ e \sqcup (\mathcal{A}\ e_1 \sqcap \mathcal{A}\ e_2)$ |
$\mathcal{A}\ (while\ (b)\ e) = \mathcal{A}\ b$ |
$\mathcal{A}\ (throw\ e) = None$ |

$\mathcal{A}s\ ([]) = \lfloor\{\}\rfloor$ |
$\mathcal{A}s\ (e\#es) = \mathcal{A}\ e \sqcup \mathcal{A}s\ es$

primrec $\mathcal{D} :: expr \Rightarrow hyperset \Rightarrow bool$ **and** $\mathcal{D}s :: expr\ list \Rightarrow hyperset \Rightarrow bool$
where
$\mathcal{D}\ (new\ C)\ A = True$ |
$\mathcal{D}\ (Cast\ C\ e)\ A = \mathcal{D}\ e\ A$ |
$\mathcal{D}\ (\!|C|\!)e)\ A = \mathcal{D}\ e\ A$ |
$\mathcal{D}\ (Val\ v)\ A = True$ |
$\mathcal{D}\ (e_1\ «bop»\ e_2)\ A = (\mathcal{D}\ e_1\ A \wedge \mathcal{D}\ e_2\ (A \sqcup \mathcal{A}\ e_1))$ |
$\mathcal{D}\ (Var\ V)\ A = (V \in\in A)$ |
$\mathcal{D}\ (LAss\ V\ e)\ A = \mathcal{D}\ e\ A$ |
$\mathcal{D}\ (e{\cdot}F\{Cs\})\ A = \mathcal{D}\ e\ A$ |
$\mathcal{D}\ (e_1{\cdot}F\{Cs\}{:=}e_2)\ A = (\mathcal{D}\ e_1\ A \wedge \mathcal{D}\ e_2\ (A \sqcup \mathcal{A}\ e_1))$ |
$\mathcal{D}\ (Call\ e\ Copt\ M\ es)\ A = (\mathcal{D}\ e\ A \wedge \mathcal{D}s\ es\ (A \sqcup \mathcal{A}\ e))$ |
$\mathcal{D}\ (\{V{:}T;\ e\})\ A = \mathcal{D}\ e\ (A \ominus V)$ |
$\mathcal{D}\ (e_1;;e_2)\ A = (\mathcal{D}\ e_1\ A \wedge \mathcal{D}\ e_2\ (A \sqcup \mathcal{A}\ e_1))$ |
$\mathcal{D}\ (if\ (e)\ e_1\ else\ e_2)\ A =$
$(\mathcal{D}\ e\ A \wedge \mathcal{D}\ e_1\ (A \sqcup \mathcal{A}\ e) \wedge \mathcal{D}\ e_2\ (A \sqcup \mathcal{A}\ e))$ |
$\mathcal{D}\ (while\ (e)\ c)\ A = (\mathcal{D}\ e\ A \wedge \mathcal{D}\ c\ (A \sqcup \mathcal{A}\ e))$ |
$\mathcal{D}\ (throw\ e)\ A = \mathcal{D}\ e\ A$ |

$\mathcal{D}s\ ([])\ A = True$ |
$\mathcal{D}s\ (e\#es)\ A = (\mathcal{D}\ e\ A \wedge \mathcal{D}s\ es\ (A \sqcup \mathcal{A}\ e))$

lemma *As-map-Val*[*simp*]: $\mathcal{A}s\ (map\ Val\ vs) = \lfloor\{\}\rfloor$
⟨*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*: $\bigwedge A.\ \mathcal{A}\ e = \lfloor A\rfloor \Longrightarrow A \subseteq fv\ e$
and $\bigwedge A.\ \mathcal{A}s\ es = \lfloor A\rfloor \Longrightarrow A \subseteq fvs\ es$

⟨*proof*⟩

lemma *sqUn-lem*: $A \sqsubseteq A' \Longrightarrow A \sqcup B \sqsubseteq A' \sqcup B$
⟨*proof*⟩

lemma *diff-lem*: $A \sqsubseteq A' \Longrightarrow A \ominus b \sqsubseteq A' \ominus b$
⟨*proof*⟩

lemma *D-mono*: $\bigwedge A\ A'.\ A \sqsubseteq A' \Longrightarrow \mathcal{D}\ e\ A \Longrightarrow \mathcal{D}\ (e{::}expr)\ A'$
and *Ds-mono*: $\bigwedge A\ A'.\ A \sqsubseteq A' \Longrightarrow \mathcal{D}s\ es\ A \Longrightarrow \mathcal{D}s\ (es{::}expr\ list)\ A'$

⟨*proof*⟩

lemma *D-mono'*: $\mathcal{D}\ e\ A \Longrightarrow A \sqsubseteq A' \Longrightarrow \mathcal{D}\ e\ A'$
and *Ds-mono'*: $\mathcal{D}s\ es\ A \Longrightarrow A \sqsubseteq A' \Longrightarrow \mathcal{D}s\ es\ A'$
⟨*proof*⟩

end

21 Runtime Well-typedness

theory *WellTypeRT* **imports** *WellType* **begin**

21.1 Run time types

primrec *typeof-h* :: *prog* ⇒ *heap* ⇒ *val* ⇒ *ty option* (- ⊢ *typeof*$_h$ -) **where**
  $P \vdash typeof_h\ Unit = Some\ Void$
| $P \vdash typeof_h\ Null = Some\ NT$
| $P \vdash typeof_h\ (Bool\ b) = Some\ Boolean$
| $P \vdash typeof_h\ (Intg\ i) = Some\ Integer$
| $P \vdash typeof_h\ (Ref\ r) = (case\ h\ (the\text{-}addr\ (Ref\ r))\ of\ None \Rightarrow None$
  | *Some*(*C*,*S*) ⇒ (*if Subobjs P C* (*the-path*(*Ref r*)) *then*
  Some(*Class*(*last*(*the-path*(*Ref r*))))
  else None))

lemma *type-eq-type*: *typeof v* = *Some T* ⟹ $P \vdash typeof_h\ v = Some\ T$
⟨*proof*⟩

lemma *typeof-Void* [*simp*]: $P \vdash typeof_h\ v = Some\ Void \Longrightarrow v = Unit$
⟨*proof*⟩

lemma *typeof-NT* [*simp*]: $P \vdash typeof_h\ v = Some\ NT \Longrightarrow v = Null$
⟨*proof*⟩

lemma *typeof-Boolean* [*simp*]: $P \vdash typeof_h\ v = Some\ Boolean \Longrightarrow \exists\, b.\ v = Bool\ b$
⟨*proof*⟩

lemma *typeof-Integer* [*simp*]: $P \vdash typeof_h\ v = Some\ Integer \Longrightarrow \exists\, i.\ v = Intg\ i$
⟨*proof*⟩

lemma *typeof-Class-Subo*:
$P \vdash typeof_h\ v = Some\ (Class\ C) \Longrightarrow$
$\exists\, a\ Cs\ D\ S.\ v = Ref(a,Cs) \wedge h\ a = Some(D,S) \wedge Subobjs\ P\ D\ Cs \wedge last\ Cs = C$
⟨*proof*⟩

21.2 The rules

inductive
  WTrt :: [*prog*,*env*,*heap*,*expr*, *ty*] ⇒ *bool*
    (-,-,- ⊢ - : - [*51*,*51*,*51*]*50*)
  and *WTrts* :: [*prog*,*env*,*heap*,*expr list*,*ty list*] ⇒ *bool*
    (-,-,- ⊢ - [:] - [*51*,*51*,*51*]*50*)
  for *P* :: *prog*
where

  WTrtNew:
  is-class P C ⟹
  P,*E*,*h* ⊢ *new C* : *Class C*

| *WTrtDynCast*:
  ⟦ *P*,*E*,*h* ⊢ *e* : *T*; *is-refT T*; *is-class P C* ⟧
  ⟹ *P*,*E*,*h* ⊢ *Cast C e* : *Class C*

| *WTrtStaticCast*:
  ⟦ *P*,*E*,*h* ⊢ *e* : *T*; *is-refT T*; *is-class P C* ⟧
  ⟹ *P*,*E*,*h* ⊢ (⌊*C*⌋)*e* : *Class C*

| *WTrtVal*:
  P ⊢ $typeof_h$ *v* = *Some T* ⟹
  P,*E*,*h* ⊢ *Val v* : *T*

| *WTrtVar*:
  E V = *Some T* ⟹
  P,*E*,*h* ⊢ *Var V* : *T*

| *WTrtBinOp*:
  ⟦ *P*,*E*,*h* ⊢ e_1 : T_1; *P*,*E*,*h* ⊢ e_2 : T_2;
    case bop of Eq ⇒ *T* = *Boolean*
      | *Add* ⇒ T_1 = *Integer* ∧ T_2 = *Integer* ∧ *T* = *Integer* ⟧
  ⟹ *P*,*E*,*h* ⊢ e_1 «*bop*» e_2 : *T*

| *WTrtLAss*:
  ⟦ *E V* = *Some T*; *P*,*E*,*h* ⊢ *e* : *T*′; *P* ⊢ *T*′ ≤ *T* ⟧

$\Longrightarrow P,E,h \vdash V{:=}e : T$

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

| *WTrtFAccNT*:
$P,E,h \vdash e : NT \Longrightarrow P,E,h \vdash e{\cdot}F\{Cs\} : T$

| *WTrtFAss*:
$[\![P,E,h \vdash e_1 : Class\ C;\ Cs \neq [];$
$P \vdash C\ has\ least\ F{:}T\ via\ Cs;\ P,E,h \vdash e_2 : T';\ P \vdash T' \leq T\ ]\!]$
$\Longrightarrow P,E,h \vdash e_1{\cdot}F\{Cs\}{:=}e_2 : T$

| *WTrtFAssNT*:
$[\![\ P,E,h \vdash e_1 : NT;\ P,E,h \vdash e_2 : T';\ P \vdash T' \leq T\ ]\!]$
$\Longrightarrow P,E,h \vdash e_1{\cdot}F\{Cs\}{:=}e_2 : T$

| *WTrtCall*:
$[\![\ P,E,h \vdash e : Class\ C;\ \ P \vdash C\ has\ least\ M = (Ts,T,m)\ via\ Cs;$
$P,E,h \vdash es\ [:]\ Ts';\ P \vdash Ts'\ [\leq]\ Ts\ ]\!]$
$\Longrightarrow P,E,h \vdash e{\cdot}M(es) : T$

| *WTrtStaticCall*:
$[\![\ P,E,h \vdash e : Class\ C';\ P \vdash Path\ C'\ to\ C\ unique;$
$P \vdash C\ has\ least\ M = (Ts,T,m)\ via\ Cs;$
$P,E,h \vdash es\ [:]\ Ts';\ P \vdash Ts'\ [\leq]\ Ts\ ]\!]$
$\Longrightarrow P,E,h \vdash e{\cdot}(C{::})M(es) : T$

| *WTrtCallNT*:
$[\![P,E,h \vdash e : NT;\ P,E,h \vdash es\ [:]\ Ts]\!] \Longrightarrow P,E,h \vdash Call\ e\ Copt\ M\ es : T$

| *WTrtBlock*:
$[\![P,E(V{\mapsto}T),h \vdash e : T';\ is\text{-}type\ P\ T]\!] \Longrightarrow$
$P,E,h \vdash \{V{:}T;\ e\} : T'$

| *WTrtSeq*:
$[\![\ P,E,h \vdash e_1 : T_1;\ \ P,E,h \vdash e_2 : T_2\ ]\!] \Longrightarrow P,E,h \vdash e_1;;e_2 : T_2$

| *WTrtCond*:
$[\![\ P,E,h \vdash e : Boolean;\ \ P,E,h \vdash e_1 : T;\ \ P,E,h \vdash e_2 : T\ ]\!]$
$\Longrightarrow P,E,h \vdash if\ (e)\ e_1\ else\ e_2 : T$

| *WTrtWhile*:
$[\![\ P,E,h \vdash e : Boolean;\ \ P,E,h \vdash c : T\ ]\!]$
$\Longrightarrow\ P,E,h \vdash while(e)\ c : Void$

| *WTrtThrow*:
$[\![P,E,h \vdash e : T';\ is\text{-}refT\ T']\!]$

$\Longrightarrow$ $P,E,h \vdash$ *throw* $e : T$

| *WTrtNil*:
$P,E,h \vdash [] \; [:] \; []$

| *WTrtCons*:
$[\![P,E,h \vdash e : T;\ P,E,h \vdash es \; [:] \; Ts]\!] \Longrightarrow P,E,h \vdash e\#es \; [:] \; T\#Ts$

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

inductive-simps [*iff*]:
$P,E,h \vdash [] \; [:] \; Ts$
$P,E,h \vdash e\#es \; [:] \; T\#Ts$
$P,E,h \vdash (e\#es) \; [:] \; Ts$
$P,E,h \vdash Val\ v : T$
$P,E,h \vdash Var\ V : T$
$P,E,h \vdash e_1;;e_2 : T_2$
$P,E,h \vdash \{V{:}T;\ e\} : T'$

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*⟩

inductive-cases *WTrt-elim-cases*[*elim*!]:
$P,E,h \vdash new\ C : T$
$P,E,h \vdash Cast\ C\ e : T$
$P,E,h \vdash (\!|C|\!)e : T$
$P,E,h \vdash e_1\ «bop»\ e_2 : T$
$P,E,h \vdash V{:=}e : T$
$P,E,h \vdash e{\cdot}F\{Cs\} : T$

$P,E,h \vdash e{\cdot}F\{Cs\} := v : T$
$P,E,h \vdash e{\cdot}M(es) : T$
$P,E,h \vdash e{\cdot}(C::)M(es) : T$
$P,E,h \vdash \mathit{if}\ (e)\ e_1\ \mathit{else}\ e_2 : T$
$P,E,h \vdash \mathit{while}(e)\ c : T$
$P,E,h \vdash \mathit{throw}\ e : T$

21.4 Some interesting lemmas

lemma *WTrts-Val*[*simp*]:
$\bigwedge Ts.\ (P,E,h \vdash map\ Val\ vs\ [:]\ Ts) = (map\ (\lambda v.\ (P \vdash typeof_h)\ v)\ vs = map\ Some\ Ts)$

⟨*proof*⟩

lemma *WTrts-same-length*: $\bigwedge Ts.\ P,E,h \vdash es\ [:]\ Ts \Longrightarrow length\ es = length\ Ts$
⟨*proof*⟩

lemma *WTrt-env-mono*:
$P,E,h \vdash e : T \Longrightarrow (\bigwedge E'.\ E \subseteq_m E' \Longrightarrow P,E',h \vdash e : T)$ **and**
$P,E,h \vdash es\ [:]\ Ts \Longrightarrow (\bigwedge E'.\ E \subseteq_m E' \Longrightarrow P,E',h \vdash es\ [:]\ Ts)$

⟨*proof*⟩

lemma *WT-implies-WTrt*: $P,E \vdash e :: T \Longrightarrow P,E,h \vdash e : T$
and *WTs-implies-WTrts*: $P,E \vdash es\ [::]\ Ts \Longrightarrow P,E,h \vdash es\ [:]\ Ts$

⟨*proof*⟩

end

22 Conformance Relations for Proofs

theory *Conform*
imports *Exceptions WellTypeRT*
begin

primrec *conf* :: *prog* ⇒ *heap* ⇒ *val* ⇒ *ty* ⇒ *bool* (-,- ⊢ - :≤ - [51,51,51,51] 50)
where
$P,h \vdash v :\leq Void = (P \vdash typeof_h\ v = Some\ Void)$
$| P,h \vdash v :\leq Boolean = (P \vdash typeof_h\ v = Some\ Boolean)$
$| P,h \vdash v :\leq Integer = (P \vdash typeof_h\ v = Some\ Integer)$
$| P,h \vdash v :\leq NT = (P \vdash typeof_h\ v = Some\ NT)$
$| P,h \vdash v :\leq (Class\ C) = (P \vdash typeof_h\ v = Some(Class\ C) \vee P \vdash typeof_h\ v =$

Some NT)

definition *fconf* :: *prog* ⇒ *heap* ⇒ ('*a* ⇀ *val*) ⇒ ('*a* ⇀ *ty*) ⇒ *bool* (-,- ⊢ - '(:≤') - [*51,51,51,51*] *50*) **where**
P,h ⊢ v_m (:≤) T_m ≡
∀ *FD T*. T_m *FD* = *Some T* ⟶ (∃ *v*. v_m *FD* = *Some v* ∧ *P,h* ⊢ *v* :≤ *T*)

definition *oconf* :: *prog* ⇒ *heap* ⇒ *obj* ⇒ *bool* (-,- ⊢ - √ [*51,51,51*] *50*) **where**
P,h ⊢ *obj* √ ≡ *let* (*C*,*S*) = *obj in*
(∀ *Cs*. *Subobjs P C Cs* ⟶ (∃!*fs'*. (*Cs*,*fs'*) ∈ *S*)) ∧
(∀ *Cs fs'*. (*Cs*,*fs'*) ∈ *S* ⟶ *Subobjs P C Cs* ∧
(∃ *fs Bs ms*. *class P* (*last Cs*) = *Some* (*Bs*,*fs*,*ms*) ∧
P,h ⊢ *fs'* (:≤) *map-of fs*))

definition *hconf* :: *prog* ⇒ *heap* ⇒ *bool* (- ⊢ - √ [*51,51*] *50*) **where**
P ⊢ *h* √ ≡
(∀ *a obj*. *h a* = *Some obj* ⟶ *P,h* ⊢ *obj* √) ∧ *preallocated h*

definition *lconf* :: *prog* ⇒ *heap* ⇒ ('*a* ⇀ *val*) ⇒ ('*a* ⇀ *ty*) ⇒ *bool* (-,- ⊢ - '(:≤')$_w$ - [*51,51,51,51*] *50*) **where**
P,h ⊢ v_m $(:\leq)_w$ T_m ≡
∀ *V v*. v_m *V* = *Some v* ⟶ (∃ *T*. T_m *V* = *Some T* ∧ *P,h* ⊢ *v* :≤ *T*)

abbreviation
confs :: *prog* ⇒ *heap* ⇒ *val list* ⇒ *ty list* ⇒ *bool*
(-,- ⊢ - [:≤] - [*51,51,51,51*] *50*) **where**
P,h ⊢ *vs* [:≤] *Ts* ≡ *list-all2* (*conf P h*) *vs Ts*

22.1 Value conformance :≤

lemma *conf-Null* [*simp*]: *P,h* ⊢ *Null* :≤ *T* = *P* ⊢ *NT* ≤ *T*
⟨*proof*⟩

lemma *typeof-conf*[*simp*]: *P* ⊢ $typeof_h$ *v* = *Some T* ⟹ *P,h* ⊢ *v* :≤ *T*
⟨*proof*⟩

lemma *typeof-lit-conf*[*simp*]: *typeof v* = *Some T* ⟹ *P,h* ⊢ *v* :≤ *T*
⟨*proof*⟩

lemma *defval-conf*[*simp*]: *is-type P T* ⟹ *P,h* ⊢ *default-val T* :≤ *T*
⟨*proof*⟩

lemma *typeof-notclass-heap*:
∀ *C*. *T* ≠ *Class C* ⟹ (*P* ⊢ $typeof_h$ *v* = *Some T*) = (*P* ⊢ $typeof_{h'}$ *v* = *Some T*)
⟨*proof*⟩

lemma assumes *h:h a = 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*]: $P,h \vdash v :\leq NT = (v = Null)$
⟨*proof*⟩

22.2 Value list conformance $[:\leq]$

lemma *confs-rev*: $P,h \vdash rev\ s\ [:\leq]\ t = (P,h \vdash s\ [:\leq]\ rev\ t)$

⟨*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*:
$class\ P\ C = Some(Bs,fs,ms) \Longrightarrow P,h \vdash init\text{-}class\text{-}fieldmap\ P\ C\ (:\leq)\ map\text{-}of\ fs$

⟨*proof*⟩

22.4 Heap conformance

lemma *hconfD*: $[\![\ P \vdash h\ \surd;\ h\ a = Some\ obj\ ]\!] \Longrightarrow P,h \vdash obj\ \surd$

⟨*proof*⟩

lemma *hconf-Subobjs*:
$[\![h\ a = Some(C,S);\ (Cs,\ fs) \in S;\ P \vdash h\ \surd]\!] \Longrightarrow Subobjs\ P\ C\ Cs$

⟨*proof*⟩

22.5 Local variable conformance

lemma *lconf-upd*:
$[\![\ P,h \vdash l\ (:\leq)_w\ E;\ P,h \vdash v :\leq T;\ E\ V = Some\ T\ ]\!] \Longrightarrow P,h \vdash l(V\mapsto v)\ (:\leq)_w\ E$

⟨*proof*⟩

lemma *lconf-empty*[*iff*]: $P,h \vdash Map.empty\ (:\leq)_w\ E$
⟨*proof*⟩

lemma *lconf-upd2*: $[\![P,h \vdash l\ (:\leq)_w\ E;\ P,h \vdash v :\leq T]\!] \Longrightarrow P,h \vdash l(V\mapsto v)\ (:\leq)_w\ E(V\mapsto T)$
⟨*proof*⟩

22.6 Environment conformance

definition *envconf* :: *prog* ⇒ *env* ⇒ *bool* (- ⊢ - $\surd$ [51,51] 50) **where**
  $P \vdash E\ \surd \equiv \forall\ V\ T.\ E\ V = Some\ T \longrightarrow$ *is-type* $P\ T$

22.7 Type conformance

primrec
  type-conf :: *prog* ⇒ *env* ⇒ *heap* ⇒ *expr* ⇒ *ty* ⇒ *bool*
    (-,-,- ⊢ - :$_{NT}$ - [51,51,51]50)
where
  type-conf-Void: $P,E,h \vdash e :_{NT} Void \longleftrightarrow (P,E,h \vdash e : Void)$
  | *type-conf-Boolean*: $P,E,h \vdash e :_{NT} Boolean \longleftrightarrow (P,E,h \vdash e : Boolean)$
  | *type-conf-Integer*: $P,E,h \vdash e :_{NT} Integer \longleftrightarrow (P,E,h \vdash e : Integer)$
  | *type-conf-NT*: $P,E,h \vdash e :_{NT} NT \longleftrightarrow (P,E,h \vdash e : NT)$
  | *type-conf-Class*: $P,E,h \vdash e :_{NT} Class\ C \longleftrightarrow$
      $(P,E,h \vdash e : Class\ C \vee P,E,h \vdash e : NT)$

fun
  types-conf :: *prog* ⇒ *env* ⇒ *heap* ⇒ *expr list* ⇒ *ty list* ⇒ *bool*
    (-,-,- ⊢ - [:]$_{NT}$ - [51,51,51]50)
where
  $P,E,h \vdash [\,]\ [:]_{NT}\ [\,] \longleftrightarrow True$
  | $P,E,h \vdash (e\#es)\ [:]_{NT}\ (T\#Ts) \longleftrightarrow$
      $(P,E,h \vdash e:_{NT}\ T \wedge P,E,h \vdash es\ [:]_{NT}\ Ts)$
  | $P,E,h \vdash es\ [:]_{NT}\ Ts \longleftrightarrow False$

lemma *wt-same-type-typeconf*:
$P,E,h \vdash e : T \Longrightarrow P,E,h \vdash e :_{NT} T$
⟨*proof*⟩

lemma *wts-same-types-typesconf*:
  $P,E,h \vdash es\ [:]\ Ts \Longrightarrow$ *types-conf* $P\ E\ h\ es\ Ts$
⟨*proof*⟩

lemma *types-conf-smaller-types*:
$\bigwedge es\ Ts.\ [\![length\ es = length\ Ts';\ types\text{-}conf\ P\ E\ h\ es\ Ts';\ P \vdash Ts'\ [\leq]\ Ts\ ]\!]$
  $\Longrightarrow \exists\ Ts''.\ P,E,h \vdash es\ [:]\ Ts'' \wedge P \vdash Ts''\ [\leq]\ Ts$

⟨*proof*⟩

end

23 Progress of Small Step Semantics

theory *Progress* **imports** *Equivalence DefAss Conform* **begin**

23.1 Some pre-definitions

lemma *final-refE*:
$[\![\ P,E,h \vdash e : Class\ C;\ final\ e;$
$\bigwedge r.\ e = ref\ r \Longrightarrow Q;$
$\bigwedge r.\ e = Throw\ r \Longrightarrow Q\ ]\!] \Longrightarrow Q$
⟨*proof*⟩

lemma *finalRefE*:
$[\![\ P,E,h \vdash e : T;\ is\text{-}refT\ T;\ final\ e;$
$e = null \Longrightarrow Q;$
$\bigwedge r.\ e = ref\ r \Longrightarrow Q;$
$\bigwedge r.\ e = Throw\ r \Longrightarrow Q]\!] \Longrightarrow Q$

⟨*proof*⟩

lemma *subE*:
$[\![\ P \vdash T \leq T';\ is\text{-}type\ P\ T';\ wf\text{-}prog\ wf\text{-}md\ P;$
$[\![\ T = T';\ \forall C.\ T \neq Class\ C\ ]\!] \Longrightarrow Q;$
$\bigwedge C\ D.\ [\![\ T = Class\ C;\ T' = Class\ D;\ P \vdash Path\ C\ to\ D\ unique\ ]\!] \Longrightarrow Q;$
$\bigwedge C.\ [\![\ T = NT;\ T' = Class\ C\ ]\!] \Longrightarrow Q\ ]\!] \Longrightarrow Q$

⟨*proof*⟩

lemma assumes *wf*:*wf-prog wf-md P*
and *typeof*: $P \vdash typeof_h\ v = Some\ T'$
and *type*:*is-type P T*
shows *sub-casts*:$P \vdash T' \leq T \Longrightarrow \exists v'.\ P \vdash T\ casts\ v\ to\ v'$

⟨*proof*⟩

Derivation of new induction scheme for well typing:

inductive
$WTrt'$ :: [*prog*,*env*,*heap*,*expr*, *ty*] ⇒ *bool*
(-,-,- ⊢ - :′′ - [*51*,*51*,*51*]*50*)
and $WTrts'$:: [*prog*,*env*,*heap*,*expr list*,*ty list*] ⇒ *bool*
(-,-,- ⊢ - [:′′] - [*51*,*51*,*51*]*50*)
for *P* :: *prog*
where
is-class P C ⟹ $P,E,h \vdash new\ C :' Class\ C$

| ⟦*is-class P C*; *P,E,h* ⊢ *e* :′ *T*; *is-refT T*⟧
  ⟹ *P,E,h* ⊢ *Cast C e* :′ *Class C*
| ⟦*is-class P C*; *P,E,h* ⊢ *e* :′ *T*; *is-refT T*⟧
  ⟹ *P,E,h* ⊢ (|*C*|)*e* :′ *Class C*
| *P* ⊢ *typeof*$_h$ *v* = *Some T* ⟹ *P,E,h* ⊢ *Val v* :′ *T*
| *E V* = *Some T* ⟹ *P,E,h* ⊢ *Var V* :′ *T*
| ⟦ *P,E,h* ⊢ *e*$_1$:′ *T*$_1$; *P,E,h* ⊢ *e*$_2$:′ *T*$_2$;
    case bop of Eq ⇒ *T* = *Boolean*
    | *Add* ⇒ *T*$_1$ = *Integer* ∧ *T*$_2$ = *Integer* ∧ *T* = *Integer* ⟧
  ⟹ *P,E,h* ⊢ *e*$_1$ «*bop*» *e*$_2$:′ *T*
| ⟦ *P,E,h* ⊢ *Var V* :′ *T*; *P,E,h* ⊢ *e* :′ *T*′ ~~*Thus*~~; *P* ⊢ *T*′ ≤ *T* ⟧
  ⟹ *P,E,h* ⊢ *V*:=*e* :′ *T*
| ⟦*P,E,h* ⊢ *e* :′ *Class C*; *Cs* ≠ []; *P* ⊢ *C has least F*:*T via Cs*⟧
  ⟹ *P,E,h* ⊢ *e*·*F*{*Cs*} :′ *T*
| *P,E,h* ⊢ *e* :′ *NT* ⟹ *P,E,h* ⊢ *e*·*F*{*Cs*} :′ *T*
| ⟦*P,E,h* ⊢ *e*$_1$:′ *Class C*; *Cs* ≠ []; *P* ⊢ *C has least F*:*T via Cs*;
    P,E,h ⊢ *e*$_2$:′ *T*′; *P* ⊢ *T*′ ≤ *T* ⟧
  ⟹ *P,E,h* ⊢ *e*$_1$·*F*{*Cs*}:=*e*$_2$:′ *T*
| ⟦ *P,E,h* ⊢ *e*$_1$:′*NT*; *P,E,h* ⊢ *e*$_2$:′ *T*′; *P* ⊢ *T*′ ≤ *T* ⟧
  ⟹ *P,E,h* ⊢ *e*$_1$·*F*{*Cs*}:=*e*$_2$:′ *T*
| ⟦ *P,E,h* ⊢ *e* :′ *Class C*; *P* ⊢ *C has least M* = (*Ts*,*T*,*m*) *via Cs*;
    P,E,h ⊢ *es* [:′] *Ts*′; *P* ⊢ *Ts*′ [≤] *Ts* ⟧
    ⟹ *P,E,h* ⊢ *e*·*M*(*es*) :′ *T*
| ⟦ *P,E,h* ⊢ *e* :′ *Class C*′; *P* ⊢ *Path C*′ *to C unique*;
    P ⊢ *C has least M* = (*Ts*,*T*,*m*) *via Cs*;
    P,E,h ⊢ *es* [:′] *Ts*′; *P* ⊢ *Ts*′ [≤] *Ts* ⟧
    ⟹ *P,E,h* ⊢ *e*·(*C*::)*M*(*es*) :′ *T*
| ⟦*P,E,h* ⊢ *e* :′ *NT*; *P,E,h* ⊢ *es* [:′] *Ts*⟧ ⟹ *P,E,h* ⊢ *Call e Copt M es* :′ *T*
| ⟦ *P* ⊢ *typeof*$_h$ *v* = *Some T*′; *P,E*(*V*↦*T*),*h* ⊢ *e*$_2$:′ *T*$_2$; *P* ⊢ *T*′ ≤ *T*; *is-type P T*
⟧
  ⟹ *P,E,h* ⊢ {*V*:*T* := *Val v*; *e*$_2$} :′ *T*$_2$
| ⟦ *P,E*(*V*↦*T*),*h* ⊢ *e* :′ *T*′; ¬ *assigned V e*; *is-type P T* ⟧
  ⟹ *P,E,h* ⊢ {*V*:*T*; *e*} :′ *T*′
| ⟦ *P,E,h* ⊢ *e*$_1$:′ *T*$_1$; *P,E,h* ⊢ *e*$_2$:′ *T*$_2$ ⟧ ⟹ *P,E,h* ⊢ *e*$_1$;;*e*$_2$:′ *T*$_2$
| ⟦ *P,E,h* ⊢ *e* :′ *Boolean*; *P,E,h* ⊢ *e*$_1$:′ *T*; *P,E,h* ⊢ *e*$_2$:′ *T* ⟧
  ⟹ *P,E,h* ⊢ *if* (*e*) *e*$_1$ *else e*$_2$:′ *T*
| ⟦ *P,E,h* ⊢ *e* :′ *Boolean*; *P,E,h* ⊢ *c*:′ *T* ⟧
  ⟹ *P,E,h* ⊢ *while*(*e*) *c* :′ *Void*
| ⟦ *P,E,h* ⊢ *e* :′ *T*′; *is-refT T*′⟧ ⟹ *P,E,h* ⊢ *throw e* :′ *T*

| *P,E,h* ⊢ [] [:′] []
| ⟦ *P,E,h* ⊢ *e* :′ *T*; *P,E,h* ⊢ *es* [:′] *Ts* ⟧ ⟹ *P,E,h* ⊢ *e*#*es* [:′] *T*#*Ts*

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

⟨*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 *wt-wt'*: $P,E,h \vdash e : T \Longrightarrow P,E,h \vdash e :' T$
and *wts-wts'*: $P,E,h \vdash es\ [:]\ Ts \Longrightarrow P,E,h \vdash es\ [:']\ Ts$

⟨*proof*⟩

lemma *wt'-wt*: $P,E,h \vdash e :' T \Longrightarrow P,E,h \vdash e : T$
and *wts'-wts*: $P,E,h \vdash es\ [:']\ Ts \Longrightarrow P,E,h \vdash es\ [:]\ Ts$

⟨*proof*⟩

corollary *wt'-iff-wt*: $(P,E,h \vdash e :' T) = (P,E,h \vdash e : T)$
⟨*proof*⟩

corollary *wts'-iff-wts*: $(P,E,h \vdash es\ [:']\ Ts) = (P,E,h \vdash es\ [:]\ Ts)$
⟨*proof*⟩

lemmas *WTrt-inducts2 = WTrt'-inducts* [*unfolded wt'-iff-wt wts'-iff-wts*,
case-names WTrtNew WTrtDynCast WTrtStaticCast WTrtVal WTrtVar WTrtBinOp
WTrtLAss WTrtFAcc WTrtFAccNT WTrtFAss WTrtFAssNT WTrtCall WTrtStaticCall WTrtCallNT
WTrtInitBlock WTrtBlock WTrtSeq WTrtCond WTrtWhile WTrtThrow
WTrtNil WTrtCons, *consumes 1*]

23.2 The theorem *progress*

lemma *mdc-leq-dyn-type*:

$P,E,h \vdash e : T \Longrightarrow$
$\forall C\ a\ Cs\ D\ S.\ T = Class\ C \wedge e = ref(a,Cs) \wedge h\ a = Some(D,S) \longrightarrow P \vdash D \preceq^* C$
and $P,E,h \vdash es\ [:]\ Ts \Longrightarrow$
$\forall T\ Ts'\ e\ es'\ C\ a\ Cs\ D\ S.\ Ts = T\#Ts' \wedge es = e\#es' \wedge T = Class\ C \wedge e = ref(a,Cs) \wedge h\ a = Some(D,S) \longrightarrow P \vdash D \preceq^* C$

⟨*proof*⟩

lemma *appendPath-append-last*:
assumes *notempty*:$Ds \neq []$
shows$(Cs\ @_p\ Ds)\ @_p\ [last\ Ds] = (Cs\ @_p\ Ds)$

⟨*proof*⟩

theorem assumes *wf*: *wwf-prog P*
shows *progress*: $P,E,h \vdash e : T \Longrightarrow$
$(\bigwedge l.\ [\![P \vdash h\ \surd;\ P \vdash E\ \surd;\ \mathcal{D}\ e\ \lfloor dom\ l \rfloor;\ \neg\ final\ e]\!] \Longrightarrow \exists e'\ s'.\ P,E \vdash \langle e,(h,l)\rangle \rightarrow \langle e',s'\rangle)$
and $P,E,h \vdash es\ [:]\ Ts \Longrightarrow$
$(\bigwedge l.\ [\![P \vdash h\ \surd;\ P \vdash E\ \surd;\ \mathcal{D}s\ es\ \lfloor dom\ l \rfloor;\ \neg\ finals\ es]\!] \Longrightarrow \exists es'\ s'.\ P,E \vdash \langle es,(h,l)\rangle\ [\rightarrow]\ \langle es',s'\rangle)$
⟨*proof*⟩

end

24 Heap Extension

theory *HeapExtension*
imports *Progress*
begin

24.1 The Heap Extension

definition *hext* :: *heap* ⇒ *heap* ⇒ *bool* (- ⊴ - [*51*,*51*] *50*) **where**
$h \trianglelefteq h' \equiv \forall a\ C\ S.\ h\ a = Some(C,S) \longrightarrow (\exists S'.\ h'\ a = Some(C,S'))$

lemma *hextI*: $\forall a\ C\ S.\ h\ a = Some(C,S) \longrightarrow (\exists S'.\ h'\ a = Some(C,S')) \Longrightarrow h \trianglelefteq h'$

⟨*proof*⟩

lemma *hext-objD*: $[\![\ h \trianglelefteq h';\ h\ a = Some(C,S)\ ]\!] \Longrightarrow \exists S'.\ h'\ a = Some(C,S')$

⟨*proof*⟩

lemma *hext-refl* [*iff*]: $h \trianglelefteq h$

⟨*proof*⟩

lemma *hext-new* [*simp*]: $h\ a = None \Longrightarrow h \trianglelefteq h(a{\mapsto}x)$

⟨*proof*⟩

lemma *hext-trans*: $[\![\ h \trianglelefteq h';\ h' \trianglelefteq h''\ ]\!] \Longrightarrow h \trianglelefteq h''$

⟨*proof*⟩

lemma *hext-upd-obj*: $h\ a = Some\ (C,S) \Longrightarrow h \trianglelefteq h(a{\mapsto}(C,S'))$

⟨*proof*⟩

24.2 $\trianglelefteq$ and preallocated

lemma *preallocated-hext*:
$[\![\ preallocated\ h;\ h \trianglelefteq h'\ ]\!] \Longrightarrow preallocated\ h'$
⟨*proof*⟩

lemmas *preallocated-upd-obj* = *preallocated-hext* [*OF* - *hext-upd-obj*]
lemmas *preallocated-new* = *preallocated-hext* [*OF* - *hext-new*]

24.3 $\trianglelefteq$ in Small- and BigStep

lemma *red-hext-incr*: $P,E \vdash \langle e,(h,l)\rangle \to \langle e',(h',l')\rangle \Longrightarrow h \trianglelefteq h'$
and *reds-hext-incr*: $P,E \vdash \langle es,(h,l)\rangle\ [\to]\ \langle es',(h',l')\rangle \Longrightarrow h \trianglelefteq h'$

⟨*proof*⟩

lemma *step-hext-incr*: $P,E \vdash \langle e,s\rangle \to* \langle e',s'\rangle \Longrightarrow hp\ s \trianglelefteq hp\ s'$

⟨*proof*⟩

lemma *steps-hext-incr*: $P,E \vdash \langle es,s\rangle\ [\to]* \langle es',s'\rangle \Longrightarrow hp\ s \trianglelefteq hp\ s'$

⟨*proof*⟩

lemma *eval-hext*: $P,E \vdash \langle e,(h,l)\rangle \Rightarrow \langle e',(h',l')\rangle \Longrightarrow h \trianglelefteq h'$
and *evals-hext*: $P,E \vdash \langle es,(h,l)\rangle\ [\Rightarrow]\ \langle es',(h',l')\rangle \Longrightarrow h \trianglelefteq h'$

⟨*proof*⟩

24.4 $\trianglelefteq$ and conformance

lemma *conf-hext*: $h \trianglelefteq h' \Longrightarrow P,h \vdash v :\leq T \Longrightarrow P,h' \vdash v :\leq T$
⟨*proof*⟩

lemma *confs-hext*: $P,h \vdash vs\ [:\leq]\ Ts \Longrightarrow h \trianglelefteq h' \Longrightarrow P,h' \vdash vs\ [:\leq]\ Ts$
⟨*proof*⟩

lemma *fconf-hext*: $[\![\ P,h \vdash fs\ (:\leq)\ E;\ h \trianglelefteq h'\ ]\!] \Longrightarrow P,h' \vdash fs\ (:\leq)\ E$

⟨*proof*⟩

lemmas *fconf-upd-obj* = *fconf-hext* [*OF* - *hext-upd-obj*]
lemmas *fconf-new* = *fconf-hext* [*OF* - *hext-new*]

lemma *oconf-hext*: $P,h \vdash obj\ \surd \Longrightarrow h \trianglelefteq h' \Longrightarrow P,h' \vdash obj\ \surd$

⟨*proof*⟩

lemmas *oconf-new* = *oconf-hext* [*OF* - *hext-new*]
lemmas *oconf-upd-obj* = *oconf-hext* [*OF* - *hext-upd-obj*]

lemma *hconf-new*: $[\![\ P \vdash h\ \surd;\ h\ a = None;\ P,h \vdash obj\ \surd\ ]\!] \Longrightarrow P \vdash h(a \mapsto obj)\ \surd$
⟨*proof*⟩

lemma $[\![P \vdash h\ \surd;\ h' = h(a \mapsto (C,\ Collect\ (init\text{-}obj\ P\ C)));\ h\ a = None;\ wf\text{-}prog\ wf\text{-}md\ P]\!]$
$\Longrightarrow P \vdash h'\ \surd$
⟨*proof*⟩

lemma *hconf-upd-obj*:
$[\![\ P \vdash h\surd;\ h\ a = Some(C,S);\ P,h \vdash (C,S')\surd\ ]\!] \Longrightarrow P \vdash h(a \mapsto (C,S'))\surd$

⟨*proof*⟩

lemma *lconf-hext*: ⟦ $P,h \vdash l\ (:\leq)_w\ E$; $h \trianglelefteq h'$ ⟧ ⟹ $P,h' \vdash l\ (:\leq)_w\ E$

⟨*proof*⟩

24.5 $\trianglelefteq$ in the runtime type system

lemma *hext-typeof-mono*: ⟦ $h \trianglelefteq h'$; $P \vdash typeof_h\ v = Some\ T$ ⟧ ⟹ $P \vdash typeof_{h'}\ v = Some\ T$

⟨*proof*⟩

lemma *WTrt-hext-mono*: $P,E,h \vdash e : T \Longrightarrow (\bigwedge h'.\ h \trianglelefteq h' \Longrightarrow P,E,h' \vdash e : T)$
and *WTrts-hext-mono*: $P,E,h \vdash es\ [:]\ Ts \Longrightarrow (\bigwedge h'.\ h \trianglelefteq h' \Longrightarrow P,E,h' \vdash es\ [:]\ Ts)$

⟨*proof*⟩

end

25 Well-formedness Constraints

theory *CWellForm* **imports** *WellForm WWellForm WellTypeRT DefAss* **begin**

definition *wf-C-mdecl* :: *prog* ⇒ *cname* ⇒ *mdecl* ⇒ *bool* **where**
wf-C-mdecl P C ≡ λ(*M*,*Ts*,*T*,(*pns*,*body*)).
length Ts = *length pns* ∧
distinct pns ∧
this ∉ *set pns* ∧
P,[*this*↦*Class C*,*pns*[↦]*Ts*] ⊢ *body* :: *T* ∧
$\mathcal{D}$ *body* ⌊{*this*} ∪ *set pns*⌋

lemma *wf-C-mdecl*[*simp*]:
wf-C-mdecl P C (*M*,*Ts*,*T*,*pns*,*body*) ≡
(*length Ts* = *length pns* ∧
distinct pns ∧
this ∉ *set pns* ∧
P,[*this*↦*Class C*,*pns*[↦]*Ts*] ⊢ *body* :: *T* ∧
$\mathcal{D}$ *body* ⌊{*this*} ∪ *set pns*⌋)
⟨*proof*⟩

abbreviation
wf-C-prog :: *prog* ⇒ *bool* **where**
wf-C-prog == *wf-prog wf-C-mdecl*

lemma *wf-C-prog-wf-C-mdecl*:
⟦ *wf-C-prog P*; (*C*,*Bs*,*fs*,*ms*) ∈ *set P*; *m* ∈ *set ms* ⟧
⟹ *wf-C-mdecl P C m*

⟨*proof*⟩

lemma *wf-mdecl-wwf-mdecl*: *wf-C-mdecl P C Md* ⟹ *wwf-mdecl P C Md*
⟨*proof*⟩

lemma *wf-prog-wwf-prog*: *wf-C-prog P* ⟹ *wwf-prog P*

⟨*proof*⟩

end

26 Type Safety Proof

theory *TypeSafe*
imports *HeapExtension CWellForm*
begin

26.1 Basic preservation lemmas

lemma assumes *wf*:*wwf-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*:*wwf-prog P*
shows *red-preserves-hconf*:
P,*E* ⊢ ⟨*e*,(*h*,*l*)⟩ → ⟨*e′*,(*h′*,*l′*)⟩ ⟹ (⋀*T*. ⟦ *P*,*E*,*h* ⊢ *e* : *T*; *P* ⊢ *h* √ ⟧ ⟹ *P* ⊢ *h′* √)
and *reds-preserves-hconf*:
P,*E* ⊢ ⟨*es*,(*h*,*l*)⟩ [→] ⟨*es′*,(*h′*,*l′*)⟩ ⟹ (⋀*Ts*. ⟦ *P*,*E*,*h* ⊢ *es* [:] *Ts*; *P* ⊢ *h* √ ⟧ ⟹ *P* ⊢ *h′* √)

⟨*proof*⟩

theorem assumes *wf*:*wwf-prog P*
shows *red-preserves-lconf*:
$P,E \vdash \langle e,(h,l)\rangle \to \langle e',(h',l')\rangle \Longrightarrow$
$(\bigwedge T.\ [\![\ P,E,h \vdash e{:}T;\ P,h \vdash l\ (:\leq)_w\ E;\ P \vdash E\ \surd\ ]\!] \Longrightarrow P,h' \vdash l'\ (:\leq)_w\ E)$
and *reds-preserves-lconf*:
$P,E \vdash \langle es,(h,l)\rangle\ [\to]\ \langle es',(h',l')\rangle \Longrightarrow$
$(\bigwedge Ts.\ [\![\ P,E,h \vdash es[:]Ts;\ P,h \vdash l\ (:\leq)_w\ E;\ P \vdash E\ \surd\ ]\!] \Longrightarrow P,h' \vdash l'\ (:\leq)_w\ E)$

⟨*proof*⟩

Preservation of definite assignment more complex and requires a few lemmas first.

lemma [*iff*]: $\bigwedge A.\ [\![\ length\ Vs = length\ Ts;\ length\ vs = length\ Ts]\!] \Longrightarrow$
$\mathcal{D}\ (blocks\ (Vs,Ts,vs,e))\ A = \mathcal{D}\ e\ (A \sqcup \lfloor set\ Vs\rfloor)$

⟨*proof*⟩

lemma *red-lA-incr*: $P,E \vdash \langle e,(h,l)\rangle \to \langle e',(h',l')\rangle \Longrightarrow \lfloor dom\ l\rfloor \sqcup \mathcal{A}\ e \sqsubseteq \lfloor dom\ l'\rfloor \sqcup \mathcal{A}\ e'$
and *reds-lA-incr*: $P,E \vdash \langle es,(h,l)\rangle\ [\to]\ \langle es',(h',l')\rangle \Longrightarrow \lfloor dom\ l\rfloor \sqcup \mathcal{A}s\ es \sqsubseteq \lfloor dom\ l'\rfloor \sqcup \mathcal{A}s\ es'$
⟨*proof*⟩

Now preservation of definite assignment.

lemma assumes *wf*: *wf-C-prog P*
shows *red-preserves-defass*:
$P,E \vdash \langle e,(h,l)\rangle \to \langle e',(h',l')\rangle \Longrightarrow \mathcal{D}\ e\ \lfloor dom\ l\rfloor \Longrightarrow \mathcal{D}\ e'\ \lfloor dom\ l'\rfloor$
and $P,E \vdash \langle es,(h,l)\rangle\ [\to]\ \langle es',(h',l')\rangle \Longrightarrow \mathcal{D}s\ es\ \lfloor dom\ l\rfloor \Longrightarrow \mathcal{D}s\ es'\ \lfloor dom\ l'\rfloor$

⟨*proof*⟩

Combining conformance of heap and local variables:

definition *sconf* :: *prog* ⇒ *env* ⇒ *state* ⇒ *bool* (-,- ⊢ - √ [51,51,51]50) **where**
$P,E \vdash s\ \surd \equiv$ *let* $(h,l) = s$ *in* $P \vdash h\ \surd \wedge P,h \vdash l\ (:\leq)_w\ E \wedge P \vdash E\ \surd$

lemma *red-preserves-sconf*:
$[\![\ P,E \vdash \langle e,s\rangle \to \langle e',s'\rangle;\ P,E,hp\ s \vdash e : T;\ P,E \vdash s\ \surd;\ wwf\text{-}prog\ P]\!]$
$\Longrightarrow P,E \vdash s'\ \surd$

⟨*proof*⟩

lemma *reds-preserves-sconf*:
$[\![\ P,E \vdash \langle es,s\rangle\ [\to]\ \langle es',s'\rangle;\ P,E,hp\ s \vdash es\ [:]\ Ts;\ P,E \vdash s\ \surd;\ wwf\text{-}prog\ P]\!]$

$\Longrightarrow P,E \vdash s' \surd$

⟨*proof*⟩

26.2 Subject reduction

lemma *wt-blocks*:
$\bigwedge E.\ [\![\ \mathit{length}\ Vs = \mathit{length}\ Ts;\ \mathit{length}\ vs = \mathit{length}\ Ts;$
$\forall\, T' \in \mathit{set}\ Ts.\ \mathit{is\text{-}type}\ P\ T']\!] \Longrightarrow$
$(P,E,h \vdash \mathit{blocks}(Vs,Ts,vs,e) : T) =$
$(P,E(Vs[\mapsto]Ts),h \vdash e{:}T \wedge$
$(\exists\, Ts'.\ \mathit{map}\ (P \vdash \mathit{typeof}_h)\ vs = \mathit{map}\ \mathit{Some}\ Ts' \wedge P \vdash Ts'\ [\leq]\ Ts))$

⟨*proof*⟩

theorem assumes *wf*: *wf-C-prog P*
shows *subject-reduction2*: $P,E \vdash \langle e,(h,l)\rangle \to \langle e',(h',l')\rangle \Longrightarrow$
$(\bigwedge T.\ [\![\ P,E \vdash (h,l)\ \surd;\ P,E,h \vdash e : T\ ]\!] \Longrightarrow P,E,h' \vdash e' :_{NT} T)$
and *subjects-reduction2*: $P,E \vdash \langle es,(h,l)\rangle\ [\to]\ \langle es',(h',l')\rangle \Longrightarrow$
$(\bigwedge Ts.[\![\ P,E \vdash (h,l)\ \surd;\ P,E,h \vdash es\ [:]\ Ts\ ]\!] \Longrightarrow \mathit{types\text{-}conf}\ P\ E\ h'\ es'\ Ts)$

⟨*proof*⟩

corollary *subject-reduction*:
$[\![\ \mathit{wf\text{-}C\text{-}prog}\ P;\ P,E \vdash \langle e,s\rangle \to \langle e',s'\rangle;\ P,E \vdash s\ \surd;\ P,E,hp\ s \vdash e{:}T\ ]\!]$
$\Longrightarrow P,E,(hp\ s') \vdash e' :_{NT} T$
⟨*proof*⟩

corollary *subjects-reduction*:
$[\![\ \mathit{wf\text{-}C\text{-}prog}\ P;\ P,E \vdash \langle es,s\rangle\ [\to]\ \langle es',s'\rangle;\ P,E \vdash s\ \surd;\ P,E,hp\ s \vdash es[:]Ts\ ]\!]$
$\Longrightarrow \mathit{types\text{-}conf}\ P\ E\ (hp\ s')\ es'\ Ts$
⟨*proof*⟩

26.3 Lifting to $\to*$

Now all these preservation lemmas are first lifted to the transitive closure …

lemma *step-preserves-sconf*:
assumes *wf*: *wf-C-prog P* **and** *step*: $P,E \vdash \langle e,s\rangle \to* \langle e',s'\rangle$
shows $\bigwedge T.\ [\![\ P,E,hp\ s \vdash e : T;\ P,E \vdash s\ \surd\ ]\!] \Longrightarrow P,E \vdash s'\ \surd$

⟨*proof*⟩

lemma *steps-preserves-sconf*:
assumes *wf*: *wf-C-prog P* **and** *step*: $P,E \vdash \langle es,s\rangle\ [\to]* \langle es',s'\rangle$

shows $\bigwedge Ts.\ [\![\ P,E,hp\ s \vdash es\ [:]\ Ts;\ P,E \vdash s\ \surd\ ]\!] \Longrightarrow P,E \vdash s'\ \surd$

⟨*proof*⟩

lemma *step-preserves-defass*:
assumes *wf*: *wf-C-prog P* **and** *step*: $P,E \vdash \langle e,s\rangle \rightarrow* \langle e',s'\rangle$
shows $\mathcal{D}\ e\ \lfloor dom(lcl\ s)\rfloor \Longrightarrow \mathcal{D}\ e'\ \lfloor dom(lcl\ s')\rfloor$

⟨*proof*⟩

lemma *step-preserves-type*:
assumes *wf*: *wf-C-prog P* **and** *step*: $P,E \vdash \langle e,s\rangle \rightarrow* \langle e',s'\rangle$
shows $\bigwedge T.\ [\![\ P,E \vdash s\ \surd;\ P,E,hp\ s \vdash e{:}T\ ]\!]$
$\Longrightarrow P,E,(hp\ s') \vdash e' :_{NT} T$

⟨*proof*⟩

predicate to show the same lemma for lists

fun
conformable :: *ty list* ⇒ *ty list* ⇒ *bool*
where
conformable [] [] ⟷ *True*
| *conformable* ($T''\#Ts''$) ($T'\#Ts'$) ⟷ ($T'' = T'$
$\vee\ (\exists C.\ T'' = NT \wedge T' = Class\ C)) \wedge conformable\ Ts''\ Ts'$
| *conformable* - - ⟷ *False*

lemma *types-conf-conf-types-conf*:
$[\![types\text{-}conf\ P\ E\ h\ es\ Ts;\ conformable\ Ts\ Ts']\!] \Longrightarrow types\text{-}conf\ P\ E\ h\ es\ Ts'$
⟨*proof*⟩

lemma *types-conf-Wtrt-conf*:
$types\text{-}conf\ P\ E\ h\ es\ Ts \Longrightarrow \exists\ Ts'.\ P,E,h \vdash es\ [:]\ Ts' \wedge conformable\ Ts'\ Ts$
⟨*proof*⟩

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 $\bigwedge Ts.\ [\![\ P,E \vdash s\ \surd;\ P,E,hp\ s \vdash es\ [:]\ Ts]\!]$
$\Longrightarrow types\text{-}conf\ P\ E\ (hp\ s')\ es'\ Ts$

⟨*proof*⟩

26.4 Lifting to ⇒

…and now to the big step semantics, just for fun.

lemma *eval-preserves-sconf*:
⟦ *wf-C-prog P*; *P*,*E* ⊢ ⟨*e*,*s*⟩ ⇒ ⟨*e*′,*s*′⟩; *P*,*E* ⊢ *e*::*T*; *P*,*E* ⊢ *s* √ ⟧ ⟹ *P*,*E* ⊢ *s*′ √

⟨*proof*⟩

lemma *evals-preserves-sconf*:
⟦ *wf-C-prog P*; *P*,*E* ⊢ ⟨*es*,*s*⟩ [⇒] ⟨*es*′,*s*′⟩; *P*,*E* ⊢ *es* [::] *Ts*; *P*,*E* ⊢ *s* √ ⟧
⟹ *P*,*E* ⊢ *s*′ √
⟨*proof*⟩

lemma *eval-preserves-type*: **assumes** *wf*: *wf-C-prog P*
shows ⟦ *P*,*E* ⊢ ⟨*e*,*s*⟩ ⇒ ⟨*e*′,*s*′⟩; *P*,*E* ⊢ *s* √; *P*,*E* ⊢ *e*::*T* ⟧
⟹ *P*,*E*,(*hp s*′) ⊢ *e*′ :$_{NT}$ *T*

⟨*proof*⟩

lemma *evals-preserves-types*: **assumes** *wf*: *wf-C-prog P*
shows ⟦ *P*,*E* ⊢ ⟨*es*,*s*⟩ [⇒] ⟨*es*′,*s*′⟩; *P*,*E* ⊢ *s* √; *P*,*E* ⊢ *es* [::] *Ts* ⟧
⟹ *types-conf P E* (*hp s*′) *es*′ *Ts*
⟨*proof*⟩

26.5 The final polish

The above preservation lemmas are now combined and packed nicely.

definition *wf-config* :: *prog* ⇒ *env* ⇒ *state* ⇒ *expr* ⇒ *ty* ⇒ *bool* (-,-,- ⊢ - : - √ [*51*,*0*,*0*,*0*,*0*]*50*) **where**
P,*E*,*s* ⊢ *e*:*T* √ ≡ *P*,*E* ⊢ *s* √ ∧ *P*,*E*,*hp s* ⊢ *e* : *T*

theorem *Subject-reduction*: **assumes** *wf*: *wf-C-prog P*
shows *P*,*E* ⊢ ⟨*e*,*s*⟩ → ⟨*e*′,*s*′⟩ ⟹ *P*,*E*,*s* ⊢ *e* : *T* √
⟹ *P*,*E*,(*hp s*′) ⊢ *e*′ :$_{NT}$ *T*

⟨*proof*⟩

theorem *Subject-reductions*:
assumes *wf*: *wf-C-prog P* **and** *reds*: *P*,*E* ⊢ ⟨*e*,*s*⟩ →* ⟨*e*′,*s*′⟩
shows ⋀*T*. *P*,*E*,*s* ⊢ *e* : *T* √ ⟹ *P*,*E*,(*hp s*′) ⊢ *e*′ :$_{NT}$ *T*

⟨*proof*⟩

corollary *Progress*: **assumes** *wf*: *wf-C-prog P*
shows ⟦ *P*,*E*,*s* ⊢ *e* : *T* √; 𝒟 *e* ⌊*dom*(*lcl s*)⌋; ¬ *final e* ⟧ ⟹ ∃ *e*′ *s*′. *P*,*E* ⊢ ⟨*e*,*s*⟩ → ⟨*e*′,*s*′⟩

⟨*proof*⟩

corollary *TypeSafety*:
fixes *s s*′ :: *state*
assumes *wf*:*wf-C-prog P* **and** *sconf*:*P*,*E* ⊢ *s* √ **and** *wte*:*P*,*E* ⊢ *e* :: *T*
and *D*:𝒟 *e* ⌊*dom*(*lcl s*)⌋ **and** *step*:*P*,*E* ⊢ ⟨*e*,*s*⟩ →* ⟨*e*′,*s*′⟩
and *nored*:¬(∃ *e*′′ *s*′′. *P*,*E* ⊢ ⟨*e*′,*s*′⟩ → ⟨*e*′′,*s*′′⟩)
shows (∃ *v*. *e*′ = *Val v* ∧ *P*,*hp s*′ ⊢ *v* :≤ *T*) ∨
(∃ *r*. *e*′ = *Throw r* ∧ *the-addr* (*Ref r*) ∈ *dom*(*hp s*′))
⟨*proof*⟩

end

27 Determinism Proof

theory *Determinism*
imports *TypeSafe*
begin

27.1 Some lemmas

lemma *maps-nth*:
⟦(*E*(*xs* [↦] *ys*)) *x* = *Some y*; *length xs* = *length ys*; *distinct xs*⟧
⟹ ∀ *i*. *x* = *xs*!*i* ∧ *i* < *length xs* ⟶ *y* = *ys*!*i*
⟨*proof*⟩

lemma *nth-maps*:⟦*length pns* = *length Ts*; *distinct pns*; *i* < *length Ts*⟧
⟹ (*E*(*pns* [↦] *Ts*)) (*pns*!*i*) = *Some* (*Ts*!*i*)
⟨*proof*⟩

lemma *casts-casts-eq-result*:
fixes *s* :: *state*
assumes *casts*:*P* ⊢ *T casts v to v*′ **and** *casts*′:*P* ⊢ *T casts v to w*′
and *type*:*is-type P T* **and** *wte*:*P*,*E* ⊢ *e* :: *T*′ **and** *leq*:*P* ⊢ *T*′ ≤ *T*
and *eval*:*P*,*E* ⊢ ⟨*e*,*s*⟩ ⇒ ⟨*Val v*,(*h*,*l*)⟩ **and** *sconf*:*P*,*E* ⊢ *s* √
and *wf*:*wf-C-prog P*
shows *v*′ = *w*′
⟨*proof*⟩

lemma *Casts-Casts-eq-result*:

assumes *wf*:*wf-C-prog P*
shows $[\![P \vdash Ts\ Casts\ vs\ to\ vs';\ P \vdash Ts\ Casts\ vs\ to\ ws';\ \forall\ T \in set\ Ts.\ is\text{-}type\ P\ T;$
$P,E \vdash es\ [::]\ Ts';\ P \vdash Ts'\ [\leq]\ Ts;\ P,E \vdash \langle es,s\rangle\ [\Rightarrow]\ \langle map\ Val\ vs,(h,l)\rangle;$
$P,E \vdash s\ \surd]\!]$
$\Longrightarrow vs' = ws'$
⟨*proof*⟩

lemma *Casts-conf*: **assumes** *wf*: *wf-C-prog P*
shows $P \vdash Ts\ Casts\ vs\ to\ vs' \Longrightarrow$
$(\bigwedge es\ s\ Ts'.\ [\![\ P,E \vdash es\ [::]\ Ts';\ P,E \vdash \langle es,s\rangle\ [\Rightarrow]\ \langle map\ Val\ vs,(h,l)\rangle;\ P,E \vdash s\ \surd;$
$P \vdash Ts'\ [\leq]\ Ts]\!] \Longrightarrow$
$\forall\ i < length\ Ts.\ P,h \vdash vs'!i :\leq Ts!i)$
⟨*proof*⟩

lemma *map-Val-throw-False*:*map Val vs* = *map Val ws* @ *throw ex* # *es* ⟹ *False*
⟨*proof*⟩

lemma *map-Val-throw-eq*:*map Val vs* @ *throw ex* # *es* = *map Val ws* @ *throw ex'* # *es'*
⟹ $vs = ws \wedge ex = ex' \wedge es = es'$
⟨*proof*⟩

27.2 The proof

lemma *deterministic-big-step*:
assumes *wf*:*wf-C-prog P*
shows $P,E \vdash \langle e,s\rangle \Rightarrow \langle e_1,s_1\rangle \Longrightarrow$
$(\bigwedge e_2\ s_2\ T.\ [\![P,E \vdash \langle e,s\rangle \Rightarrow \langle e_2,s_2\rangle;\ P,E \vdash e :: T;\ P,E \vdash s\ \surd]\!]$
$\Longrightarrow e_1 = e_2 \wedge s_1 = s_2)$
and $P,E \vdash \langle es,s\rangle\ [\Rightarrow]\ \langle es_1,s_1\rangle \Longrightarrow$
$(\bigwedge es_2\ s_2\ Ts.\ [\![P,E \vdash \langle es,s\rangle\ [\Rightarrow]\ \langle es_2,s_2\rangle;\ P,E \vdash es\ [::]\ Ts;\ P,E \vdash s\ \surd]\!]$
$\Longrightarrow es_1 = es_2 \wedge s_1 = s_2)$
⟨*proof*⟩

end

28 Program annotation

theory *Annotate* **imports** *WellType* **begin**

abbreviation (**output**)
unanFAcc :: *expr* ⇒ *vname* ⇒ *expr* ((-•-) [*10*,*10*] *90*) **where**
unanFAcc e F == *FAcc e F* []

abbreviation (**output**)
unanFAss :: *expr* ⇒ *vname* ⇒ *expr* ⇒ *expr* ((-•- := -) [*10*,*0*,*90*] *90*) **where**
unanFAss e F e′ == *FAss e F* [] *e′*

inductive
Anno :: [*prog*,*env*, *expr* , *expr*] ⇒ *bool*
(-,- ⊢ - ⇝ - [*51*,*0*,*0*,*51*]*50*)
and *Annos* :: [*prog*,*env*, *expr list*, *expr list*] ⇒ *bool*
(-,- ⊢ - [⇝] - [*51*,*0*,*0*,*51*]*50*)
for *P* :: *prog*
where

AnnoNew: *is-class P C* ⟹ *P*,*E* ⊢ *new C* ⇝ *new C*
| *AnnoCast*: *P*,*E* ⊢ *e* ⇝ *e′* ⟹ *P*,*E* ⊢ *Cast C e* ⇝ *Cast C e′*
| *AnnoStatCast*: *P*,*E* ⊢ *e* ⇝ *e′* ⟹ *P*,*E* ⊢ *StatCast C e* ⇝ *StatCast C e′*
| *AnnoVal*: *P*,*E* ⊢ *Val v* ⇝ *Val v*
| *AnnoVarVar*: *E V* = ⌊*T*⌋ ⟹ *P*,*E* ⊢ *Var V* ⇝ *Var V*
| *AnnoVarField*: ⟦ *E V* = *None*; *E this* = ⌊*Class C*⌋; *P* ⊢ *C has least V*:*T via Cs* ⟧
⟹ *P*,*E* ⊢ *Var V* ⇝ *Var this*•*V*{*Cs*}
| *AnnoBinOp*:
⟦ *P*,*E* ⊢ *e1* ⇝ *e1′*; *P*,*E* ⊢ *e2* ⇝ *e2′* ⟧
⟹ *P*,*E* ⊢ *e1* «*bop*» *e2* ⇝ *e1′* «*bop*» *e2′*
| *AnnoLAss*:
P,*E* ⊢ *e* ⇝ *e′* ⟹ *P*,*E* ⊢ *V*:=*e* ⇝ *V*:=*e′*
| *AnnoFAcc*:
⟦ *P*,*E* ⊢ *e* ⇝ *e′*; *P*,*E* ⊢ *e′* :: *Class C*; *P* ⊢ *C has least F*:*T via Cs* ⟧
⟹ *P*,*E* ⊢ *e*•*F*{[]} ⇝ *e′*•*F*{*Cs*}
| *AnnoFAss*: ⟦ *P*,*E* ⊢ *e1* ⇝ *e1′*; *P*,*E* ⊢ *e2* ⇝ *e2′*;
P,*E* ⊢ *e1′* :: *Class C*; *P* ⊢ *C has least F*:*T via Cs* ⟧
⟹ *P*,*E* ⊢ *e1*•*F*{[]} := *e2* ⇝ *e1′*•*F*{*Cs*} := *e2′*
| *AnnoCall*:
⟦ *P*,*E* ⊢ *e* ⇝ *e′*; *P*,*E* ⊢ *es* [⇝] *es′* ⟧
⟹ *P*,*E* ⊢ *Call e Copt M es* ⇝ *Call e′ Copt M es′*
| *AnnoBlock*:
P,*E*(*V* ↦ *T*) ⊢ *e* ⇝ *e′* ⟹ *P*,*E* ⊢ {*V*:*T*; *e*} ⇝ {*V*:*T*; *e′*}
| *AnnoComp*: ⟦ *P*,*E* ⊢ *e1* ⇝ *e1′*; *P*,*E* ⊢ *e2* ⇝ *e2′* ⟧
⟹ *P*,*E* ⊢ *e1*;;*e2* ⇝ *e1′*;;*e2′*
| *AnnoCond*: ⟦ *P*,*E* ⊢ *e* ⇝ *e′*; *P*,*E* ⊢ *e1* ⇝ *e1′*; *P*,*E* ⊢ *e2* ⇝ *e2′* ⟧
⟹ *P*,*E* ⊢ *if* (*e*) *e1 else e2* ⇝ *if* (*e′*) *e1′ else e2′*
| *AnnoLoop*: ⟦ *P*,*E* ⊢ *e* ⇝ *e′*; *P*,*E* ⊢ *c* ⇝ *c′* ⟧
⟹ *P*,*E* ⊢ *while* (*e*) *c* ⇝ *while* (*e′*) *c′*
| *AnnoThrow*: *P*,*E* ⊢ *e* ⇝ *e′* ⟹ *P*,*E* ⊢ *throw e* ⇝ *throw e′*

| *AnnoNil*: *P*,*E* ⊢ [] [⇝] []
| *AnnoCons*: ⟦ *P*,*E* ⊢ *e* ⇝ *e′*; *P*,*E* ⊢ *es* [⇝] *es′* ⟧
⟹ *P*,*E* ⊢ *e*#*es* [⇝] *e′*#*es′*

end

29 Code generation for Semantics and Type System

theory *Execute*
imports *BigStep WellType*
HOL−Library.AList-Mapping
HOL−Library.Code-Target-Numeral
begin

29.1 General redefinitions

inductive *app* :: *'a list* ⇒ *'a list* ⇒ *'a list* ⇒ *bool*
where
app [] *ys ys*
| *app xs ys zs* ⟹ *app* (*x* # *xs*) *ys* (*x* # *zs*)

theorem *app-eq1*: ⋀*ys zs*. *zs* = *xs* @ *ys* ⟹ *app xs ys zs*
⟨*proof*⟩

theorem *app-eq2*: *app xs ys zs* ⟹ *zs* = *xs* @ *ys*
⟨*proof*⟩

theorem *app-eq*: *app xs ys zs* = (*zs* = *xs* @ *ys*)
⟨*proof*⟩

code-pred
(*modes*:
i ⇒ *i* ⇒ *i* ⇒ *bool*, *i* ⇒ *i* ⇒ *o* ⇒ *bool*, *i* ⇒ *o* ⇒ *i* ⇒ *bool*,
o ⇒ *i* ⇒ *i* ⇒ *bool*, *o* ⇒ *o* ⇒ *i* ⇒ *bool as reverse-app*)
app
⟨*proof*⟩

declare *rtranclp-rtrancl-eq*[*code del*]

lemmas [*code-pred-intro*] = *rtranclp.rtrancl-refl converse-rtranclp-into-rtranclp*

code-pred
(*modes*:
(*i* => *o* => *bool*) => *i* => *i* => *bool*,
(*i* => *o* => *bool*) => *i* => *o* => *bool*)
rtranclp
⟨*proof*⟩

definition *Set-project* :: (*'a* × *'b*) *set* => *'a* => *'b set*
where *Set-project A a* = {*b*. (*a*, *b*) ∈ *A*}

lemma *Set-project-set* [*code*]:
Set-project (*set xs*) *a* = *set* (*List.map-filter* (λ(*a'*, *b*). *if a* = *a' then Some b else None*) *xs*)
⟨*proof*⟩

Redefine map Val vs

inductive *map-val* :: *expr list* ⇒ *val list* ⇒ *bool*
where
Nil: *map-val* [] []
| *Cons*: *map-val xs ys* ⟹ *map-val* (*Val y* # *xs*) (*y* # *ys*)

code-pred
(*modes*: *i* ⇒ *i* ⇒ *bool*, *i* ⇒ *o* ⇒ *bool*)
map-val
⟨*proof*⟩

inductive *map-val2* :: *expr list* ⇒ *val list* ⇒ *expr list* ⇒ *bool*
where
Nil: *map-val2* [] [] []
| *Cons*: *map-val2 xs ys zs* ⟹ *map-val2* (*Val y* # *xs*) (*y* # *ys*) *zs*
| *Throw*: *map-val2* (*throw e* # *xs*) [] (*throw e* # *xs*)

code-pred
(*modes*: *i* ⇒ *i* ⇒ *i* ⇒ *bool*, *i* ⇒ *o* ⇒ *o* ⇒ *bool*)
map-val2
⟨*proof*⟩

theorem *map-val-conv*: (*xs* = *map Val ys*) = *map-val xs ys*⟨*proof*⟩
theorem *map-val2-conv*:
(*xs* = *map Val ys* @ *throw e* # *zs*) = *map-val2 xs ys* (*throw e* # *zs*)⟨*proof*⟩

29.2 Code generation

lemma *subclsRp-code* [*code-pred-intro*]:
⟦ *class P C* = ⌊(*Bs*, *rest*)⌋; *Predicate-Compile.contains* (*set Bs*) (*Repeats D*) ⟧
⟹ *subclsRp P C D*
⟨*proof*⟩

code-pred
(*modes*: *i* ⇒ *i* ⇒ *i* ⇒ *bool*, *i* ⇒ *i* ⇒ *o* ⇒ *bool*)
subclsRp
⟨*proof*⟩

lemma *subclsR-code* [*code-pred-inline*]:
P ⊢ *C* ≺$_R$ *D* ⟷ *subclsRp P C D*
⟨*proof*⟩

lemma *subclsSp-code* [*code-pred-intro*]:
⟦ *class P C* = ⌊(*Bs*, *rest*)⌋; *Predicate-Compile.contains* (*set Bs*) (*Shares D*) ⟧ ⟹
subclsSp P C D

⟨*proof*⟩

code-pred
(*modes*: $i \Rightarrow i \Rightarrow i \Rightarrow bool$, $i \Rightarrow i \Rightarrow o \Rightarrow bool$)
subclsSp
⟨*proof*⟩

declare *SubobjsR-Base* [*code-pred-intro*]
lemma *SubobjsR-Rep-code* [*code-pred-intro*]:
$[\![subclsRp\ P\ C\ D;\ Subobjs_R\ P\ D\ Cs]\!] \Longrightarrow Subobjs_R\ P\ C\ (C\ \#\ Cs)$
⟨*proof*⟩

code-pred
(*modes*: $i \Rightarrow i \Rightarrow i \Rightarrow bool$, $i \Rightarrow i \Rightarrow o \Rightarrow bool$)
$Subobjs_R$
⟨*proof*⟩

lemma *subcls1p-code* [*code-pred-intro*]:
$[\![class\ P\ C = Some\ (Bs,rest);\ Predicate\text{-}Compile.contains\ (baseClasses\ Bs)\ D\ ]\!]$
$\Longrightarrow subcls1p\ P\ C\ D$
⟨*proof*⟩

code-pred (*modes*: $i \Rightarrow i \Rightarrow i \Rightarrow bool$, $i \Rightarrow i \Rightarrow o \Rightarrow bool$)
subcls1p
⟨*proof*⟩

declare *Subobjs-Rep* [*code-pred-intro*]
lemma *Subobjs-Sh-code* [*code-pred-intro*]:
$[\![\ (subcls1p\ P)\hat{}{**}\ C\ C';\ subclsSp\ P\ C'\ D;\ Subobjs_R\ P\ D\ Cs]\!]$
$\Longrightarrow Subobjs\ P\ C\ Cs$
⟨*proof*⟩

code-pred
(*modes*: $i \Rightarrow i \Rightarrow i \Rightarrow bool$, $i \Rightarrow i \Rightarrow o \Rightarrow bool$)
Subobjs
⟨*proof*⟩

definition *widen-unique* :: $prog \Rightarrow cname \Rightarrow cname \Rightarrow path \Rightarrow bool$
where *widen-unique* $P\ C\ D\ Cs \longleftrightarrow (\forall\, Cs'.\ Subobjs\ P\ C\ Cs' \longrightarrow last\ Cs' = D \longrightarrow Cs = Cs')$

code-pred [*inductify*, *skip-proof*] *widen-unique* ⟨*proof*⟩

lemma *widen-subcls′*:
$[\![Subobjs\ P\ C\ Cs';\ last\ Cs' = D;\ widen\text{-}unique\ P\ C\ D\ Cs'\ ]\!]$
$\Longrightarrow P \vdash Class\ C \leq Class\ D$
⟨*proof*⟩

declare

widen-refl [*code-pred-intro*]
widen-subcls' [*code-pred-intro widen-subcls*]
widen-null [*code-pred-intro*]

code-pred
(*modes*: $i \Rightarrow i \Rightarrow i \Rightarrow bool$)
widen
⟨*proof*⟩

code-pred
(*modes*: $i \Rightarrow i \Rightarrow i \Rightarrow i \Rightarrow bool$, $i \Rightarrow i \Rightarrow i \Rightarrow o \Rightarrow bool$, $i \Rightarrow i \Rightarrow o \Rightarrow i \Rightarrow bool$, $i \Rightarrow i \Rightarrow o \Rightarrow o \Rightarrow bool$)
leq-path1p
⟨*proof*⟩

lemma *leq-path-unfold*: $P,C \vdash Cs \sqsubseteq Ds \longleftrightarrow (leq\text{-}path1p\ P\ C)^{**}\ Cs\ Ds$
⟨*proof*⟩

code-pred
(*modes*: $i \Rightarrow i \Rightarrow i \Rightarrow o \Rightarrow bool$, $i \Rightarrow i \Rightarrow i \Rightarrow i \Rightarrow bool$)
[*inductify,skip-proof*]
path-via
⟨*proof*⟩

lemma *path-unique-eq* [*code-pred-def*]: $P \vdash Path\ C\ to\ D\ unique \longleftrightarrow$
$(\exists\, Cs.\ Subobjs\ P\ C\ Cs \wedge last\ Cs = D \wedge (\forall\, Cs'.\ Subobjs\ P\ C\ Cs' \longrightarrow last\ Cs' = D \longrightarrow Cs = Cs'))$
⟨*proof*⟩

code-pred
(*modes*: $i \Rightarrow i \Rightarrow o \Rightarrow bool$, $i \Rightarrow i \Rightarrow i \Rightarrow bool$)
[*inductify, skip-proof*]
path-unique ⟨*proof*⟩

Redefine MethodDefs and FieldDecls

definition *MethodDefs'* :: $prog \Rightarrow cname \Rightarrow mname \Rightarrow path \Rightarrow method \Rightarrow bool$ **where**
$MethodDefs'\ P\ C\ M\ Cs\ mthd \equiv (Cs, mthd) \in MethodDefs\ P\ C\ M$

lemma [*code-pred-intro*]:
$Subobjs\ P\ C\ Cs \Longrightarrow class\ P\ (last\ Cs) = \lfloor(Bs,fs,ms)\rfloor \Longrightarrow map\text{-}of\ ms\ M = \lfloor mthd\rfloor \Longrightarrow$
$MethodDefs'\ P\ C\ M\ Cs\ mthd$
⟨*proof*⟩

code-pred
(*modes*: $i \Rightarrow i \Rightarrow i \Rightarrow o \Rightarrow o \Rightarrow bool$, $i \Rightarrow i \Rightarrow i \Rightarrow i \Rightarrow i \Rightarrow bool$)
MethodDefs'

⟨*proof*⟩

definition *FieldDecls'* :: *prog* ⇒ *cname* ⇒ *vname* ⇒ *path* ⇒ *ty* ⇒ *bool* **where**
FieldDecls' P C F Cs T ≡ (*Cs*, *T*) ∈ *FieldDecls P C F*

lemma [*code-pred-intro*]:
Subobjs P C Cs ⟹ *class P* (*last Cs*) = ⌊(*Bs*,*fs*,*ms*)⌋ ⟹ *map-of fs F* = ⌊*T*⌋
⟹
FieldDecls' P C F Cs T
⟨*proof*⟩

code-pred
(*modes*: *i* ⇒ *i* ⇒ *i* ⇒ *o* ⇒ *o* ⇒ *bool*, *i* ⇒ *i* ⇒ *i* ⇒ *i* ⇒ *i* ⇒ *bool*)
FieldDecls'
⟨*proof*⟩

definition *MinimalMethodDefs'* :: *prog* ⇒ *cname* ⇒ *mname* ⇒ *path* ⇒ *method* ⇒ *bool* **where**
MinimalMethodDefs' P C M Cs mthd ≡ (*Cs*, *mthd*) ∈ *MinimalMethodDefs P C M*

definition *MinimalMethodDefs-unique* :: *prog* ⇒ *cname* ⇒ *mname* ⇒ *path* ⇒ *bool* **where**
MinimalMethodDefs-unique P C M Cs ⟷
(∀ *Cs' mthd*. *MethodDefs' P C M Cs' mthd* ⟶ (*leq-path1p P C*)^** *Cs' Cs* ⟶ *Cs'* = *Cs*)

code-pred [*inductify*, *skip-proof*] *MinimalMethodDefs-unique* ⟨*proof*⟩

lemma [*code-pred-intro*]:
MethodDefs' P C M Cs mthd ⟹ *MinimalMethodDefs-unique P C M Cs* ⟹
MinimalMethodDefs' P C M Cs mthd
⟨*proof*⟩

code-pred
(*modes*: *i* ⇒ *i* ⇒ *i* ⇒ *o* ⇒ *o* ⇒ *bool*)
MinimalMethodDefs'
⟨*proof*⟩

definition *LeastMethodDef-unique* :: *prog* ⇒ *cname* ⇒ *mname* ⇒ *path* ⇒ *bool* **where**
LeastMethodDef-unique P C M Cs ⟷
(∀ *Cs' mthd'*. *MethodDefs' P C M Cs' mthd'* ⟶ (*leq-path1p P C*)^** *Cs Cs'*)

code-pred [*inductify*, *skip-proof*] *LeastMethodDef-unique* ⟨*proof*⟩

lemma *LeastMethodDef-unfold*:
P ⊢ *C has least M* = *mthd via Cs* ⟷
MethodDefs′ P C M Cs mthd ∧ *LeastMethodDef-unique P C M Cs*
⟨*proof*⟩

lemma *LeastMethodDef-intro* [*code-pred-intro*]:
⟦ *MethodDefs′ P C M Cs mthd*; *LeastMethodDef-unique P C M Cs* ⟧
⟹ *P* ⊢ *C has least M* = *mthd via Cs*
⟨*proof*⟩

code-pred (*modes*: *i* => *i* => *i* => *o* => *o* => *bool*)
LeastMethodDef
⟨*proof*⟩

definition *OverriderMethodDefs′* :: *prog* ⇒ *subobj* ⇒ *mname* ⇒ *path* ⇒ *method* ⇒ *bool* **where**
OverriderMethodDefs′ P R M Cs mthd ≡ (*Cs*, *mthd*) ∈ *OverriderMethodDefs P R M*

lemma *Overrider1* [*code-pred-intro*]:
P ⊢ (*ldc R*) *has least M* = *mthd′ via Cs′* ⟹
MinimalMethodDefs′ P (*mdc R*) *M Cs mthd* ⟹
last (*snd R*) = *hd Cs′* ⟹ (*leq-path1p P* (*mdc R*))^** *Cs* (*snd R* @ *tl Cs′*) ⟹
OverriderMethodDefs′ P R M Cs mthd
⟨*proof*⟩

lemma *Overrider2* [*code-pred-intro*]:
P ⊢ (*ldc R*) *has least M* = *mthd′ via Cs′* ⟹
MinimalMethodDefs′ P (*mdc R*) *M Cs mthd* ⟹
last (*snd R*) ≠ *hd Cs′* ⟹ (*leq-path1p P* (*mdc R*))^** *Cs Cs′* ⟹
OverriderMethodDefs′ P R M Cs mthd
⟨*proof*⟩

code-pred
(*modes*: *i* ⇒ *i* ⇒ *i* ⇒ *o* ⇒ *o* ⇒ *bool*, *i* ⇒ *i* ⇒ *i* ⇒ *i* ⇒ *o* ⇒ *bool*, *i* ⇒ *i* ⇒ *i* ⇒ *o* ⇒ *i* ⇒ *bool*, *i* ⇒ *i* ⇒ *i* ⇒ *i* ⇒ *i* ⇒ *bool*)
OverriderMethodDefs′
⟨*proof*⟩

definition *WTDynCast-ex* :: *prog* ⇒ *cname* ⇒ *cname* ⇒ *bool*
where *WTDynCast-ex P D C* ⟷ (∃ *Cs*. *P* ⊢ *Path D to C via Cs*)

code-pred [*inductify*, *skip-proof*] *WTDynCast-ex* ⟨*proof*⟩

lemma *WTDynCast-new*:
⟦*P*,*E* ⊢ *e* :: *Class D*; *is-class P C*;
P ⊢ *Path D to C unique* ∨ ¬ *WTDynCast-ex P D C*⟧
⟹ *P*,*E* ⊢ *Cast C e* :: *Class C*
⟨*proof*⟩

definition *WTStaticCast-sub* :: *prog* ⇒ *cname* ⇒ *cname* ⇒ *bool*
where *WTStaticCast-sub P C D* ⟷
P ⊢ *Path D to C unique* ∨
((*subcls1p P*)^** *C D* ∧ (∀ *Cs*. *P* ⊢ *Path C to D via Cs* ⟶ *Subobjs*$_R$ *P C Cs*))

code-pred [*inductify*, *skip-proof*] *WTStaticCast-sub* ⟨*proof*⟩

lemma *WTStaticCast-new*:
⟦*P*,*E* ⊢ *e* :: *Class D*; *is-class P C*; *WTStaticCast-sub P C D* ⟧
⟹ *P*,*E* ⊢ (|*C*|)*e* :: *Class C*
⟨*proof*⟩

lemma *WTBinOp1*: ⟦ *P*,*E* ⊢ *e*$_1$ :: *T*; *P*,*E* ⊢ *e*$_2$ :: *T*⟧
⟹ *P*,*E* ⊢ *e*$_1$ «*Eq*» *e*$_2$ :: *Boolean*
⟨*proof*⟩

lemma *WTBinOp2*: ⟦ *P*,*E* ⊢ *e*$_1$ :: *Integer*; *P*,*E* ⊢ *e*$_2$ :: *Integer* ⟧
⟹ *P*,*E* ⊢ *e*$_1$ «*Add*» *e*$_2$ :: *Integer*
⟨*proof*⟩

lemma *LeastFieldDecl-unfold* [*code-pred-def*]:
P ⊢ *C has least F*:*T via Cs* ⟷
FieldDecls′ P C F Cs T ∧ (∀ *Cs′ T′*. *FieldDecls′ P C F Cs′ T′* ⟶ (*leq-path1p P C*)^** *Cs Cs′*)
⟨*proof*⟩

code-pred [*inductify*, *skip-proof*] *LeastFieldDecl* ⟨*proof*⟩

lemmas [*code-pred-intro*] = *WT-WTs.WTNew*
declare
WTDynCast-new[*code-pred-intro WTDynCast-new*]
WTStaticCast-new[*code-pred-intro WTStaticCast-new*]
lemmas [*code-pred-intro*] = *WT-WTs.WTVal WT-WTs.WTVar*
declare
WTBinOp1[*code-pred-intro WTBinOp1*]
WTBinOp2 [*code-pred-intro WTBinOp2*]
lemmas [*code-pred-intro*] =
WT-WTs.WTLAss WT-WTs.WTFAcc WT-WTs.WTFAss WT-WTs.WTCall WT-StaticCall
WT-WTs.WTBlock WT-WTs.WTSeq WT-WTs.WTCond WT-WTs.WTWhile WT-WTs.WTThrow

lemmas [*code-pred-intro*] = *WT-WTs.WTNil WT-WTs.WTCons*

code-pred
(*modes*: *WT*: $i \Rightarrow i \Rightarrow i \Rightarrow i \Rightarrow bool$, $i \Rightarrow i \Rightarrow i \Rightarrow o \Rightarrow bool$
and *WTs*: $i \Rightarrow i \Rightarrow i \Rightarrow i \Rightarrow bool$, $i \Rightarrow i \Rightarrow i \Rightarrow o \Rightarrow bool$)
WT
⟨*proof*⟩

lemma *casts-to-code* [*code-pred-intro*]:
(*case T of Class C* ⇒ *False* | - ⇒ *True*) ⟹ *P* ⊢ *T casts v to v*
P ⊢ *Class C casts Null to Null*
⟦*Subobjs P* (*last Cs*) *Cs'*; *last Cs'* = *C*;
last Cs = *hd Cs'*; *Cs* @ *tl Cs'* = *Ds*⟧
⟹ *P* ⊢ *Class C casts Ref*(*a*,*Cs*) *to Ref*(*a*,*Ds*)
⟦*Subobjs P* (*last Cs*) *Cs'*; *last Cs'* = *C*; *last Cs* ≠ *hd Cs'*⟧
⟹ *P* ⊢ *Class C casts Ref*(*a*,*Cs*) *to Ref*(*a*,*Cs'*)
⟨*proof*⟩

code-pred (*modes*: $i \Rightarrow i \Rightarrow i \Rightarrow o \Rightarrow bool$, $i \Rightarrow i \Rightarrow i \Rightarrow i \Rightarrow bool$)
casts-to
⟨*proof*⟩

code-pred
(*modes*: $i \Rightarrow i \Rightarrow i \Rightarrow o \Rightarrow bool$, $i \Rightarrow i \Rightarrow i \Rightarrow i \Rightarrow bool$)
Casts-to
⟨*proof*⟩

lemma *card-eq-1-iff-ex1*: *x* ∈ *A* ⟹ *card A* = *1* ⟷ *A* = {*x*}
⟨*proof*⟩

lemma *FinalOverriderMethodDef-unfold* [*code-pred-def*]:
P ⊢ *R has overrider M* = *mthd via Cs* ⟷
OverriderMethodDefs' P R M Cs mthd ∧
(∀ *Cs' mthd'*. *OverriderMethodDefs' P R M Cs' mthd'* ⟶ *Cs* = *Cs'* ∧ *mthd* = *mthd'*)
⟨*proof*⟩

code-pred
(*modes*: *i* => *i* => *i* => *o* => *o* => *bool*)
[*inductify*, *skip-proof*]
FinalOverriderMethodDef
⟨*proof*⟩

code-pred
(*modes*: *i* => *i* => *i* => *i* => *o* => *o* => *bool*, $i \Rightarrow i \Rightarrow i \Rightarrow i \Rightarrow i \Rightarrow i \Rightarrow bool$)
[*inductify*]
SelectMethodDef

⟨*proof*⟩

Isomorphic subo with mapping instead of a map

type-synonym *subo′* = (*path* × (*vname*, *val*) *mapping*)
type-synonym *obj′* = *cname* × *subo′ set*

lift-definition *init-class-fieldmap′* :: *prog* ⇒ *cname* ⇒ (*vname*, *val*) *mapping* **is** *init-class-fieldmap* ⟨*proof*⟩

lemma *init-class-fieldmap′-code* [*code*]:
init-class-fieldmap′ P C =
Mapping (*map* (λ(*F*,*T*).(*F*,*default-val T*)) (*fst*(*snd*(*the*(*class P C*)))))
⟨*proof*⟩

lift-definition *init-obj′* :: *prog* ⇒ *cname* ⇒ *subo′* ⇒ *bool* **is** *init-obj* ⟨*proof*⟩

lemma *init-obj′-intros* [*code-pred-intro*]:
Subobjs P C Cs ⟹ *init-obj′ P C* (*Cs*, *init-class-fieldmap′ P* (*last Cs*))
⟨*proof*⟩

code-pred
(*modes*: *i* ⇒ *i* ⇒ *o* ⇒ *bool as init-obj-pred*)
init-obj′
⟨*proof*⟩

lemma *init-obj-pred-conv*: *set-of-pred* (*init-obj-pred P C*) = *Collect* (*init-obj′ P C*)
⟨*proof*⟩

lift-definition *blank′* :: *prog* ⇒ *cname* ⇒ *obj′* **is** *blank* ⟨*proof*⟩

lemma *blank′-code* [*code*]:
blank′ P C = (*C*, *set-of-pred* (*init-obj-pred P C*))
⟨*proof*⟩

type-synonym *heap′* = *addr* ⇀ *obj′*

abbreviation
cname-of′ :: *heap′* ⇒ *addr* ⇒ *cname* **where**
⋀*hp*. *cname-of′ hp a* == *fst* (*the* (*hp a*))

lift-definition *new-Addr′* :: *heap′* ⇒ *addr option* **is** *new-Addr* ⟨*proof*⟩

lift-definition *start-heap′* :: *prog* ⇒ *heap′* **is** *start-heap* ⟨*proof*⟩

lemma *start-heap′-code* [*code*]:
start-heap′ P = *Map.empty* (*addr-of-sys-xcpt NullPointer* ↦ *blank′ P Null-Pointer*)
(*addr-of-sys-xcpt ClassCast* ↦ *blank′ P ClassCast*)

(*addr-of-sys-xcpt OutOfMemory* ↦ *blank′ P OutOfMemory*)
⟨*proof*⟩

type-synonym
state′ = *heap′* × *locals*

lift-definition *hp′* :: *state′* ⇒ *heap′* **is** *hp* ⟨*proof*⟩

lemma *hp′-code* [*code*]: *hp′* = *fst*
⟨*proof*⟩

lift-definition *lcl′* :: *state′* ⇒ *locals* **is** *lcl* ⟨*proof*⟩

lemma *lcl-code* [*code*]: *lcl′* = *snd*
⟨*proof*⟩

lift-definition *eval′* :: *prog* ⇒ *env* ⇒ *expr* ⇒ *state′* ⇒ *expr* ⇒ *state′* ⇒ *bool*
(-,- ⊢ ((1⟨-,/-⟩) ⇒′/ (1⟨-,/-⟩))) [51,0,0,0,0] 81)
is *eval* ⟨*proof*⟩
lift-definition *evals′* :: *prog* ⇒ *env* ⇒ *expr list* ⇒ *state′* ⇒ *expr list* ⇒ *state′* ⇒ *bool*
(-,- ⊢ ((1⟨-,/-⟩) [⇒′]/ (1⟨-,/-⟩))) [51,0,0,0,0] 81)
is *evals* ⟨*proof*⟩

lemma *New′*:
⟦ *new-Addr′ h* = *Some a*; *h′* = *h*(*a*↦(*blank′ P C*)) ⟧
⟹ *P*,*E* ⊢ ⟨*new C*,(*h*,*l*)⟩ ⇒′ ⟨*ref* (*a*,[*C*]),(*h′*,*l*)⟩
⟨*proof*⟩

lemma *NewFail′*:
new-Addr′ h = *None* ⟹
P,*E* ⊢ ⟨*new C*, (*h*,*l*)⟩ ⇒′ ⟨*THROW OutOfMemory*,(*h*,*l*)⟩
⟨*proof*⟩

lemma *StaticUpCast′*:
⟦ *P*,*E* ⊢ ⟨*e*,*s*$_0$⟩ ⇒′ ⟨*ref* (*a*,*Cs*),*s*$_1$⟩; *P* ⊢ *Path last Cs to C via Cs′*; *Ds* = *Cs*@$_p$*Cs′*
⟧
⟹ *P*,*E* ⊢ ⟨(|*C*|)*e*,*s*$_0$⟩ ⇒′ ⟨*ref* (*a*,*Ds*),*s*$_1$⟩
⟨*proof*⟩

lemma *StaticDownCast′-new*:
⟦*P*,*E* ⊢ ⟨*e*,*s*$_0$⟩ ⇒′ ⟨*ref* (*a*,*Ds*),*s*$_1$⟩; *app Cs* [*C*] *Ds′*; *app Ds′ Cs′ Ds*⟧
⟹ *P*,*E* ⊢ ⟨(|*C*|)*e*,*s*$_0$⟩ ⇒′ ⟨*ref*(*a*,*Cs*@[*C*]),*s*$_1$⟩
⟨*proof*⟩

lemma *StaticCastNull′*:
P,*E* ⊢ ⟨*e*,*s*$_0$⟩ ⇒′ ⟨*null*,*s*$_1$⟩ ⟹
P,*E* ⊢ ⟨(|*C*|)*e*,*s*$_0$⟩ ⇒′ ⟨*null*,*s*$_1$⟩

⟨*proof*⟩

lemma *StaticCastFail′-new*:
⟦ $P,E \vdash \langle e,s_0\rangle \Rightarrow' \langle ref\ (a,Cs),s_1\rangle$; $\neg\ (subcls1p\ P)^{**}\ (last\ Cs)\ C$; $C \notin set\ Cs$⟧
$\Longrightarrow P,E \vdash \langle (\!|\, C \,|\!)\, e,s_0\rangle \Rightarrow' \langle THROW\ ClassCast,s_1\rangle$
⟨*proof*⟩

lemma *StaticCastThrow′*:
$P,E \vdash \langle e,s_0\rangle \Rightarrow' \langle throw\ e',s_1\rangle \Longrightarrow$
$P,E \vdash \langle (\!|\, C \,|\!)\, e,s_0\rangle \Rightarrow' \langle throw\ e',s_1\rangle$
⟨*proof*⟩

lemma *StaticUpDynCast′*:
⟦$P,E \vdash \langle e,s_0\rangle \Rightarrow' \langle ref(a,Cs),s_1\rangle$; $P \vdash Path\ last\ Cs\ to\ C\ unique$;
$P \vdash Path\ last\ Cs\ to\ C\ via\ Cs'$; $Ds = Cs@_p Cs'$ ⟧
$\Longrightarrow P,E \vdash \langle Cast\ C\ e,s_0\rangle \Rightarrow' \langle ref(a,Ds),s_1\rangle$
⟨*proof*⟩

lemma *StaticDownDynCast′-new*:
⟦$P,E \vdash \langle e,s_0\rangle \Rightarrow' \langle ref\ (a,Ds),s_1\rangle$; $app\ Cs\ [C]\ Ds'$; $app\ Ds'\ Cs'\ Ds$⟧
$\Longrightarrow P,E \vdash \langle Cast\ C\ e,s_0\rangle \Rightarrow' \langle ref(a,Cs@[C]),s_1\rangle$
⟨*proof*⟩

lemma *DynCast′*:
⟦ $P,E \vdash \langle e,s_0\rangle \Rightarrow' \langle ref\ (a,Cs),(h,l)\rangle$; $h\ a = Some(D,S)$;
$P \vdash Path\ D\ to\ C\ via\ Cs'$; $P \vdash Path\ D\ to\ C\ unique$ ⟧
$\Longrightarrow P,E \vdash \langle Cast\ C\ e,s_0\rangle \Rightarrow' \langle ref\ (a,Cs'),(h,l)\rangle$
⟨*proof*⟩

lemma *DynCastNull′*:
$P,E \vdash \langle e,s_0\rangle \Rightarrow' \langle null,s_1\rangle \Longrightarrow$
$P,E \vdash \langle Cast\ C\ e,s_0\rangle \Rightarrow' \langle null,s_1\rangle$
⟨*proof*⟩

lemma *DynCastFail′*:
⟦ $P,E \vdash \langle e,s_0\rangle \Rightarrow' \langle ref\ (a,Cs),(h,l)\rangle$; $h\ a = Some(D,S)$; $\neg\ P \vdash Path\ D\ to\ C\ unique$;
$\neg\ P \vdash Path\ last\ Cs\ to\ C\ unique$; $C \notin set\ Cs$ ⟧
$\Longrightarrow P,E \vdash \langle Cast\ C\ e,s_0\rangle \Rightarrow' \langle null,(h,l)\rangle$
⟨*proof*⟩

lemma *DynCastThrow′*:
$P,E \vdash \langle e,s_0\rangle \Rightarrow' \langle throw\ e',s_1\rangle \Longrightarrow$
$P,E \vdash \langle Cast\ C\ e,s_0\rangle \Rightarrow' \langle throw\ e',s_1\rangle$
⟨*proof*⟩

lemma *Val′*:
$P,E \vdash \langle Val\ v,s\rangle \Rightarrow' \langle Val\ v,s\rangle$
⟨*proof*⟩

lemma *BinOp'*:
⟦ $P,E \vdash \langle e_1,s_0\rangle \Rightarrow' \langle Val\ v_1,s_1\rangle$; $P,E \vdash \langle e_2,s_1\rangle \Rightarrow' \langle Val\ v_2,s_2\rangle$;
binop$(bop,v_1,v_2) = Some\ v$ ⟧
⟹ $P,E \vdash \langle e_1\ «bop»\ e_2,s_0\rangle \Rightarrow' \langle Val\ v,s_2\rangle$
⟨*proof*⟩

lemma *BinOpThrow1'*:
$P,E \vdash \langle e_1,s_0\rangle \Rightarrow' \langle throw\ e,s_1\rangle$ ⟹
$P,E \vdash \langle e_1\ «bop»\ e_2,\ s_0\rangle \Rightarrow' \langle throw\ e,s_1\rangle$
⟨*proof*⟩

lemma *BinOpThrow2'*:
⟦ $P,E \vdash \langle e_1,s_0\rangle \Rightarrow' \langle Val\ v_1,s_1\rangle$; $P,E \vdash \langle e_2,s_1\rangle \Rightarrow' \langle throw\ e,s_2\rangle$ ⟧
⟹ $P,E \vdash \langle e_1\ «bop»\ e_2,s_0\rangle \Rightarrow' \langle throw\ e,s_2\rangle$
⟨*proof*⟩

lemma *Var'*:
$l\ V = Some\ v$ ⟹
$P,E \vdash \langle Var\ V,(h,l)\rangle \Rightarrow' \langle Val\ v,(h,l)\rangle$
⟨*proof*⟩

lemma *LAss'*:
⟦ $P,E \vdash \langle e,s_0\rangle \Rightarrow' \langle Val\ v,(h,l)\rangle$; $E\ V = Some\ T$;
$P \vdash T\ casts\ v\ to\ v'$; $l' = l(V \mapsto v')$ ⟧
⟹ $P,E \vdash \langle V:=e,s_0\rangle \Rightarrow' \langle Val\ v',(h,l')\rangle$
⟨*proof*⟩

lemma *LAssThrow'*:
$P,E \vdash \langle e,s_0\rangle \Rightarrow' \langle throw\ e',s_1\rangle$ ⟹
$P,E \vdash \langle V:=e,s_0\rangle \Rightarrow' \langle throw\ e',s_1\rangle$
⟨*proof*⟩

lemma *FAcc'-new*:
⟦ $P,E \vdash \langle e,s_0\rangle \Rightarrow' \langle ref\ (a,Cs'),(h,l)\rangle$; $h\ a = Some(D,S)$;
$Ds = Cs' @_p Cs$; *Predicate-Compile.contains* (*Set-project S Ds*) *fs*; *Mapping.lookup* *fs* $F = Some\ v$ ⟧
⟹ $P,E \vdash \langle e{\cdot}F\{Cs\},s_0\rangle \Rightarrow' \langle Val\ v,(h,l)\rangle$
⟨*proof*⟩

lemma *FAccNull'*:
$P,E \vdash \langle e,s_0\rangle \Rightarrow' \langle null,s_1\rangle$ ⟹
$P,E \vdash \langle e{\cdot}F\{Cs\},s_0\rangle \Rightarrow' \langle THROW\ NullPointer,s_1\rangle$
⟨*proof*⟩

lemma *FAccThrow'*:
$P,E \vdash \langle e,s_0\rangle \Rightarrow' \langle throw\ e',s_1\rangle$ ⟹
$P,E \vdash \langle e{\cdot}F\{Cs\},s_0\rangle \Rightarrow' \langle throw\ e',s_1\rangle$
⟨*proof*⟩

lemma *FAss′-new*:
⟦ $P,E \vdash \langle e_1,s_0\rangle \Rightarrow' \langle ref\ (a,Cs'),s_1\rangle$; $P,E \vdash \langle e_2,s_1\rangle \Rightarrow' \langle Val\ v,(h_2,l_2)\rangle$;
$h_2\ a = Some(D,S)$; $P \vdash (last\ Cs')$ *has least* $F{:}T$ *via* Cs; $P \vdash T$ *casts* v *to* v';
$Ds = Cs'@_pCs$; *Predicate-Compile.contains* (*Set-project* $S\ Ds$) fs; $fs' = Mapping.update\ F\ v'\ fs$;
$S' = S - \{(Ds,fs)\} \cup \{(Ds,fs')\}$; $h_2' = h_2(a{\mapsto}(D,S'))$⟧
$\Longrightarrow P,E \vdash \langle e_1{\cdot}F\{Cs\}{:=}e_2,s_0\rangle \Rightarrow' \langle Val\ v',(h_2',l_2)\rangle$
⟨*proof*⟩

lemma *FAssNull′*:
⟦ $P,E \vdash \langle e_1,s_0\rangle \Rightarrow' \langle null,s_1\rangle$; $P,E \vdash \langle e_2,s_1\rangle \Rightarrow' \langle Val\ v,s_2\rangle$ ⟧ $\Longrightarrow$
$P,E \vdash \langle e_1{\cdot}F\{Cs\}{:=}e_2,s_0\rangle \Rightarrow' \langle THROW\ NullPointer,s_2\rangle$
⟨*proof*⟩

lemma *FAssThrow1′*:
$P,E \vdash \langle e_1,s_0\rangle \Rightarrow' \langle throw\ e',s_1\rangle \Longrightarrow$
$P,E \vdash \langle e_1{\cdot}F\{Cs\}{:=}e_2,s_0\rangle \Rightarrow' \langle throw\ e',s_1\rangle$
⟨*proof*⟩

lemma *FAssThrow2′*:
⟦ $P,E \vdash \langle e_1,s_0\rangle \Rightarrow' \langle Val\ v,s_1\rangle$; $P,E \vdash \langle e_2,s_1\rangle \Rightarrow' \langle throw\ e',s_2\rangle$ ⟧
$\Longrightarrow P,E \vdash \langle e_1{\cdot}F\{Cs\}{:=}e_2,s_0\rangle \Rightarrow' \langle throw\ e',s_2\rangle$
⟨*proof*⟩

lemma *CallObjThrow′*:
$P,E \vdash \langle e,s_0\rangle \Rightarrow' \langle throw\ e',s_1\rangle \Longrightarrow$
$P,E \vdash \langle Call\ e\ Copt\ M\ es,s_0\rangle \Rightarrow' \langle throw\ e',s_1\rangle$
⟨*proof*⟩

lemma *CallParamsThrow′-new*:
⟦ $P,E \vdash \langle e,s0\rangle \Rightarrow' \langle Val\ v,s1\rangle$; $P,E \vdash \langle es,s1\rangle\ [\Rightarrow']\ \langle evs,s2\rangle$;
$map\text{-}val2\ evs\ vs\ (throw\ ex\ \#\ es')$ ⟧
$\Longrightarrow P,E \vdash \langle Call\ e\ Copt\ M\ es,s0\rangle \Rightarrow' \langle throw\ ex,s2\rangle$
⟨*proof*⟩

lemma *Call′-new*:
⟦ $P,E \vdash \langle e,s_0\rangle \Rightarrow' \langle ref\ (a,Cs),s_1\rangle$; $P,E \vdash \langle ps,s_1\rangle\ [\Rightarrow']\ \langle evs,(h_2,l_2)\rangle$;
$map\text{-}val\ evs\ vs$;
$h_2\ a = Some(C,S)$; $P \vdash last\ Cs\ has\ least\ M = (Ts',T',pns',body')\ via\ Ds$;
$P \vdash (C,Cs@_pDs)\ selects\ M = (Ts,T,pns,body)\ via\ Cs'$; $length\ vs = length\ pns$;
$P \vdash Ts\ Casts\ vs\ to\ vs'$; $l_2' = [this{\mapsto}Ref\ (a,Cs'),\ pns[{\mapsto}]vs']$;
$new\text{-}body = (case\ T'\ of\ Class\ D \Rightarrow (\!|D|\!)body\ \ |\ -\ \Rightarrow body)$;
$P,E(this{\mapsto}Class(last\ Cs'),\ pns[{\mapsto}]Ts) \vdash \langle new\text{-}body,(h_2,l_2')\rangle \Rightarrow' \langle e',(h_3,l_3)\rangle$ ⟧
$\Longrightarrow P,E \vdash \langle e{\cdot}M(ps),s_0\rangle \Rightarrow' \langle e',(h_3,l_2)\rangle$
⟨*proof*⟩

lemma *StaticCall′-new*:
⟦ $P,E \vdash \langle e,s_0\rangle \Rightarrow' \langle ref\ (a,Cs),s_1\rangle$; $P,E \vdash \langle ps,s_1\rangle\ [\Rightarrow']\ \langle evs,(h_2,l_2)\rangle$;
$map\text{-}val\ evs\ vs$;

$P \vdash Path\ (last\ Cs)\ to\ C\ unique$; $P \vdash Path\ (last\ Cs)\ to\ C\ via\ Cs''$;
$P \vdash C\ has\ least\ M = (Ts,T,pns,body)\ via\ Cs'$; $Ds = (Cs@_p Cs'')@_p Cs'$;
$length\ vs = length\ pns$; $P \vdash Ts\ Casts\ vs\ to\ vs'$;
$l_2' = [this \mapsto Ref\ (a,Ds),\ pns[\mapsto]vs']$;
$P,E(this \mapsto Class(last\ Ds),\ pns[\mapsto]Ts) \vdash \langle body,(h_2,l_2')\rangle \Rightarrow' \langle e',(h_3,l_3)\rangle$ ⟧
$\Longrightarrow P,E \vdash \langle e\bullet(C::)M(ps),s_0\rangle \Rightarrow' \langle e',(h_3,l_2)\rangle$
⟨*proof*⟩

lemma *CallNull′-new*:
⟦ $P,E \vdash \langle e,s_0\rangle \Rightarrow' \langle null,s_1\rangle$; $P,E \vdash \langle es,s_1\rangle\ [\Rightarrow']\ \langle evs,s_2\rangle$; *map-val evs vs* ⟧
$\Longrightarrow P,E \vdash \langle Call\ e\ Copt\ M\ es,s_0\rangle \Rightarrow' \langle THROW\ NullPointer,s_2\rangle$
⟨*proof*⟩

lemma *Block′*:
$[\![P,E(V \mapsto T) \vdash \langle e_0,(h_0,l_0(V:=None))\rangle \Rightarrow' \langle e_1,(h_1,l_1)\rangle\ ]\!] \Longrightarrow$
$P,E \vdash \langle\{V:T;\ e_0\},(h_0,l_0)\rangle \Rightarrow' \langle e_1,(h_1,l_1(V:=l_0\ V))\rangle$
⟨*proof*⟩

lemma *Seq′*:
⟦ $P,E \vdash \langle e_0,s_0\rangle \Rightarrow' \langle Val\ v,s_1\rangle$; $P,E \vdash \langle e_1,s_1\rangle \Rightarrow' \langle e_2,s_2\rangle$ ⟧
$\Longrightarrow P,E \vdash \langle e_0;;e_1,s_0\rangle \Rightarrow' \langle e_2,s_2\rangle$
⟨*proof*⟩

lemma *SeqThrow′*:
$P,E \vdash \langle e_0,s_0\rangle \Rightarrow' \langle throw\ e,s_1\rangle \Longrightarrow$
$P,E \vdash \langle e_0;;e_1,s_0\rangle \Rightarrow' \langle throw\ e,s_1\rangle$
⟨*proof*⟩

lemma *CondT′*:
⟦ $P,E \vdash \langle e,s_0\rangle \Rightarrow' \langle true,s_1\rangle$; $P,E \vdash \langle e_1,s_1\rangle \Rightarrow' \langle e',s_2\rangle$ ⟧
$\Longrightarrow P,E \vdash \langle if\ (e)\ e_1\ else\ e_2,s_0\rangle \Rightarrow' \langle e',s_2\rangle$
⟨*proof*⟩

lemma *CondF′*:
⟦ $P,E \vdash \langle e,s_0\rangle \Rightarrow' \langle false,s_1\rangle$; $P,E \vdash \langle e_2,s_1\rangle \Rightarrow' \langle e',s_2\rangle$ ⟧
$\Longrightarrow P,E \vdash \langle if\ (e)\ e_1\ else\ e_2,s_0\rangle \Rightarrow' \langle e',s_2\rangle$
⟨*proof*⟩

lemma *CondThrow′*:
$P,E \vdash \langle e,s_0\rangle \Rightarrow' \langle throw\ e',s_1\rangle \Longrightarrow$
$P,E \vdash \langle if\ (e)\ e_1\ else\ e_2,\ s_0\rangle \Rightarrow' \langle throw\ e',s_1\rangle$
⟨*proof*⟩

lemma *WhileF′*:
$P,E \vdash \langle e,s_0\rangle \Rightarrow' \langle false,s_1\rangle \Longrightarrow$
$P,E \vdash \langle while\ (e)\ c,s_0\rangle \Rightarrow' \langle unit,s_1\rangle$
⟨*proof*⟩

lemma *WhileT′*:

⟦ $P,E \vdash \langle e,s_0\rangle \Rightarrow' \langle true,s_1\rangle$; $P,E \vdash \langle c,s_1\rangle \Rightarrow' \langle Val\ v_1,s_2\rangle$;
$P,E \vdash \langle while\ (e)\ c,s_2\rangle \Rightarrow' \langle e_3,s_3\rangle$ ⟧
⟹ $P,E \vdash \langle while\ (e)\ c,s_0\rangle \Rightarrow' \langle e_3,s_3\rangle$
⟨*proof*⟩

lemma *WhileCondThrow'*:
$P,E \vdash \langle e,s_0\rangle \Rightarrow' \langle\ throw\ e',s_1\rangle \Longrightarrow$
$P,E \vdash \langle while\ (e)\ c,s_0\rangle \Rightarrow' \langle throw\ e',s_1\rangle$
⟨*proof*⟩

lemma *WhileBodyThrow'*:
⟦ $P,E \vdash \langle e,s_0\rangle \Rightarrow' \langle true,s_1\rangle$; $P,E \vdash \langle c,s_1\rangle \Rightarrow' \langle throw\ e',s_2\rangle$⟧
⟹ $P,E \vdash \langle while\ (e)\ c,s_0\rangle \Rightarrow' \langle throw\ e',s_2\rangle$
⟨*proof*⟩

lemma *Throw'*:
$P,E \vdash \langle e,s_0\rangle \Rightarrow' \langle ref\ r,s_1\rangle \Longrightarrow$
$P,E \vdash \langle throw\ e,s_0\rangle \Rightarrow' \langle Throw\ r,s_1\rangle$
⟨*proof*⟩

lemma *ThrowNull'*:
$P,E \vdash \langle e,s_0\rangle \Rightarrow' \langle null,s_1\rangle \Longrightarrow$
$P,E \vdash \langle throw\ e,s_0\rangle \Rightarrow' \langle THROW\ NullPointer,s_1\rangle$
⟨*proof*⟩

lemma *ThrowThrow'*:
$P,E \vdash \langle e,s_0\rangle \Rightarrow' \langle throw\ e',s_1\rangle \Longrightarrow$
$P,E \vdash \langle throw\ e,s_0\rangle \Rightarrow' \langle throw\ e',s_1\rangle$
⟨*proof*⟩

lemma *Nil'*:
$P,E \vdash \langle[],s\rangle\ [\Rightarrow']\ \langle[],s\rangle$
⟨*proof*⟩

lemma *Cons'*:
⟦ $P,E \vdash \langle e,s_0\rangle \Rightarrow' \langle Val\ v,s_1\rangle$; $P,E \vdash \langle es,s_1\rangle\ [\Rightarrow']\ \langle es',s_2\rangle$ ⟧
⟹ $P,E \vdash \langle e\#es,s_0\rangle\ [\Rightarrow']\ \langle Val\ v\ \#\ es',s_2\rangle$
⟨*proof*⟩

lemma *ConsThrow'*:
$P,E \vdash \langle e,\ s_0\rangle \Rightarrow' \langle throw\ e',\ s_1\rangle \Longrightarrow$
$P,E \vdash \langle e\#es,\ s_0\rangle\ [\Rightarrow']\ \langle throw\ e'\ \#\ es,\ s_1\rangle$
⟨*proof*⟩

Axiomatic heap address model refinement

partial-function (*option*) *lowest* :: (*nat* ⇒ *bool*) ⇒ *nat* ⇒ *nat option*
where
[*code*]: *lowest P n* = (*if P n then Some n else lowest P* (*Suc n*))

axiomatization
where
new-Addr'-code [*code*]: *new-Addr' h = lowest* (*Option.is-none* ∘ *h*) *0*
— admissible: a tightening of the specification of *new-Addr'*

lemma *eval'-cases*
[*consumes 1*,
case-names New NewFail StaticUpCast StaticDownCast StaticCastNull StaticCastFail
StaticCastThrow StaticUpDynCast StaticDownDynCast DynCast DynCastNull DynCastFail
DynCastThrow Val BinOp BinOpThrow1 BinOpThrow2 Var LAss LAssThrow FAcc FAccNull FAccThrow
FAss FAssNull FAssThrow1 FAssThrow2 CallObjThrow CallParamsThrow Call StaticCall CallNull
Block Seq SeqThrow CondT CondF CondThrow WhileF WhileT WhileCondThrow WhileBodyThrow
Throw ThrowNull ThrowThrow]:
assumes $P,x \vdash \langle y,z\rangle \Rightarrow' \langle u,v\rangle$
and $\bigwedge h\ a\ h'\ C\ E\ l.\ x = E \Longrightarrow y = new\ C \Longrightarrow z = (h,\ l) \Longrightarrow u = ref\ (a,\ [C]) \Longrightarrow$
$v = (h',\ l) \Longrightarrow new\text{-}Addr'\ h = \lfloor a \rfloor \Longrightarrow h' = h(a \mapsto blank'\ P\ C) \Longrightarrow thesis$
and $\bigwedge h\ E\ C\ l.\ x = E \Longrightarrow y = new\ C \Longrightarrow z = (h,\ l) \Longrightarrow$
$u = Throw\ (addr\text{-}of\text{-}sys\text{-}xcpt\ OutOfMemory,\ [OutOfMemory]) \Longrightarrow$
$v = (h,\ l) \Longrightarrow new\text{-}Addr'\ h = None \Longrightarrow thesis$
and $\bigwedge E\ e\ s_0\ a\ Cs\ s_1\ C\ Cs'\ Ds.\ x = E \Longrightarrow y = (\!|C|\!)e \Longrightarrow z = s_0 \Longrightarrow$
$u = ref\ (a,\ Ds) \Longrightarrow v = s_1 \Longrightarrow P,E \vdash \langle e,s_0\rangle \Rightarrow' \langle ref\ (a,\ Cs),s_1\rangle \Longrightarrow$
$P \vdash Path\ last\ Cs\ to\ C\ via\ Cs' \Longrightarrow Ds = Cs\ @_p\ Cs' \Longrightarrow thesis$
and $\bigwedge E\ e\ s_0\ a\ Cs\ C\ Cs'\ s_1.\ x = E \Longrightarrow y = (\!|C|\!)e \Longrightarrow z = s_0 \Longrightarrow u = ref\ (a,\ Cs\ @\ [C]) \Longrightarrow$
$v = s_1 \Longrightarrow P,E \vdash \langle e,s_0\rangle \Rightarrow' \langle ref\ (a,\ Cs\ @\ [C]\ @\ Cs'),s_1\rangle \Longrightarrow thesis$
and $\bigwedge E\ e\ s_0\ s_1\ C.\ x = E \Longrightarrow y = (\!|C|\!)e \Longrightarrow z = s_0 \Longrightarrow u = null \Longrightarrow v = s_1 \Longrightarrow$
$P,E \vdash \langle e,s_0\rangle \Rightarrow' \langle null,s_1\rangle \Longrightarrow thesis$
and $\bigwedge E\ e\ s_0\ a\ Cs\ s_1\ C.\ x = E \Longrightarrow y = (\!|C|\!)e \Longrightarrow z = s_0 \Longrightarrow$
$u = Throw\ (addr\text{-}of\text{-}sys\text{-}xcpt\ ClassCast,\ [ClassCast]) \Longrightarrow\ v = s_1 \Longrightarrow$
$P,E \vdash \langle e,s_0\rangle \Rightarrow' \langle ref\ (a,\ Cs),s_1\rangle \Longrightarrow (last\ Cs,\ C) \notin (subcls1\ P)^* \Longrightarrow C \notin set\ Cs \Longrightarrow thesis$
and $\bigwedge E\ e\ s_0\ e'\ s_1\ C.\ x = E \Longrightarrow y = (\!|C|\!)e \Longrightarrow z = s_0 \Longrightarrow u = throw\ e' \Longrightarrow v = s_1 \Longrightarrow$
$P,E \vdash \langle e,s_0\rangle \Rightarrow' \langle throw\ e',s_1\rangle \Longrightarrow thesis$
and $\bigwedge E\ e\ s_0\ a\ Cs\ s_1\ C\ Cs'\ Ds.\ x = E \Longrightarrow y = Cast\ C\ e \Longrightarrow z = s_0 \Longrightarrow u = ref\ (a,\ Ds) \Longrightarrow$
$v = s_1 \Longrightarrow P,E \vdash \langle e,s_0\rangle \Rightarrow' \langle ref\ (a,\ Cs),s_1\rangle \Longrightarrow P \vdash Path\ last\ Cs\ to\ C\ unique \Longrightarrow$
$P \vdash Path\ last\ Cs\ to\ C\ via\ Cs' \Longrightarrow Ds = Cs\ @_p\ Cs' \Longrightarrow thesis$
and $\bigwedge E\ e\ s_0\ a\ Cs\ C\ Cs'\ s_1.\ x = E \Longrightarrow y = Cast\ C\ e \Longrightarrow z = s_0 \Longrightarrow$
$u = ref\ (a,\ Cs\ @\ [C]) \Longrightarrow v = s_1 \Longrightarrow P,E \vdash \langle e,s_0\rangle \Rightarrow' \langle ref\ (a,\ Cs\ @\ [C]\ @\ Cs'),s_1\rangle \Longrightarrow thesis$

and $\bigwedge E\ e\ s_0\ a\ Cs\ h\ l\ D\ S\ C\ Cs'.\ x = E \Longrightarrow y = Cast\ C\ e \Longrightarrow z = s_0 \Longrightarrow$
$u = ref\ (a,\ Cs') \Longrightarrow v = (h,\ l) \Longrightarrow P,E \vdash \langle e,s_0\rangle \Rightarrow' \langle ref\ (a,\ Cs),(h,\ l)\rangle \Longrightarrow$
$h\ a = \lfloor (D,\ S)\rfloor \Longrightarrow P \vdash Path\ D\ to\ C\ via\ Cs' \Longrightarrow P \vdash Path\ D\ to\ C\ unique \Longrightarrow$
thesis

and $\bigwedge E\ e\ s_0\ s_1\ C.\ x = E \Longrightarrow y = Cast\ C\ e \Longrightarrow z = s_0 \Longrightarrow u = null \Longrightarrow v = s_1 \Longrightarrow$
$P,E \vdash \langle e,s_0\rangle \Rightarrow' \langle null,s_1\rangle \Longrightarrow thesis$

and $\bigwedge E\ e\ s_0\ a\ Cs\ h\ l\ D\ S\ C.\ x = E \Longrightarrow y = Cast\ C\ e \Longrightarrow z = s_0 \Longrightarrow u = null \Longrightarrow$
$v = (h,\ l) \Longrightarrow P,E \vdash \langle e,s_0\rangle \Rightarrow' \langle ref\ (a,\ Cs),(h,\ l)\rangle \Longrightarrow h\ a = \lfloor (D,\ S)\rfloor \Longrightarrow$
$\neg\ P \vdash Path\ D\ to\ C\ unique \Longrightarrow \neg\ P \vdash Path\ last\ Cs\ to\ C\ unique \Longrightarrow C \notin set\ Cs \Longrightarrow thesis$

and $\bigwedge E\ e\ s_0\ e'\ s_1\ C.\ x = E \Longrightarrow y = Cast\ C\ e \Longrightarrow z = s_0 \Longrightarrow u = throw\ e' \Longrightarrow v = s_1$
$\Longrightarrow P,E \vdash \langle e,s_0\rangle \Rightarrow' \langle throw\ e',s_1\rangle \Longrightarrow thesis$

and $\bigwedge E\ va\ s.\ x = E \Longrightarrow y = Val\ va \Longrightarrow z = s \Longrightarrow u = Val\ va \Longrightarrow v = s \Longrightarrow thesis$

and $\bigwedge E\ e_1\ s_0\ v_1\ s_1\ e_2\ v_2\ s_2\ bop\ va.\ x = E \Longrightarrow y = e_1\ «bop»\ e_2 \Longrightarrow z = s_0 \Longrightarrow$
$u = Val\ va \Longrightarrow v = s_2 \Longrightarrow P,E \vdash \langle e_1,s_0\rangle \Rightarrow' \langle Val\ v_1,s_1\rangle \Longrightarrow$
$P,E \vdash \langle e_2,s_1\rangle \Rightarrow' \langle Val\ v_2,s_2\rangle \Longrightarrow binop\ (bop,\ v_1,\ v_2) = \lfloor va\rfloor \Longrightarrow thesis$

and $\bigwedge E\ e_1\ s_0\ e\ s_1\ bop\ e_2.\ x = E \Longrightarrow y = e_1\ «bop»\ e_2 \Longrightarrow z = s_0 \Longrightarrow u = throw\ e \Longrightarrow v = s_1\ \Longrightarrow$
$P,E \vdash \langle e_1,s_0\rangle \Rightarrow' \langle throw\ e,s_1\rangle \Longrightarrow thesis$

and $\bigwedge E\ e_1\ s_0\ v_1\ s_1\ e_2\ e\ s_2\ bop.\ x = E \Longrightarrow y = e_1\ «bop»\ e_2 \Longrightarrow z = s_0 \Longrightarrow u = throw\ e \Longrightarrow$
$v = s_2 \Longrightarrow P,E \vdash \langle e_1,s_0\rangle \Rightarrow' \langle Val\ v_1,s_1\rangle \Longrightarrow P,E \vdash \langle e_2,s_1\rangle \Rightarrow' \langle throw\ e,s_2\rangle \Longrightarrow thesis$

and $\bigwedge l\ V\ va\ E\ h.\ x = E \Longrightarrow y = Var\ V \Longrightarrow z = (h,\ l) \Longrightarrow u = Val\ va \Longrightarrow v = (h,\ l) \Longrightarrow$
$l\ V = \lfloor va\rfloor \Longrightarrow thesis$

and $\bigwedge E\ e\ s_0\ va\ h\ l\ V\ T\ v'\ l'.\ x = E \Longrightarrow y = V{:=}e \Longrightarrow z = s_0 \Longrightarrow u = Val\ v' \Longrightarrow$
$v = (h,\ l') \Longrightarrow P,E \vdash \langle e,s_0\rangle \Rightarrow' \langle Val\ va,(h,\ l)\rangle \Longrightarrow$
$E\ V = \lfloor T\rfloor \Longrightarrow P \vdash T\ casts\ va\ to\ v' \Longrightarrow l' = l(V \mapsto v') \Longrightarrow thesis$

and $\bigwedge E\ e\ s_0\ e'\ s_1\ V.\ x = E \Longrightarrow y = V{:=}e \Longrightarrow z = s_0 \Longrightarrow u = throw\ e' \Longrightarrow v = s_1 \Longrightarrow$
$P,E \vdash \langle e,s_0\rangle \Rightarrow' \langle throw\ e',s_1\rangle \Longrightarrow thesis$

and $\bigwedge E\ e\ s_0\ a\ Cs'\ h\ l\ D\ S\ Ds\ Cs\ fs\ F\ va.\ x = E \Longrightarrow y = e{\cdot}F\{Cs\} \Longrightarrow z = s_0 \Longrightarrow$
$u = Val\ va \Longrightarrow v = (h,\ l) \Longrightarrow P,E \vdash \langle e,s_0\rangle \Rightarrow' \langle ref\ (a,\ Cs'),(h,\ l)\rangle \Longrightarrow$
$h\ a = \lfloor (D,\ S)\rfloor \Longrightarrow Ds = Cs' \mathbin{@_p} Cs \Longrightarrow (Ds,\ fs) \in S \Longrightarrow Mapping.lookup\ fs\ F = \lfloor va\rfloor \Longrightarrow thesis$

and $\bigwedge E\ e\ s_0\ s_1\ F\ Cs.\ x = E \Longrightarrow y = e{\cdot}F\{Cs\} \Longrightarrow z = s_0 \Longrightarrow$
$u = Throw\ (addr\text{-}of\text{-}sys\text{-}xcpt\ NullPointer,\ [NullPointer]) \Longrightarrow$
$v = s_1 \Longrightarrow P,E \vdash \langle e,s_0\rangle \Rightarrow' \langle null,s_1\rangle \Longrightarrow thesis$

and $\bigwedge E\ e\ s_0\ e'\ s_1\ F\ Cs.\ x = E \Longrightarrow y = e{\cdot}F\{Cs\} \Longrightarrow z = s_0 \Longrightarrow u = throw\ e' \Longrightarrow v = s_1 \Longrightarrow$
$P,E \vdash \langle e,s_0\rangle \Rightarrow' \langle throw\ e',s_1\rangle \Longrightarrow thesis$

and $\bigwedge E\ e_1\ s_0\ a\ Cs'\ s_1\ e_2\ va\ h_2\ l_2\ D\ S\ F\ T\ Cs\ v'\ Ds\ fs\ fs'\ S'\ h_2'.$

$x = E \Longrightarrow y = e_1{\cdot}F\{Cs\} := e_2 \Longrightarrow z = s_0 \Longrightarrow u = Val\ v' \Longrightarrow v = (h_2', l_2) \Longrightarrow$
$P,E \vdash \langle e_1,s_0\rangle \Rightarrow' \langle ref\ (a,\ Cs'),s_1\rangle \Longrightarrow P,E \vdash \langle e_2,s_1\rangle \Rightarrow' \langle Val\ va,(h_2,\ l_2)\rangle \Longrightarrow$
$h_2\ a = \lfloor (D,\ S)\rfloor \Longrightarrow P \vdash last\ Cs'\ has\ least\ F{:}T\ via\ Cs \Longrightarrow$
$P \vdash T\ casts\ va\ to\ v' \Longrightarrow Ds = Cs'\ @_p\ Cs \Longrightarrow (Ds,\ fs) \in S \Longrightarrow fs' = Mapping.update\ F\ v'\ fs \Longrightarrow$
$S' = S - \{(Ds,\ fs)\} \cup \{(Ds,\ fs')\} \Longrightarrow h_2' = h_2(a \mapsto (D,\ S')) \Longrightarrow thesis$
and $\bigwedge E\ e_1\ s_0\ s_1\ e_2\ va\ s_2\ F\ Cs.\ x = E \Longrightarrow y = e_1{\cdot}F\{Cs\} := e_2 \Longrightarrow z = s_0 \Longrightarrow$
$u = Throw\ (addr\text{-}of\text{-}sys\text{-}xcpt\ NullPointer,\ [NullPointer]) \Longrightarrow$
$v = s_2 \Longrightarrow P,E \vdash \langle e_1,s_0\rangle \Rightarrow' \langle null,s_1\rangle \Longrightarrow P,E \vdash \langle e_2,s_1\rangle \Rightarrow' \langle Val\ va,s_2\rangle \Longrightarrow thesis$
and $\bigwedge E\ e_1\ s_0\ e'\ s_1\ F\ Cs\ e_2.\ x = E \Longrightarrow y = e_1{\cdot}F\{Cs\} := e_2 \Longrightarrow$
$z = s_0 \Longrightarrow u = throw\ e' \Longrightarrow v = s_1 \Longrightarrow P,E \vdash \langle e_1,s_0\rangle \Rightarrow' \langle throw\ e',s_1\rangle \Longrightarrow thesis$
and $\bigwedge E\ e_1\ s_0\ va\ s_1\ e_2\ e'\ s_2\ F\ Cs.\ x = E \Longrightarrow y = e_1{\cdot}F\{Cs\} := e_2 \Longrightarrow z = s_0 \Longrightarrow$
$u = throw\ e' \Longrightarrow v = s_2 \Longrightarrow P,E \vdash \langle e_1,s_0\rangle \Rightarrow' \langle Val\ va,s_1\rangle \Longrightarrow P,E \vdash \langle e_2,s_1\rangle \Rightarrow' \langle throw\ e',s_2\rangle \Longrightarrow$
$thesis$
and $\bigwedge E\ e\ s_0\ e'\ s_1\ Copt\ M\ es.\ x = E \Longrightarrow y = Call\ e\ Copt\ M\ es \Longrightarrow$
$z = s_0 \Longrightarrow u = throw\ e' \Longrightarrow v = s_1 \Longrightarrow P,E \vdash \langle e,s_0\rangle \Rightarrow' \langle throw\ e',s_1\rangle \Longrightarrow thesis$
and $\bigwedge E\ e\ s_0\ va\ s_1\ es\ vs\ ex\ es'\ s_2\ Copt\ M.\ x = E \Longrightarrow y = Call\ e\ Copt\ M\ es \Longrightarrow$
$z = s_0 \Longrightarrow u = throw\ ex \Longrightarrow v = s_2 \Longrightarrow P,E \vdash \langle e,s_0\rangle \Rightarrow' \langle Val\ va,s_1\rangle \Longrightarrow$
$P,E \vdash \langle es,s_1\rangle\ [\Rightarrow']\ \langle map\ Val\ vs\ @\ throw\ ex\ \#\ es',s_2\rangle \Longrightarrow thesis$
and $\bigwedge E\ e\ s_0\ a\ Cs\ s_1\ ps\ vs\ h_2\ l_2\ C\ S\ M\ Ts'\ T'\ pns'\ body'\ Ds\ Ts\ T\ pns\ body\ Cs'\ vs'\ l_2'\ new\text{-}body\ e'$
$h_3\ l_3.\ x = E \Longrightarrow y = Call\ e\ None\ M\ ps \Longrightarrow z = s_0 \Longrightarrow u = e' \Longrightarrow v = (h_3,\ l_2) \Longrightarrow$
$P,E \vdash \langle e,s_0\rangle \Rightarrow' \langle ref\ (a,\ Cs),s_1\rangle \Longrightarrow P,E \vdash \langle ps,s_1\rangle\ [\Rightarrow']\ \langle map\ Val\ vs,(h_2,\ l_2)\rangle \Longrightarrow$
$h_2\ a = \lfloor (C,\ S)\rfloor \Longrightarrow P \vdash last\ Cs\ has\ least\ M = (Ts',\ T',\ pns',\ body')\ via\ Ds \Longrightarrow$
$P \vdash (C,Cs\ @_p\ Ds)\ selects\ M = (Ts,\ T,\ pns,\ body)\ via\ Cs' \Longrightarrow length\ vs = length\ pns \Longrightarrow$
$P \vdash Ts\ Casts\ vs\ to\ vs' \Longrightarrow l_2' = [this \mapsto Ref\ (a,\ Cs'),\ pns\ [\mapsto]\ vs'] \Longrightarrow$
$new\text{-}body = (case\ T'\ of\ Class\ D \Rightarrow (\!|D|\!)body \mid \text{-} \Rightarrow body) \Longrightarrow$
$P,E(this \mapsto Class\ (last\ Cs'),\ pns\ [\mapsto]\ Ts) \vdash \langle new\text{-}body,(h_2,\ l_2')\rangle \Rightarrow' \langle e',(h_3,\ l_3)\rangle \Longrightarrow$
$thesis$
and $\bigwedge E\ e\ s_0\ a\ Cs\ s_1\ ps\ vs\ h_2\ l_2\ C\ Cs''\ M\ Ts\ T\ pns\ body\ Cs'\ Ds\ vs'\ l_2'\ e'\ h_3\ l_3.$
$x = E \Longrightarrow y = Call\ e\ \lfloor C\rfloor\ M\ ps \Longrightarrow z = s_0 \Longrightarrow u = e' \Longrightarrow v = (h_3,\ l_2) \Longrightarrow$
$P,E \vdash \langle e,s_0\rangle \Rightarrow' \langle ref\ (a,\ Cs),s_1\rangle \Longrightarrow P,E \vdash \langle ps,s_1\rangle\ [\Rightarrow']\ \langle map\ Val\ vs,(h_2,\ l_2)\rangle \Longrightarrow$
$P \vdash Path\ last\ Cs\ to\ C\ unique \Longrightarrow P \vdash Path\ last\ Cs\ to\ C\ via\ Cs'' \Longrightarrow$
$P \vdash C\ has\ least\ M = (Ts,\ T,\ pns,\ body)\ via\ Cs' \Longrightarrow Ds = (Cs\ @_p\ Cs'')\ @_p\ Cs' \Longrightarrow$
$length\ vs = length\ pns \Longrightarrow P \vdash Ts\ Casts\ vs\ to\ vs' \Longrightarrow$
$l_2' = [this \mapsto Ref\ (a,\ Ds),\ pns\ [\mapsto]\ vs'] \Longrightarrow$

$P,E(this \mapsto Class\ (last\ Ds),\ pns\ [\mapsto]\ Ts) \vdash \langle body,(h_2,\ l_2')\rangle \Rightarrow' \langle e',(h_3,\ l_3)\rangle \Longrightarrow$
thesis

and $\bigwedge E\ e\ s_0\ s_1\ es\ vs\ s_2\ Copt\ M.\ x = E \Longrightarrow y = Call\ e\ Copt\ M\ es \Longrightarrow z = s_0 \Longrightarrow$
$u = Throw\ (addr\text{-}of\text{-}sys\text{-}xcpt\ NullPointer,\ [NullPointer]) \Longrightarrow$
$v = s_2 \Longrightarrow P,E \vdash \langle e,s_0\rangle \Rightarrow' \langle null,s_1\rangle \Longrightarrow P,E \vdash \langle es,s_1\rangle\ [\Rightarrow']\ \langle map\ Val\ vs,s_2\rangle \Longrightarrow thesis$

and $\bigwedge E\ V\ T\ e_0\ h_0\ l_0\ e_1\ h_1\ l_1.$
$x = E \Longrightarrow y = \{V{:}T;\ e_0\} \Longrightarrow z = (h_0,\ l_0) \Longrightarrow u = e_1 \Longrightarrow$
$v = (h_1,\ l_1(V := l_0\ V)) \Longrightarrow P,E(V \mapsto T) \vdash \langle e_0,(h_0,\ l_0(V := None))\rangle \Rightarrow' \langle e_1,(h_1,\ l_1)\rangle \Longrightarrow thesis$

and $\bigwedge E\ e_0\ s_0\ va\ s_1\ e_1\ e_2\ s_2.\ x = E \Longrightarrow y = e_0;;\ e_1 \Longrightarrow z = s_0 \Longrightarrow u = e_2 \Longrightarrow$
$v = s_2 \Longrightarrow P,E \vdash \langle e_0,s_0\rangle \Rightarrow' \langle Val\ va,s_1\rangle \Longrightarrow P,E \vdash \langle e_1,s_1\rangle \Rightarrow' \langle e_2,s_2\rangle \Longrightarrow thesis$

and $\bigwedge E\ e_0\ s_0\ e\ s_1\ e_1.\ x = E \Longrightarrow y = e_0;;\ e_1 \Longrightarrow z = s_0 \Longrightarrow u = throw\ e \Longrightarrow v = s_1 \Longrightarrow$
$P,E \vdash \langle e_0,s_0\rangle \Rightarrow' \langle throw\ e,s_1\rangle \Longrightarrow thesis$

and $\bigwedge E\ e\ s_0\ s_1\ e_1\ e'\ s_2\ e_2.\ x = E \Longrightarrow y = if\ (e)\ e_1\ else\ e_2 \Longrightarrow z = s_0 \Longrightarrow u = e' \Longrightarrow$
$v = s_2 \Longrightarrow P,E \vdash \langle e,s_0\rangle \Rightarrow' \langle true,s_1\rangle \Longrightarrow P,E \vdash \langle e_1,s_1\rangle \Rightarrow' \langle e',s_2\rangle \Longrightarrow thesis$

and $\bigwedge E\ e\ s_0\ s_1\ e_2\ e'\ s_2\ e_1.\ x = E \Longrightarrow y = if\ (e)\ e_1\ else\ e_2 \Longrightarrow z = s_0 \Longrightarrow$
$u = e' \Longrightarrow v = s_2 \Longrightarrow P,E \vdash \langle e,s_0\rangle \Rightarrow' \langle false,s_1\rangle \Longrightarrow P,E \vdash \langle e_2,s_1\rangle \Rightarrow' \langle e',s_2\rangle \Longrightarrow thesis$

and $\bigwedge E\ e\ s_0\ e'\ s_1\ e_1\ e_2.\ x = E \Longrightarrow y = if\ (e)\ e_1\ else\ e_2 \Longrightarrow$
$z = s_0 \Longrightarrow u = throw\ e' \Longrightarrow v = s_1 \Longrightarrow P,E \vdash \langle e,s_0\rangle \Rightarrow' \langle throw\ e',s_1\rangle \Longrightarrow thesis$

and $\bigwedge E\ e\ s_0\ s_1\ c.\ x = E \Longrightarrow y = while\ (e)\ c \Longrightarrow z = s_0 \Longrightarrow u = unit \Longrightarrow v = s_1 \Longrightarrow$
$P,E \vdash \langle e,s_0\rangle \Rightarrow' \langle false,s_1\rangle \Longrightarrow thesis$

and $\bigwedge E\ e\ s_0\ s_1\ c\ v_1\ s_2\ e_3\ s_3.\ x = E \Longrightarrow y = while\ (e)\ c \Longrightarrow z = s_0 \Longrightarrow u = e_3 \Longrightarrow$
$v = s_3 \Longrightarrow P,E \vdash \langle e,s_0\rangle \Rightarrow' \langle true,s_1\rangle \Longrightarrow P,E \vdash \langle c,s_1\rangle \Rightarrow' \langle Val\ v_1,s_2\rangle \Longrightarrow$
$P,E \vdash \langle while\ (e)\ c,s_2\rangle \Rightarrow' \langle e_3,s_3\rangle \Longrightarrow thesis$

and $\bigwedge E\ e\ s_0\ e'\ s_1\ c.\ x = E \Longrightarrow y = while\ (e)\ c \Longrightarrow z = s_0 \Longrightarrow u = throw\ e' \Longrightarrow v = s_1 \Longrightarrow$
$P,E \vdash \langle e,s_0\rangle \Rightarrow' \langle throw\ e',s_1\rangle \Longrightarrow thesis$

and $\bigwedge E\ e\ s_0\ s_1\ c\ e'\ s_2.\ x = E \Longrightarrow y = while\ (e)\ c \Longrightarrow z = s_0 \Longrightarrow u = throw\ e' \Longrightarrow$
$v = s_2 \Longrightarrow P,E \vdash \langle e,s_0\rangle \Rightarrow' \langle true,s_1\rangle \Longrightarrow P,E \vdash \langle c,s_1\rangle \Rightarrow' \langle throw\ e',s_2\rangle \Longrightarrow thesis$

and $\bigwedge E\ e\ s_0\ r\ s_1.\ x = E \Longrightarrow y = throw\ e \Longrightarrow$
$z = s_0 \Longrightarrow u = Throw\ r \Longrightarrow v = s_1 \Longrightarrow P,E \vdash \langle e,s_0\rangle \Rightarrow' \langle ref\ r,s_1\rangle \Longrightarrow thesis$

and $\bigwedge E\ e\ s_0\ s_1.\ x = E \Longrightarrow y = throw\ e \Longrightarrow z = s_0 \Longrightarrow$
$u = Throw\ (addr\text{-}of\text{-}sys\text{-}xcpt\ NullPointer,\ [NullPointer]) \Longrightarrow$
$v = s_1 \Longrightarrow P,E \vdash \langle e,s_0\rangle \Rightarrow' \langle null,s_1\rangle \Longrightarrow thesis$

and $\bigwedge E\ e\ s_0\ e'\ s_1.\ x = E \Longrightarrow y = throw\ e \Longrightarrow$
$z = s_0 \Longrightarrow u = throw\ e' \Longrightarrow v = s_1 \Longrightarrow P,E \vdash \langle e,s_0\rangle \Rightarrow' \langle throw\ e',s_1\rangle \Longrightarrow thesis$

shows *thesis*

⟨*proof*⟩

lemmas [*code-pred-intro*] = *New′ NewFail′ StaticUpCast′*
declare *StaticDownCast′-new*[*code-pred-intro StaticDownCast′*]
lemmas [*code-pred-intro*] = *StaticCastNull′*
declare *StaticCastFail′-new*[*code-pred-intro StaticCastFail′*]
lemmas [*code-pred-intro*] = *StaticCastThrow′ StaticUpDynCast′*
declare
StaticDownDynCast′-new[*code-pred-intro StaticDownDynCast′*]
DynCast′[*code-pred-intro DynCast′*]
lemmas [*code-pred-intro*] = *DynCastNull′*
declare *DynCastFail′*[*code-pred-intro DynCastFail′*]
lemmas [*code-pred-intro*] = *DynCastThrow′ Val′ BinOp′ BinOpThrow1′*
declare *BinOpThrow2′*[*code-pred-intro BinOpThrow2′*]
lemmas [*code-pred-intro*] = *Var′ LAss′ LAssThrow′*
declare *FAcc′-new*[*code-pred-intro FAcc′*]
lemmas [*code-pred-intro*] = *FAccNull′ FAccThrow′*
declare *FAss′-new*[*code-pred-intro FAss′*]
lemmas [*code-pred-intro*] = *FAssNull′ FAssThrow1′*
declare *FAssThrow2′*[*code-pred-intro FAssThrow2′*]
lemmas [*code-pred-intro*] = *CallObjThrow′*
declare
CallParamsThrow′-new[*code-pred-intro CallParamsThrow′*]
Call′-new[*code-pred-intro Call′*]
StaticCall′-new[*code-pred-intro StaticCall′*]
CallNull′-new[*code-pred-intro CallNull′*]
lemmas [*code-pred-intro*] = *Block′ Seq′*
declare *SeqThrow′*[*code-pred-intro SeqThrow′*]
lemmas [*code-pred-intro*] = *CondT′*
declare
CondF′[*code-pred-intro CondF′*]
CondThrow′[*code-pred-intro CondThrow′*]
lemmas [*code-pred-intro*] = *WhileF′ WhileT′*
declare
WhileCondThrow′[*code-pred-intro WhileCondThrow′*]
WhileBodyThrow′[*code-pred-intro WhileBodyThrow′*]
lemmas [*code-pred-intro*] = *Throw′*
declare *ThrowNull′*[*code-pred-intro ThrowNull′*]
lemmas [*code-pred-intro*] = *ThrowThrow′*
lemmas [*code-pred-intro*] = *Nil′ Cons′ ConsThrow′*

code-pred
(*modes*: *eval′*: $i \Rightarrow i \Rightarrow i \Rightarrow i \Rightarrow o \Rightarrow o \Rightarrow$ *bool as big-step*
and *evals′*: $i \Rightarrow i \Rightarrow i \Rightarrow i \Rightarrow o \Rightarrow o \Rightarrow$ *bool as big-steps*)
eval′
⟨*proof*⟩

29.3 Examples

declare [[*values-timeout = 180*]]

values [*expected* {*Val (Intg 5)*}]
{*fst* (*e′*, *s′*) | *e′ s′*.
[],*Map.empty* ⊢ ⟨{*''V''*:*Integer*; *''V''* := *Val*(*Intg 5*);; *Var ''V''*},(*Map.empty*,*Map.empty*)⟩
⇒′ ⟨*e′*, *s′*⟩}

values [*expected* {*Val (Intg 11)*}]
{*fst* (*e′*, *s′*) | *e′ s′*.
[],*Map.empty* ⊢ ⟨(*Val*(*Intg 5*)) «*Add*» (*Val*(*Intg 6*)),(*Map.empty*,*Map.empty*)⟩
⇒′ ⟨*e′*, *s′*⟩}

values [*expected* {*Val (Intg 83)*}]
{*fst* (*e′*, *s′*) | *e′ s′*.
[],[*''V''*↦*Integer*] ⊢ ⟨(*Var ''V''*) «*Add*» (*Val*(*Intg 6*)),
(*Map.empty*,[*''V''*↦*Intg 77*])⟩ ⇒′ ⟨*e′*, *s′*⟩}

values [*expected* {*Some (Intg 6)*}]
{*lcl′* (*snd* (*e′*, *s′*)) *''V''* | *e′ s′*.
[],[*''V''*↦*Integer*] ⊢ ⟨*''V''* := *Val*(*Intg 6*),(*Map.empty*,*Map.empty*)⟩ ⇒′ ⟨*e′*, *s′*⟩}

values [*expected* {*Some (Intg 12)*}]
{*lcl′* (*snd* (*e′*, *s′*)) *''mult''* | *e′ s′*.
[],[*''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*)))),
(*Map.empty*,*Map.empty*)⟩ ⇒′ ⟨*e′*, *s′*⟩}

values [*expected* {*Val (Intg 30)*}]
{*fst* (*e′*, *s′*) | *e′ s′*.
[],[*''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''*),
(*Map.empty*,*Map.empty*)⟩ ⇒′ ⟨*e′*,*s′*⟩}

progOverrider examples

definition
classBottom :: *cdecl* **where**
classBottom = (*''Bottom''*, [*Repeats ''Left''*, *Repeats ''Right''*],
[(*''x''*,*Integer*)],[])

definition
classLeft :: *cdecl* **where**
classLeft = (*''Left''*, [*Repeats ''Top''*],[],[(*''f''*, [*Class ''Top''*, *Integer*],*Integer*,

$[''V'',''W''], Var\ this \cdot\ ''x''\ \{[''Left'',''Top'']\}\ \langle\langle Add \rangle\rangle\ Val\ (Intg\ 5))])$

definition
$classRight :: cdecl$ **where**
$classRight = (''Right'', [Shares\ ''Right2''],[],$
$[(''f'', [Class\ ''Top'', Integer], Integer,[''V'',''W''], Var\ this \cdot\ ''x''\ \{[''Right2'',''Top'']\}$
$\langle\langle Add \rangle\rangle\ Val\ (Intg\ 7)),(''g'',[],Class\ ''Left'',[],new\ ''Left'')])$

definition
$classRight2 :: cdecl$ **where**
$classRight2 = (''Right2'', [Repeats\ ''Top''],[],$
$[(''f'', [Class\ ''Top'', Integer], Integer,[''V'',''W''], Var\ this \cdot\ ''x''\ \{[''Right2'',''Top'']\}$
$\langle\langle Add \rangle\rangle\ Val\ (Intg\ 9)),(''g'',[],Class\ ''Top'',[],new\ ''Top'')])$

definition
$classTop :: cdecl$ **where**
$classTop = (''Top'', [], [(''x'',Integer)],[])$

definition
$progOverrider :: cdecl\ list$ **where**
$progOverrider = [classBottom, classLeft, classRight, classRight2, classTop]$

values $[expected\ \{Val(Ref(0,[''Bottom'',''Left'']))\}]$ — dynCastSide
$\{fst\ (e', s')\ |\ e'\ s'.$
$progOverrider,[''V''\mapsto Class\ ''Right''] \vdash$
$\langle ''V'' := new\ ''Bottom''\ ;;\ Cast\ ''Left''\ (Var\ ''V''),(Map.empty,Map.empty)\rangle$
$\Rightarrow' \langle e', s'\rangle\}$

values $[expected\ \{Val(Ref(0,[''Right'']))\}]$ — dynCastViaSh
$\{fst\ (e', s')\ |\ e'\ s'.$
$progOverrider,[''V''\mapsto Class\ ''Right2''] \vdash$
$\langle ''V'' := new\ ''Right''\ ;;\ Cast\ ''Right''\ (Var\ ''V''),(Map.empty,Map.empty)\rangle \Rightarrow'$
$\langle e', s'\rangle\}$

values $[expected\ \{Val\ (Intg\ 42)\}]$ — block
$\{fst\ (e', s')\ |\ e'\ s'.$
$progOverrider,[''V''\mapsto Integer]$
$\vdash \langle ''V'' := Val(Intg\ 42)\ ;;\ \{''V'':Class\ ''Left''; ''V'' := new\ ''Bottom''\}\ ;;\ Var$
$''V'',$
$(Map.empty,Map.empty)\rangle \Rightarrow' \langle e', s'\rangle\}$

values $[expected\ \{Val\ (Intg\ 8)\}]$ — staticCall
$\{fst\ (e', s')\ |\ e'\ s'.$
$progOverrider,[''V''\mapsto Class\ ''Right'',''W''\mapsto Class\ ''Bottom'']$
$\vdash \langle ''V'' := new\ ''Bottom''\ ;;\ ''W'' := new\ ''Bottom''\ ;;$
$((Cast\ ''Left''\ (Var\ ''W''))\cdot ''x''\{[''Left'',''Top'']\} := Val(Intg\ 3));;$
$(Var\ ''W''\cdot(''Left''::)''f''([Var\ ''V'', Val(Intg\ 2)])),(Map.empty,Map.empty)\rangle$
$\Rightarrow' \langle e', s'\rangle\}$

values [*expected* {*Val* (*Intg 12*)}] — call
{*fst* (*e′*, *s′*) | *e′ s′*.
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*)])),
(*Map.empty*,*Map.empty*)⟩ ⇒′ ⟨*e′*, *s′*⟩}

values [*expected* {*Val*(*Intg 13*)}] — callOverrider
{*fst* (*e′*, *s′*) | *e′ s′*.
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*)]),
(*Map.empty*,*Map.empty*)⟩ ⇒′ ⟨*e′*, *s′*⟩}

values [*expected* {*Val*(*Ref*(*1*,[*''Left''*,*''Top''*]))}] — callClass
{*fst* (*e′*, *s′*) | *e′ s′*.
progOverrider,[*''V''*↦*Class ''Right2''*]
⊢ ⟨*''V''* := *new ''Right''* ;; *Var ''V''*•*''g''*([]),(*Map.empty*,*Map.empty*)⟩ ⇒′ ⟨*e′*, *s′*⟩}

values [*expected* {*Val*(*Intg 42*)}] — fieldAss
{*fst* (*e′*, *s′*) | *e′ s′*.
progOverrider,[*''V''*↦*Class ''Right2''*]
⊢ ⟨*''V''* := *new ''Right''* ;;
(*Var ''V''*•*''x''*{[*''Right2''*,*''Top''*]} := (*Val*(*Intg 42*))) ;;
(*Var ''V''*•*''x''*{[*''Right2''*,*''Top''*]}),(*Map.empty*,*Map.empty*)⟩ ⇒′ ⟨*e′*, *s′*⟩}

typing rules

values [*expected* {*Class ''Bottom''*}] — typeNew
{*T*. *progOverrider*,*Map.empty* ⊢ *new ''Bottom''* :: *T*}

values [*expected* {*Class ''Left''*}] — typeDynCast
{*T*. *progOverrider*,*Map.empty* ⊢ *Cast ''Left''* (*new ''Bottom''*) :: *T*}

values [*expected* {*Class ''Left''*}] — typeStaticCast
{*T*. *progOverrider*,*Map.empty* ⊢ (|*''Left''*|) (*new ''Bottom''*) :: *T*}

values [*expected* {*Integer*}] — typeVal
{*T*. [],*Map.empty* ⊢ *Val*(*Intg 17*) :: *T*}

values [*expected* {*Integer*}] — typeVar
{*T*. [],[*''V''* ↦ *Integer*] ⊢ *Var ''V''* :: *T*}

values [*expected* {*Boolean*}] — typeBinOp
{*T*. [],*Map.empty* ⊢ (*Val*(*Intg 5*)) «*Eq*» (*Val*(*Intg 6*)) :: *T*}

values [*expected* {*Class ''Top''*}] — typeLAss

$\{T.\ progOverrider,[''V'' \mapsto Class\ ''Top'']\ \vdash\ ''V'' := (new\ ''Left'')\ ::\ T\}$

values [*expected* {*Integer*}] — typeFAcc
$\{T.\ progOverrider,Map.empty \vdash (new\ ''Right'')\cdot''x''\{[''Right2'',''Top'']\}\ ::\ T\}$

values [*expected* {*Integer*}] — typeFAss
$\{T.\ progOverrider,Map.empty \vdash (new\ ''Right'')\cdot''x''\{[''Right2'',''Top'']\}\ ::\ T\}$

values [*expected* {*Integer*}] — typeStaticCall
$\{T.\ progOverrider,[''V''\mapsto Class\ ''Left'']$
$\vdash\ ''V'' := new\ ''Left''\ ;;\ Var\ ''V''\cdot(''Left''::)''f''([new\ ''Top'',\ Val(Intg\ 13)])$
$::\ T\}$

values [*expected* {*Class ''Top''*}] — typeCall
$\{T.\ progOverrider,[''V''\mapsto Class\ ''Right2'']$
$\vdash\ ''V'' := new\ ''Right''\ ;;\ Var\ ''V''\cdot''g''([])\ ::\ T\}$

values [*expected* {*Class ''Top''*}] — typeBlock
$\{T.\ progOverrider,Map.empty \vdash \{''V'':Class\ ''Top''; ''V'' := new\ ''Left''\}\ ::\ T\}$

values [*expected* {*Integer*}] — typeCond
$\{T.\ [],Map.empty \vdash if\ (true)\ Val(Intg\ 6)\ else\ Val(Intg\ 9)\ ::\ T\}$

values [*expected* {*Void*}] — typeWhile
$\{T.\ [],Map.empty \vdash while\ (false)\ Val(Intg\ 17)\ ::\ T\}$

values [*expected* {*Void*}] — typeThrow
$\{T.\ progOverrider,Map.empty \vdash throw\ (new\ ''Bottom'')\ ::\ T\}$

values [*expected* {*Integer*}] — typeBig
$\{T.\ progOverrider,[''V''\mapsto Class\ ''Right2'',''W''\mapsto Class\ ''Left'']$
$\vdash\ ''V'' := new\ ''Right''\ ;;\ ''W'' := new\ ''Left''\ ;;$
$(Var\ ''V''\cdot''f''([Var\ ''W'',\ Val(Intg\ 7)]))\ «Add»\ (Var\ ''W''\cdot''f''([Var\ ''V'',$
$Val(Intg\ 13)]))$
$::\ T\}$

progDiamond examples

definition
classDiamondBottom :: *cdecl* **where**
classDiamondBottom = (*''Bottom''*, [*Repeats ''Left''*, *Repeats ''Right''*],[(*''x''*,*Integer*)],
[(*''g''*, [],*Integer*, [],*Var this* · *''x''* {[*''Bottom''*]} «*Add*» *Val* (*Intg 5*))])

definition
classDiamondLeft :: *cdecl* **where**
classDiamondLeft = (*''Left''*, [*Repeats ''TopRep''*,*Shares ''TopSh''*],[],[])

definition
classDiamondRight :: *cdecl* **where**
classDiamondRight = (*''Right''*, [*Repeats ''TopRep''*,*Shares ''TopSh''*],[],

[(''f'', [Integer], Boolean,[''i''], Var ''i'' «Eq» Val (Intg 7))])

definition
classDiamondTopRep :: cdecl **where**
classDiamondTopRep = (''TopRep'', [], [(''x'',Integer)],
[(''g'', [],Integer, [], Var this · ''x'' {[''TopRep'']} «Add» Val (Intg 10))])

definition
classDiamondTopSh :: cdecl **where**
classDiamondTopSh = (''TopSh'', [], [],
[(''f'', [Integer], Boolean,[''i''], Var ''i'' «Eq» Val (Intg 3))])

definition
progDiamond :: cdecl list **where**
progDiamond = [classDiamondBottom, classDiamondLeft, classDiamondRight, classDiamondTopRep, classDiamondTopSh]

values [expected {Val(Ref(0,[''Bottom'',''Left'']))}] — cast1
{fst (e', s') | e' s'.
progDiamond,[''V''↦Class ''Left''] ⊢ ⟨''V'' := new ''Bottom'',
(Map.empty,Map.empty)⟩ ⇒' ⟨e', s'⟩}

values [expected {Val(Ref(0,[''TopSh'']))}] — cast2
{fst (e', s') | e' s'.
progDiamond,[''V''↦Class ''TopSh''] ⊢ ⟨''V'' := new ''Bottom'',
(Map.empty,Map.empty)⟩ ⇒' ⟨e', s'⟩}

values [expected {}] — typeCast3 not typeable
{T. progDiamond,[''V''↦Class ''TopRep''] ⊢ ''V'' := new ''Bottom'' :: T}

values [expected {
Val(Ref(0,[''Bottom'', ''Left'', ''TopRep'']),
Val(Ref(0,[''Bottom'', ''Right'', ''TopRep'']))
}] — cast3
{fst (e', s') | e' s'.
progDiamond,[''V''↦Class ''TopRep''] ⊢ ⟨''V'' := new ''Bottom'',
(Map.empty,Map.empty)⟩ ⇒' ⟨e', s'⟩}

values [expected {Val(Intg 17)}] — fieldAss
{fst (e', s') | e' s'.
progDiamond,[''V''↦Class ''Bottom'']
⊢ ⟨''V'' := new ''Bottom'' ;;
((Var ''V'')·''x''{[''Bottom'']} := (Val(Intg 17))) ;;
((Var ''V'')·''x''{[''Bottom'']}),(Map.empty,Map.empty)⟩ ⇒' ⟨e',s'⟩}

values [expected {Val Null}] — dynCastNull
{fst (e', s') | e' s'.
progDiamond,Map.empty ⊢ ⟨Cast ''Right'' null,(Map.empty,Map.empty)⟩ ⇒' ⟨e',s'⟩}

values [*expected* {*Val* (*Ref*(*0*, [*''Right''*]))}] — dynCastViaSh
{*fst* (*e'*, *s'*) | *e' s'*.
progDiamond,[*''V''*↦*Class ''TopSh''*]
⊢ ⟨*''V''* := *new ''Right''* ;; *Cast ''Right''* (*Var ''V''*),(*Map.empty*,*Map.empty*)⟩
⇒′ ⟨*e'*,*s'*⟩}

values [*expected* {*Val Null*}] — dynCastFail
{*fst* (*e'*, *s'*) | *e' s'*.
progDiamond,[*''V''*↦*Class ''TopRep''*]
⊢ ⟨*''V''* := *new ''Right''* ;; *Cast ''Bottom''* (*Var ''V''*),(*Map.empty*,*Map.empty*)⟩
⇒′ ⟨*e'*,*s'*⟩}

values [*expected* {*Val* (*Ref*(*0*, [*''Bottom''*, *''Left''*]))}] — dynCastSide
{*fst* (*e'*, *s'*) | *e' s'*.
progDiamond,[*''V''*↦*Class ''Right''*]
⊢ ⟨*''V''* := *new ''Bottom''* ;; *Cast ''Left''* (*Var ''V''*),(*Map.empty*,*Map.empty*)⟩
⇒′ ⟨*e'*,*s'*⟩}

failing g++ example

definition
classD :: *cdecl* **where**
classD = (*''D''*, [*Shares ''A''*, *Shares ''B''*, *Repeats ''C''*],[],[])

definition
classC :: *cdecl* **where**
classC = (*''C''*, [*Shares ''A''*, *Shares ''B''*],[],
[(*''f''*,[],*Integer*,[],*Val*(*Intg 42*))])

definition
classB :: *cdecl* **where**
classB = (*''B''*, [],[],
[(*''f''*,[],*Integer*,[],*Val*(*Intg 17*))])

definition
classA :: *cdecl* **where**
classA = (*''A''*, [],[],
[(*''f''*,[],*Integer*,[],*Val*(*Intg 13*))])

definition
ProgFailing :: *cdecl list* **where**
ProgFailing = [*classA*,*classB*,*classC*,*classD*]

values [*expected* {*Val* (*Intg 42*)}] — callFailGplusplus
{*fst* (*e'*, *s'*) | *e' s'*.
ProgFailing,*Map.empty*
⊢ ⟨{*''V''*:*Class ''D''*; *''V''* := *new ''D''*;; *Var ''V''*·*''f''*([])},
(*Map.empty*,*Map.empty*)⟩ ⇒′ ⟨*e'*, *s'*⟩}

end
theory *CoreC++*
imports *Determinism Annotate Execute*
begin

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.