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/

June 16, 2019

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*]:
$((n::nat) + max\ i\ j \leq m) = (n + i \leq m \wedge n + j \leq m)$
by *arith*

lemma *Suc-add-max-le*[*simp*]:
$(Suc(n + max\ i\ j) \leq m) = (Suc(n + i) \leq m \wedge Suc(n + j) \leq m)$
by *arith*

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

lemma *butlast-tail*:
$butlast\ (Xs@[X, Y]) = Xs@[X]$
by (*induct Xs*) *auto*

lemma *butlast-noteq*:$Cs \neq [] \Longrightarrow butlast\ Cs \neq Cs$
by(*induct Cs*)*simp-all*

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

apply (*subgoal-tac* $[hd\ Cs] @ tl\ Cs = Cs' @ tl\ Cs$)
apply *fast*
apply *simp*
done

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

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

apply −
apply (*simp add*:*append-eq-append-conv2*)
apply (*auto simp*:*in-set-conv-decomp*)
apply (*subgoal-tac hd* $(us @ C'\#Ds') = C'$)
apply (*case-tac us*)
apply *simp*
apply *fastforce*
apply *simp*
apply (*subgoal-tac hd* $(us @ C'\#Ds') = C'$)
apply (*case-tac us*)
apply *simp*
apply *fastforce*

```
  apply simp
 apply (subgoal-tac hd (us @ C'#Cs') = C')
  apply (case-tac us)
   apply simp
  apply fastforce
 apply (subgoal-tac hd(C'#Ds') = C')
  apply simp
 apply (simp (no-asm))
apply (subgoal-tac hd (us @ C'#Cs') = C')
 apply (case-tac us)
  apply simp
 apply fastforce
apply (subgoal-tac hd(C'#Ds') = C')
 apply simp
apply (simp (no-asm))
done


definition pick :: 'a set ⇒ 'a where
  pick A ≡ SOME x. x ∈ A


lemma pick-is-element:x ∈ A ⟹ pick A ∈ A
by (unfold pick-def,rule-tac x=x in someI)


definition set2list :: 'a set ⇒ 'a list where
  set2list A ≡ fst (while (λ(Es,S). S ≠ {})
                          (λ(Es,S). let x = pick S in (x#Es,S−{x}))
                          ([],A) )

lemma card-pick:⟦finite A; A ≠ {}⟧ ⟹ Suc(card(A−{pick(A)})) = card A
by (drule card-Suc-Diff1,auto dest!:pick-is-element simp:ex-in-conv)


lemma set2list-prop:⟦finite A; A ≠ {}⟧ ⟹
  ∃xs. while (λ(Es,S). S ≠ {})
             (λ(Es,S). let x = pick S in (x#Es,S−{x}))
             ([],A) = (xs,{}) ∧ (set xs ∪ {} = A)

apply(rule-tac P=(λxs. (set(fst xs) ∪ snd xs = A)) and
              r=measure (card o snd)  in while-rule)
apply(auto dest:pick-is-element)
apply(auto dest:card-pick simp:ex-in-conv measure-def inv-image-def)
done


lemma set2list-correct:⟦finite A; A ≠ {}; set2list A = xs⟧ ⟹ set xs = A
by (auto dest:set2list-prop simp:set2list-def)
```

1.1 *distinct-fst*

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

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

apply (*unfold distinct-fst-def*)
apply (*simp* (*no-asm*))
done

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

apply (*unfold distinct-fst-def*)
apply (*auto simp*:*image-def*)
done

lemma *map-of-SomeI*:
$[\![$ *distinct-fst kxs*; (*k*,*x*) $\in$ *set kxs* $]\!] \Longrightarrow$ *map-of kxs k* = *Some x*
by (*induct kxs*) (*auto simp*:*fun-upd-apply*)

1.2 Using *list-all2* for relations

definition *fun-of* :: $('a \times 'b)$ *set* $\Rightarrow$ $'a \Rightarrow 'b \Rightarrow$ *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*)
by *simp*

lemma *rel-list-all2-Cons1*:
list-all2 (*fun-of S*) (*x*#*xs*) *ys* =
($\exists$ *z zs*. *ys* = *z*#*zs* $\wedge$ (*x*,*z*) $\in$ *S* $\wedge$ *list-all2* (*fun-of S*) *xs zs*)
by (*cases ys*) *auto*

lemma *rel-list-all2-Cons2*:
list-all2 (*fun-of S*) *xs* (*y*#*ys*) =
($\exists$ *z zs*. *xs* = *z*#*zs* $\wedge$ (*z*,*y*) $\in$ *S* $\wedge$ *list-all2* (*fun-of S*) *zs ys*)
by (*cases xs*) *auto*

lemma *rel-list-all2-refl*:
($\bigwedge$*x*. (*x*,*x*) $\in$ *S*) $\Longrightarrow$ *list-all2* (*fun-of S*) *xs xs*

by (*simp add*: *list-all2-refl*)

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*
by (*rule list-all2-antisym*) *auto*

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*
by (*rule list-all2-trans*) *auto*

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*])
by (*simp add*: *list-all2-update-cong*)

lemma *rel-list-all2-nthD*:
⟦ *list-all2* (*fun-of S*) *xs ys*; *p* < *size xs* ⟧ ⟹ (*xs*!*p*,*ys*!*p*) ∈ *S*
by (*drule list-all2-nthD*) *auto*

lemma *rel-list-all2I*:
⟦ *length a* = *length b*; ⋀*n*. *n* < *length a* ⟹ (*a*!*n*,*b*!*n*) ∈ *S* ⟧ ⟹ *list-all2* (*fun-of S*) *a b*
by (*erule list-all2-all-nthI*) *simp*

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*
by(*simp add*:*is-refT-def*)

lemma [*iff*]: *is-refT*(*Class C*)
by(*simp add*:*is-refT-def*)

lemma *refTE*:
⟦*is-refT T*; *T* = *NT* ⟹ *Q*; ⋀*C*. *T* = *Class C* ⟹ *Q* ⟧ ⟹ *Q*
by (*auto simp add*: *is-refT-def*)

lemma *not-refTE*:
⟦ ¬*is-refT T*; *T* = *Void* ∨ *T* = *Boolean* ∨ *T* = *Integer* ⟹ *Q* ⟧ ⟹ *Q*
by (*cases T*, *auto simp add*: *is-refT-def*)

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*
by(*cases T*)*simp-all*

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*)
by(*induct v*) *auto*

lemma [*simp*]: (*typeof v* = *Some Integer*) = (∃ *i*. *v* = *Intg i*)
by(*cases v*) *auto*

lemma [*simp*]: (*typeof v* = *Some NT*) = (*v* = *Null*)
 by(*cases v*) *auto*

lemma [*simp*]: (*typeof v* = *Some Void*) = (*v* = *Unit*)
 by(*cases v*) *auto*

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*$_2$))
apply(*cases v*$_1$)

apply *auto*
apply(*cases* v_2)
apply *auto*
done

lemma *binop-not-ref*[*simp*]:
  binop(*bop*,v_1,v_2) = *Some* (*Ref r*) $\Longrightarrow$ *False*
by(*cases bop*)*auto*

4.2 Free Variables

primrec
  fv :: *expr* $\Rightarrow$ *vname set*
  and *fvs* :: *expr list* $\Rightarrow$ *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
by (*induct* es_1 *type*:*list*) *auto*

lemma [*simp*]: *fvs*(*map Val vs*) = {}
by (*induct vs*) *auto*

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*
by (*induct rule*: *list.induct*, *auto*)

lemma *not-getbase-shares*:
D ∉ *set* (*map getbase xs*) ⟹ *Shares D* ∉ *set xs*
by (*induct rule*: *list.induct*, *auto*)

lemma *RepBaseclass-isBaseclass*:
⟦*class P C* = *Some*(*Bs*,*fs*,*ms*); *Repeats D* ∈ *set Bs*⟧
⟹ *D* ∈ *baseClasses Bs*

by (*simp add*:*baseClasses-def*, *induct rule*: *list.induct*,
auto simp:*not-getbase-repeats*)

lemma *ShBaseclass-isBaseclass*:
⟦*class P C* = *Some*(*Bs*,*fs*,*ms*); *Shares D* ∈ *set Bs*⟧
⟹ *D* ∈ *baseClasses Bs*
by (*simp add*:*baseClasses-def*, *induct rule*: *list.induct*,
auto simp:*not-getbase-shares*)

lemma *base-repeats-or-shares*:
⟦*B* ∈ *set Bs*; *D* = *getbase B*⟧
⟹ *Repeats D* ∈ *set Bs* ∨ *Shares D* ∈ *set Bs*
by(*induct B rule*:*base.induct*) *simp*+

lemma *baseClasses-repeats-or-shares*:
D ∈ *baseClasses Bs* ⟹ *Repeats D* ∈ *set Bs* ∨ *Shares D* ∈ *set Bs*
by (*auto elim*!:*bexE base-repeats-or-shares*
simp add:*baseClasses-def image-def*)

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

apply (*unfold is-class-def class-def*)
apply (*fold dom-def*)
apply (*rule finite-dom-map-of*)
done

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

apply (*unfold is-class-def class-def baseClasses-def*)
apply *clarsimp*
done

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*
by(*simp add*:*is-type-def*)

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

lemma *typeof-lit-is-type*:
typeof v = *Some T* ⟹ *is-type P T*
by (*induct v*) (*auto*)

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*)
by(*auto elim*: *subclsR.cases*)

lemma *subclsSD*:
$P \vdash C \prec_S D \Longrightarrow \exists fs\ ms\ Bs.\ (class\ P\ C = Some\ (Bs,fs,ms)) \wedge (Shares(D) \in set\ Bs)$
by(*auto elim*: *subclsS.cases*)

lemma *subcls1D*:
$P \vdash C \prec^1 D \Longrightarrow \exists fs\ ms\ Bs.\ (class\ P\ C = Some\ (Bs,fs,ms)) \wedge (D \in baseClasses\ Bs)$
by(*auto elim*: *subcls1.cases*)

lemma *subclsR-subcls1*:
$P \vdash C \prec_R D \Longrightarrow P \vdash C \prec^1 D$
by (*auto elim*!:*subclsR.cases intro*:*subcls1I simp*:*RepBaseclass-isBaseclass*)

lemma *subclsS-subcls1*:
$P \vdash C \prec_S D \Longrightarrow P \vdash C \prec^1 D$
by (*auto elim*!:*subclsS.cases intro*:*subcls1I simp*:*ShBaseclass-isBaseclass*)

lemma *subcls1-subclsR-or-subclsS*:
$P \vdash C \prec^1 D \Longrightarrow P \vdash C \prec_R D \vee P \vdash C \prec_S D$
by (*auto dest*!:*subcls1D intro*:*subclsRI*
dest:*baseClasses-repeats-or-shares subclsSI*)

lemma *finite-subcls1*: *finite* (*subcls1 P*)

apply(*subgoal-tac subcls1 P* = (*SIGMA C*: {*C*. *is-class P C*} .
{*D*. *D* ∈ *baseClasses* (*fst*(*the*(*class P C*)))}))
prefer *2*
apply(*fastforce simp*:*is-class-def dest*: *subcls1D elim*: *subcls1I*)
apply *simp*
apply(*rule finite-SigmaI* [*OF finite-is-class*])
apply(*rule-tac B* = *baseClasses* (*fst* (*the* (*class P C*))) **in** *finite-subset*)
apply (*auto intro*:*finite-baseClasses simp*:*is-class-def*)
done

lemma *finite-subclsR*: *finite* (*subclsR P*)
by(*rule-tac B* = *subcls1 P* **in** *finite-subset*,
auto simp:*subclsR-subcls1 finite-subcls1*)

lemma *finite-subclsS*: *finite* (*subclsS P*)
by(*rule-tac B* = *subcls1 P* **in** *finite-subset*,
auto simp:*subclsS-subcls1 finite-subcls1*)

lemma *subcls1-class*:
$P \vdash C \prec^1 D \Longrightarrow is\text{-}class\ P\ C$
by (*auto dest*:*subcls1D simp*:*is-class-def*)

lemma *subcls-is-class*:
$[\![P \vdash D \preceq^* C;\ is\text{-}class\ P\ C]\!] \Longrightarrow is\text{-}class\ P\ D$
by (*induct rule*:*rtrancl-induct*,*auto dest*:*subcls1-class*)

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

lemma *ldc-tuple* [*simp*]: *ldc* (*C*,*Cs*) = *last Cs*
by(*simp add*:*ldc-def*)

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 −
  obtain *Cs'* **where** *Cs'*:*Cs'* = *rev Cs* **by** *simp*
  hence *rev*:*Cs'*@[*C''*] = *rev Cs*@[*C''*] **by** *simp*
  from *this* **obtain** *D Ds* **where** *DDs*:*Cs'*@[*C''*] = *D*#*Ds* **by** (*cases Cs'*) *auto*

with *1 rev* **have** *subo*:*is-subobj P* ((*C*,*C'*#*D*#*Ds*)) **by** *simp*
from *DDs* **have** *butlast* (*C'*#*D*#*Ds*) = *C'*#*Cs'* **by** (*cases Cs'*) *auto*
with *subo* **have** *is-subobj P* ((*C*,*C'*#*Cs'*)) **by** *simp*
with *Cs'* **show** *?thesis* **by** *simp*
qed

lemma *subobj-aux*:
assumes *1*:*is-subobj P* ((*C*,*C'*#*Cs*@[*C''*]))
shows *is-subobj P* ((*C*,*C'*#*Cs*))
proof −
from *1* **obtain** *Cs'* **where** *Cs'*:*Cs'* = *rev Cs* **by** *simp*
with *1* **have** *is-subobj P* ((*C*,*C'*#*rev Cs'*@[*C''*])) **by** *simp*
hence *is-subobj P* ((*C*,*C'*#*rev Cs'*)) **by** (*rule subobj-aux-rev*)
with *Cs'* **show** *?thesis* **by** *simp*
qed

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

proof −
obtain *C' Cs'* **where** *R*:*R* = (*C'*,*Cs'*) **by**(*cases R*) *auto*
with *1* **have** *ne*:*Cs'* ≠ [] **by** (*cases Cs'*) *auto*
from *this* **obtain** *C'' Cs''* **where** *C''Cs''*:*Cs'* = *C''*#*Cs''* **by** (*cases Cs'*) *auto*
from *this* **obtain** *Ds* **where** *Ds* = *rev Cs''* **by** *simp*
with *1 R C''Cs''* **have** *subo1*:*is-subobj P* ((*C'*,*C''*#*rev Ds*)) **by** *simp*
with *R* **show** *?thesis*
by (*induct Ds*,*auto simp*:*mdc-def split*:*if-split-asm dest*:*subobj-aux*,
auto elim:*converse-rtranclE dest*!:*subclsS-subcls1 elim*:*subcls1-class*)
qed

lemma *isSubobjs-subclsR-rev*:
assumes *1*:*is-subobj P* ((*C*,*Cs*@[*D*,*D'*]@(*rev Cs'*)))
shows *P* ⊢ *D* ≺$_R$ *D'*
using *1*
proof (*induct Cs'*)
case *Nil*
from *this* **obtain** *Cs' X Y Xs* **where** *Cs'1*:*Cs'* = *Cs*@[*D*,*D'*]
and *X* = *hd*(*Cs*@[*D*,*D'*]) **and** *Y* = *hd*(*tl*(*Cs*@[*D*,*D'*]))
and *Xs* = *tl*(*tl*(*Cs*@[*D*,*D'*])) **by** *simp*
hence *Cs'2*:*Cs'* = *X*#*Y*#*Xs* **by** (*cases Cs*) *auto*
from *Cs'1* **have** *last*:*last Cs'* = *D'* **by** *simp*

```
  from Cs'1 have butlast:last(butlast Cs') = D by (simp add:butlast-tail)
  from Nil Cs'1 Cs'2 have is-subobj P ((C,X#Y#Xs)) by simp
  with last butlast Cs'2 show ?case by simp
next
  case (Cons C'' Cs'')
  have IH:is-subobj P ( (C, Cs @ [D, D'] @ rev Cs'')) ⟹ P ⊢ D ≺R D' by fact
  from Cons obtain Cs' X Y Xs where Cs'1:Cs' = Cs@[D,D']@(rev (C''#Cs''))

    and X = hd(Cs@[D,D']@(rev (C''#Cs'')))
    and Y = hd(tl(Cs@[D,D']@(rev (C''#Cs''))))
    and Xs =  tl(tl(Cs@[D,D']@(rev (C''#Cs'')))) by simp
  hence Cs'2:Cs' = X#Y#Xs by (cases Cs) auto
  from Cons Cs'1 Cs'2 have is-subobj P ((C,X#Y#Xs)) by simp
  hence sub:is-subobj P ((C,butlast (X#Y#Xs))) by simp
  from Cs'1 obtain E Es where Cs'3:Cs' = Es@[E] by (cases Cs') auto
  with Cs'1 have butlast:Es = Cs@[D,D']@(rev Cs'') by simp
  from Cs'3 have butlast Cs' = Es by simp
  with butlast have butlast Cs' = Cs@[D,D']@(rev Cs'') by simp
  with Cs'2 sub have is-subobj P ((C,Cs@[D,D']@(rev Cs'')))
    by simp
  with IH show ?case by simp
qed



lemma isSubobjs-subclsR:
assumes 1:is-subobj P ((C,Cs@[D,D']@Cs'))
shows P ⊢ D ≺R D'

proof −
  from 1 obtain Cs'' where Cs'' = rev Cs' by simp
  with 1 have is-subobj P ((C,Cs@[D,D']@(rev Cs''))) by simp
  thus ?thesis by (rule isSubobjs-subclsR-rev)
qed



lemma mdc-leq-ldc-aux:
assumes 1:is-subobj P ((C,C'#rev Cs'))
shows P ⊢ C ≼* last (C'#rev Cs')
using 1
proof (induct Cs')
  case Nil
  from 1 have is-class P C
    by (drule-tac R=(C,C'#rev Cs') in isSubobj-isClass, simp add:mdc-def)
  with Nil show ?case
    proof (cases C=C')
      case True
```

thus *?thesis* **by** *simp*
next
case *False*
with *Nil* **show** *?thesis*
by (*auto dest*!:*subclsS-subcls1*)
qed
next
case (*Cons C'' Cs''*)
have *IH*:*is-subobj P* ((*C*, *C'* # *rev Cs''*)) ⟹ *P* ⊢ *C* ≼* *last* (*C'* # *rev Cs''*)
and *subo*:*is-subobj P* ((*C*, *C'* # *rev* (*C''* # *Cs''*))) **by** *fact+*
hence *is-subobj P* ((*C*, *C'* # *rev Cs''*)) **by** (*simp add*:*subobj-aux-rev*)
with *IH* **have** *rel*:*P* ⊢ *C* ≼* *last* (*C'* # *rev Cs''*) **by** *simp*
from *subo* **obtain** *D Ds* **where** *DDs*:*C'* # *rev Cs''* = *Ds*@[*D*]
by (*cases Cs''*) *auto*
hence *C'* # *rev* (*C''* # *Cs''*) = *Ds*@[*D*,*C''*] **by** *simp*
with *subo* **have** *is-subobj P* ((*C*,*Ds*@[*D*,*C''*])) **by** (*cases Ds*) *auto*
hence *P* ⊢ *D* ≺$_R$ *C''* **by** (*rule-tac Cs'*=[] **in** *isSubobjs-subclsR*) *simp*
hence *rel1*:*P* ⊢ *D* ≺¹ *C''* **by** (*rule subclsR-subcls1*)
from *DDs* **have** *D* = *last* (*C'* # *rev Cs''*) **by** *simp*
with *rel1* **have** *lastrel1*:*P* ⊢ *last* (*C'* # *rev Cs''*) ≺¹ *C''* **by** *simp*
with *rel* **have** *P* ⊢ *C* ≼* *C''*
by(*rule-tac b*=*last* (*C'* # *rev Cs''*) **in** *rtrancl-into-rtrancl*) *simp*
thus *?case* **by** *simp*
qed

lemma *mdc-leq-ldc*:
assumes *1*:*is-subobj P* (*R*)
shows *P* ⊢ *mdc R* ≼* *ldc R*

proof −
from *1* **obtain** *C Cs* **where** *R*:*R* = (*C*,*Cs*) **by** (*cases R*) *auto*
with *1* **have** *ne*:*Cs* ≠ [] **by** (*cases Cs*) *auto*
from *this* **obtain** *C' Cs'* **where** *Cs*:*Cs* = *C'*#*Cs'* **by** (*cases Cs*) *auto*
from *this* **obtain** *Cs''* **where** *Cs'*:*Cs''* = *rev Cs'* **by** *simp*
with *R Cs 1* **have** *is-subobj P* ((*C*,*C'*#*rev Cs''*)) **by** *simp*
hence *rel*:*P* ⊢ *C* ≼* *last* (*C'*#*rev Cs''*) **by** (*rule mdc-leq-ldc-aux*)
from *R Cs Cs'* **have** *ldc*:*last* (*C'*#*rev Cs''*) = *ldc R* **by**(*simp add*:*ldc-def*)
from *R* **have** *mdc R* = *C* **by**(*simp add*:*mdc-def*)
with *ldc rel* **show** *?thesis* **by** *simp*
qed

Next three lemmas show subobject property as presented in literature

lemma *class-isSubobj*:
is-class P C ⟹ *is-subobj P* ((*C*,[*C*]))
by *simp*

lemma *repSubobj-isSubobj*:
assumes *1*:*is-subobj P* ((*C*,*Xs*@[*X*])) **and** *2*:*P* ⊢ *X* ≺$_R$ *Y*
shows *is-subobj P* ((*C*,*Xs*@[*X*, *Y*]))

using *1*
proof −
obtain *Cs D E Cs′* **where** *Cs1*:*Cs* = *Xs*@[*X*, *Y*] **and** *D* = *hd*(*Xs*@[*X*, *Y*])
and *E* = *hd*(*tl*(*Xs*@[*X*, *Y*])) **and** *Cs′* = *tl*(*tl*(*Xs*@[*X*, *Y*]))**by** *simp*
hence *Cs2*:*Cs* = *D*#*E*#*Cs′* **by** (*cases Xs*) *auto*
with *1 Cs1* **have** *subobj-butlast*:*is-subobj P* ((*C*,*butlast*(*D*#*E*#*Cs′*)))
by (*simp add*:*butlast-tail*)
with *2 Cs1 Cs2* **have** *P* ⊢ (*last*(*butlast*(*D*#*E*#*Cs′*))) ≺$_R$ *last*(*D*#*E*#*Cs′*)
by (*simp add*:*butlast-tail*)
with *subobj-butlast* **have** *is-subobj P* ((*C*,(*D*#*E*#*Cs′*))) **by** *simp*
with *Cs1 Cs2* **show** *?thesis* **by** *simp*
qed

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

using *1*
proof −
from *1* **have** *classC*:*is-class P C*
by (*drule-tac R*=(*C*,*Xs*@[*X*]) **in** *isSubobj-isClass*, *simp add*:*mdc-def*)
from *1* **have** *P* ⊢ *C* ≼* *X*
by (*drule-tac R*=(*C*,*Xs*@[*X*]) **in** *mdc-leq-ldc*, *simp add*:*mdc-def ldc-def*)
with *classC 2* **show** *?thesis* **by** *fastforce*
qed

Auxiliary lemmas

lemma *build-rec-isSubobj-rev*:
assumes *1*:*is-subobj P* ((*D*,*D*#*rev Cs*)) **and** *2*: *P* ⊢ *C* ≺$_R$ *D*
shows *is-subobj P* ((*C*,*C*#*D*#*rev Cs*))
using *1*
proof (*induct Cs*)
case *Nil*
from *2* **have** *is-class P C* **by** (*auto dest*:*subclsRD simp add*:*is-class-def*)
with *1 2* **show** *?case* **by** *simp*
next
case (*Cons C′ Cs′*)
have *suboD*:*is-subobj P* ((*D*,*D*#*rev* (*C′*#*Cs′*)))
and *IH*:*is-subobj P* ((*D*,*D*#*rev Cs′*)) ⟹ *is-subobj P* ((*C*,*C*#*D*#*rev Cs′*)) **by**
fact+
obtain *E Es* **where** *E*:*E* = *hd* (*rev* (*C′*#*Cs′*)) **and** *Es*:*Es* = *tl* (*rev* (*C′*#*Cs′*))
by *simp*
with *E* **have** *E-Es*:*rev* (*C′*#*Cs′*) = *E*#*Es* **by** *simp*

with *E Es* **have** *butlast*:*butlast* (*D*#*E*#*Es*) = *D*#*rev Cs'* **by** *simp*
from *E-Es suboD* **have** *suboDE*:*is-subobj P* ((*D*,*D*#*E*#*Es*)) **by** *simp*
hence *is-subobj P* ((*D*,*butlast* (*D*#*E*#*Es*))) **by** *simp*
with *butlast* **have** *is-subobj P* ((*D*,*D*#*rev Cs'*)) **by** *simp*
with *IH* **have** *suboCD*:*is-subobj P* ((*C*, *C*#*D*#*rev Cs'*)) **by** *simp*
from *suboDE* **obtain** *Xs X Y Xs'* **where** *Xs'*:*Xs'* = *D*#*E*#*Es*
and *bb*:*Xs* = *butlast* (*butlast* (*D*#*E*#*Es*))
and *lb*:*X* = *last*(*butlast* (*D*#*E*#*Es*)) **and** *l*:*Y* = *last* (*D*#*E*#*Es*) **by** *simp*
from *this* **obtain** *Xs''* **where** *Xs''*:*Xs''* = *Xs*@[*X*] **by** *simp*
with *bb lb* **have** *Xs''* = *butlast* (*D*#*E*#*Es*) **by** *simp*
with *l* **have** *D*#*E*#*Es* = *Xs''*@[*Y*] **by** *simp*
with *Xs''* **have** *D*#*E*#*Es* = *Xs*@[*X*]@[*Y*] **by** *simp*
with *suboDE* **have** *is-subobj P* ((*D*,*Xs*@[*X*,*Y*])) **by** *simp*
hence *subR*:*P* ⊢ *X* ≺$_R$ *Y* **by**(*rule-tac Cs*=*Xs* **and** *Cs'*=[] **in** *isSubobjs-subclsR*)
simp
from *E-Es Es* **have** *last* (*D*#*E*#*Es*) = *C'* **by** *simp*
with *subR lb l butlast* **have** *P* ⊢ *last*(*D*#*rev Cs'*) ≺$_R$ *C'*
by (*auto split*:*if-split-asm*)
with *suboCD* **show** *?case* **by** *simp*
qed

lemma *build-rec-isSubobj*:
assumes *1*:*is-subobj P* ((*D*,*D*#*Cs*)) **and** *2*: *P* ⊢ *C* ≺$_R$ *D*
shows *is-subobj P* ((*C*,*C*#*D*#*Cs*))

proof −
obtain *Cs'* **where** *Cs'*:*Cs'* = *rev Cs* **by** *simp*
with *1* **have** *is-subobj P* ((*D*,*D*#*rev Cs'*)) **by** *simp*
with *2* **have** *is-subobj P* ((*C*,*C*#*D*#*rev Cs'*))
by − (*rule build-rec-isSubobj-rev*)
with *Cs'* **show** *?thesis* **by** *simp*
qed

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*)))
using *2*
proof (*induct Cs*)
case *Nil*
with *1* **show** *?case* **by** *simp*
next
case (*Cons C' Cs'*)
have *IH*:*is-subobj P* ((*D*,*D*#*rev Cs'*)) ⟹ *is-subobj P* ((*C*,*D*#*rev Cs'*))

and *is-subobj P* ((*D*,*D*#*rev* (*C′*#*Cs′*))) **by** *fact+*
hence *suboD*:*is-subobj P* ((*D*,*D*#*rev Cs′*@[*C′*])) **by** *simp*
hence *is-subobj P* ((*D*,*D*#*rev Cs′*)) **by** (*rule subobj-aux-rev*)
with *IH* **have** *suboC*:*is-subobj P* ((*C*,*D*#*rev Cs′*)) **by** *simp*
obtain *C″* **where** *C″*: *C″* = *last* (*D* # *rev Cs′*) **by** *simp*
moreover have *D* # *rev Cs′* = *butlast* (*D* # *rev Cs′*) @ [*last* (*D* # *rev Cs′*)]
by (*rule append-butlast-last-id* [*symmetric*]) *simp*
ultimately have *butlast*: *D* # *rev Cs′* = *butlast* (*D* #*rev Cs′*) @ [*C″*]
by *simp*
hence *butlast2*:*D*#*rev Cs′*@[*C′*] = *butlast*(*D*#*rev Cs′*)@[*C″*]@[*C′*] **by** *simp*
with *suboD* **have** *is-subobj P* ((*D*,*butlast*(*D*#*rev Cs′*)@[*C″*]@[*C′*]))
by *simp*
with *C″* **have** *subR*:*P* ⊢ *C″* ≺$_R$ *C′*
by (*rule-tac Cs*=*butlast*(*D*#*rev Cs′*) **and** *Cs′*=[] **in** *isSubobjs-subclsR*)*simp*
with *C″ suboC butlast* **have** *is-subobj P* ((*C*,*butlast*(*D*#*rev Cs′*)@[*C″*]@[*C′*]))
by (*auto intro*:*repSubobj-isSubobj simp del*:*butlast.simps*)
with *butlast2* **have** *is-subobj P* ((*C*,*D*#*rev Cs′*@[*C′*]))
by (*cases Cs′*)*auto*
thus *?case* **by** *simp*
qed

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 −
obtain *Cs′* **where** *Cs′*:*Cs′* = *rev Cs* **by** *simp*
with *2* **have** *is-subobj P* ((*D*,*D*#*rev Cs′*)) **by** *simp*
with *1* **have** *is-subobj P* ((*C*,*D*#*rev Cs′*))
by − (*rule isSubobj-isSubobj-isSubobj-rev*)
with *Cs′* **show** *?thesis* **by** *simp*
qed

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* ⊢ *C* ≺$_R$ *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* ⊢ *C* ≼* *C′*; *P* ⊢ *C′* ≺$_S$ *D*; *Subobjs*$_R$ *P D Cs*⟧
⟹ *Subobjs P C Cs*

lemma *Subobjs-Base*:*is-class P C* $\Longrightarrow$ *Subobjs P C* [*C*]
by (*fastforce intro*:*Subobjs-Rep SubobjsR-Base*)

lemma *SubobjsR-nonempty*: *Subobjs*$_R$ *P C Cs* $\Longrightarrow$ *Cs* $\neq$ []
by (*induct rule*: *Subobjs*$_R$.*induct*, *simp-all*)

lemma *Subobjs-nonempty*: *Subobjs P C Cs* $\Longrightarrow$ *Cs* $\neq$ []
by (*erule Subobjs.induct*)(*erule SubobjsR-nonempty*)+

lemma *hd-SubobjsR*:
Subobjs$_R$ *P C Cs* $\Longrightarrow$ $\exists$ *Cs'*. *Cs* = *C*#*Cs'*
by(*erule Subobjs*$_R$.*induct*,*simp*+)

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

apply(*erule Subobjs*$_R$.*induct*)
apply *simp*
apply(*simp add*: *SubobjsR-nonempty*)
done

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

apply(*erule Subobjs*$_R$.*induct*)
apply *simp*
apply(*simp add*: *SubobjsR-nonempty*)
apply(*blast intro*:*subclsR-subcls1 rtrancl-trans*)
done

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

apply(*erule Subobjs.induct*)
apply(*erule SubobjsR-subclass*)
apply(*erule rtrancl-trans*)
apply(*blast intro*:*subclsS-subcls1 SubobjsR-subclass rtrancl-trans*)
done

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*
apply (*induct rule*: *Subobjs.induct*)

apply *clarsimp*
apply *fastforce*
done

lemma assumes *subo*:$Subobjs_R\ P\ (hd\ (Cs@\ C'\#Cs'))\ (Cs@\ C'\#Cs')$
shows *SubobjsR-Subobjs*:*Subobjs P* $C'\ (C'\#Cs')$
using *subo*
proof (*induct Cs*)
case *Nil*
thus *?case* **by** −(*frule hd-SubobjsR*,*fastforce intro*:*Subobjs-Rep*)
next
case (*Cons D Ds*)
have *subo'*:$Subobjs_R\ P\ (hd\ ((D\#Ds)\ @\ C'\#Cs'))\ ((D\#Ds)\ @\ C'\#Cs')$
and *IH*:$Subobjs_R\ P\ (hd\ (Ds\ @\ C'\#Cs'))\ (Ds\ @\ C'\#Cs') \Longrightarrow Subobjs\ P\ C'$
$(C'\#Cs')$ **by** *fact+*
from *subo'* **have** $Subobjs_R\ P\ (hd\ (Ds\ @\ C'\ \#\ Cs'))\ (Ds\ @\ C'\ \#\ Cs')$
apply −
apply (*drule* $Subobjs_R$*.cases*)
apply *auto*
apply (*rename-tac D'*)
apply (*subgoal-tac* $D' = hd\ (Ds\ @\ C'\ \#\ Cs')$)
apply (*auto dest*:*hd-SubobjsR*)
done
with *IH* **show** *?case* **by** *simp*
qed

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

apply −
apply (*drule Subobjs.cases*)
apply *auto*
apply (*subgoal-tac* $C = hd(Cs\ @\ C'\ \#\ Cs')$)
apply (*fastforce intro*:*SubobjsR-Subobjs*)
apply (*fastforce dest*:*hd-SubobjsR*)
apply (*subgoal-tac* $D = hd(Cs\ @\ C'\ \#\ Cs')$)
apply (*fastforce intro*:*SubobjsR-Subobjs*)
apply (*fastforce dest*:*hd-SubobjsR*)
done

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

using *subo*
proof (*induct rule*:$Subobjs_R$*.induct*)

case *SubobjsR-Base* **thus** *?case* **by** *assumption*
next
case *SubobjsR-Rep* **thus** *?case* **by** (*fastforce intro*:*subclsR-subcls1 subcls1-class*)
qed

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

using *subo*
proof (*induct rule*:*Subobjs.induct*)
case *Subobjs-Rep* **thus** *?case* **by** (*rule SubobjsR-isClass*)
next
case (*Subobjs-Sh C C' D Cs*)
have *leq*:$P \vdash C \preceq^* C'$ **and** *leqS*:$P \vdash C' \prec_S D$ **by** *fact+*
hence $(C,D) \in (subcls1\ P)^+$ **by** (*fastforce intro*:*rtrancl-into-trancl1 subclsS-subcls1*)
thus *?case* **by** (*induct rule*:*trancl-induct*, *fastforce intro*:*subcls1-class*)
qed

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

using *subo*
proof −
from *subo* **have** *Subobjs P D* (*D*#*D'*#*Cs'*) **by** −(*rule Subobjs-Subobjs*,*simp*)
then obtain *C'* **where** *subo'*:$Subobjs_R\ P\ C'$ (*D*#*D'*#*Cs'*)
by (*induct rule*:*Subobjs.induct*,*blast+*)
hence $C' = D$ **by** −(*drule hd-SubobjsR*,*simp*)
with *subo'* **have** $Subobjs_R\ P\ D$ (*D*#*D'*#*Cs'*) **by** *simp*
thus *?thesis* **by** (*fastforce elim*:$Subobjs_R$.*cases dest*:*hd-SubobjsR*)
qed

lemma assumes *subo*:$Subobjs_R\ P$ (*hd Cs*) (*Cs*@[*D*]) **and** *notempty*:$Cs \neq []$
shows *butlast-Subobjs-Rep*:$Subobjs_R\ P$ (*hd Cs*) *Cs*
using *subo notempty*
proof (*induct Cs*)
case *Nil* **thus** *?case* **by** *simp*
next
case (*Cons C' Cs'*)
have *subo*:$Subobjs_R\ P$ (*hd*(*C'*#*Cs'*)) ((*C'*#*Cs'*)@[*D*])
and *IH*:$[\![Subobjs_R\ P\ (hd\ Cs')\ (Cs'@[D]);\ Cs' \neq []]\!] \Longrightarrow Subobjs_R\ P\ (hd\ Cs')$ *Cs'* **by** *fact+*
from *subo* **have** *subo'*:$Subobjs_R\ P\ C'$ (*C'*#*Cs'*@[*D*]) **by** *simp*

```
  show ?case
  proof (cases Cs' = [])
    case True
    with subo' have Subobjs_R P C' [C',D] by simp
    hence is-class P C' by(rule SubobjsR-isClass)
    hence Subobjs_R P C' [C'] by (rule SubobjsR-Base)
    with True show ?thesis by simp
  next
    case False
    with subo' obtain D' where subo'':Subobjs_R P D' (Cs'@[D])
      and subR:P ⊢ C' ≺_R D'
      by (auto elim:Subobjs_R.cases)
    from False subo'' have hd:D' = hd Cs'
      by (induct Cs',auto dest:hd-SubobjsR)
    with subo'' False IH have Subobjs_R P (hd Cs') Cs' by simp
   with subR hd have Subobjs_R P C' (C'#Cs') by (fastforce intro:SubobjsR-Rep)
    thus ?thesis by simp
  qed
qed



lemma assumes subo:Subobjs P C (Cs@[D]) and notempty:Cs ≠ []
  shows butlast-Subobjs:Subobjs P C Cs

using subo
proof (rule Subobjs.cases,auto)
  assume subo R:Subobjs_R P C (Cs@[D]) and Subobjs P C (Cs@[D])
  from subo R notempty have hd:C = hd Cs
    by (induct Cs,auto dest:hd-SubobjsR)
  with subo R notempty have Subobjs_R P (hd Cs) Cs
    by(fastforce intro:butlast-Subobjs-Rep)
  with hd show Subobjs P C Cs by (fastforce intro:Subobjs-Rep)
next
  fix C' D' assume leq:P ⊢ C ≼* C' and subS:P ⊢ C' ≺_S D'
  and subo R:Subobjs_R P D' (Cs@[D]) and Subobjs P C (Cs@[D])
  from subo R notempty have hd:D' = hd Cs
    by (induct Cs,auto dest:hd-SubobjsR)
  with subo R notempty have Subobjs_R P (hd Cs) Cs
    by(fastforce intro:butlast-Subobjs-Rep)
  with hd leq subS show Subobjs P C Cs
    by(fastforce intro:Subobjs-Sh)
qed



lemma assumes subo:Subobjs P C (Cs@(rev Cs')) and notempty:Cs ≠ []
  shows rev-appendSubobj:Subobjs P C Cs
```

using *subo*
proof(*induct Cs'*)
case *Nil* **thus** *?case* **by** *simp*
next
case (*Cons D Ds*)
have *subo'*:*Subobjs P C* (*Cs*@*rev*(*D*#*Ds*))
and *IH*:*Subobjs P C* (*Cs*@*rev Ds*) ⟹ *Subobjs P C Cs* **by** *fact+*
from *notempty subo'* **have** *Subobjs P C* (*Cs*@*rev Ds*)
by (*fastforce intro*:*butlast-Subobjs*)
with *IH* **show** *?case* **by** *simp*
qed

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

proof −
obtain *Cs''* **where** *Cs''*:*Cs''* = *rev Cs'* **by** *simp*
with *subo* **have** *Subobjs P C* (*Cs*@(*rev Cs''*)) **by** *simp*
with *notempty* **show** *?thesis* **by** − (*rule rev-appendSubobj*)
qed

lemma *SubobjsR-isSubobj*:
Subobjs$_R$ *P C Cs* ⟹ *is-subobj P* ((*C*,*Cs*))
by(*erule Subobjs*$_R$.*induct*,*simp*,
auto dest:*hd-SubobjsR intro*:*build-rec-isSubobj*)

lemma *leq-SubobjsR-isSubobj*:
⟦*P* ⊢ *C* ≼* *C'*; *P* ⊢ *C'* ≺$_S$ *D*; *Subobjs*$_R$ *P D Cs*⟧
⟹ *is-subobj P* ((*C*,*Cs*))

apply (*subgoal-tac is-subobj P* ((*C*,[*D*])))
apply (*frule hd-SubobjsR*)
apply (*drule SubobjsR-isSubobj*)
apply (*erule exE*)
apply (*simp del*: *is-subobj.simps*)
apply (*erule isSubobj-isSubobj-isSubobj*)
apply *simp*
apply *auto*
done

lemma *Subobjs-isSubobj*:
Subobjs P C Cs ⟹ *is-subobj P* ((*C*,*Cs*))

by (*auto elim*:*Subobjs.induct SubobjsR-isSubobj*
simp add:*leq-SubobjsR-isSubobj*)

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* ≠ [] ⟹ *last Cs* = *last* (*Cs'*$@_p$*Cs*)
by(*auto simp*:*appendPath-def last-append*)(*cases Cs*, *simp-all*)+

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* ⟹ *P* ⊢ *T casts v to v*

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

| *casts-ref*: ⟦ *P* ⊢ *Path last Cs to C via Cs'*; *Ds* = *Cs*$@_p$*Cs'* ⟧
⟹ *P* ⊢ *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* ⊢ [] *Casts* [] *to* []

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

lemma *length-Casts-vs*:
P ⊢ *Ts Casts vs to vs′* ⟹ *length Ts* = *length vs*
by (*induct rule*:*Casts-to.induct*,*simp-all*)

lemma *length-Casts-vs′*:
P ⊢ *Ts Casts vs to vs′* ⟹ *length Ts* = *length vs′*
by (*induct rule*:*Casts-to.induct*,*simp-all*)

7.6 The relation on paths

inductive-set
leq-path1 :: *prog* ⇒ *cname* ⇒ (*path* × *path*) *set*
and *leq-path1′* :: *prog* ⇒ *cname* ⇒ [*path*, *path*] ⇒ *bool* (-,- ⊢ - ⊏¹ - [*71*,*71*,*71*] *70*)
for *P* :: *prog* **and** *C* :: *cname*
where
P,*C* ⊢ *Cs* ⊏¹ *Ds* ≡ (*Cs*,*Ds*) ∈ *leq-path1 P C*
| *leq-pathRep*: ⟦ *Subobjs P C Cs*; *Subobjs P C Ds*; *Cs* = *butlast Ds*⟧
⟹ *P*,*C* ⊢ *Cs* ⊏¹ *Ds*
| *leq-pathSh*: ⟦ *Subobjs P C Cs*; *P* ⊢ *last Cs* ≺*S D* ⟧
⟹ *P*,*C* ⊢ *Cs* ⊏¹ [*D*]

abbreviation
leq-path :: *prog* ⇒ *cname* ⇒ [*path*, *path*] ⇒ *bool* (-,- ⊢ - ⊑ - [*71*,*71*,*71*] *70*)
where
P,*C* ⊢ *Cs* ⊑ *Ds* ≡ (*Cs*,*Ds*) ∈ (*leq-path1 P C*)*

lemma *leq-path-rep*:
⟦ *Subobjs P C* (*Cs*@[*C′*]); *Subobjs P C* (*Cs*@[*C′*,*C″*])⟧
⟹ *P*,*C* ⊢ (*Cs*@[*C′*]) ⊏¹ (*Cs*@[*C′*,*C″*])
by(*rule leq-pathRep*,*simp-all add*:*butlast-tail*)

lemma *leq-path-sh*:
⟦ *Subobjs P C* (*Cs*@[*C′*]); *P* ⊢ *C′* ≺*S C″*⟧
⟹ *P*,*C* ⊢ (*Cs*@[*C′*]) ⊏¹ [*C″*]
by(*erule leq-pathSh*)*simp*

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**

$P \vdash C$ *has least* $F{:}T$ *via* $Cs \equiv$
$(Cs,T) \in$ *FieldDecls* $P\ C\ F \wedge$
$(\forall (Cs',T') \in$ *FieldDecls* $P\ C\ F.\ P,C \vdash Cs \sqsubseteq Cs')$

definition *MethodDefs* :: *prog* ⇒ *cname* ⇒ *mname* ⇒ (*path* × *method*)*set* **where**
MethodDefs $P\ C\ M \equiv$
$\{(Cs,mthd).$ *Subobjs* $P\ C\ Cs \wedge (\exists\, Bs\ fs\ ms.$ *class* P (*last* Cs) = *Some*(Bs,fs,ms)
$\wedge$ *map-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**
$P \vdash C$ *has* M = *mthd via* $Cs \equiv (Cs,mthd) \in$ *MethodDefs* $P\ C\ M$

definition *LeastMethodDef* :: *prog* ⇒ *cname* ⇒ *mname* ⇒ *method* ⇒ *path* ⇒ *bool*
(- ⊢ - *has least* - = - *via* - [51,0,0,0,51] 50) **where**
$P \vdash C$ *has least* M = *mthd via* $Cs \equiv$
$(Cs,mthd) \in$ *MethodDefs* $P\ C\ M \wedge$
$(\forall (Cs',mthd') \in$ *MethodDefs* $P\ C\ M.\ P,C \vdash Cs \sqsubseteq Cs')$

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

definition *OverriderMethodDefs* :: *prog* ⇒ *subobj* ⇒ *mname* ⇒ (*path* × *method*)*set*
where
OverriderMethodDefs $P\ R\ M \equiv$
$\{(Cs,mthd).\ \exists\, Cs'\ mthd'.\ P \vdash (ldc\ R)$ *has least* $M = mthd'$ *via* $Cs' \wedge$
$(Cs,mthd) \in$ *MinimalMethodDefs* P $(mdc\ R)\ M \wedge$
$P,mdc\ R \vdash Cs \sqsubseteq (snd\ R)@_p Cs'\}$

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*:
$[\![\forall\, mthd\ Cs'.\ \neg\ P \vdash C$ *has least* $M = mthd$ *via* $Cs';$
$P \vdash (C,Cs)$ *has overrider* $M = mthd$ *via* $Cs']\!]$
$\Longrightarrow P \vdash (C,Cs)$ *selects* $M = mthd$ *via* Cs'

lemma *sees-fields-fun*:
$(Cs,T) \in FieldDecls\ P\ C\ F \Longrightarrow (Cs,T') \in FieldDecls\ P\ C\ F \Longrightarrow T = T'$
by(*fastforce simp*:*FieldDecls-def*)

lemma *sees-field-fun*:
$[\![P \vdash C$ *has least* F:T *via* $Cs;\ P \vdash C$ *has least* F:T' *via* $Cs]\!]$
$\Longrightarrow T = T'$
by (*fastforce simp*:*LeastFieldDecl-def dest*:*sees-fields-fun*)

lemma *has-least-method-has-method*:
$P \vdash C$ *has least* $M = mthd$ *via* $Cs \Longrightarrow P \vdash C$ *has* $M = mthd$ *via* Cs
by (*simp add*:*LeastMethodDef-def HasMethodDef-def*)

lemma *visible-methods-exist*:
$(Cs,mthd) \in MethodDefs\ P\ C\ M \Longrightarrow$
$(\exists\, Bs\ fs\ ms.\ class\ P\ (last\ Cs) = Some(Bs,fs,ms) \wedge map\text{-}of\ ms\ M = Some\ mthd)$
by(*auto simp*:*MethodDefs-def*)

lemma *sees-methods-fun*:
$(Cs,mthd) \in MethodDefs\ P\ C\ M \Longrightarrow (Cs,mthd') \in MethodDefs\ P\ C\ M \Longrightarrow mthd = mthd'$
by(*fastforce simp*:*MethodDefs-def*)

lemma *sees-method-fun*:
$[\![P \vdash C$ *has least* $M = mthd$ *via* $Cs;\ P \vdash C$ *has least* $M = mthd'$ *via* $Cs]\!]$
$\Longrightarrow mthd = mthd'$
by (*fastforce simp*:*LeastMethodDef-def dest*:*sees-methods-fun*)

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 −
from *overrider'* **have** *omd*:$(Cs'',mthd') \in OverriderMethodDefs\ P\ (C,Cs)\ M$
by(*simp-all add*:*FinalOverriderMethodDef-def*)
from *overrider* **have** $(Cs',mthd) \in OverriderMethodDefs\ P\ (C,Cs)\ M$

and *card*(*OverriderMethodDefs P* (*C*,*Cs*) *M*) = *1*
by(*simp-all add*:*FinalOverriderMethodDef-def*)
hence ∀(*Ds*,*mthd''*) ∈ *OverriderMethodDefs P* (*C*,*Cs*) *M*. (*Cs'*,*mthd*) = (*Ds*,*mthd''*)
by(*fastforce simp*:*card-Suc-eq*)
with *omd* **show** *?thesis* **by** *fastforce*
qed

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 ⟹ *init-obj P C* (*Cs*,*init-class-fieldmap P* (*last Cs*))

lemma *init-obj-nonempty*: *init-obj P C* (*Cs*,*fs*) ⟹ *Cs* ≠ []
by (*fastforce elim*:*init-obj.cases dest*:*Subobjs-nonempty*)

lemma *init-obj-no-Ref*:
⟦*init-obj P C* (*Cs*,*fs*); *fs F* = *Some*(*Ref*(*a'*,*Cs'*))⟧ ⟹ *False*
by (*fastforce elim*:*init-obj.cases default-val-no-Ref*
simp:*init-class-fieldmap-def map-of-map*)

lemma *SubobjsSet-init-objSet*:
{*Cs*. *Subobjs P C Cs*} = {*Cs*. ∃ *vmap*. *init-obj P C* (*Cs*,*vmap*)}
by (*fastforce intro*:*init-obj.intros elim*:*init-obj.cases*)

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*
by (*simp add*: *obj-ty-def*)

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*
by(*fastforce simp add*:*new-Addr-def split*:*if-splits intro*:*someI*)

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*
by(*simp add*: *sys-xcpts-def*)

lemma *sys-xcpts-cases* [*consumes 1*, *cases set*]:
⟦ *C* ∈ *sys-xcpts*; *P NullPointer*; *P OutOfMemory*; *P ClassCast*⟧ ⟹ *P C*
by (*auto simp add*: *sys-xcpts-def*)

9.3 *preallocated*

lemma *preallocated-dom* [*simp*]:
⟦ *preallocated h*; *C* ∈ *sys-xcpts* ⟧ ⟹ *addr-of-sys-xcpt C* ∈ *dom h*
by (*fastforce simp*:*preallocated-def dom-def*)

lemma *preallocatedD*:
⟦ *preallocated h*; *C* ∈ *sys-xcpts* ⟧ ⟹ ∃ *S*. *h* (*addr-of-sys-xcpt C*) = *Some* (*C*,*S*)
by(*auto simp add*: *preallocated-def sys-xcpts-def*)

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

lemma *cname-of-xcp* [*simp*]:
⟦ *preallocated h*; *C* ∈ *sys-xcpts* ⟧ ⟹ *cname-of h* (*addr-of-sys-xcpt C*) = *C*
by (*auto elim*: *preallocatedE*)

lemma *preallocated-start*:
preallocated (*start-heap P*)
by (*auto simp add*: *start-heap-def blank-def sys-xcpts-def fun-upd-apply*
addr-of-sys-xcpt-def preallocated-def)

9.4 *start-heap*

lemma *start-Subobj*:
⟦*start-heap P a = Some*(*C*, *S*); (*Cs*,*fs*) ∈ *S*⟧ ⟹ *Subobjs P C Cs*
by (*fastforce elim*:*init-obj.cases simp*:*start-heap-def blank-def*
fun-upd-apply split:*if-split-asm*)

lemma *start-SuboSet*:
⟦*start-heap P a = Some*(*C*, *S*); *Subobjs P C Cs*⟧ ⟹ ∃*fs*. (*Cs*,*fs*) ∈ *S*
by (*fastforce intro*:*init-obj.intros simp*:*start-heap-def blank-def*
split:*if-split-asm*)

lemma *start-init-obj*: *start-heap P a = Some*(*C*,*S*) ⟹ *S = Collect* (*init-obj P C*)
by (*auto simp*:*start-heap-def blank-def split*:*if-split-asm*)

lemma *start-subobj*:
⟦*start-heap P a = Some*(*C*, *S*); ∃*fs*. (*Cs*, *fs*) ∈ *S*⟧ ⟹ *Subobjs P C Cs*
by (*fastforce elim*:*init-obj.cases simp*:*start-heap-def blank-def*
split:*if-split-asm*)

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*:

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

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

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

$\Longrightarrow P,E \vdash \langle e_1 \ll bop \gg e_2,s_0\rangle \Rightarrow \langle Val\ v,s_2\rangle$

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

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

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

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

| *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*:
$P,E \vdash \langle e_1,s_0\rangle \Rightarrow \langle throw\ e',s_1\rangle \Longrightarrow$

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

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

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

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

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

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

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**
  $\mathit{final}\ e \equiv (\exists v.\ e = \mathit{Val}\ v) \vee (\exists r.\ e = \mathit{Throw}\ r)$

definition *finals*:: *expr list* ⇒ *bool* **where**
  $\mathit{finals}\ es \equiv (\exists vs.\ es = \mathit{map}\ \mathit{Val}\ vs) \vee (\exists vs\ r\ es'.\ es = \mathit{map}\ \mathit{Val}\ vs\ @\ \mathit{Throw}\ r\ \#\ es')$

lemma [*simp*]: *final*(*Val v*)
by(*simp add*:*final-def*)

lemma [*simp*]: *final*(*throw e*) = (∃ *r*. *e* = *ref r*)
by(*simp add*:*final-def*)

lemma *finalE*: ⟦ *final e*; ⋀*v*. *e* = *Val v* ⟹ *Q*; ⋀*r*. *e* = *Throw r* ⟹ *Q* ⟧ ⟹ *Q*
by(*auto simp*:*final-def*)

lemma [*iff*]: *finals* []

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

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

apply(*clarsimp simp add*:*finals-def*)
apply(*rule iffI*)
 apply(*erule disjE*)
  apply *simp*
 apply(*rule disjI2*)
 apply *clarsimp*
 apply(*case-tac vs*)
  apply *simp*
 apply *fastforce*
apply(*erule disjE*)
 apply (*rule disjI1*)
 apply *clarsimp*
apply(*rule disjI2*)
apply *clarsimp*
apply(*rule-tac x* = *v*#*vs* **in** *exI*)
apply *simp*
done

lemma *finals-app-map*[*iff*]: *finals* (*map Val vs* @ *es*) = *finals es*
by(*induct-tac vs*, *auto*)

lemma [*iff*]: *finals* (*map Val vs*)
using *finals-app-map*[*of vs* []]**by**(*simp*)

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

apply(*simp add*:*finals-def*)
apply(*rule iffI*)
 apply *clarsimp*
 apply(*case-tac vs*)
  apply *simp*
 apply *fastforce*
apply *fastforce*
done

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

apply(*auto simp add*:*finals-def final-def*)
apply(*case-tac vs*)
apply *auto*
done

lemma *eval-final*: $P,E \vdash \langle e,s\rangle \Rightarrow \langle e',s'\rangle \Longrightarrow final\ e'$
and *evals-final*: $P,E \vdash \langle es,s\rangle\ [\Rightarrow]\ \langle es',s'\rangle \Longrightarrow finals\ es'$
by(*induct rule*:*eval-evals.inducts*, *simp-all*)

lemma *eval-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 *evals-lcl-incr*: $P,E \vdash \langle es,(h_0,l_0)\rangle\ [\Rightarrow]\ \langle es',(h_1,l_1)\rangle \Longrightarrow dom\ l_0 \subseteq dom\ l_1$
by (*induct rule*:*eval-evals-inducts*) (*auto simp del*:*fun-upd-apply*)

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

lemma *eval-finalId*: *final e* $\Longrightarrow P,E \vdash \langle e,s\rangle \Rightarrow \langle e,s\rangle$
by (*erule finalE*) (*fastforce intro*: *eval-evals.intros*)+

lemma *eval-finalsId*:
assumes *finals*: *finals es* **shows** $P,E \vdash \langle es,s\rangle\ [\Rightarrow]\ \langle es,s\rangle$

using *finals*
proof (*induct es type*: *list*)
case *Nil* **show** *?case* **by** (*rule eval-evals.intros*)
next
case (*Cons e es*)
have *hyp*: *finals es* $\Longrightarrow P,E \vdash \langle es,s\rangle\ [\Rightarrow]\ \langle es,s\rangle$
and *finals*: *finals* (*e # es*) **by** *fact*+
show $P,E \vdash \langle e\ \#\ es,s\rangle\ [\Rightarrow]\ \langle e\ \#\ es,s\rangle$
proof *cases*
assume *final e*
thus *?thesis*
proof (*cases rule*: *finalE*)
fix *v* **assume** *e*: *e* = *Val v*
have $P,E \vdash \langle Val\ v,s\rangle \Rightarrow \langle Val\ v,s\rangle$ **by** (*simp add*: *eval-finalId*)
moreover from *finals e* **have** $P,E \vdash \langle es,s\rangle\ [\Rightarrow]\ \langle es,s\rangle$ **by**(*fast intro*:*hyp*)
ultimately have $P,E \vdash \langle Val\ v\#es,s\rangle\ [\Rightarrow]\ \langle Val\ v\#es,s\rangle$
by (*rule eval-evals.intros*)
with *e* **show** *?thesis* **by** *simp*
next
fix *a* **assume** *e*: *e* = *Throw a*
have $P,E \vdash \langle Throw\ a,s\rangle \Rightarrow \langle Throw\ a,s\rangle$ **by** (*simp add*: *eval-finalId*)
hence $P,E \vdash \langle Throw\ a\#es,s\rangle\ [\Rightarrow]\ \langle Throw\ a\#es,s\rangle$ **by** (*rule eval-evals.intros*)
with *e* **show** *?thesis* **by** *simp*
qed
next
assume ¬ *final e*
with *not-finals-ConsI finals* **have** *False* **by** *blast*
thus *?thesis* **..**
qed
qed

lemma
eval-preserves-obj:P,E $\vdash$ $\langle e,(h,l)\rangle$ $\Rightarrow$ $\langle e',(h',l')\rangle$ $\Longrightarrow$ ($\bigwedge S.$ h a = *Some*(D,S)
$\Longrightarrow$ $\exists S'.$ h' a = *Some*(D,S'))
and *evals-preserves-obj*:P,E $\vdash$ $\langle es,(h,l)\rangle$ [$\Rightarrow$] $\langle es',(h',l')\rangle$
$\Longrightarrow$ ($\bigwedge S.$ h a = *Some*(D,S) $\Longrightarrow$ $\exists S'.$ h' a = *Some*(D,S'))
by(*induct rule*:*eval-evals-inducts*)(*fastforce dest*:*new-Addr-SomeD*)+

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
⟦$\bigwedge aj\ ak\ al.$ P [] [] (aj # ak) al; $\bigwedge ad\ ae\ a\ b.$ P [] (ad # ae) a b;
$\bigwedge V\ Vs\ a\ b.$ P (V # Vs) [] a b; $\bigwedge V\ Vs\ T\ Ts\ aw.$ P (V # Vs) (T # Ts) [] aw;
$\bigwedge V\ Vs\ T\ Ts\ v\ vs\ e.$ P Vs Ts vs e $\Longrightarrow$ P (V # Vs) (T # Ts) (v # vs) e; $\bigwedge e.$ P [] [] [] e⟧
$\Longrightarrow$ P u v w x
by (*induction-schema*) (*pat-completeness*, *lexicographic-order*)

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

apply(*induct rule*:*blocks-old-induct*)
apply *simp-all*
apply *blast*
done

definition *assigned* :: *vname* ⇒ *expr* ⇒ *bool* **where**
assigned V e ≡ $\exists v\ e'.$ 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 ⊢ ⟨e,s⟩ → ⟨e',s'⟩ ⟹
  P,E ⊢ ⟨(⦇C⦈)e, s⟩ → ⟨(⦇C⦈)e', s'⟩

| RedStaticCastNull:
  P,E ⊢ ⟨(⦇C⦈)null, s⟩ → ⟨null,s⟩

| RedStaticUpCast:
  ⟦ P ⊢ Path last Cs to C via Cs'; Ds = Cs@_p Cs' ⟧
  ⟹ P,E ⊢ ⟨(⦇C⦈)(ref (a,Cs)), s⟩ → ⟨ref (a,Ds), s⟩

| RedStaticDownCast:
  P,E ⊢ ⟨(⦇C⦈)(ref (a,Cs@[C]@Cs')), s⟩ → ⟨ref (a,Cs@[C]), s⟩

| RedStaticCastFail:
  ⟦C ∉ set Cs; ¬ P ⊢ (last Cs) ≼* C⟧
  ⟹ P,E ⊢ ⟨(⦇C⦈)(ref (a,Cs)), s⟩ → ⟨THROW ClassCast, s⟩

| DynCastRed:
  P,E ⊢ ⟨e,s⟩ → ⟨e',s'⟩ ⟹
  P,E ⊢ ⟨Cast C e, s⟩ → ⟨Cast C e', s'⟩

| RedDynCastNull:
  P,E ⊢ ⟨Cast C null, s⟩ → ⟨null,s⟩

| RedStaticUpDynCast:
  ⟦ P ⊢ Path last Cs to C unique; P ⊢ Path last Cs to C via Cs'; Ds = Cs@_p Cs' ⟧
  ⟹ P,E ⊢ ⟨Cast C(ref(a,Cs)),s⟩ → ⟨ref(a,Ds),s⟩
```

| *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*:
$P,E \vdash \langle e,s\rangle \to \langle e',s'\rangle \Longrightarrow$
$P,E \vdash \langle e\ \ll bop \gg\ e_2,\ s\rangle \to \langle e'\ \ll bop \gg\ e_2,\ s'\rangle$

| *BinOpRed2*:
$P,E \vdash \langle e,s\rangle \to \langle e',s'\rangle \Longrightarrow$
$P,E \vdash \langle (Val\ v_1)\ \ll bop \gg\ e,\ s\rangle \to \langle (Val\ v_1)\ \ll bop \gg\ e',\ s'\rangle$

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

| *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 $\rrbracket$
$\Longrightarrow P,E \vdash \langle\{V{:}T;\ e\},\ (h,l)\rangle \to \langle\{V{:}T := Val\ v;\ e'\},\ (h',l'(V := l\ V))\rangle$

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

| *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*: $P,E \vdash \langle (Throw\ r) \ll bop \gg e_2,\ s\rangle \to \langle Throw\ r,\ s\rangle$
| *BinOpThrow2*: $P,E \vdash \langle (Val\ v_1) \ll bop \gg (Throw\ r),\ s\rangle \to \langle Throw\ r,\ s\rangle$
| *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*:$P,E \vdash \langle es,s\rangle\ [\to]\ \langle es',s'\rangle \Longrightarrow length\ es = length\ es'$
by (*induct rule*: *red-reds.inducts*) *auto*

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*]: $((e,s),e',s') \in Red\ P\ E = P,E \vdash \langle e,s\rangle \to \langle e',s'\rangle$
by (*simp add*:*Red-def*)

lemma[*simp*]: $((es,s),es',s') \in Reds\ P\ E = P,E \vdash \langle es,s\rangle\ [\to]\ \langle es',s'\rangle$
by (*simp add*:*Reds-def*)

abbreviation
Step :: *prog* ⇒ *env* ⇒ *expr* ⇒ *state* ⇒ *expr* ⇒ *state* ⇒ *bool*
(-,- ⊢ ((1⟨-,/-⟩) →∗/ (1⟨-,/-⟩))) [51,0,0,0,0] 81) **where**
P,*E* ⊢ ⟨*e*,*s*⟩ →∗ ⟨*e*′,*s*′⟩ ≡ ((*e*,*s*), *e*′,*s*′) ∈ (*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* ⊢ ⟨*es*,*s*⟩ [→]∗ ⟨*es*′,*s*′⟩ ≡ ((*es*,*s*), *es*′,*s*′) ∈ (*Reds P E*)∗

lemma *converse-rtrancl-induct-red*[*consumes 1*]:
assumes *P*,*E* ⊢ ⟨*e*,(*h*,*l*)⟩ →∗ ⟨*e*′,(*h*′,*l*′)⟩
and ⋀*e h l*. *R e h l e h l*
and ⋀*e*0 *h*0 *l*0 *e*1 *h*1 *l*1 *e*′ *h*′ *l*′.
⟦ *P*,*E* ⊢ ⟨*e*0,(*h*0,*l*0)⟩ → ⟨*e*1,(*h*1,*l*1)⟩; *R e*1 *h*1 *l*1 *e*′ *h*′ *l*′ ⟧ ⟹ *R e*0 *h*0 *l*0 *e*′ *h*′ *l*′
shows *R e h l e*′ *h*′ *l*′

proof −
{ fix *s s*′
assume *reds*: *P*,*E* ⊢ ⟨*e*,*s*⟩ →∗ ⟨*e*′,*s*′⟩
and *base*: ⋀*e s*. *R e* (*hp s*) (*lcl s*) *e* (*hp s*) (*lcl s*)
and *IH*: ⋀*e*0 *s*0 *e*1 *s*1 *e*′ *s*′.
⟦ *P*,*E* ⊢ ⟨*e*0,*s*0⟩ → ⟨*e*1,*s*1⟩; *R e*1 (*hp s*1) (*lcl s*1) *e*′ (*hp s*′) (*lcl s*′) ⟧
⟹ *R e*0 (*hp s*0) (*lcl s*0) *e*′ (*hp s*′) (*lcl s*′)
from *reds* **have** *R e* (*hp s*) (*lcl s*) *e*′ (*hp s*′) (*lcl s*′)
proof (*induct rule*:*converse-rtrancl-induct2*)
case *refl* **show** *?case* **by**(*rule base*)
next
case (*step e*0 *s*0 *e s*)
have *Red*:((*e*0,*s*0),*e*,*s*) ∈ *Red P E*
and *R*:*R e* (*hp s*) (*lcl s*) *e*′ (*hp s*′) (*lcl s*′) **by** *fact*+
from *IH*[*OF Red*[*simplified*] *R*] **show** *?case* **.**
qed
}
with *assms* **show** *?thesis* **by** *fastforce*
qed

lemma *steps-length*:*P*,*E* ⊢ ⟨*es*,*s*⟩ [→]∗ ⟨*es*′,*s*′⟩ ⟹ *length es* = *length es*′
by(*induct rule*:*rtrancl-induct2*,*auto intro*:*reds-length*)

13.4 Some easy lemmas

lemma [*iff*]: ¬ *P*,*E* ⊢ ⟨[],*s*⟩ [→] ⟨*es*′,*s*′⟩

by(*blast elim*: *reds.cases*)

lemma [*iff*]: ¬ $P,E \vdash \langle Val\ v,s\rangle \to \langle e',s'\rangle$
by(*fastforce elim*: *red.cases*)

lemma [*iff*]: ¬ $P,E \vdash \langle Throw\ r,s\rangle \to \langle e',s'\rangle$
by(*fastforce elim*: *red.cases*)

lemma *red-lcl-incr*: $P,E \vdash \langle e,(h_0,l_0)\rangle \to \langle e',(h_1,l_1)\rangle \Longrightarrow dom\ l_0 \subseteq dom\ l_1$
and $P,E \vdash \langle es,(h_0,l_0)\rangle\ [\to]\ \langle es',(h_1,l_1)\rangle \Longrightarrow dom\ l_0 \subseteq dom\ l_1$
by (*induct rule*: *red-reds-inducts*) (*auto simp del*:*fun-upd-apply*)

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

proof (*induct rule*:*red-reds-inducts*)
case *RedLAss* **thus** *?case* **by**(*auto intro*:*red-reds.intros simp del*:*fun-upd-apply*)
next
case *RedStaticDownCast* **thus** *?case* **by**(*fastforce intro*:*red-reds.intros*)
next
case *RedStaticUpDynCast* **thus** *?case* **by**(*fastforce intro*:*red-reds.intros*)
next
case *RedStaticDownDynCast* **thus** *?case* **by**(*fastforce intro*:*red-reds.intros*)
next
case *RedDynCast* **thus** *?case* **by**(*fastforce intro*:*red-reds.intros*)
next
case *RedDynCastFail* **thus** *?case* **by**(*fastforce intro*:*red-reds.intros*)
next
case *RedFAcc* **thus** *?case* **by**(*fastforce intro*:*red-reds.intros*)
next
case *RedFAss* **thus** *?case* **by** (*fastforce intro*:*red-reds.intros*)
next
case *RedCall* **thus** *?case* **by** (*fastforce intro*!:*red-reds.RedCall*)
next
case *RedStaticCall* **thus** *?case* **by**(*fastforce intro*:*red-reds.intros*)
next
case (*InitBlockRed E V T e h l v' e' h' l' v'' v* l_0)
have *IH*: $\bigwedge l_0.\ P,E(V \mapsto T) \vdash \langle e,(h,\ l_0\ {+}{+}\ l(V \mapsto v'))\rangle \to \langle e',(h',\ l_0\ {+}{+}\ l')\rangle$
and $l'V$: $l'\ V = Some\ v''$ **and** *casts*:$P \vdash T$ *casts v to v'* **by** *fact+*
from *IH* **have** *IH'*: $P,E(V \mapsto T) \vdash \langle e,(h,\ (l_0\ {+}{+}\ l)(V \mapsto v'))\rangle \to \langle e',(h',l_0$
${+}{+}\ l')\rangle$
by *simp*
have $(l_0\ {+}{+}\ l')(V := (l_0\ {+}{+}\ l)\ V) = l_0\ {+}{+}\ l'(V := l\ V)$
by(*rule ext*)(*simp add*:*map-add-def*)
with *red-reds.InitBlockRed*[*OF IH' - casts*] $l'V$ **show** *?case*
by(*simp del*:*fun-upd-apply*)

next
case (*BlockRedNone E V T e h l e′ h′ l′ l*$_0$)
have *IH*: $\bigwedge l_0.\ P,E(V \mapsto T) \vdash \langle e,(h,\ l_0 +\!\!+ l(V := None))\rangle \to \langle e',(h',\ l_0 +\!\!+ l')\rangle$
and $l'V$: $l'\ V = None$ **and** *unass*: $\neg$ *assigned V e* **by** *fact+*
have $l_0(V := None) +\!\!+ l(V := None) = (l_0 +\!\!+ l)(V := None)$
by(*simp add*:*fun-eq-iff map-add-def*)
hence *IH′*: $P,E(V \mapsto T) \vdash \langle e,(h,\ (l_0 +\!\!+ l)(V := None))\rangle \to \langle e',(h',\ l_0(V := None) +\!\!+ l')\rangle$
using *IH*[*of* $l_0(V := None)$] **by** *simp*
have $(l_0(V := None) +\!\!+ l')(V := (l_0 +\!\!+ l)\ V) = l_0 +\!\!+ l'(V := l\ V)$
by(*simp add*:*fun-eq-iff map-add-def*)
with *red-reds.BlockRedNone*[*OF IH′ - unass*] $l'V$ **show** *?case*
by(*simp add*: *map-add-def*)
next
case (*BlockRedSome E V T e h l e′ h′ l′ v l*$_0$)
have *IH*: $\bigwedge l_0.\ P,E(V \mapsto T) \vdash \langle e,(h,\ l_0 +\!\!+ l(V := None))\rangle \to \langle e',(h',\ l_0 +\!\!+ l')\rangle$
and $l'V$: $l'\ V = Some\ v$ **and** *unass*: $\neg$ *assigned V e* **by** *fact+*
have $l_0(V := None) +\!\!+ l(V := None) = (l_0 +\!\!+ l)(V := None)$
by(*simp add*:*fun-eq-iff map-add-def*)
hence *IH′*: $P,E(V \mapsto T) \vdash \langle e,(h,\ (l_0 +\!\!+ l)(V := None))\rangle \to \langle e',(h',\ l_0(V := None) +\!\!+ l')\rangle$
using *IH*[*of* $l_0(V := None)$] **by** *simp*
have $(l_0(V := None) +\!\!+ l')(V := (l_0 +\!\!+ l)\ V) = l_0 +\!\!+ l'(V := l\ V)$
by(*simp add*:*fun-eq-iff map-add-def*)
with *red-reds.BlockRedSome*[*OF IH′ - unass*] $l'V$ **show** *?case*
by(*simp add*:*map-add-def*)
next
qed (*simp-all add*:*red-reds.intros*)

lemma *Red-lcl-add*:
assumes $P,E \vdash \langle e,(h,l)\rangle \to* \langle e',(h',l')\rangle$ **shows** $P,E \vdash \langle e,(h,l_0 +\!\!+ l)\rangle \to* \langle e',(h',l_0 +\!\!+ l')\rangle$
using *assms*
proof(*induct rule*:*converse-rtrancl-induct-red*)
case *1* **thus** *?case* **by** *simp*
next
case *2* **thus** *?case*
by(*auto dest*: *red-lcl-add intro*: *converse-rtrancl-into-rtrancl simp*:*Red-def*)
qed

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

$\Longrightarrow \exists S'.\ h'\ a = Some(D,S')$
by (*induct rule*:*red-reds-inducts*) (*auto dest*:*new-Addr-SomeD*)

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*

$P \vdash Integer \leq T$
$P \vdash T \leq NT$

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

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

| *WTDynCast*:
⟦*P*,*E* ⊢ *e* :: *Class D*; *is-class P C*;
P ⊢ *Path D to C unique* ∨ (∀ *Cs*. ¬ *P* ⊢ *Path D to C via Cs*)⟧
⟹ *P*,*E* ⊢ *Cast C e* :: *Class C*

| *WTStaticCast*:
⟦*P*,*E* ⊢ *e* :: *Class D*; *is-class P C*;
P ⊢ *Path D to C unique* ∨
(*P* ⊢ *C* ≼* *D* ∧ (∀ *Cs*. *P* ⊢ *Path C to D via Cs* ⟶ *Subobjs*$_R$ *P C Cs*)) ⟧
⟹ *P*,*E* ⊢ (|*C*|)*e* :: *Class C*

| *WTVal*:
typeof v = *Some T* ⟹
P,*E* ⊢ *Val v* :: *T*

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

| *WTBinOp*:
⟦ *P*,*E* ⊢ *e*$_1$ :: *T*$_1$; *P*,*E* ⊢ *e*$_2$ :: *T*$_2$;
case bop of Eq ⇒ *T*$_1$ = *T*$_2$ ∧ *T* = *Boolean*
| *Add* ⇒ *T*$_1$ = *Integer* ∧ *T*$_2$ = *Integer* ∧ *T* = *Integer* ⟧

$\Longrightarrow P,E \vdash e_1 \ll bop \gg e_2 :: T$

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

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

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

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

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

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

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

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

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

— well-typed expression lists

| *WTNil*:

P,E ⊢ [] [::] []

| *WTCons*:
[⟦ P,E ⊢ e :: T; P,E ⊢ es [::] Ts ⟧
⟹ P,E ⊢ 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 ⊢ [] [::] Ts) = (Ts = [])

apply(*rule iffI*)
apply (*auto elim*: *WTs.cases*)
done

lemma [*iff*]: (P,E ⊢ e#es [::] T#Ts) = (P,E ⊢ e :: T ∧ P,E ⊢ es [::] Ts)

apply(*rule iffI*)
apply (*auto elim*: *WTs.cases*)
done

lemma [*iff*]: (P,E ⊢ (e#es) [::] Ts) =
(∃ U Us. Ts = U#Us ∧ P,E ⊢ e :: U ∧ P,E ⊢ es [::] Us)

apply(*rule iffI*)
apply (*auto elim*: *WTs.cases*)
done

lemma [*iff*]: ⋀Ts. (P,E ⊢ es_1 @ es_2 [::] Ts) =
(∃ Ts_1 Ts_2. Ts = Ts_1 @ Ts_2 ∧ P,E ⊢ es_1 [::] Ts_1 ∧ P,E ⊢ es_2[::]Ts_2)

apply(*induct* es_1 *type*:*list*)
apply *simp*
apply *clarsimp*
apply(*erule thin-rl*)
apply (*rule iffI*)
apply *clarsimp*
apply(*rule exI*)+
apply(*rule conjI*)
prefer *2* **apply** *blast*
apply *simp*

apply *fastforce*
done

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

apply(*rule iffI*)
apply (*auto elim*: *WT.cases*)
done

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

apply(*rule iffI*)
apply (*auto elim*: *WT.cases*)
done

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

apply(*rule iffI*)
apply (*auto elim*: *WT.cases*)
done

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

apply(*rule iffI*)
apply (*auto elim*: *WT.cases*)
done

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**
$P,E \vdash es\ [::]\ Ts \Longrightarrow (\bigwedge E'.\ E \subseteq_m E' \Longrightarrow P,E' \vdash es\ [::]\ Ts)$

apply(*induct rule*: *WT-WTs-inducts*)
apply(*simp add*: *WTNew*)
apply(*fastforce simp*: *WTDynCast*)
apply(*fastforce simp*: *WTStaticCast*)
apply(*fastforce simp*: *WTVal*)
apply(*simp add*: *WTVar map-le-def dom-def*)
apply(*fastforce simp*: *WTBinOp*)
apply(*force simp*:*map-le-def*)
apply(*fastforce simp*: *WTFAcc*)
apply(*fastforce simp*: *WTFAss*)
apply(*fastforce simp*: *WTCall*)
apply(*fastforce simp*: *WTStaticCall*)
apply(*fastforce simp*: *map-le-def WTBlock*)
apply(*fastforce simp*: *WTSeq*)
apply(*fastforce simp*: *WTCond*)
apply(*fastforce simp*: *WTWhile*)
apply(*fastforce simp*: *WTThrow*)
apply(*simp add*: *WTNil*)
apply(*simp add*: *WTCons*)
done

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

apply(*induct rule*:*WT-WTs.inducts*)
apply(*simp-all del*: *fun-upd-apply*)
apply *fast+*
done

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 ≡ λ(*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*)

apply (*unfold wf-prog-def class-def*)
apply (*fast dest*: *map-of-SomeD*)
done

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

apply (*simp add*: *wf-prog-def wf-syscls-def is-class-def class-def*)
apply (*fastforce intro*!: *map-of-SomeI*)
done

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* $\subseteq$ *types P*
proof
  from *assms* **have** *wf-mdecl wf-md P C* (*M*,*Ts*,*T*,*m*)
    by (*unfold wf-prog-def wf-cdecl-def*) *auto*
  hence $\forall\, t \in$ *set Ts*. *is-type P t* **by** (*unfold wf-mdecl-def*) *auto*
  moreover fix *t* **assume** *t* $\in$ *set Ts*
  ultimately have *is-type P t* **by** *blast*
  thus *t* $\in$ *types P* **..**
qed

17.2 Well-formedness subclass lemmas

lemma *subcls1-wfD*:
  $[\![$ *P* $\vdash$ *C* $\prec^1$ *D*; *wf-prog wf-md P* $]\!] \Longrightarrow$ *D* $\neq$ *C* $\wedge$ (*D*,*C*) $\notin$ (*subcls1 P*)$^+$

apply(*frule r-into-trancl*)
apply(*drule subcls1D*)
apply(*clarify*)
apply(*drule* (*1*) *class-wf*)
apply(*unfold wf-cdecl-def baseClasses-def*)
apply(*force simp add*: *reflcl-trancl* [*THEN sym*] *simp del*: *reflcl-trancl*)
done

lemma *wf-cdecl-supD*:
  $[\![$*wf-cdecl wf-md P* (*C*,*Bs*,*r*); *D* $\in$ *baseClasses Bs*$]\!] \Longrightarrow$ *is-class P D*
by (*auto simp*: *wf-cdecl-def baseClasses-def*)

lemma *subcls-asym*:
  $[\![$ *wf-prog wf-md P*; (*C*,*D*) $\in$ (*subcls1 P*)$^+$ $]\!] \Longrightarrow$ (*D*,*C*) $\notin$ (*subcls1 P*)$^+$

apply(*erule trancl.cases*)
apply(*fast dest*!: *subcls1-wfD*)
apply(*fast dest*!: *subcls1-wfD intro*: *trancl-trans*)
done

lemma *subcls-irrefl*:
  $[\![$ *wf-prog wf-md P*; (*C*,*D*) $\in$ (*subcls1 P*)$^+$ $]\!] \Longrightarrow$ *C* $\neq$ *D*

apply (*erule trancl-trans-induct*)
apply (*auto dest*: *subcls1-wfD subcls-asym*)
done

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

apply (*induct rule*:*rtrancl.induct*)
apply *simp*
apply (*drule rtrancl-into-trancl1*)
apply *simp*
apply (*drule subcls-asym*)
apply *simp*
apply(*drule rtranclD*)
apply *simp*
done

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

apply (*unfold acyclic-def*)
apply (*fast dest*: *subcls-irrefl*)
done

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

apply (*rule finite-acyclic-wf-converse*)
apply (*rule finite-subcls1*)
apply (*erule acyclic-subcls1*)
done

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 −
assume *p*: *PROP ?P*
assume *?A* **thus** *?thesis* **apply** −
apply(*drule wf-subcls1*)
apply(*drule wf-trancl*)
apply(*simp only*: *trancl-converse*)
apply(*erule-tac a* = *C* **in** *wf-induct*)
apply(*rule p*)
apply(*auto*)

done
qed

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

using *leq*
proof (*induct rule*:*leq-path1.induct*)
fix *Cs Ds* **assume** *suboCs*:*Subobjs P C Cs* **and** *suboDs*:*Subobjs P C Ds*
and *butlast*:$Cs = butlast\ Ds$
from *suboDs* **have** *notempty*:$Ds \neq []$ **by** $-$(*drule Subobjs-nonempty*)
with *butlast* **have** *DsCs*:$Ds = Cs\ @\ [last\ Ds]$ **by** *simp*
from *suboCs* **have** *notempty*:$Cs \neq []$ **by** $-$(*drule Subobjs-nonempty*)
with *DsCs* **have** $Ds = ((butlast\ Cs)\ @\ [last\ Cs])\ @\ [last\ Ds]$ **by** *simp*
with *suboDs* **have** *Subobjs P C* $((butlast\ Cs)\ @\ [last\ Cs,last\ Ds])$
by *simp*
thus $P \vdash last\ Cs \prec^1 last\ Ds$ **by** (*fastforce intro*:*subclsR-subcls1 Subobjs-subclsR*)
next
fix *Cs D* **assume** $P \vdash last\ Cs \prec_S D$
thus $P \vdash last\ Cs \prec^1 last\ [D]$ **by** (*fastforce intro*:*subclsS-subcls1*)
qed

lemma *last-leq-paths*:
assumes *leq*:$(Cs,Ds) \in (leq\text{-}path1\ P\ C)^+$ **and** *wf*:*wf-prog wf-md P*
shows $(last\ Cs,last\ Ds) \in (subcls1\ P)^+$

using *leq*
proof (*induct rule*:*trancl.induct*)
fix *Cs Ds* **assume** $P,C \vdash Cs \sqsubset^1 Ds$
thus $(last\ Cs,\ last\ Ds) \in (subcls1\ P)^+$ **using** *wf*
by (*fastforce intro*:*r-into-trancl elim*:*last-leq-path*)
next
fix *Cs Cs' Ds* **assume** $(last\ Cs,\ last\ Cs') \in (subcls1\ P)^+$
and $P,C \vdash Cs' \sqsubset^1 Ds$
thus $(last\ Cs,\ last\ Ds) \in (subcls1\ P)^+$ **using** *wf*
by (*fastforce dest*:*last-leq-path*)
qed

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

apply (*rule conjI*)
 apply (*erule leq-path1.cases*)
  apply *simp*
  apply (*drule-tac Cs=Ds* **in** *Subobjs-nonempty*)
  apply (*rule butlast-noteq*) **apply** *assumption*
 apply *clarsimp*
 apply (*drule subclsS-subcls1*)
 apply (*drule subcls1-wfD*) **apply** *simp-all*
apply *clarsimp*
apply (*frule last-leq-path*)
 apply *simp*
apply (*drule last-leq-paths*)
 apply *simp*
apply (*drule-tac r=subcls1 P* **in** *r-into-trancl*)
apply (*drule subcls-asym*)
apply *auto*
done

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

apply(*erule tranclE*)
apply(*fast dest*!:*leq-path1-wfD*)
apply(*fast dest*!:*leq-path1-wfD intro*: *trancl-trans*)
done

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

apply (*induct rule*:*rtrancl.induct*)
 apply *simp*
apply (*drule rtrancl-into-trancl1*)
 apply *simp*
apply (*drule leq-path-asym*)
 apply *simp*
apply (*drule-tac a=c* **and** *b=a* **in** *rtranclD*)
apply *simp*
done

lemma *leq-path-Subobjs*:
$[\![P,C \vdash [C] \sqsubseteq Cs;\ is\text{-}class\ P\ C;\ wf\text{-}prog\ wf\text{-}md\ P]\!] \Longrightarrow Subobjs\ P\ C\ Cs$
by (*induct rule*:*rtrancl-induct*,*auto intro*:*Subobjs-Base elim*!:*leq-path1.cases*,
auto dest!:*Subobjs-subclass intro*!:*Subobjs-Sh SubobjsR-Base dest*!:*subclsSD*

intro:*wf-cdecl-supD class-wf ShBaseclass-isBaseclass subclsSI*)

17.4 Lemmas concerning Subobjs

lemma *Subobj-last-isClass*:⟦*wf-prog wf-md P*; *Subobjs P C Cs*⟧ ⟹ *is-class P* (*last Cs*)

apply (*frule Subobjs-isClass*)
apply (*drule Subobjs-subclass*)
apply (*drule rtranclD*)
apply (*erule disjE*)
apply *simp*
apply *clarsimp*
apply (*erule trancl-induct*)
apply (*fastforce dest*:*subcls1D class-wf elim*:*wf-cdecl-supD*)
apply (*fastforce dest*:*subcls1D class-wf elim*:*wf-cdecl-supD*)
done

lemma *converse-SubobjsR-Rep*:
⟦*Subobjs*$_R$ *P C Cs*; *P* ⊢ *last Cs* ≺$_R$ *C'*; *wf-prog wf-md P*⟧
⟹ *Subobjs*$_R$ *P C* (*Cs*@[*C'*])

apply (*induct rule*:*Subobjs*$_R$.*induct*)
apply (*frule subclsR-subcls1*)
apply (*fastforce dest*!:*subcls1D class-wf wf-cdecl-supD SubobjsR-Base SubobjsR-Rep*)
apply (*fastforce elim*:*SubobjsR-Rep simp*: *SubobjsR-nonempty split*:*if-split-asm*)
done

lemma *converse-Subobjs-Rep*:
⟦*Subobjs P C Cs*; *P* ⊢ *last Cs* ≺$_R$ *C'*; *wf-prog wf-md P*⟧
⟹ *Subobjs P C* (*Cs*@[*C'*])
by (*induct rule*:*Subobjs.induct*, *fastforce dest*:*converse-SubobjsR-Rep Subobjs-Rep*,

fastforce dest:*converse-SubobjsR-Rep Subobjs-Sh*)

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'*)
using *subo*
proof (*induct Cs'*)
case *Nil*
show *?case*
proof (*cases C*=*C'*)

```
    case True
    have is-subobj P ((C,C'#rev [])) by fact
    with True have is-subobj P ((C,[C])) by simp
    hence is-class P C
      by (fastforce elim:converse-rtranclE dest:subclsS-subcls1 elim:subcls1-class)
    with True show ?thesis by (fastforce intro:Subobjs-Base)
  next
    case False
    have is-subobj P ((C,C'#rev [])) by fact
    with False obtain D where sup:P ⊢ C ≼* D and subS:P ⊢ D ≺S C'
      by fastforce
    with wf have is-class P C'
      by (fastforce dest:subclsS-subcls1 subcls1D class-wf elim:wf-cdecl-supD)
    hence Subobjs_R P C' [C'] by (fastforce elim:SubobjsR-Base)
    with sup subS have Subobjs P C [C'] by −(erule Subobjs-Sh, simp)
    thus ?thesis by simp
  qed
next
  case (Cons C'' Cs'')
  have IH:is-subobj P ((C,C'#rev Cs'')) ⟹ Subobjs P C (C'#rev Cs'')
    and subo:is-subobj P ((C,C'#rev(C''# Cs''))) by fact+
  obtain Ds' where Ds':Ds' = rev Cs'' by simp
  obtain D Ds where DDs:D#Ds = Ds'@[C''] by (cases Ds') auto
  with Ds' subo have is-subobj P ((C,C'#D#Ds)) by simp
  hence subobl:is-subobj P ((C,butlast(C'#D#Ds)))
    and subRbl:P ⊢ last(butlast(C'#D#Ds)) ≺R last(C'#D#Ds) by simp+
 with DDs Ds' have is-subobj P ((C,C'#rev Cs'')) by (simp del: is-subobj.simps)
  with IH have suborev:Subobjs P C (C'#rev Cs'') by simp
  from subRbl DDs Ds' have subR:P ⊢ last(C'#rev Cs'') ≺R C'' by simp
  with suborev wf show ?case by (fastforce dest:converse-Subobjs-Rep)
qed



lemma isSubobj-Subobjs:
assumes subo:is-subobj P ((C,Cs)) and wf:wf-prog wf-md P
shows Subobjs P C Cs

using subo
proof (induct Cs)
  case Nil
  thus ?case by simp
next
  case (Cons C' Cs')
  have subo:is-subobj P ((C,C'#Cs')) by fact
  obtain Cs'' where Cs'':Cs'' = rev Cs' by simp
  with subo have is-subobj P ((C,C'#rev Cs'')) by simp
  with wf have Subobjs P C (C'#rev Cs'') by − (rule isSubobj-Subobjs-rev)
  with Cs'' show ?case by simp
```

qed

lemma *isSubobj-eq-Subobjs*:
wf-prog wf-md P $\Longrightarrow$ *is-subobj P* ((*C*,*Cs*)) = (*Subobjs P C Cs*)
by(*auto elim*:*isSubobj-Subobjs Subobjs-isSubobj*)

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

using *subo*
proof (*induct Cs*′)
case *Nil*
thus *?case* **by** *simp*
next
case (*Cons D Ds*)
have *IH*:*Subobjs P C* (*Cs* @ *C*′ # *rev Ds*) $\Longrightarrow$
$\forall$ *C*′′$\in$*set Ds*. (*C*′, *C*′′) $\in$ (*subcls1 P*)$^+$
and *Subobjs P C* (*Cs* @ *C*′ # *rev* (*D* # *Ds*)) **by** *fact*+
hence *subo*′:*Subobjs P C* (*Cs*@ *C*′#*rev Ds* @ [*D*]) **by** *simp*
hence *Subobjs P C* (*Cs*@ *C*′#*rev Ds*)
by −(*rule appendSubobj*,*simp-all*)
with *IH* **have** *set*:$\forall$ *C*′′$\in$*set Ds*. (*C*′, *C*′′) $\in$ (*subcls1 P*)$^+$ **by** *simp*
hence *revset*:$\forall$ *C*′′$\in$*set* (*rev Ds*). (*C*′, *C*′′) $\in$ (*subcls1 P*)$^+$ **by** *simp*
have (*C*′,*D*) $\in$ (*subcls1 P*)$^+$
proof (*cases Ds* = [])
case *True*
with *subo*′ **have** *Subobjs P C* (*Cs*@[*C*′,*D*]) **by** *simp*
thus *?thesis*
by (*fastforce intro*: *subclsR-subcls1 Subobjs-subclsR*)
next
case *False*
with *revset* **have** *hd*:(*C*′,*hd Ds*) $\in$ (*subcls1 P*)$^+$
apply −
apply (*erule ballE*)
apply *simp*
apply (*simp add*:*in-set-conv-decomp*)
apply (*erule-tac x*=[] **in** *allE*)
apply (*erule-tac x*=*tl Ds* **in** *allE*)
apply *simp*
done
from *False subo*′ **have** (*hd Ds*,*D*) $\in$ (*subcls1 P*)$^+$
apply (*cases Ds*)
apply *simp*
apply *simp*

```
    apply (rule r-into-trancl)
    apply (rule subclsR-subcls1)
    apply (rule-tac Cs=Cs @ C' # rev list in Subobjs-subclsR)
    apply simp
    done
  with hd show ?thesis by (rule trancl-trans)
 qed
 with set show ?case by simp
qed



lemma unique1:
 assumes subo:Subobjs P C (Cs@ C'#Cs') and wf:wf-prog wf-md P
 shows C' ∉ set Cs'

proof −
 obtain Ds where Ds:Ds = rev Cs' by simp
 with subo have Subobjs P C (Cs@ C'#rev Ds) by simp
 with Ds subo have ∀ C'' ∈ set Cs'. (C',C'') ∈ (subcls1 P)^+
  by (fastforce dest:subo-trans-subcls)
 with wf have ∀ C'' ∈ set Cs'. C' ≠ C''
  by (auto dest:subcls-irrefl)
 thus ?thesis by fastforce
qed



lemma subo-subcls-trans:
 assumes subo:Subobjs P C (Cs@ C'#Cs')
 shows ∀ C'' ∈ set Cs. (C'',C') ∈ (subcls1 P)^+

proof −
 from wf subo have ⋀C''. C'' ∈ set Cs ⟹ (C'',C') ∈ (subcls1 P)^+
  apply (auto simp:in-set-conv-decomp)
  apply (case-tac zs)
   apply (fastforce intro: subclsR-subcls1 Subobjs-subclsR)
  apply simp
  apply (rule-tac b=a in trancl-rtrancl-trancl)
   apply (fastforce intro: subclsR-subcls1 Subobjs-subclsR)
  apply (subgoal-tac P ⊢ a ≼* last (a # list @ [C']))
   apply simp
  apply (rule Subobjs-subclass)
  apply (rule-tac C=C and Cs= ys @[C''] in Subobjs-Subobjs)
  apply (rule-tac Cs'=Cs' in appendSubobj)
  apply simp-all
  done
 thus ?thesis by fastforce
qed
```

lemma *unique2*:
assumes *subo*:*Subobjs P C* (*Cs*@ *C*′#*Cs*′) **and** *wf*:*wf-prog wf-md P*
shows $C' \notin set\ Cs$

proof −
from *subo wf* **have** $\forall\, C'' \in set\ Cs.\ (C'', C') \in (subcls1\ P)^+$
by (*fastforce dest*:*subo-subcls-trans*)
with *wf* **have** $\forall\, C'' \in set\ Cs.\ C' \neq C''$
by (*auto dest*:*subcls-irrefl*)
thus *?thesis* **by** *fastforce*
qed

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 −
from *subo set* **obtain** *Ds Ds*′ **where** *Cs*:$Cs = Ds @ C \# Ds'$
by (*auto simp*:*in-set-conv-decomp*)
then obtain *Cs*′ **where** *Cs*′:$Cs' = rev\ Ds$ **by** *simp*
with *Cs subo* **have** *subo*′:*Subobjs P C* ((*rev Cs*′)@ *C*#*Ds*′) **by** *simp*
thus *?thesis*
proof (*cases Cs*′)
case *Nil*
with *Cs Cs*′ **show** *?thesis* **by** *simp*
next
case (*Cons X Xs*)
with *subo*′ **have** *suboX*:*Subobjs P C* ((*rev Xs*)@[*X*,*C*]@*Ds*′) **by** *simp*
hence *leq*:$P \vdash X \prec^1 C$
by (*fastforce intro*:*subclsR-subcls1 Subobjs-subclsR*)
from *suboX wf* **have** $P \vdash C \preceq^* last\ ((rev\ Xs)@[X])$
by (*fastforce intro*:*Subobjs-subclass appendSubobj*)
with *leq* **have** $(C, C) \in (subcls1\ P)^+$ **by** *simp*
with *wf* **show** *?thesis* **by** (*fastforce dest*:*subcls-irrefl*)
qed
qed

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 −
  from subo have notempty:Cs ≠ [] by − (drule Subobjs-nonempty)
  hence lastset:last Cs ∈ set Cs
    apply (auto simp add:in-set-conv-decomp)
    apply (rule-tac x=butlast Cs in exI)
    apply (rule-tac x=[] in exI)
    apply simp
    done
  with last have C:C ∈ set Cs by simp
  with subo wf have hd:C = hd Cs by −(rule mdc-hd-path)
  then obtain Cs' where Cs':Cs' = tl Cs by simp
  thus ?thesis
  proof (cases Cs')
    case Nil
   with hd subo Cs' show ?thesis by (fastforce dest:Subobjs-nonempty hd-Cons-tl)
  next
    case (Cons D Ds)
    with Cs' hd notempty have Cs:Cs=C#D#Ds by simp
    with subo have Subobjs P C (C#D#Ds) by simp
   with wf have notset:C ∉ set (D#Ds) by −(rule-tac Cs=[] in unique1,simp-all)
    from Cs last have last Cs = last (D#Ds) by simp
    hence last Cs ∈ set (D#Ds)
      apply (auto simp add:in-set-conv-decomp)
      apply (erule-tac x=butlast Ds in allE)
      apply (erule-tac x=[] in allE)
      apply simp
      done
    with last have C ∈ set (D#Ds) by simp
    with notset show ?thesis by simp
  qed
qed



lemma assumes leq:P ⊢ C ≼* D and wf:wf-prog wf-md P
  shows subcls-leq-path:∃ Cs. P,C ⊢ [C] ⊑ Cs@[D]

using leq
proof (induct rule:rtrancl.induct)
  fix C show ∃ Cs. P,C ⊢ [C] ⊑ Cs@[C] by (rule-tac x=[] in exI,simp)
next
  fix C C' D assume leq':P ⊢ C ≼* C' and IH:∃ Cs. P,C ⊢ [C] ⊑ Cs@[C']
    and sub:P ⊢ C' ≺1 D
  from sub have is-class P C' by (rule subcls1-class)
  with leq' have class: is-class P C by (rule subcls-is-class)
  from IH obtain Cs where steps:P,C ⊢ [C] ⊑ Cs@[C'] by auto
  hence subo:Subobjs P C (Cs@[C']) using class wf
    by (fastforce intro:leq-path-Subobjs)
  { assume P ⊢ C' ≺R D
```

```
    with subo wf have Subobjs P C (Cs@[C',D])
      by (fastforce dest:converse-Subobjs-Rep)
    with subo have P,C ⊢ (Cs@[C']) ⊏¹ (Cs@[C']@[D])
      by (fastforce intro:leq-path-rep) }
  moreover
  { assume P ⊢ C' ≺_S D
    with subo have P,C ⊢ (Cs@[C']) ⊏¹ [D] by (rule leq-path-sh) }
  ultimately show ∃ Cs. P,C ⊢ [C] ⊑ Cs@[D] using sub steps
    apply (auto dest!:subcls1-subclsR-or-subclsS)
    apply (rule-tac x=Cs@[C'] in exI) apply simp
    apply (rule-tac x=[] in exI) apply simp
    done
qed



lemma assumes subo:Subobjs P C (rev Cs) and wf:wf-prog wf-md P
  shows subobjs-rel-rev:P,C ⊢ [C] ⊑ (rev Cs)
using subo
proof (induct Cs)
  case Nil
  thus ?case by (fastforce dest:Subobjs-nonempty)
next
  case (Cons C' Cs')
  have subo':Subobjs P C (rev (C'#Cs'))
    and IH:Subobjs P C (rev Cs') ⟹ P,C ⊢ [C] ⊑ rev Cs' by fact+
  from subo' have class: is-class P C by(rule Subobjs-isClass)
  show ?case
  proof (cases Cs' = [])
    case True hence empty:Cs' = [] .
    with subo' have subo'':Subobjs P C [C'] by simp
    thus ?thesis
    proof (cases C = C')
      case True
      with empty show ?thesis by simp
    next
      case False
      with subo'' obtain D D' where leq:P ⊢ C ≼* D and subS:P ⊢ D ≺_S D'
        and suboR:Subobjs_R P D' [C']
        by (auto elim:Subobjs.cases dest:hd-SubobjsR)
      from suboR have C':C' = D' by (fastforce dest:hd-SubobjsR)
      from leq wf obtain Ds where steps:P,C ⊢ [C] ⊑ Ds@[D]
        by (auto dest:subcls-leq-path)
      hence suboSteps:Subobjs P C (Ds@[D]) using class wf
        apply (induct rule:rtrancl-induct)
         apply (erule Subobjs-Base)
        apply (auto elim!:leq-path1.cases)
        apply (subgoal-tac Subobjs_R P D [D])
```

```
      apply (fastforce dest:Subobjs-subclass intro:Subobjs-Sh)
      apply (fastforce dest!:subclsSD intro:SubobjsR-Base wf-cdecl-supD
                                             class-wf ShBaseclass-isBaseclass)
      done
    hence step:P,C ⊢ (Ds@[D]) ⊏¹ [D'] using subS by (rule leq-path-sh)
    with steps empty False C' show ?thesis by simp
  qed
 next
  case False
  with subo' have subo'':Subobjs P C (rev Cs')
   by (fastforce intro:butlast-Subobjs)
  with IH have steps:P,C ⊢ [C] ⊑ rev Cs' by simp
  from subo' subo'' have P,C ⊢ rev Cs' ⊏¹ rev (C'#Cs')
   by (fastforce intro:leq-pathRep)
  with steps show ?thesis by simp
 qed
qed



lemma subobjs-rel:
assumes subo:Subobjs P C Cs and wf:wf-prog wf-md P
shows P,C ⊢ [C] ⊑ Cs

proof −
 obtain Cs' where Cs':Cs' = rev Cs by simp
 with subo have Subobjs P C (rev Cs') by simp
 hence P,C ⊢ [C] ⊑ rev Cs' using wf by (rule subobjs-rel-rev)
 with Cs' show ?thesis by simp
qed



lemma assumes wf:wf-prog wf-md P
 shows leq-path-last:⟦P,C ⊢ Cs ⊑ Cs'; last Cs = last Cs'⟧ ⟹ Cs = Cs'

proof(induct rule:rtrancl-induct)
 show Cs = Cs by simp
next
 fix Cs' Cs''
 assume leqs:P,C ⊢ Cs ⊑ Cs' and leq:P,C ⊢ Cs' ⊏¹ Cs''
   and last:last Cs = last Cs''
   and IH:last Cs = last Cs' ⟹ Cs = Cs'
 from leq wf have sup1:P ⊢ last Cs' ≺¹ last Cs''
   by(rule last-leq-path)
 { assume Cs = Cs'
   with last have eq:last Cs'' = last Cs' by simp
   with eq wf sup1 have Cs = Cs'' by(fastforce dest:subcls1-wfD) }
 moreover
```

{ assume (*Cs*,*Cs'*) ∈ (*leq-path1 P C*)$^+$
hence *sub*:(*last Cs*,*last Cs'*) ∈ (*subcls1 P*)$^+$ **using** *wf*
by(*rule last-leq-paths*)
with *sup1 last* **have** (*last Cs''*,*last Cs''*) ∈ (*subcls1 P*)$^+$ **by** *simp*
with *wf* **have** *Cs* = *Cs''* **by**(*fastforce dest*:*subcls-irrefl*) **}**
ultimately show *Cs* = *Cs''* **using** *leqs*
by(*fastforce dest*:*rtranclD*)
qed

17.5 Well-formedness and appendPath

lemma *appendPath1*:
⟦*Subobjs P C Cs*; *Subobjs P* (*last Cs*) *Ds*; *last Cs* ≠ *hd Ds*⟧
⟹ *Subobjs P C Ds*

apply(*subgoal-tac* ¬ *Subobjs*$_R$ *P* (*last Cs*) *Ds*)
apply (*subgoal-tac* ∃ *C' D*. *P* ⊢ *last Cs* ≼* *C'* ∧ *P* ⊢ *C'* ≺$_S$ *D* ∧ *Subobjs*$_R$ *P D Ds*)
apply *clarsimp*
apply (*drule Subobjs-subclass*)
apply (*subgoal-tac P* ⊢ *C* ≼* *C'*)
apply (*erule-tac C'*=*C'* **and** *D*=*D* **in** *Subobjs-Sh*)
apply *simp*
apply *simp*
apply *fastforce*
apply (*erule Subobjs-notSubobjsR*)
apply *simp*
apply (*fastforce dest*:*hd-SubobjsR*)
done

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*)))
using *subo2*
proof (*induct Ds*)
case *Nil*
with *subo1* **show** *?case* **by** *simp*
next
case (*Cons D' Ds'*)
have *IH*:*Subobjs P* (*last Cs*) (*last Cs*#*rev Ds'*)
⟹ *Subobjs P C* (*Cs*@*tl*(*last Cs*#*rev Ds'*))
and *subo*:*Subobjs P* (*last Cs*) (*last Cs*#*rev* (*D'*#*Ds'*)) **by** *fact*+
from *subo* **have** *Subobjs P* (*last Cs*) (*last Cs*#*rev Ds'*)
by (*fastforce intro*:*butlast-Subobjs*)
with *IH* **have** *subo'*:*Subobjs P C* (*Cs*@*tl*(*last Cs*#*rev Ds'*))

by *simp*
have *last*:*last*(*last Cs*#*rev Ds'*) = *last* (*Cs*@*tl*(*last Cs*#*rev Ds'*))
by (*cases Ds'*)*auto*
obtain *C' Cs'* **where** *C'*:*C'* = *last*(*last Cs*#*rev Ds'*) **and**
Cs' = *butlast*(*last Cs*#*rev Ds'*) **by** *simp*
then have *Cs'* @ [*C'*] = *last Cs* # *rev Ds'*
using *append-butlast-last-id* **by** *blast*
hence *last Cs*#*rev* (*D'*#*Ds'*) = *Cs'*@[*C'*,*D'*] **by** *simp*
with *subo* **have** *Subobjs P* (*last Cs*) (*Cs'*@[*C'*,*D'*]) **by** (*cases Cs'*) *auto*
hence *P* ⊢ *C'* ≺$_R$ *D'* **by** − (*rule Subobjs-subclsR*,*simp*)
with *C' last* **have** *P* ⊢ *last* (*Cs*@*tl*(*last Cs*#*rev Ds'*)) ≺$_R$ *D'* **by** *simp*
with *subo' wf* **have** *Subobjs P C* ((*Cs*@*tl*(*last Cs*#*rev Ds'*))@[*D'*])
by (*erule-tac Cs*=(*Cs*@*tl*(*last Cs*#*rev Ds'*)) **in** *converse-Subobjs-Rep*) *simp*
thus *?case* **by** *simp*
qed

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*))

using *subo2*
proof (*cases Ds*)
case *Nil*
with *subo1* **show** *?thesis* **by** *simp*
next
case (*Cons D' Ds'*)
with *subo2 eq* **have** *subo*:*Subobjs P* (*last Cs*) (*last Cs*#*Ds'*) **by** *simp*
obtain *Ds''* **where** *Ds''*:*Ds''* = *rev Ds'* **by** *simp*
with *subo* **have** *Subobjs P* (*last Cs*) (*last Cs*#*rev Ds''*) **by** *simp*
with *subo1 wf* **have** *Subobjs P C* (*Cs*@(*tl* (*last Cs*#*rev Ds''*)))
by −(*rule appendPath2-rev*)
with *Ds''* *eq Cons* **show** *?thesis* **by** *simp*
qed

lemma *Subobjs-appendPath*:
⟦*Subobjs P C Cs*; *Subobjs P* (*last Cs*) *Ds*;*wf-prog wf-md P*⟧
⟹ *Subobjs P C* (*Cs*@$_p$*Ds*)
by(*fastforce elim*:*appendPath2 appendPath1 simp*:*appendPath-def*)

17.6 Path and program size

lemma assumes *subo*:*Subobjs P C Cs* **and** *wf*:*wf-prog wf-md P*
shows *path-contains-classes*:∀ *C'* ∈ *set Cs*. *is-class P C'*
using *subo*

proof *clarsimp*
fix C' **assume** *subo*:*Subobjs P C Cs* **and** *set*:$C' \in$ *set Cs*
from *set* **obtain** *Ds Ds'* **where** *Cs*:$Cs = Ds@C'\#Ds'$
by (*fastforce simp*:*in-set-conv-decomp*)
with *Cs* **show** *is-class P C'*
proof (*cases Ds* = [])
case *True*
with *Cs subo* **have** *subo'*:*Subobjs P C* (*C'*#*Ds'*) **by** *simp*
thus *?thesis* **by** (*rule Subobjs.cases*,
auto dest:*hd-SubobjsR intro*:*SubobjsR-isClass*)
next
case *False*
then obtain *C'' Cs''* **where** *Cs''*:$Cs'' =$ *butlast Ds*
and *last*:$C'' =$ *last Ds* **by** *auto*
with *False* **have** *Ds*:$Ds = Cs''@[C'']$ **by** *simp*
with *Cs subo* **have** *subo'*:*Subobjs P C* (*Cs''*@[*C''*,*C'*]@*Ds'*)
by *simp*
hence $P \vdash C'' \prec_R C'$ **by**(*fastforce intro*:*isSubobjs-subclsR Subobjs-isSubobj*)
with *wf* **show** *?thesis*
by (*fastforce dest*!:*subclsRD*
intro:*wf-cdecl-supD class-wf RepBaseclass-isBaseclass subclsSI*)
qed
qed

lemma *path-subset-classes*:⟦*Subobjs P C Cs*; *wf-prog wf-md P*⟧
$\Longrightarrow$ *set Cs* $\subseteq$ {*C*. *is-class P C*}
by (*auto dest*:*path-contains-classes*)

lemma assumes *subo*:*Subobjs P C* (*rev Cs*) **and** *wf*:*wf-prog wf-md P*
shows *rev-path-distinct-classes*:*distinct Cs*
using *subo*
proof (*induct Cs*)
case *Nil* **thus** *?case* **by**(*fastforce dest*:*Subobjs-nonempty*)
next
case (*Cons C' Cs'*)
have *subo'*:*Subobjs P C* (*rev*(*C'*#*Cs'*))
and *IH*:*Subobjs P C* (*rev Cs'*) $\Longrightarrow$ *distinct Cs'* **by** *fact+*
show *?case*
proof (*cases Cs'* = [])
case *True* **thus** *?thesis* **by** *simp*
next
case *False*
hence *rev*:*rev Cs'* $\neq$ [] **by** *simp*
from *subo'* **have** *subo''*:*Subobjs P C* (*rev Cs'*@[*C'*]) **by** *simp*
hence *Subobjs P C* (*rev Cs'*) **using** *rev wf*
by(*fastforce dest*:*appendSubobj*)

```
    with IH have dist:distinct Cs' by simp
    from subo'' wf have C' ∉ set (rev Cs')
      by(fastforce dest:unique2)
    with dist show ?thesis by simp
  qed
qed


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

proof −
  obtain Cs' where Cs':Cs' = rev Cs by simp
  with subo have Subobjs P C (rev Cs') by simp
  with wf have distinct Cs'
    by −(rule rev-path-distinct-classes)
  with Cs' show ?thesis by simp
qed



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

proof −
  from wf have dist-fst:distinct-fst P by (simp add:wf-prog-def)
  hence distinct P by (simp add:distinct-fst-def,induct P,auto)
  hence card-set:card (set P) = length P by (rule distinct-card)
  from dist-fst have set:{C. is-class P C} = fst ‘ (set P)
    by (simp add:is-class-def class-def,auto simp:distinct-fst-def,
      auto dest:map-of-eq-Some-iff intro!:image-eqI)
  from dist-fst have card(fst ‘ (set P)) = card (set P)
    by(auto intro:card-image simp:distinct-map distinct-fst-def)
  with card-set set show ?thesis by simp
qed



lemma assumes subo:Subobjs P C Cs and wf:wf-prog wf-md P
  shows path-length:length Cs ≤ length P

proof −
  from subo wf have distinct Cs by (rule path-distinct-classes)
  hence card-eq-length:card (set Cs) = length Cs by (rule distinct-card)
  from subo wf have card (set Cs) ≤ card {C. is-class P C}
    by (auto dest:path-subset-classes intro:card-mono finite-is-class)
  with card-eq-length have length Cs ≤ card {C. is-class P C} by simp
  with wf show ?thesis by(fastforce dest:prog-length)
```

qed

lemma *empty-path-empty-set*:$\{Cs.\ Subobjs\ P\ C\ Cs \wedge length\ Cs \leq 0\} = \{\}$
by (*auto dest*:*Subobjs-nonempty*)

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

lemma *empty-list-set*:$\{xs.\ set\ xs \subseteq F \wedge xs = []\} = \{[]\}$
by *auto*

lemma *suc-n-union-of-union*:$\{xs.\ set\ xs \subseteq F \wedge length\ xs = Suc\ n\} = (UN\ x{:}F.\ UN\ xs : \{xs.\ set\ xs \leq F \wedge length\ xs = n\}.\ \{x\#xs\})$
by (*auto simp*:*length-Suc-conv*)

lemma *max-length-finite-set*:*finite F* $\Longrightarrow$ *finite*$\{xs.\ set\ xs <= F \wedge length\ xs = n\}$
by(*induct n*,*simp add*:*empty-list-set*, *simp add*:*suc-n-union-of-union*)

lemma *path-length-n-finite-set*:
wf-prog wf-md P $\Longrightarrow$ *finite*$\{Cs.\ Subobjs\ P\ C\ Cs \wedge length\ Cs = n\}$
by (*rule-tac B*=$\{Cs.\ set\ Cs <= \{C.\ is\text{-}class\ P\ C\} \wedge length\ Cs = n\}$ **in** *finite-subset*,
auto dest:*path-contains-classes intro*:*max-length-finite-set simp*:*finite-is-class*)

lemma *path-finite-leq*:
wf-prog wf-md P $\Longrightarrow$ *finite*$\{Cs.\ Subobjs\ P\ C\ Cs \wedge length\ Cs \leq length\ P\}$
by (*induct* (*length P*), *simp only*:*empty-path-empty-set*,
auto intro:*path-length-n-finite-set simp*:*split-set-path-length*)

lemma *path-finite*:*wf-prog wf-md P* $\Longrightarrow$ *finite*$\{Cs.\ Subobjs\ P\ C\ Cs\}$
by (*subgoal-tac* $\{Cs.\ Subobjs\ P\ C\ Cs\} =$
$\{Cs.\ Subobjs\ P\ C\ Cs \wedge length\ Cs \leq length\ P\}$,
auto intro:*path-finite-leq path-length*)

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 −
from *path-via* **have** *subo*:*Subobjs P C Cs* **and** *last*:*last Cs* = *D*
by(*simp add*:*path-via-def*)+
hence *leq*:$P \vdash C \preceq^* D$ **by**(*fastforce dest*:*Subobjs-subclass*)
{ fix Cs' **assume** $P \vdash Path\ D\ to\ C\ via\ Cs'$
hence *subo'*:*Subobjs P D Cs'* **and** *last'*:*last Cs'* = *C*
by(*simp add*:*path-via-def*)+

hence *leq'*:$P \vdash D \preceq^* C$ **by**(*fastforce dest*:*Subobjs-subclass*)
with *leq wf* **have** *CeqD*:$C = D$ **by**(*rule subcls-asym2*)
moreover have *Cs*:$Cs = [C]$ **using** *CeqD subo last wf* **by**(*fastforce intro*:*mdc-eq-last*)
moreover have $Cs' = [C]$ **using** *CeqD subo' last' wf* **by**(*fastforce intro*:*mdc-eq-last*)
ultimately have $Cs = [C] \wedge Cs' = [C] \wedge C = D$ **by** *simp* **}**
thus *?thesis* **by** *blast*
qed

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 $Cs' = [hd\ Cs]$

using *path*
proof −
from *path notemptyCs last* **have** *path2*:$P,C \vdash Cs \sqsubseteq Cs'@\ tl\ Cs$
by (*simp add*:*appendPath-def*)
thus *?thesis*
proof (*auto dest*!:*rtranclD*)
assume $Cs = Cs'@\ tl\ Cs$
with *notemptyCs* **show** $Cs' = [hd\ Cs]$ **by** (*rule app-hd-tl*)
next
assume *trancl*:$(Cs, Cs'@\ tl\ Cs) \in (leq\text{-}path1\ P\ C)^+$
from *notemptyCs' last* **have** *butlastLast*:$Cs' = butlast\ Cs' @ [hd\ Cs]$
by −(*drule append-butlast-last-id*,*simp*)
with *trancl* **have** *trancl'*:$(Cs, (butlast\ Cs' @ [hd\ Cs]) @\ tl\ Cs) \in (leq\text{-}path1\ P\ C)^+$
by *simp*
from *notemptyCs* **have** $(butlast\ Cs' @ [hd\ Cs]) @\ tl\ Cs = butlast\ Cs' @ Cs$
by *simp*
with *trancl'* **have** $(Cs, butlast\ Cs' @ Cs) \in (leq\text{-}path1\ P\ C)^+$ **by** *simp*
hence $(last\ Cs, last\ (butlast\ Cs' @ Cs)) \in (subcls1\ P)^+$ **using** *wf*
by (*rule last-leq-paths*)
with *notemptyCs* **have** $(last\ Cs, last\ Cs) \in (subcls1\ P)^+$
by −(*drule-tac xs*=*butlast Cs'* **in** *last-appendR*,*simp*)
with *wf* **show** *?thesis* **by** (*auto dest*:*subcls-irrefl*)
qed
qed

lemma *path-via-C*: $[\![P \vdash Path\ C\ to\ C\ via\ Cs;\ wf\text{-}prog\ wf\text{-}md\ P]\!] \Longrightarrow Cs = [C]$
by (*fastforce intro*:*mdc-eq-last simp*:*path-via-def*)

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

proof −
from *path-via* **have** *notempty*:$Cs' \neq []$
by(*fastforce intro*!:*Subobjs-nonempty simp*:*path-via-def*)
{ assume *eq*:*last Cs* = *hd Cs'*
and *Cs*:$Cs = Cs@tl\ Cs'$
from *Cs* **have** *tl Cs'* = [] **by** *simp*
with *eq notempty* **have** *Cs'* = [*last Cs*]
by −(*drule hd-Cons-tl*,*simp*) **}**
moreover
{ assume *Cs* = *Cs'*
with *wf path-via* **have** *Cs'* = [*last Cs*]
by(*fastforce intro*:*mdc-eq-last simp*:*path-via-def*) **}**
ultimately have *eq*:*Cs'* = [*last Cs*] **using** *appendPath*
by(*simp add*:*appendPath-def*,*split if-split-asm*,*simp-all*)
with *path-via* **have** *C* = *last Cs*
by(*simp add*:*path-via-def*)
with *wf path-via'* **have** *Cs''* = [*last Cs*]
by *simp*(*rule path-via-C*)
thus *?thesis* **by** (*simp add*:*appendPath-def*)
qed

lemma *subo-no-path*:
assumes *subo*:*Subobjs P C'* (*Cs* @ *C*#*Cs'*) **and** *wf*:*wf-prog wf-md P*
and *notempty*:$Cs' \neq []$
shows ¬ *P* ⊢ *Path last Cs' to C via Ds*

proof
assume *P* ⊢ *Path last Cs' to C via Ds*
hence *subo'*:*Subobjs P* (*last Cs'*) *Ds* **and** *last*:*last Ds* = *C*
by (*auto simp*:*path-via-def*)
hence *notemptyDs*:$Ds \neq []$ **by** −(*drule Subobjs-nonempty*)
then obtain *D' Ds'* **where** *D'Ds'*:$Ds = D'\#Ds'$ **by**(*cases Ds*)*auto*
from *subo* **have** *suboC*:*Subobjs P C* (*C*#*Cs'*) **by** (*rule Subobjs-Subobjs*)
with *wf subo' notempty* **have** *suboapp*:*Subobjs P C* ((*C*#*Cs'*)@$_p$*Ds*)
by −(*rule Subobjs-appendPath*,*simp-all*)
with *notemptyDs last* **have** *last'*:*last* ((*C*#*Cs'*)@$_p$*Ds*) = *C*
by −(*drule-tac Cs'*=(*C*#*Cs'*) **in** *appendPath-last*,*simp*)
from *notemptyDs* **have** (*C*#*Cs'*)@$_p$*Ds* ≠ []
by (*simp add*:*appendPath-def*)
with *last'* **have** *C* ∈ *set* ((*C*#*Cs'*)@$_p$*Ds*)
apply (*auto simp add*:*in-set-conv-decomp*)
apply (*rule-tac x*=*butlast*((*C*#*Cs'*)@$_p$*Ds*) **in** *exI*)
apply (*rule-tac x*=[] **in** *exI*)
apply (*drule append-butlast-last-id*)

```
  apply simp
  done
with suboapp wf have hd:C = hd ((C#Cs')@_pDs) by -(rule  mdc-hd-path)
thus False
proof (cases last (C#Cs') = hd Ds)
  case True
  hence eq:(C#Cs')@_pDs = (C#Cs')@(tl Ds) by (simp add:appendPath-def)
  show ?thesis
  proof (cases Ds')
    case Nil
    with D'Ds' have Ds:Ds = [D'] by simp
    with last have C = D' by simp
    with True notempty Ds have last (C#Cs') = C by simp
    with notempty have last Cs' = C by simp
    with notempty have Cset:C ∈ set Cs'
      apply (auto simp add:in-set-conv-decomp)
      apply (rule-tac x=butlast Cs' in exI)
      apply (rule-tac x=[] in exI)
      apply (drule append-butlast-last-id)
      apply simp
      done
    from subo wf have C ∉ set Cs' by (rule unique1)
    with Cset show ?thesis by simp
  next
    case (Cons X Xs)
    with D'Ds' have tlnotempty:tl Ds ≠ [] by simp
    with Cons last D'Ds' have last (tl Ds) = C by simp
    with tlnotempty have C ∈ set (tl Ds)
      apply (auto simp add:in-set-conv-decomp)
      apply (rule-tac x=butlast (tl Ds) in exI)
      apply (rule-tac x=[] in exI)
      apply (drule append-butlast-last-id)
      apply simp
      done
    hence Cset:C ∈ set (Cs'@(tl Ds)) by simp
    from suboapp eq wf have C ∉ set (Cs'@(tl Ds))
      by (subgoal-tac Subobjs P C (C#(Cs'@(tl Ds))),
        rule-tac Cs=[] in unique1,simp-all)
    with Cset show ?thesis by simp
  qed
next
  case False
 with notemptyDs have eq:(C#Cs')@_pDs = Ds by (simp add:appendPath-def)
  with subo' last have lastleq:P ⊢ last Cs' ≼* C
    by (fastforce dest:Subobjs-subclass)
  from notempty obtain X Xs where X:X = last Cs' and Xs = butlast Cs'
    by auto
  with notempty have XXs:Cs' = Xs@[X] by simp
  hence CleqX:(C,X) ∈ (subcls1 P)+
```

```
    proof (cases Xs)
      case Nil
      with subo C XXs have Subobjs P C [C,X] by simp
      thus ?thesis
        apply −
        apply (rule r-into-trancl)
        apply (rule subclsR-subcls1)
        apply (rule-tac Cs=[] in Subobjs-subclsR)
        apply simp
        done
    next
      case (Cons Y Ys)
      with subo C XXs have subo'':Subobjs P C ([C,Y]@Ys@[X]) by simp
      hence plus:(C,Y) ∈ (subcls1 P)+
        apply −
        apply (rule r-into-trancl)
        apply (rule subclsR-subcls1)
        apply (rule-tac Cs=[] in Subobjs-subclsR)
        apply simp
        done
      from subo'' have P ⊢ Y ≼* X
        apply −
        apply (subgoal-tac Subobjs P C ([C]@Y#(Ys@[X])))
         apply (drule Subobjs-Subobjs)
         apply (drule-tac C=Y in Subobjs-subclass) apply simp-all
        done
      with plus show ?thesis by (fastforce elim:trancl-rtrancl-trancl)
    qed
    from lastleq X have leq:P ⊢ X ≼* C by simp
    with CleqX have (C,C) ∈ (subcls1 P)+
      by (rule trancl-rtrancl-trancl)
    with wf show ?thesis by (fastforce dest:subcls-irrefl)
  qed
qed



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

using leq class
proof(induct rule:rtrancl.induct)
  fix C assume is-class P C
  thus ∃ Cs. P ⊢ Path C to C via Cs
    by (rule-tac x=[C] in exI,fastforce intro:Subobjs-Base simp:path-via-def)
next
  fix C C' D assume CleqC':P ⊢ C ≼* C' and C'leqD:P ⊢ C' ≺1 D
```

and *classC*:*is-class P C* **and** *IH*:*is-class P C* $\Longrightarrow \exists$ *Cs*. *P* $\vdash$ *Path C to C'* *via Cs*
from *IH*[*OF classC*] **obtain** *Cs* **where** *subo*:*Subobjs P C Cs* **and** *last*:*last Cs* = *C'*
by (*auto simp*:*path-via-def*)
with *C'leqD* **show** $\exists$ *Cs*. *P* $\vdash$ *Path C to D via Cs*
proof (*auto dest*!:*subcls1-subclsR-or-subclsS*)
assume *P* $\vdash$ *last Cs* $\prec_R$ *D*
with *subo* **have** *Subobjs P C* (*Cs*@[*D*]) **using** *wf*
by (*rule converse-Subobjs-Rep*)
thus *?thesis* **by** (*fastforce simp*:*path-via-def*)
next
assume *subS*:*P* $\vdash$ *last Cs* $\prec_S$ *D*
from *CleqC' last* **have** *Cleqlast*:*P* $\vdash$ *C* $\preceq^*$ *last Cs* **by** *simp*
from *subS* **have** *classLast*:*is-class P* (*last Cs*)
by (*auto intro*:*subcls1-class subclsS-subcls1*)
then obtain *Bs fs ms* **where** *class P* (*last Cs*) = *Some*(*Bs*,*fs*,*ms*)
by (*fastforce simp*:*is-class-def*)
hence *classD*:*is-class P D* **using** *subS wf*
by (*auto intro*:*wf-cdecl-supD dest*:*class-wf dest*!:*subclsSD*
elim:*ShBaseclass-isBaseclass*)
with *Cleqlast subS* **have** *Subobjs P C* [*D*]
by (*fastforce intro*:*Subobjs-Sh SubobjsR-Base*)
thus *?thesis* **by** (*fastforce simp*:*path-via-def*)
qed
qed

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

proof (*auto simp add*:*path-unique-def*)

from *least* **have** *Subobjs P C Cs*
by (*simp add*:*LeastMethodDef-def MethodDefs-def*)
thus $\exists$ *Cs'*. *Subobjs P C Cs'* $\wedge$ *last Cs'* = *last Cs*
by *fastforce*
next

fix *Cs' Cs''*
assume *suboCs'*:*Subobjs P C Cs'* **and** *suboCs''*:*Subobjs P C Cs''*
and *lastCs'*:*last Cs'* = *last Cs* **and** *lastCs''*:*last Cs''* = *last Cs*
from *suboCs'* **have** *notemptyCs'*:*Cs'* $\neq$ [] **by** (*rule Subobjs-nonempty*)
from *suboCs''* **have** *notemptyCs''*:*Cs''* $\neq$ [] **by** (*rule Subobjs-nonempty*)
from *least* **have** *suboCs*:*Subobjs P C Cs*
and *all*:$\forall$ *Ds*. *Subobjs P C Ds* $\wedge$
($\exists$ *Ts T m Bs ms*. ($\exists$ *fs*. *class P* (*last Ds*) = *Some* (*Bs*, *fs*, *ms*)) $\wedge$
map-of ms M = *Some*(*Ts*,*T*,*m*)) $\longrightarrow$ *P*,*C* $\vdash$ *Cs* $\sqsubseteq$ *Ds*

by (*auto simp*:*LeastMethodDef-def MethodDefs-def*)
from *least* **obtain** *Bs fs ms T Ts m* **where**
class: *class P* (*last Cs*) = *Some*(*Bs*, *fs*, *ms*) **and** *map*:*map-of ms M* = *Some*(*Ts*,*T*,*m*)
by (*auto simp*:*LeastMethodDef-def MethodDefs-def intro*:*that*)
from *suboCs′ lastCs′ class map all* **have** *pathCs′*:*P*,*C* ⊢ *Cs* ⊑ *Cs′*
by *simp*
with *wf lastCs′* **have** *eq*:*Cs* = *Cs′* **by**(*fastforce intro*:*leq-path-last*)
from *suboCs″ lastCs″ class map all* **have** *pathCs″*:*P*,*C* ⊢ *Cs* ⊑ *Cs″*
by *simp*
with *wf lastCs″* **have** *Cs* = *Cs″* **by**(*fastforce intro*:*leq-path-last*)
with *eq* **show** *Cs′* = *Cs″* **by** *simp*
qed

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 (*auto simp add*:*path-unique-def*)

from *least* **have** *Subobjs P C Cs*
by (*simp add*:*LeastFieldDecl-def FieldDecls-def*)
hence *Subobjs P C* ([*hd Cs*]@*tl Cs*)
by − (*frule Subobjs-nonempty*,*simp*)
with *wf* **have** *Subobjs P C* [*hd Cs*]
by (*fastforce intro*:*appendSubobj*)
thus ∃ *Cs′*. *Subobjs P C Cs′* ∧ *last Cs′* = *hd Cs*
by *fastforce*
next

fix *Cs′ Cs″*
assume *suboCs′*:*Subobjs P C Cs′* **and** *suboCs″*:*Subobjs P C Cs″*
and *lastCs′*:*last Cs′* = *hd Cs* **and** *lastCs″*:*last Cs″* = *hd Cs*
from *suboCs′* **have** *notemptyCs′*:*Cs′* ≠ [] **by** (*rule Subobjs-nonempty*)
from *suboCs″* **have** *notemptyCs″*:*Cs″* ≠ [] **by** (*rule Subobjs-nonempty*)
from *least* **have** *suboCs*:*Subobjs P C Cs*
and *all*:∀ *Ds*. *Subobjs P C Ds* ∧
(∃ *T Bs fs*. (∃ *ms*. *class P* (*last Ds*) = *Some* (*Bs*, *fs*, *ms*)) ∧
map-of fs F = *Some T*) ⟶ *P*,*C* ⊢ *Cs* ⊑ *Ds*
by (*auto simp*:*LeastFieldDecl-def FieldDecls-def*)
from *least* **obtain** *Bs fs ms T* **where**
class: *class P* (*last Cs*) = *Some*(*Bs*, *fs*, *ms*) **and** *map*:*map-of fs F* = *Some T*
by (*auto simp*:*LeastFieldDecl-def FieldDecls-def*)
from *suboCs* **have** *notemptyCs*:*Cs* ≠ [] **by** (*rule Subobjs-nonempty*)
from *suboCs notemptyCs* **have** *suboHd*:*Subobjs P* (*hd Cs*) (*hd Cs*#*tl Cs*)
by −(*rule-tac C*=*C* **and** *Cs*=[] **in** *Subobjs-Subobjs*,*simp*)
with *suboCs′ notemptyCs lastCs′ wf* **have** *suboCs′App*:*Subobjs P C* (*Cs′*@$_p$*Cs*)

by −(*rule Subobjs-appendPath*,*simp-all*)
from *suboHd suboCs''* *notemptyCs lastCs''* *wf*
have *suboCs''App*:*Subobjs P C* (*Cs''*@$_p$*Cs*)
by −(*rule Subobjs-appendPath*,*simp-all*)
from *suboCs'App all class map notemptyCs* **have** *pathCs'*:*P*,*C* ⊢ *Cs* ⊑ *Cs'*@$_p$*Cs*
by −(*erule-tac x*=*Cs'*@$_p$*Cs* **in** *allE*,*drule-tac Cs'*=*Cs'* **in** *appendPath-last*,*simp*)
from *suboCs''App all class map notemptyCs* **have** *pathCs''*:*P*,*C* ⊢ *Cs* ⊑ *Cs''*@$_p$*Cs*
by −(*erule-tac x*=*Cs''*@$_p$*Cs* **in** *allE*,*drule-tac Cs'*=*Cs''* **in** *appendPath-last*,*simp*)
from *pathCs'* *lastCs'* *notemptyCs notemptyCs'* *wf* **have** *Cs'*:*Cs'* = [*hd Cs*]
by (*rule path-hd-appendPath*)
from *pathCs''* *lastCs''* *notemptyCs notemptyCs''* *wf* **have** *Cs''* = [*hd Cs*]
by (*rule path-hd-appendPath*)
with *Cs'* **show** *Cs'* = *Cs''* **by** *simp*
qed

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*]

apply (*simp add*:*LeastFieldDecl-def FieldDecls-def*)
apply *clarsimp*
apply (*simp add*:*path-via-def*)
apply (*frule Subobjs-nonempty*)
apply (*rule-tac Cs'*=*tl Cs* **in** *appendSubobj*)
apply *auto*
done

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

apply (*unfold path-unique-def*)
apply (*rule-tac a*=[*C*] **in** *ex1I*)
apply (*auto intro*:*Subobjs-Base mdc-eq-last*)
done

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

apply (*induct rule*:*rtrancl-induct*)
apply (*drule SubobjsR-Base*)
apply (*rule-tac x*=[] **in** *exI*)
apply *simp*
apply (*auto dest*:*converse-SubobjsR-Rep*)
done

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 −
from *path-unique* **have** *is-class P C*
by (*auto intro*:*Subobjs-isClass simp*:*path-unique-def*)
with *leq wf* **obtain** *Cs* **where** *path-via*:$P \vdash$ *Path C to C' via Cs*
by (*auto dest*:*leq-implies-path*)
with *wf* **have** *classC'*:*is-class P C'*
by (*fastforce intro*:*Subobj-last-isClass simp*:*path-via-def*)
with *leqR wf* **obtain** *Cs'* **where** *subo*:$Subobjs_R\ P\ C'\ Cs'$ **and** *last*:*last Cs' = D*
by (*auto dest*:*leqR-SubobjsR*)
hence *hd*:*hd Cs' = C'*
by (*fastforce dest*:*hd-SubobjsR*)
with *path-via subo wf* **have** *suboApp*:*Subobjs P C* (*Cs@tl Cs'*)
by (*auto dest*!:*Subobjs-Rep dest*:*Subobjs-appendPath*
simp:*path-via-def appendPath-def*)
hence *last'*:*last* (*Cs@tl Cs'*) = *D*
proof (*cases tl Cs' = []*)
case *True*
with *subo hd last* **have** *C' = D*
by (*subgoal-tac Cs' = [C']*,*auto dest*!:*SubobjsR-nonempty hd-Cons-tl*)
with *path-via* **have** *last Cs = D*
by (*auto simp*:*path-via-def*)
with *True* **show** *?thesis* **by** *simp*
next
case *False*
from *subo* **have** *Cs'*:*Cs' = hd Cs'#tl Cs'*
by (*auto dest*:*SubobjsR-nonempty*)
from *False* **have** *last*(*hd Cs'#tl Cs'*) = *last* (*tl Cs'*)
by (*rule last-ConsR*)
with *False Cs' last* **show** *?thesis* **by** *simp*
qed
with *path-unique suboApp*
have *all*:$\forall$ *Ds. Subobjs P C Ds* $\wedge$ *last Ds = D* $\longrightarrow$ *Ds = Cs@tl Cs'*
by (*auto simp add*:*path-unique-def*)
{ fix *Cs''* **assume** *path-via2*:$P \vdash$ *Path C to C' via Cs''* **and** *noteq*:$Cs'' \neq Cs$
with *suboApp* **have** *last* (*Cs''@tl Cs'*) = *D*
proof (*cases tl Cs' = []*)
case *True*
with *subo hd last* **have** *C' = D*
by (*subgoal-tac Cs' = [C']*,*auto dest*!:*SubobjsR-nonempty hd-Cons-tl*)
with *path-via2* **have** *last Cs'' = D*
by (*auto simp*:*path-via-def*)
with *True* **show** *?thesis* **by** *simp*
next

case *False*
from *subo* **have** *Cs'*:*Cs'* = *hd Cs'*#*tl Cs'*
by (*auto dest*:*SubobjsR-nonempty*)
from *False* **have** *last*(*hd Cs'*#*tl Cs'*) = *last* (*tl Cs'*)
by (*rule last-ConsR*)
with *False Cs' last* **show** *?thesis* **by** *simp*
qed
with *path-via2 noteq* **have** *False* **using** *all subo hd wf*
apply (*auto simp*:*path-via-def*)
apply (*drule Subobjs-Rep*)
apply (*drule Subobjs-appendPath*)
apply (*auto simp*:*appendPath-def*)
done }
with *path-via* **show** *?thesis*
by (*auto simp*:*path-via-def path-unique-def*)
qed

17.8 Well-formedness and member lookup

lemma *has-path-has*:
⟦*P* ⊢ *Path D to C via Ds*; *P* ⊢ *C has M* = (*Ts*,*T*,*m*) *via Cs*; *wf-prog wf-md P*⟧
⟹ *P* ⊢ *D has M* = (*Ts*,*T*,*m*) *via Ds*@$_p$*Cs*
by (*clarsimp simp*:*HasMethodDef-def MethodDefs-def*,*frule Subobjs-nonempty*,
drule-tac Cs'=*Ds* **in** *appendPath-last*,
fastforce intro:*Subobjs-appendPath simp*:*path-via-def*)

lemma *has-least-wf-mdecl*:
⟦ *wf-prog wf-md P*; *P* ⊢ *C has least M* = *m via Cs* ⟧
⟹ *wf-mdecl wf-md P* (*last Cs*) (*M*,*m*)
by(*fastforce dest*:*visible-methods-exist class-wf map-of-SomeD*
simp:*LeastMethodDef-def wf-cdecl-def*)

lemma *has-overrider-wf-mdecl*:
⟦ *wf-prog wf-md P*; *P* ⊢ (*C*,*Cs*) *has overrider M* = *m via Cs'* ⟧
⟹ *wf-mdecl wf-md P* (*last Cs'*) (*M*,*m*)
by(*fastforce dest*:*visible-methods-exist map-of-SomeD class-wf*
simp:*FinalOverriderMethodDef-def OverriderMethodDefs-def*
MinimalMethodDefs-def wf-cdecl-def)

lemma *select-method-wf-mdecl*:
⟦ *wf-prog wf-md P*; *P* ⊢ (*C*,*Cs*) *selects M* = *m via Cs'* ⟧
⟹ *wf-mdecl wf-md P* (*last Cs'*) (*M*,*m*)
by(*fastforce elim*:*SelectMethodDef.induct*
intro:*has-least-wf-mdecl has-overrider-wf-mdecl*)

lemma *wf-sees-method-fun*:
⟦$P \vdash C$ *has least* M = *mthd via* Cs; $P \vdash C$ *has least* M = *mthd'* *via* Cs';
wf-prog wf-md P⟧
⟹ *mthd* = *mthd'* ∧ $Cs = Cs'$

apply (*auto simp*:*LeastMethodDef-def*)
apply (*erule-tac* x=(Cs', *mthd'*) **in** *ballE*)
apply (*erule-tac* x=(Cs, *mthd*) **in** *ballE*)
apply *auto*
apply (*drule leq-path-asym2*) **apply** *simp-all*
apply (*rule sees-methods-fun*) **apply** *simp-all*
apply (*erule-tac* x=(Cs', *mthd'*) **in** *ballE*)
apply (*erule-tac* x=(Cs, *mthd*) **in** *ballE*)
apply (*auto intro*:*leq-path-asym2*)
done

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''⟧
⟹ *mthd* = *mthd'* ∧ $Cs' = Cs''$
proof(*induct rule*:*SelectMethodDef.induct*)
case (*dyn-unique* C M *mthd* Cs' Cs)
have $P \vdash (C, Cs)$ *selects* M = *mthd'* *via* Cs''
and $P \vdash C$ *has least* M = *mthd via* Cs' **by** *fact+*
thus *?case*
proof(*induct rule*:*SelectMethodDef.induct*)
case (*dyn-unique* D M' *mthd'* Ds' Ds)
have $P \vdash D$ *has least* M' = *mthd'* *via* Ds'
and $P \vdash D$ *has least* M' = *mthd via* Cs' **by** *fact+*
with *wf* **show** *?case*
by −(*rule wf-sees-method-fun*,*simp-all*)
next
case (*dyn-ambiguous* D M' Ds *mthd'* Ds')
have ∀ *mthd* Cs'. ¬ $P \vdash D$ *has least* M' = *mthd via* Cs'
and $P \vdash D$ *has least* M' = *mthd via* Cs' **by** *fact+*
thus *?case* **by** *blast*
qed
next
case (*dyn-ambiguous* C M Cs *mthd* Cs')
have $P \vdash (C, Cs)$ *selects* M = *mthd'* *via* Cs''
and $P \vdash (C, Cs)$ *has overrider* M = *mthd via* Cs'
and ∀ *mthd* Cs'. ¬ $P \vdash C$ *has least* M = *mthd via* Cs' **by** *fact+*
thus *?case*
proof(*induct rule*:*SelectMethodDef.induct*)
case (*dyn-unique* D M' *mthd'* Ds' Ds)

```
    have P ⊢ D has least M′ = mthd′ via Ds′
      and ∀ mthd Cs′. ¬ P ⊢ D has least M′ = mthd via Cs′ by fact+
    thus ?case by blast
  next
    case (dyn-ambiguous D M′ Ds mthd′ Ds′)
    have P ⊢ (D, Ds) has overrider M′ = mthd′ via Ds′
      and P ⊢ (D, Ds) has overrider M′ = mthd via Cs′ by fact+
    thus ?case by(fastforce dest:overrider-method-fun)
  qed
qed



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

proof −
  from field have (Cs,T) ∈ FieldDecls P C F
    by (simp add:LeastFieldDecl-def)
  from this obtain Bs fs ms
    where map-of fs F = Some T
    and class: class P (last Cs) = Some (Bs,fs,ms)
    by (auto simp add:FieldDecls-def)
  hence (F,T) ∈ set fs by (simp add:map-of-SomeD)
  with class wf show ?thesis
    by(fastforce dest!: class-wf simp: wf-cdecl-def wf-fdecl-def)
qed



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

proof −
  from method have (Cs,Ts,T,m) ∈ MethodDefs P C M
    by (simp add:LeastMethodDef-def)
  from this obtain Bs fs ms
    where map-of ms M = Some(Ts,T,m)
    and class: class P (last Cs) = Some (Bs,fs,ms)
    by (auto simp add:MethodDefs-def)
  hence (M,Ts,T,m) ∈ set ms by (simp add:map-of-SomeD)
  with class wf show ?thesis
    by(fastforce dest!: class-wf simp: wf-cdecl-def wf-mdecl-def)
qed
```

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 −
from *method* **have** $(Cs',Ts,T,m) \in$ *MethodDefs* P C M
by(*clarsimp simp*:*FinalOverriderMethodDef-def OverriderMethodDefs-def MinimalMethodDefs-def*)
from *this* **obtain** *Bs fs ms*
where *map-of ms* M = *Some*(Ts,T,m)
and *class*: *class* P (*last* Cs') = *Some* (Bs,fs,ms)
by (*auto simp add*:*MethodDefs-def*)
hence $(M,Ts,T,m) \in$ *set ms* **by** (*simp add*:*map-of-SomeD*)
with *class wf* **show** *?thesis*
by(*fastforce dest*!: *class-wf simp*: *wf-cdecl-def wf-mdecl-def*)
qed

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
by(*auto elim*:*SelectMethodDef.cases intro*:*least-method-is-type least-overrider-is-type*)

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*$]\!]$
$\Longrightarrow Ts' = Ts \wedge P \vdash T \leq T'$

apply (*simp add*:*wf-cdecl-def*)
apply *clarsimp*
apply (*rotate-tac* −*1*)
apply (*erule-tac* x=C' **in** *ballE*)
apply *clarsimp*
apply (*rotate-tac* −*1*)
apply (*erule-tac* x=$(M,\ Ts,\ T,\ m)$ **in** *ballE*)
apply *clarsimp*
apply (*erule-tac* x=Ts' **in** *allE*)
apply (*erule-tac* x=T' **in** *allE*)
apply (*auto simp*:*HasMethodDef-def*)
apply (*erule-tac* x=*fst* m' **in** *allE*)
apply (*erule-tac* x=*snd* m' **in** *allE*)
apply (*erule-tac* x=$Cs@_p[D]$ **in** *allE*)
apply *simp*
apply (*erule-tac* x=*fst* m' **in** *allE*)
apply (*erule-tac* x=*snd* m' **in** *allE*)

apply (*erule-tac* x=$Cs@_p[D]$ **in** *allE*)
apply *simp*
done

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) ∈ (*subcls1* P)$^+$ **and** *wf*:*wf-prog wf-md* P
shows ∀ Bs fs ms Ts T m. *class* P C = *Some*(Bs,fs,ms) ∧ *map-of* ms M = *Some*(Ts,T,m)
⟶ Ts' = Ts ∧ P ⊢ T ≤ T'

using *leq classD mapMs'*
proof (*erule-tac* a=C **and** b=D **in** *converse-trancl-induct*)
fix C
assume *CleqD*:P ⊢ C ≺^{1} D **and** *classD1*:*class* P D = *Some*(Bs',fs',ms')
{ fix Bs fs ms Ts T m
assume *classC*:*class* P C = *Some*(Bs,fs,ms) **and** *mapMs*:*map-of* ms M = *Some*(Ts,T,m)
from *classD1 mapMs'* **have** *hasViaD*:P ⊢ D *has* M = (Ts',T',m') *via* [D]
by (*fastforce intro*:*Subobjs-Base simp*:*HasMethodDef-def MethodDefs-def is-class-def*)
from *CleqD classC* **have** *base*:D ∈ *baseClasses* Bs
by (*fastforce dest*:*subcls1D*)
from *classC wf* **have** *cdecl*:*wf-cdecl wf-md* P (C,Bs,fs,ms)
by (*rule class-wf*)
from *classC mapMs* **have** (M,Ts,T,m)∈*set* ms
by −(*drule map-of-SomeD*)
with *cdecl base hasViaD* **have** Ts' = Ts ∧ P ⊢ T ≤ T'
by −(*rule-tac* Cs=[D] **in** *base-subtype*,*auto simp*:*appendPath-def*) **}**
thus ∀ Bs fs ms Ts T m. *class* P C = *Some*(Bs, fs, ms) ∧ *map-of* ms M = *Some*(Ts,T,m)
⟶ Ts' = Ts ∧ P ⊢ T ≤ T' **by** *blast*
next
fix C C'
assume *classD1*:*class* P D = *Some*(Bs',fs',ms') **and** *CleqC'*:P ⊢ C ≺^{1} C'
and *subcls*:(C',D) ∈ (*subcls1* P)$^+$
and *IH*:∀ Bs fs ms Ts T m. *class* P C' = *Some*(Bs,fs,ms) ∧
map-of ms M = *Some*(Ts,T,m) ⟶
Ts' = Ts ∧ P ⊢ T ≤ T'
{ fix Bs fs ms Ts T m
assume *classC*:*class* P C = *Some*(Bs,fs,ms) **and** *mapMs*:*map-of* ms M = *Some*(Ts,T,m)
from *classD1 mapMs'* **have** *hasViaD*:P ⊢ D *has* M = (Ts',T',m') *via* [D]
by (*fastforce intro*:*Subobjs-Base simp*:*HasMethodDef-def MethodDefs-def is-class-def*)
from *subcls* **have** *C'leqD*:P ⊢ C' ≼* D **by** *simp*
from *classC wf CleqC'* **have** *is-class* P C'
by (*fastforce intro*:*wf-cdecl-supD class-wf dest*:*subcls1D*)

with *C′leqD wf* **obtain** *Cs* **where** $P \vdash Path\ C'\ to\ D\ via\ Cs$
by (*auto dest*!:*leq-implies-path simp*:*is-class-def*)
hence *hasVia*:$P \vdash C'\ has\ M = (Ts',T',m')\ via\ Cs@_p[D]$ **using** *hasViaD wf*
by (*rule has-path-has*)
from *CleqC′ classC* **have** *base*:$C' \in baseClasses\ Bs$
by (*fastforce dest*:*subcls1D*)
from *classC wf* **have** *cdecl*:*wf-cdecl wf-md P* (C,Bs,fs,ms)
by (*rule class-wf*)
from *classC mapMs* **have** $(M,Ts,T,m) \in set\ ms$
by −(*drule map-of-SomeD*)
with *cdecl base hasVia* **have** $Ts' = Ts \wedge P \vdash T \leq T'$
by(*rule base-subtype*) **}**
thus $\forall Bs\ fs\ ms\ Ts\ T\ m.\ class\ P\ C = Some(Bs,\ fs,\ ms) \wedge map\text{-}of\ ms\ M = Some(Ts,T,m) \longrightarrow Ts' = Ts \wedge P \vdash T \leq T'$ **by** *blast*
qed

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 $\forall Ts\ T\ m\ Cs.\ P \vdash C\ has\ M = (Ts,T,m)\ via\ Cs \longrightarrow Ts = Ts' \wedge P \vdash T' \leq T$
using *assms*
proof (*induct rule*:*rtrancl.induct*)
fix *C*
assume *Cleast*:$P \vdash C\ has\ least\ M = (Ts',T',m')\ via\ Ds$
{ fix *Ts T m Cs*
assume *Chas*:$P \vdash C\ has\ M = (Ts,T,m)\ via\ Cs$
with *Cleast* **have** *path*:$P,C \vdash Ds \sqsubseteq Cs$
by (*fastforce simp*:*LeastMethodDef-def HasMethodDef-def*)
{ assume $Ds = Cs$
with *Cleast Chas* **have** $Ts = Ts' \wedge T' = T$
by (*auto simp*:*LeastMethodDef-def HasMethodDef-def MethodDefs-def*)
hence $Ts = Ts' \wedge P \vdash T' \leq T$ **by** *auto* **}**
moreover
{ assume $(Ds,Cs) \in (leq\text{-}path1\ P\ C)^+$
hence *subcls*:$(last\ Ds,last\ Cs) \in (subcls1\ P)^+$ **using** *wf*
by −(*rule last-leq-paths*)
from *Chas* **obtain** *Bs fs ms* **where** $class\ P\ (last\ Cs) = Some(Bs,fs,ms)$
and $map\text{-}of\ ms\ M = Some(Ts,T,m)$
by (*auto simp*:*HasMethodDef-def MethodDefs-def*)
hence *ex*:$\forall Bs'\ fs'\ ms'\ Ts'\ T'\ m'.\ class\ P\ (last\ Ds) = Some(Bs',fs',ms') \wedge map\text{-}of\ ms'\ M = Some(Ts',T',m') \longrightarrow Ts = Ts' \wedge P \vdash T' \leq T$
using *subcls wf*
by −(*rule subclsPlus-subtype*,*auto*)
from *Cleast* **obtain** $Bs'\ fs'\ ms'$ **where** $class\ P\ (last\ Ds) = Some(Bs',fs',ms')$

```
      and map-of ms' M = Some(Ts',T',m')
      by (auto simp:LeastMethodDef-def MethodDefs-def)
    with ex have Ts = Ts' and P ⊢ T' ≤ T by auto }
    ultimately have Ts = Ts' and P ⊢ T' ≤ T using path
      by (auto dest!:rtranclD) }
  thus ∀ Ts T m Cs. P ⊢ C has M = (Ts, T, m) via Cs ⟶
                    Ts = Ts' ∧ P ⊢ T' ≤ T
    by (simp add:HasMethodDef-def MethodDefs-def)
next
  fix D C' C
  assume DleqC':P ⊢ D ≼* C' and C'leqC:P ⊢ C' ≺1 C
  and Dleast:P ⊢ D has least M = (Ts',T',m') via Ds
  and IH:⟦P ⊢ D has least M = (Ts',T',m') via Ds; wf-prog wf-md P⟧
   ⟹ ∀ Ts T m Cs. P ⊢ C' has M = (Ts, T, m) via Cs ⟶
          Ts = Ts' ∧ P ⊢ T' ≤ T
  { fix Ts T m Cs
    assume Chas:P ⊢ C has M = (Ts,T,m) via Cs
    from Dleast have classD:is-class P D
      by (auto intro:Subobjs-isClass simp:LeastMethodDef-def MethodDefs-def)
    from DleqC' C'leqC have P ⊢ D ≼* C by simp
    then obtain Cs' where P ⊢ Path D to C via Cs' using classD wf
      by (auto dest:leq-implies-path)
    hence Dhas:P ⊢ D has M = (Ts,T,m) via Cs'@pCs using Chas wf
      by (fastforce intro:has-path-has)
    with Dleast have path:P,D ⊢ Ds ⊑ Cs'@pCs
      by (auto simp:LeastMethodDef-def HasMethodDef-def)
    { assume Ds = Cs'@pCs
      with Dleast Dhas have Ts = Ts' ∧ T' = T
        by (auto simp:LeastMethodDef-def HasMethodDef-def MethodDefs-def)
      hence Ts = Ts' ∧ T' = T by auto }
    moreover
    { assume (Ds,Cs'@pCs) ∈ (leq-path1 P D)+
      hence subcls:(last Ds,last (Cs'@pCs)) ∈ (subcls1 P)+ using wf
        by −(rule last-leq-paths)
    from Dhas obtain Bs fs ms where class P (last (Cs'@pCs)) = Some(Bs,fs,ms)

      and map-of ms M = Some(Ts,T,m)
      by (auto simp:HasMethodDef-def MethodDefs-def)
    hence ex:∀ Bs' fs' ms' Ts' T' m'. class P (last Ds) = Some(Bs',fs',ms') ∧
            map-of ms' M = Some(Ts',T',m') ⟶
              Ts = Ts' ∧ P ⊢ T' ≤ T
      using subcls wf
      by −(rule subclsPlus-subtype,auto)
   from Dleast obtain Bs' fs' ms' where class P (last Ds) = Some(Bs',fs',ms')

      and map-of ms' M = Some(Ts',T',m')
      by (auto simp:LeastMethodDef-def MethodDefs-def)
    with ex have Ts = Ts' and P ⊢ T' ≤ T by auto }
  ultimately have Ts = Ts' and P ⊢ T' ≤ T using path
```

```
    by (auto dest!:rtranclD) }
  thus ∀ Ts T m Cs. P ⊢ C has M = (Ts, T, m) via Cs ⟶
         Ts = Ts' ∧ P ⊢ T' ≤ T
    by simp
qed



lemma leq-methods-subtypes:
  assumes leq:P ⊢ D ≼* C and least:(Ds,(Ts',T',m')) ∈ MinimalMethodDefs P
D M
  and wf:wf-prog wf-md P
  shows ∀ Ts T m Cs Cs'. P ⊢ Path D to C via Cs' ∧ P,D ⊢ Ds ⊑ Cs'@_pCs ∧
Cs ≠ [] ∧
                          P ⊢ C has M = (Ts,T,m) via Cs
                               ⟶ Ts = Ts' ∧ P ⊢ T' ≤ T
using assms
proof (induct rule:rtrancl.induct)
  fix C
  assume Cleast:(Ds,(Ts',T',m')) ∈ MinimalMethodDefs P C M
  { fix Ts T m Cs Cs'
    assume path':P ⊢ Path C to C via Cs'
      and leq-path:P,C ⊢ Ds ⊑ Cs' @_p Cs and notempty:Cs ≠ []
      and Chas:P ⊢ C has M = (Ts,T,m) via Cs
    from path' wf have Cs':Cs' = [C] by(rule path-via-C)
    from leq-path Cs' notempty have leq':P,C ⊢ Ds ⊑ Cs
      by(auto simp:appendPath-def split:if-split-asm)
    { assume Ds = Cs
      with Cleast Chas have Ts = Ts' ∧ T' = T
        by (auto simp:MinimalMethodDefs-def HasMethodDef-def MethodDefs-def)
      hence Ts = Ts' ∧ P ⊢ T' ≤ T by auto }
    moreover
    { assume (Ds,Cs) ∈ (leq-path1 P C)^+
      hence subcls:(last Ds,last Cs) ∈ (subcls1 P)^+ using wf
        by -(rule last-leq-paths)
      from Chas obtain Bs fs ms where class P (last Cs) = Some(Bs,fs,ms)
        and map-of ms M = Some(Ts,T,m)
        by (auto simp:HasMethodDef-def MethodDefs-def)
      hence ex:∀ Bs' fs' ms' Ts' T' m'. class P (last Ds) = Some(Bs',fs',ms') ∧
        map-of ms' M = Some(Ts',T',m') ⟶ Ts = Ts' ∧ P ⊢ T' ≤ T
        using subcls wf
        by -(rule subclsPlus-subtype,auto)
     from Cleast obtain Bs' fs' ms' where class P (last Ds) = Some(Bs',fs',ms')

        and map-of ms' M = Some(Ts',T',m')
        by (auto simp:MinimalMethodDefs-def MethodDefs-def)
      with ex have Ts = Ts' and P ⊢ T' ≤ T by auto }
      ultimately have Ts = Ts' and P ⊢ T' ≤ T using leq'
        by (auto dest!:rtranclD) }
```

thus $\forall\ Ts\ T\ m\ Cs\ Cs'.\ P \vdash Path\ C\ to\ C\ via\ Cs' \wedge P,C \vdash Ds \sqsubseteq Cs' @_p\ Cs \wedge Cs \neq [] \wedge$
$$P \vdash C\ has\ M = (Ts,\ T,\ m)\ via\ Cs \longrightarrow Ts = Ts' \wedge P \vdash T' \leq T$$ **by** *blast*

next

fix $D\ C'\ C$

assume $DleqC'{:}P \vdash D \preceq^*\ C'$ **and** $C'leqC{:}P \vdash C' \prec^1\ C$

and $Dleast{:}(Ds,\ Ts',\ T',\ m') \in MinimalMethodDefs\ P\ D\ M$

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

{ fix $Ts\ T\ m\ Cs\ Cs'$

assume $path{:}P \vdash Path\ D\ to\ C\ via\ Cs'$

and $leq\text{-}path{:}P,D \vdash Ds \sqsubseteq Cs' @_p\ Cs$

and $notempty{:}Cs \neq []$

and $Chas{:}P \vdash C\ has\ M = (Ts,T,m)\ via\ Cs$

from $Dleast$ **have** $classD{:}is\text{-}class\ P\ D$

by $(auto\ intro{:}Subobjs\text{-}isClass\ simp{:}MinimalMethodDefs\text{-}def\ MethodDefs\text{-}def)$

from $path$ **have** $Dhas{:}P \vdash D\ has\ M = (Ts,T,m)\ via\ Cs'@_pCs$ **using** $Chas\ wf$

by $(fastforce\ intro{:}has\text{-}path\text{-}has)$

{ assume $Ds = Cs'@_pCs$

with $Dleast\ Dhas$ **have** $Ts = Ts' \wedge T' = T$

by $(auto\ simp{:}MinimalMethodDefs\text{-}def\ HasMethodDef\text{-}def\ MethodDefs\text{-}def)$

hence $Ts = Ts' \wedge T' = T$ **by** $auto$ **}**

moreover

{ assume $(Ds,Cs'@_pCs) \in (leq\text{-}path1\ P\ D)^+$

hence $subcls{:}(last\ Ds,last\ (Cs'@_pCs)) \in (subcls1\ P)^+$ **using** wf

by $-(rule\ last\text{-}leq\text{-}paths)$

from $Dhas$ **obtain** $Bs\ fs\ ms$ **where** $class\ P\ (last\ (Cs'@_pCs)) = Some(Bs,fs,ms)$

and $map\text{-}of\ ms\ M = Some(Ts,T,m)$

by $(auto\ simp{:}HasMethodDef\text{-}def\ MethodDefs\text{-}def)$

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

using $subcls\ wf$

by $-(rule\ subclsPlus\text{-}subtype,auto)$

from $Dleast$ **obtain** $Bs'\ fs'\ ms'$ **where** $class\ P\ (last\ Ds) = Some(Bs',fs',ms')$

and $map\text{-}of\ ms'\ M = Some(Ts',T',m')$

by $(auto\ simp{:}MinimalMethodDefs\text{-}def\ MethodDefs\text{-}def)$

with ex **have** $Ts = Ts'$ **and** $P \vdash T' \leq T$ **by** $auto$ **}**

ultimately have $Ts = Ts'$ **and** $P \vdash T' \leq T$ **using** $leq\text{-}path$

by $(auto\ dest!{:}rtranclD)$ **}**

thus $\forall\ Ts\ T\ m\ Cs\ Cs'.\ P \vdash Path\ D\ to\ C\ via\ Cs' \wedge P,D \vdash Ds \sqsubseteq Cs' @_p\ Cs \wedge Cs \neq [] \wedge$
$$P \vdash C\ has\ M = (Ts,\ T,\ m)\ via\ Cs \longrightarrow$$

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

by *blast*

qed

lemma *select-least-methods-subtypes*:
assumes *select-method*:$P \vdash (C,Cs@_pDs)$ *selects* $M = (Ts,T,pns,body)$ *via* Cs'
and *least-method*:$P \vdash$ *last Cs has least* $M = (Ts',T',pns',body')$ *via Ds*
and *path*:$P \vdash$ *Path C to* (*last Cs*) *via Cs*
and *wf*:*wf-prog wf-md P*
shows $Ts' = Ts \wedge P \vdash T \leq T'$
using *select-method*
proof −
from *path* **have** *sub*:$P \vdash C \preceq^*$ *last Cs*
by(*fastforce intro*:*Subobjs-subclass simp*:*path-via-def*)
from *least-method* **have** *has*:$P \vdash$ *last Cs has* $M = (Ts',T',pns',body')$ *via Ds*
by(*rule has-least-method-has-method*)
from *select-method* **show** *?thesis*
proof *cases*
case *dyn-unique*
hence *dyn*:$P \vdash C$ *has least* $M = (Ts,T,pns,body)$ *via* Cs' **by** *simp*
with *sub has wf* **show** *?thesis*
by −(*drule leq-method-subtypes*,*assumption*,*simp*,*blast*)+
next
case *dyn-ambiguous*
hence *overrider*:$P \vdash (C,Cs@_pDs)$ *has overrider* $M = (Ts,T,pns,body)$ *via* Cs'

by *simp*
from *least-method* **have** *notempty*:$Ds \neq []$
by(*auto intro*!:*Subobjs-nonempty simp*:*LeastMethodDef-def MethodDefs-def*)
have *last Cs* = *hd Ds* $\Longrightarrow$ *last* (*Cs* @ *tl Ds*) = *last Ds*
proof(*cases tl Ds* = [])
case *True*
assume *last*:*last Cs* = *hd Ds*
with *True notempty* **have** *Ds* = [*last Cs*] **by** (*fastforce dest*:*hd-Cons-tl*)
hence *last Ds* = *last Cs* **by** *simp*
with *True* **show** *?thesis* **by** *simp*
next
case *False*
assume *last*:*last Cs* = *hd Ds*
from *notempty False* **have** *last* (*tl Ds*) = *last Ds*
by −(*drule hd-Cons-tl*,*drule-tac x*=*hd Ds* **in** *last-ConsR*,*simp*)
with *False* **show** *?thesis* **by** *simp*
qed
hence *eq*:(*Cs* $@_p$ *Ds*) $@_p$ [*last Ds*] = (*Cs* $@_p$ *Ds*)
by(*simp add*:*appendPath-def*)
from *least-method wf*
have $P \vdash$ *last Ds has least* $M = (Ts',T',pns',body')$ *via* [*last Ds*]
by(*auto dest*:*Subobj-last-isClass intro*:*Subobjs-Base subobjs-rel*

simp:*LeastMethodDef-def MethodDefs-def*)
with *notempty*
have $P \vdash$ *last* ($Cs@_p Ds$) *has least* $M = (Ts', T', pns', body')$ *via* [*last Ds*]
by −(*drule-tac* Cs'=*Cs* **in** *appendPath-last*,*simp*)
with *overrider wf eq* **have** $(Cs', Ts, T, pns, body) \in$ *MinimalMethodDefs P C M*
and $P, C \vdash Cs' \sqsubseteq Cs\ @_p\ Ds$
by −(*auto simp*:*FinalOverriderMethodDef-def OverriderMethodDefs-def*,
drule wf-sees-method-fun,*auto*)
with *sub wf path notempty has* **show** *?thesis*
by −(*drule leq-methods-subtypes*,*simp-all*,*blast*)+
qed
qed

lemma *wf-syscls*:
set SystemClasses ⊆ *set P* ⟹ *wf-syscls P*
by (*simp add*: *image-def SystemClasses-def wf-syscls-def sys-xcpts-def*
NullPointerC-def ClassCastC-def OutOfMemoryC-def,*force intro*:*conjI*)

17.9 Well formedness and widen

lemma *Class-widen*: ⟦$P \vdash$ *Class* $C \leq T$; *wf-prog wf-md P*; *is-class P C*⟧
⟹ ∃ *D*. T = *Class* $D \wedge P \vdash$ *Path C to D unique*

apply (*ind-cases* $P \vdash$ *Class* $C \leq T$)
apply (*auto intro*:*path-C-to-C-unique*)
done

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

apply (*rule iffI*)
apply (*ind-cases* $P \vdash$ *Class* $C \leq$ *Class* D)
apply (*auto elim*: *widen-subcls intro*:*path-C-to-C-unique*)
done

lemma *widen-Class*: ⟦*wf-prog wf-md P*; *is-class P C*⟧ ⟹
($P \vdash T \leq$ *Class* C) =
($T = NT \vee$ (∃ *D*. T = *Class* $D \wedge P \vdash$ *Path D to C unique*))

apply(*induct* T) **apply** (*auto intro*:*widen-subcls*)
apply (*ind-cases* $P \vdash$ *Class* $D \leq$ *Class* C **for** D) **apply** (*auto intro*:*path-C-to-C-unique*)
done

17.10 Well formedness and well typing

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

shows *WT-determ*: $P,E \vdash e :: T \Longrightarrow (\bigwedge T'.\ P,E \vdash e :: T' \Longrightarrow T = T')$
and *WTs-determ*: $P,E \vdash es\ [::]\ Ts \Longrightarrow (\bigwedge Ts'.\ P,E \vdash es\ [::]\ Ts' \Longrightarrow Ts = Ts')$

proof(*induct rule*:*WT-WTs-inducts*)
  case (*WTDynCast E e D C*)
  have $P,E \vdash Cast\ C\ e :: T'$ **by** *fact*
  thus *?case* **by** (*fastforce elim*:*WT.cases*)
next
  case (*WTStaticCast E e D C*)
  have $P,E \vdash (\!|C|\!)e :: T'$ **by** *fact*
  thus *?case* **by** (*fastforce elim*:*WT.cases*)
next
  case (*WTBinOp E* e_1 T_1 e_2 T_2 *bop T*)
  have *bop*:*case bop of Eq* $\Rightarrow T_1 = T_2 \wedge T = Boolean$
    | *Add* $\Rightarrow T_1 = Integer \wedge T_2 = Integer \wedge T = Integer$
    and *wt*:$P,E \vdash e_1 \ll bop \gg e_2 :: T'$ **by** *fact+*
  from *wt* **obtain** *T1' T2'* **where**
    bop':*case bop of Eq* $\Rightarrow T1' = T2' \wedge T' = Boolean$
    | *Add* $\Rightarrow T1' = Integer \wedge T2' = Integer \wedge T' = Integer$
    by *auto*
  from *bop* **show** *?case*
  proof (*cases bop*)
    assume *Eq*:*bop* = *Eq*
    with *bop* **have** *T* = *Boolean* **by** *auto*
    with *Eq bop'* **show** *?thesis* **by** *simp*
  next
    assume *Add*:*bop* = *Add*
    with *bop* **have** *T* = *Integer*
      by *auto*
    with *Add bop'* **show** *?thesis* **by** *simp*
  qed
next
  case (*WTLAss E V T e T' T''*)
  have $P,E \vdash V{:=}e :: T''$
    and *E V* = *Some T* **by** *fact+*
  thus *?case* **by** *auto*
next
  case (*WTFAcc E e C F T Cs*)
  have *IH*:$\bigwedge T'.\ P,E \vdash e :: T' \Longrightarrow Class\ C = T'$
    and *least*:$P \vdash C\ has\ least\ F{:}T\ via\ Cs$
    and *wt*:$P,E \vdash e{\cdot}F\{Cs\} :: T'$ **by** *fact+*
  from *wt* **obtain** *C'* **where** *wte'*:$P,E \vdash e :: Class\ C'$
    and *least'*:$P \vdash C'\ has\ least\ F{:}T'\ via\ Cs$ **by** *auto*
  from *IH*[*OF wte'*] **have** *C* = *C'* **by** *simp*
  with *least least'* **show** *?case*
    by (*fastforce simp*:*sees-field-fun*)
next
  case (*WTFAss E* e_1 *C F T Cs* e_2 *T' T''*)
  have *least*:$P \vdash C\ has\ least\ F{:}T\ via\ Cs$

```
    and wt:P,E ⊢ e₁·F{Cs} := e₂ :: T''
    and IH:⋀S. P,E ⊢ e₁ :: S ⟹ Class C = S by fact+
  from wt obtain C' where wte':P,E ⊢ e₁ :: Class C'
    and least':P ⊢ C' has least F:T'' via Cs by auto
  from IH[OF wte'] have C = C' by simp
  with least least' show ?case
    by (fastforce simp:sees-field-fun)
next
  case (WTCall E e C M Ts T pns body Cs es Ts')
  have IH:⋀T'. P,E ⊢ e :: T' ⟹ Class C = T'
    and least:P ⊢ C has least M = (Ts, T, pns, body) via Cs
    and wt:P,E ⊢ e·M(es) :: T' by fact+
  from wt obtain C' Ts' pns' body' Cs' where wte':P,E ⊢ e :: Class C'
    and least':P ⊢ C' has least M = (Ts',T',pns',body') via Cs' by auto
  from IH[OF wte'] have C = C' by simp
  with least least' wf show ?case by (auto dest:wf-sees-method-fun)
next
  case (WTStaticCall E e C' C M Ts T pns body Cs es Ts')
  have IH:⋀T'. P,E ⊢ e :: T' ⟹ Class C' = T'
    and unique:P ⊢ Path C' to C unique
    and least:P ⊢ C has least M = (Ts, T, pns, body) via Cs
    and wt:P,E ⊢ e·(C::)M(es) :: T' by fact+
  from wt obtain Ts' pns' body' Cs'
    where P ⊢ C has least M = (Ts',T',pns',body') via Cs' by auto
  with least wf show ?case by (auto dest:wf-sees-method-fun)
next
  case WTBlock thus ?case by (clarsimp simp del:fun-upd-apply)
next
  case (WTSeq E e₁ T₁ e₂ T₂)
  have IH:⋀T'. P,E ⊢ e₂ :: T' ⟹ T₂ = T'
    and wt:P,E ⊢ e₁;; e₂ :: T' by fact+
  from wt have wt':P,E ⊢ e₂ :: T' by auto
  from IH[OF wt'] show ?case .
next
  case (WTCond E e e₁ T e₂)
  have IH:⋀S. P,E ⊢ e₁ :: S ⟹ T = S
    and wt:P,E ⊢ if (e) e₁ else e₂ :: T' by fact+
  from wt have P,E ⊢ e₁ :: T' by auto
  from IH[OF this] show ?case .
next
  case (WTCons E e T es Ts)
  have IHe:⋀T'. P,E ⊢ e :: T' ⟹ T = T'
    and IHes:⋀Ts'. P,E ⊢ es [::] Ts' ⟹ Ts = Ts'
    and wt:P,E ⊢ e # es [::] Ts' by fact+
  from wt show ?case
  proof (cases Ts')
    case Nil with wt show ?thesis by simp
  next
    case (Cons T'' Ts'')
```

with *wt* **have** *wte'*:$P,E \vdash e :: T''$ **and** *wtes'*:$P,E \vdash es$ [::] *Ts''*
by *auto*
from *IHe*[*OF wte'*] *IHes*[*OF wtes'*] *Cons* **show** *?thesis* **by** *simp*
qed
qed *clarsimp+*

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*)
by(*simp add*:*wwf-mdecl-def*)

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'* ⟹ $P \vdash T$ *casts v' to v'*

proof(*induct rule*:*casts-to.induct*)
case *casts-prim* **thus** *?case* **by**(*rule casts-to.casts-prim*)
next
case (*casts-null C*) **thus** *?case* **by**(*rule casts-to.casts-null*)
next
case (*casts-ref Cs C Cs' Ds a*)

have *path-via*:$P \vdash$ *Path last Cs to C via Cs'* **and** *Ds*:$Ds = Cs \ @_p \ Cs'$ **by** *fact*+
with *wf* **have** *last Cs'* $= C$ **and** $Cs' \neq []$ **and** *class*: *is-class P C*
by(*auto intro*!:*Subobjs-nonempty Subobj-last-isClass simp*:*path-via-def*)
with *Ds* **have** *last*:*last Ds* $= C$
by −(*drule-tac Cs'* $= Cs$ **in** *appendPath-last*,*simp*)
hence *Ds'*:$Ds = Ds \ @_p \ [C]$ **by**(*simp add*:*appendPath-def*)
from *last class* **have** $P \vdash$ *Path last Ds to C via* $[C]$
by(*fastforce intro*:*Subobjs-Base simp*:*path-via-def*)
with *Ds'* **show** *?case* **by**(*fastforce intro*:*casts-to.casts-ref*)
qed

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

apply −
apply(*erule casts-to.cases*)
apply *clarsimp*
apply(*erule casts-to.cases*)
apply *simp*
apply *simp*
apply (*simp* (*asm-lr*))
apply(*erule casts-to.cases*)
apply *simp*
apply *simp*
apply *simp*
apply *simp*
apply(*erule casts-to.cases*)
apply *simp*
apply *simp*
apply *clarsimp*
apply(*erule appendPath-path-via*)
by *auto*

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

proof(*induct rule*:*red-reds-inducts*)
case (*RedLAss E V T' w w' h l V'*)
have *env*:$E\ V = Some\ T'$ **and** *env'*:$E\ V' = Some\ T$

```
    and l:l V' = None and lupd:(l(V ↦ w')) V' = Some v'
    and casts:P ⊢ T' casts w to w' by fact+
  show ?case
  proof(cases V = V')
    case True
    with lupd have v':v' = w' by simp
    from True env env' have T = T' by simp
    with v' casts wf show ?thesis by(fastforce intro:casts-casts)
  next
    case False
    with lupd have l V' = Some v' by(fastforce split:if-split-asm)
    with l show ?thesis by simp
  qed
next
  case (BlockRedNone E V T' e h l e' h' l' V')
  have l:l V' = None
    and l'upd:(l'(V := l V)) V' = Some v' and env:E V' = Some T
    and IH:⋀V'. ⟦(l(V := None)) V' = None; (E(V ↦ T')) V' = Some T;
                 l' V' = Some v'⟧
          ⟹ P ⊢ T casts v' to v' by fact+
  show ?case
  proof(cases V = V')
    case True
    with l'upd l show ?thesis by fastforce
  next
    case False
    with l l'upd have lnew:(l(V := None)) V' = None
      and l'new:l' V' = Some v' by (auto split:if-split-asm)
    from env False have env':(E(V ↦ T')) V' = Some T by fastforce
    from IH[OF lnew env' l'new] show ?thesis .
  qed
next
  case (BlockRedSome E V T' e h l e' h' l' v V')
  have l:l V' = None
    and l'upd:(l'(V := l V)) V' = Some v' and env:E V' = Some T
    and IH:⋀V'. ⟦(l(V := None)) V' = None; (E(V ↦ T')) V' = Some T;
                 l' V' = Some v'⟧
          ⟹ P ⊢ T casts v' to v' by fact+
  show ?case
  proof(cases V = V')
    case True
    with l l'upd show ?thesis by fastforce
  next
    case False
    with l l'upd have lnew:(l(V := None)) V' = None
      and l'new:l' V' = Some v' by (auto split:if-split-asm)
    from env False have env':(E(V ↦ T')) V' = Some T by fastforce
    from IH[OF lnew env' l'new] show ?thesis .
  qed
```

next
case (*InitBlockRed E V T′ e h l w′ e′ h′ l′ w″ w V′*)
have *l*:*l V′* = *None*
and *l′upd*:(*l′*(*V* := *l V*)) *V′* = *Some v′* **and** *env*:*E V′* = *Some T*
and *IH*:⋀*V′*. ⟦(*l*(*V* ↦ *w′*)) *V′* = *None*; (*E*(*V* ↦ *T′*)) *V′* = *Some T*;
l′ V′ = *Some v′*⟧
⟹ *P* ⊢ *T casts v′ to v′* **by** *fact*+
show *?case*
proof(*cases V* = *V′*)
case *True*
with *l l′upd* **show** *?thesis* **by** *fastforce*
next
case *False*
with *l l′upd* **have** *lnew*:(*l*(*V* ↦ *w′*)) *V′* = *None*
and *l′new*:*l′ V′* = *Some v′* **by** (*auto split*:*if-split-asm*)
from *env False* **have** *env′*:(*E*(*V* ↦ *T′*)) *V′* = *Some T* **by** *fastforce*
from *IH*[*OF lnew env′ l′new*] **show** *?thesis* **.**
qed
qed (*auto intro*:*casts-casts wf*)

lemma assumes *wf*:*wf-prog wf-md P*
shows *Some-lcl-casts-values*:
P,*E* ⊢ ⟨*e*,(*h*,*l*)⟩ → ⟨*e′*,(*h′*,*l′*)⟩ ⟹
(⋀*V*. ⟦*l V* = *Some v*; *E V* = *Some T*;
P ⊢ *T casts v″ to v*; *l′ V* = *Some v′*⟧
⟹ *P* ⊢ *T casts v′ to v′*)
and *P*,*E* ⊢ ⟨*es*,(*h*,*l*)⟩ [→] ⟨*es′*,(*h′*,*l′*)⟩ ⟹
(⋀*V*. ⟦*l V* = *Some v*; *E V* = *Some T*;
P ⊢ *T casts v″ to v*; *l′ V* = *Some v′*⟧
⟹ *P* ⊢ *T casts v′ to v′*)

proof(*induct rule*:*red-reds-inducts*)
case (*RedNew h a h′ C′ E l V*)
have *l1*:*l V* = *Some v* **and** *l2*:*l V* = *Some v′*
and *casts*:*P* ⊢ *T casts v″ to v* **by** *fact*+
from *l1 l2* **have** *eq*:*v* = *v′* **by** *simp*
with *casts wf* **show** *?case* **by**(*fastforce intro*:*casts-casts*)
next
case (*RedLAss E V T′ w w′ h l V′*)
have *l*:*l V′* = *Some v* **and** *lupd*:(*l*(*V* ↦ *w′*)) *V′* = *Some v′*
and *T′casts*:*P* ⊢ *T′ casts w to w′*
and *env*:*E V* = *Some T′* **and** *env′*:*E V′* = *Some T*
and *casts*:*P* ⊢ *T casts v″ to v* **by** *fact*+
show *?case*
proof (*cases V* = *V′*)
case *True*
with *lupd* **have** *v′*:*v′* = *w′* **by** *simp*

```
    from True env env' have T = T' by simp
    with T'casts v' wf show ?thesis by(fastforce intro:casts-casts)
  next
    case False
    with l lupd have v = v' by (auto split:if-split-asm)
    with casts wf show ?thesis by(fastforce intro:casts-casts)
  qed
next
  case (RedFAss h a D S Cs' F T' Cs w w' Ds fs E l V)
  have l1:l V = Some v and l2:l V = Some v'
    and hp:h a = Some(D, S)
    and T'casts:P ⊢ T' casts w to w'
    and casts:P ⊢ T casts v'' to v by fact+
  from l1 l2 have eq:v = v' by simp
  with casts wf show ?case by(fastforce intro:casts-casts)
next
  case (BlockRedNone E V T' e h l e' h' l' V')
  have l':l' V = None and l:l V' = Some v
    and l'upd:(l'(V := l V)) V' = Some v' and env:E V' = Some T
    and casts:P ⊢ T casts v'' to v
    and IH:⋀V'. ⟦(l(V := None)) V' = Some v; (E(V ↦ T')) V' = Some T;
                 P ⊢ T casts v'' to v ; l' V' = Some v'⟧
          ⟹ P ⊢ T casts v' to v' by fact+
  show ?case
  proof(cases V = V')
    case True
    with l' l'upd have l V = Some v' by auto
    with True l have eq:v = v' by simp
    with casts wf show ?thesis by(fastforce intro:casts-casts)
  next
    case False
    with l l'upd have lnew:(l(V := None)) V' = Some v
      and l'new:l' V' = Some v' by (auto split:if-split-asm)
    from env False have env':(E(V ↦ T')) V' = Some T by fastforce
    from IH[OF lnew env' casts l'new] show ?thesis .
  qed
next
  case (BlockRedSome E V T' e h l e' h' l' w V')
  have l':l' V = Some w and l:l V' = Some v
    and l'upd:(l'(V := l V)) V' = Some v' and env:E V' = Some T
    and casts:P ⊢ T casts v'' to v
    and IH:⋀V'. ⟦(l(V := None)) V' = Some v; (E(V ↦ T')) V' = Some T;
                 P ⊢ T casts v'' to v ; l' V' = Some v'⟧
          ⟹ P ⊢ T casts v' to v' by fact+
  show ?case
  proof(cases V = V')
    case True
    with l' l'upd have l V = Some v' by auto
    with True l have eq:v = v' by simp
```

with *casts wf* **show** *?thesis* **by**(*fastforce intro*:*casts-casts*)
next
case *False*
with *l l'upd* **have** *lnew*:(*l*(*V* := *None*)) *V'* = *Some v*
and *l'new*:*l' V'* = *Some v'* **by** (*auto split*:*if-split-asm*)
from *env False* **have** *env'*:(*E*(*V* ↦ *T'*)) *V'* = *Some T* **by** *fastforce*
from *IH*[*OF lnew env' casts l'new*] **show** *?thesis* **.**
qed
next
case (*InitBlockRed E V T' e h l w' e' h' l' w'' w V'*)
have *l*:*l V'* = *Some v* **and** *l'*:*l' V* = *Some w''*
and *l'upd*:(*l'*(*V* := *l V*)) *V'* = *Some v'* **and** *env*:*E V'* = *Some T*
and *casts*:*P* ⊢ *T casts v'' to v*
and *IH*:⋀*V'*. ⟦(*l*(*V* ↦ *w'*)) *V'* = *Some v*; (*E*(*V* ↦ *T'*)) *V'* = *Some T*;
P ⊢ *T casts v'' to v* ; *l' V'* = *Some v'*⟧
⟹ *P* ⊢ *T casts v' to v'* **by** *fact+*
show *?case*
proof(*cases V* = *V'*)
case *True*
with *l' l'upd* **have** *l V* = *Some v'* **by** *auto*
with *True l* **have** *eq*:*v* = *v'* **by** *simp*
with *casts wf* **show** *?thesis* **by**(*fastforce intro*:*casts-casts*)
next
case *False*
with *l l'upd* **have** *lnew*:(*l*(*V* ↦ *w'*)) *V'* = *Some v*
and *l'new*:*l' V'* = *Some v'* **by** (*auto split*:*if-split-asm*)
from *env False* **have** *env'*:(*E*(*V* ↦ *T'*)) *V'* = *Some T* **by** *fastforce*
from *IH*[*OF lnew env' casts l'new*] **show** *?thesis* **.**
qed
qed (*auto intro*:*casts-casts wf*)

19.2 Small steps simulate big step

19.3 Cast

lemma *StaticCastReds*:
P,*E* ⊢ ⟨*e*,*s*⟩ →* ⟨*e'*,*s'*⟩ ⟹ *P*,*E* ⊢ ⟨(⌊*C*⌋)*e*,*s*⟩ →* ⟨(⌊*C*⌋)*e'*,*s'*⟩

apply(*erule rtrancl-induct2*)
apply *blast*
apply(*erule rtrancl-into-rtrancl*)
apply (*simp add*:*StaticCastRed*)
done

lemma *StaticCastRedsNull*:
P,*E* ⊢ ⟨*e*,*s*⟩ →* ⟨*null*,*s'*⟩ ⟹ *P*,*E* ⊢ ⟨(⌊*C*⌋)*e*,*s*⟩ →* ⟨*null*,*s'*⟩

apply(*rule rtrancl-into-rtrancl*)
apply(*erule StaticCastReds*)

apply(*simp add*:*RedStaticCastNull*)
done

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

apply(*rule rtrancl-into-rtrancl*)
apply(*erule StaticCastReds*)
apply(*fastforce intro*:*RedStaticUpCast*)
done

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

apply(*rule rtrancl-into-rtrancl*)
apply(*erule StaticCastReds*)
apply *simp*
apply(*subgoal-tac* $P,E \vdash \langle(\!|\, C\,|\!)ref(a,Cs@[C]@Cs'),s'\rangle \to \langle ref(a,Cs@[C]),s'\rangle$)
apply *simp*
apply(*rule RedStaticDownCast*)
done

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

apply(*rule rtrancl-into-rtrancl*)
apply(*erule StaticCastReds*)
apply(*fastforce intro*:*RedStaticCastFail*)
done

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

apply(*rule rtrancl-into-rtrancl*)
apply(*erule StaticCastReds*)
apply(*simp add*:*red-reds.StaticCastThrow*)
done

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

apply(*erule rtrancl-induct2*)
apply *blast*
apply(*erule rtrancl-into-rtrancl*)
apply (*simp add*:*DynCastRed*)
done

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

apply(*rule rtrancl-into-rtrancl*)
apply(*erule DynCastReds*)
apply(*simp add*:*RedDynCastNull*)
done

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

apply(*rule rtrancl-into-rtrancl*)
apply(*erule DynCastReds*)
apply(*fastforce intro*:*RedDynCast*)
done

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

apply(*rule rtrancl-into-rtrancl*)
apply(*erule DynCastReds*)
apply(*fastforce intro*:*RedStaticUpDynCast*)
done

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

apply(*rule rtrancl-into-rtrancl*)
apply(*erule DynCastReds*)
apply *simp*
apply(*subgoal-tac* $P,E \vdash \langle Cast\ C\ (ref(a,Cs@[C]@Cs')),s'\rangle \to \langle ref(a,Cs@[C]),s'\rangle$)
apply *simp*

apply(*rule RedStaticDownDynCast*)
done

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

apply(*rule rtrancl-into-rtrancl*)
apply(*erule DynCastReds*)
apply(*fastforce intro*:*RedDynCastFail*)
done

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

apply(*rule rtrancl-into-rtrancl*)
apply(*erule DynCastReds*)
apply(*simp add*:*red-reds.DynCastThrow*)
done

19.4 LAss

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

apply(*erule rtrancl-induct2*)
apply *blast*
apply(*erule rtrancl-into-rtrancl*)
apply(*simp add*:*LAssRed*)
done

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

apply(*rule rtrancl-into-rtrancl*)
apply(*erule LAssReds*)
apply(*simp add*:*RedLAss*)
done

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

apply(*rule rtrancl-into-rtrancl*)
 apply(*erule LAssReds*)
apply(*simp add*:*red-reds.LAssThrow*)
done

19.5 BinOp

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

apply(*erule rtrancl-induct2*)
 apply *blast*
apply(*erule rtrancl-into-rtrancl*)
apply(*simp add*:*BinOpRed1*)
done

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

apply(*erule rtrancl-induct2*)
 apply *blast*
apply(*erule rtrancl-into-rtrancl*)
apply(*simp add*:*BinOpRed2*)
done

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

apply(*rule rtrancl-trans*)
 apply(*erule BinOp1Reds*)
apply(*rule rtrancl-into-rtrancl*)
 apply(*erule BinOp2Reds*)
apply(*simp add*:*RedBinOp*)
done

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

apply(*rule rtrancl-into-rtrancl*)
 apply(*erule BinOp1Reds*)
apply(*simp add*:*red-reds.BinOpThrow1*)
done

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

apply(*rule rtrancl-trans*)
apply(*erule BinOp1Reds*)
apply(*rule rtrancl-into-rtrancl*)
apply(*erule BinOp2Reds*)
apply(*simp add*:*red-reds.BinOpThrow2*)
done

19.6 FAcc

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

apply(*erule rtrancl-induct2*)
apply *blast*
apply(*erule rtrancl-into-rtrancl*)
apply(*simp add*:*FAccRed*)
done

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

apply(*rule rtrancl-into-rtrancl*)
apply(*erule FAccReds*)
apply (*fastforce intro*:*RedFAcc*)
done

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

apply(*rule rtrancl-into-rtrancl*)
apply(*erule FAccReds*)
apply(*simp add*:*RedFAccNull*)
done

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

apply(*rule rtrancl-into-rtrancl*)
apply(*erule FAccReds*)

apply(*simp add*:*red-reds.FAccThrow*)
done

19.7 FAss

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

apply(*erule rtrancl-induct2*)
apply *blast*
apply(*erule rtrancl-into-rtrancl*)
apply(*simp add*:*FAssRed1*)
done

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

apply(*erule rtrancl-induct2*)
apply *blast*
apply(*erule rtrancl-into-rtrancl*)
apply(*simp add*:*FAssRed2*)
done

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

apply(*rule rtrancl-trans*)
apply(*erule FAssReds1*)
apply(*rule rtrancl-into-rtrancl*)
apply(*erule FAssReds2*)
apply(*fastforce intro*:*RedFAss*)
done

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

apply(*rule rtrancl-trans*)
apply(*erule FAssReds1*)
apply(*rule rtrancl-into-rtrancl*)
apply(*erule FAssReds2*)

apply(*simp add*:*RedFAssNull*)
done

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

apply(*rule rtrancl-into-rtrancl*)
apply(*erule FAssReds1*)
apply(*simp add*:*red-reds.FAssThrow1*)
done

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

apply(*rule rtrancl-trans*)
apply(*erule FAssReds1*)
apply(*rule rtrancl-into-rtrancl*)
apply(*erule FAssReds2*)
apply(*simp add*:*red-reds.FAssThrow2*)
done

19.8 ;;

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

apply(*erule rtrancl-induct2*)
apply *blast*
apply(*erule rtrancl-into-rtrancl*)
apply(*simp add*:*SeqRed*)
done

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

apply(*rule rtrancl-into-rtrancl*)
apply(*erule SeqReds*)
apply(*simp add*:*red-reds.SeqThrow*)
done

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

apply(*rule rtrancl-trans*)
 apply(*erule SeqReds*)
apply(*rule-tac b*=(e_2,s_1) **in** *converse-rtrancl-into-rtrancl*)
 apply(*simp add*:*RedSeq*)
apply *assumption*
done

19.9 If

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

apply(*erule rtrancl-induct2*)
 apply *blast*
apply(*erule rtrancl-into-rtrancl*)
apply(*simp add*:*CondRed*)
done

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

apply(*rule rtrancl-into-rtrancl*)
 apply(*erule CondReds*)
apply(*simp add*:*red-reds.CondThrow*)
done

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

apply(*rule rtrancl-trans*)
 apply(*erule CondReds*)
apply(*rule-tac b*=(e_1, s_1) **in** *converse-rtrancl-into-rtrancl*)
 apply(*simp add*:*RedCondT*)
apply *assumption*
done

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

apply(*rule rtrancl-trans*)
 apply(*erule CondReds*)
apply(*rule-tac b*=(e_2, s_1) **in** *converse-rtrancl-into-rtrancl*)
 apply(*simp add*:*RedCondF*)
apply *assumption*

done

19.10 While

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

apply(*rule-tac b=*(*if*(*b*) (*c*;;*while*(*b*) *c*) *else unit*, *s*) **in** *converse-rtrancl-into-rtrancl*)
apply(*simp add*:*RedWhile*)
apply(*rule rtrancl-into-rtrancl*)
apply(*erule CondReds*)
apply(*simp add*:*RedCondF*)
done

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

apply(*rule-tac b=*(*if*(*b*) (*c*;;*while*(*b*) *c*) *else unit*, *s*) **in** *converse-rtrancl-into-rtrancl*)
apply(*simp add*:*RedWhile*)
apply(*rule rtrancl-into-rtrancl*)
apply(*erule CondReds*)
apply(*simp add*:*red-reds.CondThrow*)
done

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

apply(*rule-tac b=*(*if*(*b*) (*c*;;*while*(*b*) *c*) *else unit*, s_0) **in** *converse-rtrancl-into-rtrancl*)
apply(*simp add*:*RedWhile*)
apply(*rule rtrancl-trans*)
apply(*erule CondReds*)
apply(*rule-tac b=*(*c*;;*while*(*b*) *c*,s_1) **in** *converse-rtrancl-into-rtrancl*)
apply(*simp add*:*RedCondT*)
apply(*rule rtrancl-trans*)
apply(*erule SeqReds*)
apply(*rule-tac b=*(*while*(*b*) *c*,s_2) **in** *converse-rtrancl-into-rtrancl*)
apply(*simp add*:*RedSeq*)
apply *assumption*
done

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

apply(*rule-tac b*=(*if*(*b*) (*c*;;*while*(*b*) *c*) *else unit*, s_0) **in** *converse-rtrancl-into-rtrancl*)
 apply(*simp add*:*RedWhile*)
apply(*rule rtrancl-trans*)
 apply(*erule CondReds*)
apply(*rule-tac b*=(*c*;;*while*(*b*) *c*,s_1) **in** *converse-rtrancl-into-rtrancl*)
 apply(*simp add*:*RedCondT*)
apply(*rule rtrancl-trans*)
 apply(*erule SeqReds*)
apply(*rule-tac b*=(*Throw r*,s_2) **in** *converse-rtrancl-into-rtrancl*)
 apply(*simp add*:*red-reds.SeqThrow*)
apply *simp*
done

19.11 Throw

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

apply(*erule rtrancl-induct2*)
 apply *blast*
apply(*erule rtrancl-into-rtrancl*)
apply(*simp add*:*ThrowRed*)
done

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

apply(*rule rtrancl-into-rtrancl*)
 apply(*erule ThrowReds*)
apply(*simp add*:*RedThrowNull*)
done

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

apply(*rule rtrancl-into-rtrancl*)
 apply(*erule ThrowReds*)
apply(*simp add*:*red-reds.ThrowThrow*)
done

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 (*erule converse-rtrancl-induct2*)
{ fix $h\ l\ h'\ l'\ v\ v'$
assume $s' = (h,\ l(V \mapsto v'))$ **and** $s' = (h',\ l')$
hence $h{:}h = h'$ **and** $l'{:}l' = l(V \mapsto v')$ **by** *simp-all*
hence $P{,}E \vdash \langle\{V{:}T;\ V{:=}Val\ v;;\ e'\},(h,\ l)\rangle \to*$
$\langle\{V{:}T;\ V{:=}Val\ v;;\ e'\},(h',\ l'(V := l\ V))\rangle$
by(*fastforce simp*: *fun-upd-same simp del*:*fun-upd-apply*) **}**
hence $\forall h\ l\ h'\ l'\ v\ v'.$
$s' = (h,\ l(V \mapsto v')) \longrightarrow$
$P \vdash T\ casts\ v\ to\ v' \longrightarrow$
$s' = (h',\ l') \longrightarrow$
$P{,}E \vdash \langle\{V{:}T;\ V{:=}Val\ v;;\ e'\},(h,\ l)\rangle \to*$
$\langle\{V{:}T;\ V{:=}Val\ v;;\ e'\},(h',\ l'(V := l\ V))\rangle \wedge$
$P \vdash T\ casts\ v\ to\ v'$
by *auto*
thus $\forall h\ l\ h'\ l'\ v\ v'.$
$s' = (h,\ l(V \mapsto v')) \longrightarrow$
$P \vdash T\ casts\ v\ to\ v' \longrightarrow$
$s' = (h',\ l') \longrightarrow$
$(\exists v''\ w.\ P{,}E \vdash \langle\{V{:}T;\ V{:=}Val\ v;;\ e'\},(h,\ l)\rangle \to*$
$\langle\{V{:}T;\ V{:=}Val\ v'';;\ e'\},(h',\ l'(V := l\ V))\rangle \wedge$
$P \vdash T\ casts\ v''\ to\ w)$
by *auto*
next
fix $e\ s\ e''\ s''$
assume *Red*:$((e,s),e'',s'') \in Red\ P\ (E(V \mapsto T))$
and *reds*:$P{,}E(V \mapsto T) \vdash \langle e'',s''\rangle \to* \langle e',s'\rangle$
and *IH*:$\forall h\ l\ h'\ l'\ v\ v'.$
$s'' = (h,\ l(V \mapsto v')) \longrightarrow$
$P \vdash T\ casts\ v\ to\ v' \longrightarrow$
$s' = (h',\ l') \longrightarrow$
$(\exists v''\ w.\ P{,}E \vdash \langle\{V{:}T;\ V{:=}Val\ v;;\ e''\},(h,\ l)\rangle \to*$
$\langle\{V{:}T;\ V{:=}Val\ v'';;\ e'\},(h',\ l'(V := l\ V))\rangle \wedge$
$P \vdash T\ casts\ v''\ to\ w)$
{ fix $h\ l\ h'\ l'\ v\ v'$
assume $s{:}s = (h,\ l(V \mapsto v'))$ **and** $s'{:}s' = (h',\ l')$
and *casts*:$P \vdash T\ casts\ v\ to\ v'$
obtain $h''\ l''$ **where** $s''{:}s'' = (h'',l'')$ **by** (*cases* s'') *auto*
with *Red* s **have** $V \in dom\ l''$ **by** (*fastforce dest*:*red-lcl-incr*)
then obtain v'' **where** $l''{:}l''\ V = Some\ v''$ **by** *auto*
with *Red* $s\ s''$ *casts*
have *step*:$P{,}E \vdash \langle\{V{:}T := Val\ v;\ e\},(h,\ l)\rangle \to$
$\langle\{V{:}T := Val\ v'';\ e''\},\ (h'',l''(V := l\ V))\rangle$
by(*fastforce intro*:*InitBlockRed*)
from *Red* $s\ s''\ l''$ *casts wf*
have *casts'*:$P \vdash T\ casts\ v''\ to\ v''$ **by**(*fastforce intro*:*Some-lcl-casts-values*)
with *IH* $s''\ s'\ l''$ **obtain** $v'''\ w$

```
    where P,E ⊢ ⟨{V:T := Val v''; e''}, (h'',l''(V := l V))⟩ →*
                ⟨{V:T := Val v'''; e'},(h', l'(V := l V))⟩ ∧
          P ⊢ T casts v''' to w
    apply simp
    apply (erule-tac x = l''(V := l V) in allE)
    apply (erule-tac x = v'' in allE)
    apply (erule-tac x = v'' in allE)
    by(auto intro:ext)
  with step have ∃ v'' w. P,E ⊢ ⟨{V:T; V:=Val v;; e},(h, l)⟩ →*
                                ⟨{V:T; V:=Val v'';; e'},(h', l'(V := l V))⟩ ∧
                           P ⊢ T casts v'' to w
    apply(rule-tac x=v''' in exI)
    apply auto
     apply (rule converse-rtrancl-into-rtrancl)
    by simp-all }
 thus ∀ h l h' l' v v'.
         s = (h, l(V ↦ v')) ⟶
         P ⊢ T casts v to v' ⟶
         s' = (h', l') ⟶
         (∃ v'' w. P,E ⊢ ⟨{V:T; V:=Val v;; e},(h, l)⟩ →*
                         ⟨{V:T; V:=Val v'';; e'},(h', l'(V := l V))⟩ ∧
                   P ⊢ T casts v'' to w)
  by auto
qed



lemma InitBlockReds:
 ⟦P,E(V ↦ T) ⊢ ⟨e, (h,l(V↦v'))⟩ →* ⟨e', (h',l')⟩;
   P ⊢ T casts v to v'; wf-prog wf-md P ⟧ ⟹
 ∃ v'' w. P,E ⊢ ⟨{V:T := Val v; e}, (h,l)⟩ →*
           ⟨{V:T := Val v''; e'}, (h',l'(V:=(l V)))⟩ ∧
         P ⊢ T casts v'' to w
by(blast dest:InitBlockReds-aux)

lemma InitBlockRedsFinal:
  assumes reds:P,E(V ↦ T) ⊢ ⟨e,(h,l(V↦v'))⟩ →* ⟨e',(h',l')⟩
  and final:final e' and casts:P ⊢ T casts v to v'
  and wf:wf-prog wf-md P
  shows P,E ⊢ ⟨{V:T := Val v; e},(h,l)⟩ →* ⟨e',(h', l'(V := l V))⟩
proof −
  from reds casts wf obtain v'' and w
    where steps:P,E ⊢ ⟨{V:T := Val v; e},(h,l)⟩ →*
                      ⟨{V:T := Val v''; e'}, (h',l'(V:=(l V)))⟩
    and casts':P ⊢ T casts v'' to w
    by (auto dest:InitBlockReds)
  from final casts casts'
  have step:P,E ⊢ ⟨{V:T := Val v''; e'}, (h',l'(V:=(l V)))⟩ →
                  ⟨e',(h',l'(V := l V))⟩
```

by(*auto elim*!:*finalE intro*:*RedInitBlock InitBlockThrow*)
from *step steps* **show** *?thesis*
by(*fastforce intro*:*rtrancl-into-rtrancl*)
qed

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

using *reds*
proof (*induct rule*:*converse-rtrancl-induct2*)
case *refl* **thus** *?case*
by(*fastforce intro*:*finalE*[*OF fin*] *RedBlock BlockThrow*
simp del:*fun-upd-apply*)
next
case (*step* $e_0\ s_0\ e_1\ s_1$)
have *Red*: $((e_0,s_0),e_1,s_1) \in Red\ P\ (E(V \mapsto T))$
and *reds*: $P,E(V \mapsto T) \vdash \langle e_1,s_1\rangle \to* \langle e_2,(h_2,l_2)\rangle$
and *IH*: $\bigwedge h\ l.\ s_1 = (h,l(V := None)) \Longrightarrow P,E \vdash \langle\{V:T;\ e_1\},(h,l)\rangle \to* \langle e_2,(h_2,\ l_2(V := l\ V))\rangle$
and s_0: $s_0 = (h_0,\ l_0(V := None))$ **by** *fact+*
obtain $h_1\ l_1$ **where** s_1: $s_1 = (h_1,l_1)$ **by** *fastforce*
show *?case*
proof *cases*
assume *assigned* $V\ e_0$
then obtain $v\ e$ **where** e_0: $e_0 = V:= Val\ v;;\ e$
by (*unfold assigned-def*)*blast*
from *Red* $e_0\ s_0$ **obtain** v' **where** e_1: $e_1 = Val\ v';;e$
and s_1: $s_1 = (h_0,\ l_0(V \mapsto v'))$ **and** *casts*:$P \vdash T$ *casts* v *to* v'
by *auto*
from e_1 *fin* **have** $e_1 \neq e_2$ **by** (*auto simp*:*final-def*)
then obtain $e'\ s'$ **where** *red1*: $P,E(V \mapsto T) \vdash \langle e_1,s_1\rangle \to \langle e',s'\rangle$
and *reds'*: $P,E(V \mapsto T) \vdash \langle e',s'\rangle \to* \langle e_2,(h_2,l_2)\rangle$
using *converse-rtranclE2*[*OF reds*] **by** *simp blast*
from *red1* e_1 **have** *es'*: $e' = e\ s' = s_1$ **by** *auto*
show *?thesis* **using** $e_0\ s_1$ *es' reds'*
by(*fastforce intro*!: *InitBlockRedsFinal*[*OF - fin casts wf*]
simp del:*fun-upd-apply*)
next
assume *unass*: ¬ *assigned* $V\ e_0$
show *?thesis*
proof (*cases* $l_1\ V$)
assume *None*: $l_1\ V = None$
hence $P,E \vdash \langle\{V:T;\ e_0\},(h_0,l_0)\rangle \to \langle\{V:T;\ e_1\},(h_1,\ l_1(V := l_0\ V))\rangle$
using $s_0\ s_1$ *Red* **by**(*simp add*: *BlockRedNone*[*OF - - unass*])

moreover
have $P,E \vdash \langle\{V{:}T;\ e_1\},(h_1,\ l_1(V\ :=\ l_0\ V))\rangle \to* \langle e_2,(h_2,\ l_2(V\ :=\ l_0\ V))\rangle$
using *IH*[*of* - $l_1(V\ :=\ l_0\ V)$] s_1 *None* **by**(*simp add*:*fun-upd-idem*)
ultimately show *?case*
by(*rule-tac b*=$(\{V{:}T;\ e_1\},(h_1,\ l_1(V\ :=\ l_0\ V)))$ **in** *converse-rtrancl-into-rtrancl*,*simp*)
next
fix v **assume** *Some*: $l_1\ V\ =\ Some\ v$
with *Red Some* $s_0\ s_1$ *wf*
have *casts*:$P \vdash T$ *casts* v *to* v
by(*fastforce intro*:*None-lcl-casts-values*)
from *Some*
have $P,E \vdash \langle\{V{:}T;e_0\},(h_0,l_0)\rangle \to \langle\{V{:}T\ :=\ Val\ v;\ e_1\},(h_1,l_1(V\ :=\ l_0\ V))\rangle$
using $s_0\ s_1$ *Red* **by**(*simp add*: *BlockRedSome*[*OF* - - *unass*])
moreover
have $P,E \vdash \langle\{V{:}T\ :=\ Val\ v;\ e_1\},(h_1,l_1(V{:=}\ l_0\ V))\rangle \to*$
$\langle e_2,(h_2,l_2(V{:=}l_0\ V))\rangle$
using *InitBlockRedsFinal*[*OF* - *fin casts wf*,*of* - - $l_1(V{:=}l_0\ V)\ V$]
Some reds s_1
by (*simp add*:*fun-upd-idem*)
ultimately show *?case*
by(*rule-tac b*=$(\{V{:}T;\ V{:=}Val\ v;;\ e_1\},(h_1,\ l_1(V\ :=\ l_0\ V)))$ **in** *converse-rtrancl-into-rtrancl*,*simp*)
qed
qed
qed

19.14 List

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

apply(*erule rtrancl-induct2*)
apply *blast*
apply(*erule rtrancl-into-rtrancl*)
apply(*simp add*:*ListRed1*)
done

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

apply(*erule rtrancl-induct2*)
apply *blast*
apply(*erule rtrancl-into-rtrancl*)
apply(*simp add*:*ListRed2*)
done

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

$\Longrightarrow P,E \vdash \langle e\#es,s_0\rangle\ [\rightarrow]*\ \langle Val\ v\ \#\ es',s_2\rangle$

apply(*rule rtrancl-trans*)
apply(*erule ListReds1*)
apply(*erule ListReds2*)
done

19.15 Call

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

lemma assumes *wf*: *wwf-prog P*
shows *Red-fv*: $P,E \vdash \langle e,(h,l)\rangle \rightarrow \langle e',(h',l')\rangle \Longrightarrow fv\ e' \subseteq fv\ e$
and $P,E \vdash \langle es,(h,l)\rangle\ [\rightarrow]\ \langle es',(h',l')\rangle \Longrightarrow fvs\ es' \subseteq fvs\ es$
proof (*induct rule*:*red-reds-inducts*)
case (*RedCall h l a C S Cs M Ts' T' pns' body' Ds Ts T pns body Cs' vs bs new-body E*)
hence *fv body* $\subseteq$ {*this*} $\cup$ *set pns*
using *assms* **by**(*fastforce dest*!:*select-method-wf-mdecl simp*:*wf-mdecl-def*)
with *RedCall.hyps* **show** *?case*
by(*cases T'*) *auto*
next
case (*RedStaticCall Cs C Cs'' M Ts T pns body Cs' Ds vs E a a' b*)
hence *fv body* $\subseteq$ {*this*} $\cup$ *set pns*
using *assms* **by**(*fastforce dest*!:*has-least-wf-mdecl simp*:*wf-mdecl-def*)
with *RedStaticCall.hyps* **show** *?case*
by *auto*
qed *auto*

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

proof (*induct rule*:*red-reds-inducts*)
case *RedLAss* **thus** *?case* **by**(*force split*:*if-splits*)
next
case *CallParams* **thus** *?case* **by**(*force split*:*if-splits*)
next
case *BlockRedNone* **thus** *?case* **by** *clarsimp* (*fastforce split*:*if-splits*)
next
case *BlockRedSome* **thus** *?case* **by** *clarsimp* (*fastforce split*:*if-splits*)
next
case *InitBlockRed* **thus** *?case* **by** *clarsimp* (*fastforce split*:*if-splits*)
qed *auto*

lemma *Reds-dom-lcl*:

⟦ *wwf-prog P*; *P*,*E* ⊢ ⟨*e*,(*h*,*l*)⟩ →∗ ⟨*e*′,(*h*′,*l*′)⟩ ⟧ ⟹ *dom l*′ ⊆ *dom l* ∪ *fv e*

apply(*erule converse-rtrancl-induct-red*)
apply *blast*
apply(*blast dest*: *Red-fv Red-dom-lcl*)
done

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

lemma *override-on-upd-lemma*:
(*override-on f* (*g*(*a*↦*b*)) *A*)(*a* := *g a*) = *override-on f g* (*insert a A*)

apply(*rule ext*)
apply(*simp add*:*override-on-def*)
done

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

lemma assumes *wf*:*wf-prog wf-md P*
shows *blocksReds*:
⋀*l*$_0$ *E vs*′. ⟦ *length Vs* = *length Ts*; *length vs* = *length Ts*;
distinct Vs; ~~*P*,*E*(*Vs* [↦] *Ts*) ⊢ *Ts Casts vs to vs*′~~; *P* ⊢ *Ts Casts vs to vs*′;
P,*E*(*Vs* [↦] *Ts*) ⊢ ⟨*e*, (*h*$_0$,*l*$_0$(*Vs* [↦] *vs*′))⟩ →∗ ⟨*e*′, (*h*$_1$,*l*$_1$)⟩ ⟧
⟹ ∃ *vs*′′. *P*,*E* ⊢ ⟨*blocks*(*Vs*,*Ts*,*vs*,*e*), (*h*$_0$,*l*$_0$)⟩ →∗
⟨*blocks*(*Vs*,*Ts*,*vs*′′,*e*′), (*h*$_1$,*override-on l*$_1$ *l*$_0$ (*set Vs*))⟩ ∧
(∃ *ws*. *P* ⊢ *Ts Casts vs*′′ *to ws*) ∧ *length vs* = *length vs*′′

proof(*induct Vs Ts vs e rule*:*blocks-old-induct*)
case (5 *V Vs T Ts v vs e*)
have *length1*:*length* (*V*#*Vs*) = *length* (*T*#*Ts*)
and *length2*:*length* (*v*#*vs*) = *length* (*T*#*Ts*)
and *dist*:*distinct* (*V*#*Vs*)
and *casts*:*P* ⊢ (*T*#*Ts*) *Casts* (*v*#*vs*) *to vs*′
and *reds*:*P*,*E*(*V*#*Vs* [↦] *T*#*Ts*) ⊢ ⟨*e*,(*h*$_0$,*l*$_0$(*V*#*Vs* [↦] *vs*′))⟩ →∗ ⟨*e*′,(*h*$_1$,*l*$_1$)⟩
and *IH*:⋀*l*$_0$ *E vs*′′. ⟦*length Vs* = *length Ts*; *length vs* = *length Ts*;
distinct Vs; *P* ⊢ *Ts Casts vs to vs*′′;
P,*E*(*Vs* [↦] *Ts*) ⊢ ⟨*e*,(*h*$_0$,*l*$_0$(*Vs* [↦] *vs*′′))⟩ →∗ ⟨*e*′,(*h*$_1$,*l*$_1$)⟩⟧
⟹ ∃ *vs*′′. *P*,*E* ⊢ ⟨*blocks* (*Vs*,*Ts*,*vs*,*e*),(*h*$_0$,*l*$_0$)⟩ →∗
⟨*blocks* (*Vs*,*Ts*,*vs*′′,*e*′),(*h*$_1$, *override-on l*$_1$ *l*$_0$ (*set Vs*))⟩ ∧
(∃ *ws*. *P* ⊢ *Ts Casts vs*′′ *to ws*) ∧ *length vs* = *length vs*′′ **by** *fact*+
from *length1* **have** *length1*′:*length Vs* = *length Ts* **by** *simp*
from *length2* **have** *length2*′:*length vs* = *length Ts* **by** *simp*
from *dist* **have** *dist*′:*distinct Vs* **by** *simp*
from *casts* **obtain** *x xs* **where** *vs*′:*vs*′ = *x*#*xs*
by(*cases vs*′,*auto dest*:*length-Casts-vs*′)
with *reds*
have *reds*′:*P*,*E*(*V* ↦ *T*)(*Vs* [↦] *Ts*) ⊢ ⟨*e*,(*h*$_0$,*l*$_0$(*V* ↦ *x*)(*Vs* [↦] *xs*))⟩

→∗ ⟨*e*′,(h_1,l_1)⟩
by *simp*
from *casts vs*′ **have** *casts*′:*P* ⊢ *Ts Casts vs to xs*
and *cast*′:*P* ⊢ *T casts v to x*
by(*auto elim*:*Casts-to.cases*)
from *IH*[*OF length1*′ *length2*′ *dist*′ *casts*′ *reds*′]
obtain *vs*′′ *ws*
where *blocks*:*P*,*E*(*V* ↦ *T*) ⊢ ⟨*blocks* (*Vs*, *Ts*, *vs*, *e*),(h_0, l_0(*V* ↦ *x*))⟩ →∗
⟨*blocks* (*Vs*, *Ts*, *vs*′′, *e*′),(h_1, *override-on* l_1 (l_0(*V* ↦ *x*)) (*set Vs*))⟩
and *castsws*:*P* ⊢ *Ts Casts vs*′′ *to ws*
and *lengthvs*′′:*length vs* = *length vs*′′ **by** *auto*
from *InitBlockReds*[*OF blocks cast*′ *wf*] **obtain** *v*′′ *w* **where**
blocks′:*P*,*E* ⊢ ⟨{*V*:*T*; *V*:=*Val v*;; *blocks* (*Vs*, *Ts*, *vs*, *e*)},(h_0, l_0)⟩ →∗
⟨{*V*:*T*; *V*:=*Val v*′′;; *blocks* (*Vs*, *Ts*, *vs*′′, *e*′)},
(h_1, (*override-on* l_1 (l_0(*V* ↦ *x*)) (*set Vs*))(*V* := l_0 *V*))⟩
and *P* ⊢ *T casts v*′′ *to w* **by** *auto*
with *castsws* **have** *P* ⊢ *T*#*Ts Casts v*′′#*vs*′′ *to w*#*ws*
by −(*rule Casts-Cons*)
with *blocks*′ *lengthvs*′′ **show** *?case*
by(*rule-tac x*=*v*′′#*vs*′′ **in** *exI*,*auto simp*:*override-on-upd-lemma*)
next
case (*6 e*)
have *casts*:*P* ⊢ [] *Casts* [] *to vs*′
and *step*:*P*,*E*([] [↦] []) ⊢ ⟨*e*,(h_0, l_0([] [↦] *vs*′))⟩ →∗ ⟨*e*′,(h_1, l_1)⟩ **by** *fact*+
from *casts* **have** *vs*′ = [] **by**(*fastforce dest*:*length-Casts-vs*′)
with *step* **have** *P*,*E* ⊢ ⟨*e*,(h_0, l_0)⟩ →∗ ⟨*e*′,(h_1, l_1)⟩ **by** *simp*
with *casts* **show** *?case* **by** *auto*
qed *simp-all*

lemma assumes *wf*:*wf-prog wf-md P*
shows *blocksFinal*:
⋀*E l vs*′. ⟦ *length Vs* = *length Ts*; *length vs* = *length Ts*;
~~∀*T*∈*set Ts*. *is-type P T*;~~ *final e*; *P* ⊢ *Ts Casts vs to vs*′ ⟧ ⟹
P,*E* ⊢ ⟨*blocks*(*Vs*,*Ts*,*vs*,*e*), (*h*,*l*)⟩ →∗ ⟨*e*, (*h*,*l*)⟩

proof(*induct Vs Ts vs e rule*:*blocks-old-induct*)
case (*5 V Vs T Ts v vs e*)
have *length1*:*length* (*V* # *Vs*) = *length* (*T* # *Ts*)
and *length2*:*length* (*v* # *vs*) = *length* (*T* # *Ts*)
and *final*:*final e* **and** *casts*:*P* ⊢ *T* # *Ts Casts v* # *vs to vs*′
and *IH*:⋀*E l vs*′. ⟦*length Vs* = *length Ts*; *length vs* = *length Ts*; *final e*;
P ⊢ *Ts Casts vs to vs*′ ⟧
⟹ *P*,*E* ⊢ ⟨*blocks* (*Vs*, *Ts*, *vs*, *e*),(*h*, *l*)⟩ →∗ ⟨*e*,(*h*, *l*)⟩ **by** *fact*+
from *length1 length2*
have *length1*′:*length Vs* = *length Ts* **and** *length2*′:*length vs* = *length Ts*
by *simp-all*
from *casts* **obtain** *x xs* **where** *vs*′:*vs*′ = *x*#*xs*

by(*cases vs'*,*auto dest*:*length-Casts-vs'*)
with *casts* **have** *casts'*:$P \vdash$ *Ts Casts vs to xs*
and *cast'*:$P \vdash$ *T casts v to x*
by(*auto elim*:*Casts-to.cases*)
from *InitBlockReds*[*OF IH*[*OF length1' length2' final casts'*] *cast' wf*, *of V l*]
obtain *v'' w*
where *blocks*:$P,E \vdash \langle\{V{:}T;\ V{:=}Val\ v;;\ blocks\ (Vs,\ Ts,\ vs,\ e)\},(h,\ l)\rangle \to*
$\langle\{V{:}T;\ V{:=}Val\ v'';;\ e\},(h,l)\rangle$
and $P \vdash$ *T casts v'' to w* **by** *auto blast*
with *final* **have** $P,E \vdash \langle\{V{:}T;\ V{:=}Val\ v'';;\ e\},(h,l)\rangle \to \langle e,(h,l)\rangle$
by(*auto elim*!:*finalE intro*:*RedInitBlock InitBlockThrow*)
with *blocks* **show** *?case*
by −(*rule-tac b*=($\{V{:}T;\ V{:=}Val\ v'';;\ e\},(h,\ l)$) **in** *rtrancl-into-rtrancl*,*simp-all*)
qed *auto*

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 −
obtain *vs'' ws* **where** *blocks*:$P,E \vdash \langle blocks(Vs,Ts,vs,e),\ (h_0,\ l_0)\rangle \to*
$\langle blocks(Vs,Ts,vs'',e'),\ (h_1,l_2)\rangle$
and *length*:*length vs = length vs''*
and *casts'*:$P \vdash$ *Ts Casts vs'' to ws*
using *l2 blocksReds*[*OF wfmd wf casts reds*]
by *auto*
have $P,E \vdash \langle blocks(Vs,Ts,vs'',e'),\ (h_1,l_2)\rangle \to* \langle e',\ (h_1,l_2)\rangle$
using *blocksFinal*[*OF wfmd - - fin casts'*] *wf length* **by** *simp*
with *blocks* **show** *?thesis* **by** *simp*
qed

An now the actual method call reduction lemmas.

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

apply(*erule rtrancl-induct2*)
apply *blast*
apply(*erule rtrancl-into-rtrancl*)
apply(*simp add*:*CallObj*)
done

lemma *CallRedsParams*:

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

apply(*erule rtrancl-induct2*)
apply *blast*
apply(*erule rtrancl-into-rtrancl*)
apply(*simp add*:*CallParams*)
done

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

apply(*erule red.cases*)
apply(*auto intro*:*red-reds.intros*)
apply(*subgoal-tac* $P,E \vdash \langle(\!|C|\!)ref\ (a,Cs@[C]@Cs'),(h,l')\rangle \rightarrow \langle ref\ (a,Cs@[C]),(h,l')\rangle$)
apply *simp*
apply(*rule RedStaticDownCast*)
done

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

apply(*erule red.cases*)
apply(*auto intro*:*red-reds.intros*)
apply(*subgoal-tac* $P,E' \vdash \langle(\!|C|\!)ref\ (a,Cs@[C]@Cs'),(h,l)\rangle \rightarrow \langle ref\ (a,Cs@[C]),(h,l)\rangle$)
apply *simp*
apply(*rule RedStaticDownCast*)
done

lemma *Cast-step-Cast-or-fin*:
$P,E \vdash \langle(\!|C|\!)e,s\rangle \rightarrow* \langle e',s'\rangle \Longrightarrow final\ e' \vee (\exists e''.\ e' = (\!|C|\!)e'')$
by(*induct rule*:*rtrancl-induct2*,*auto elim*:*red.cases simp*:*final-def*)

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

proof(*induct rule*:*rtrancl-induct2*)
case *refl* **thus** *?case* **by** *simp*
next
case (*step* $e''\ s''\ e'\ s'$)
have *step*:$P,E \vdash \langle e,s\rangle \rightarrow* \langle e'',s''\rangle$
and *Red*:$((e'',\ s''),\ e',\ s') \in Red\ P\ E$

```
    and cast:e = (|C|)e0 and cast':e' = (|C|)e1
   and IH:⋀e1. ⟦e = (|C|)e0; e'' = (|C|)e1⟧ ⟹ P,E ⊢ ⟨e0,s⟩ →* ⟨e1,s''⟩ by fact+
  from Red have red:P,E ⊢ ⟨e'',s''⟩ → ⟨e',s'⟩ by simp
  from step cast have final e'' ∨ (∃ ex. e'' = (|C|)ex)
    by simp(rule Cast-step-Cast-or-fin)
  thus ?case
  proof(rule disjE)
    assume final e''
    with red show ?thesis by(auto simp:final-def)
  next
    assume ∃ ex. e'' = (|C|)ex
    then obtain ex where e'':e'' = (|C|)ex by blast
    with cast' red have P,E ⊢ ⟨ex,s''⟩ → ⟨e1,s'⟩
      by(auto elim:red.cases)
    with  IH[OF cast e''] show ?thesis
      by(rule-tac b=(ex,s'') in rtrancl-into-rtrancl,simp-all)
  qed
qed


lemma Cast-final:⟦P,E ⊢ ⟨(|C|)e,s⟩ →* ⟨e',s'⟩; final e'⟧ ⟹
∃ e'' s''. P,E ⊢ ⟨e,s⟩ →* ⟨e'',s''⟩ ∧ P,E ⊢ ⟨(|C|)e'',s''⟩ → ⟨e',s'⟩ ∧ final e''

proof(induct rule:rtrancl-induct2)
  case refl thus ?case by (simp add:final-def)
next
  case (step e'' s'' e' s')
  have step:P,E ⊢ ⟨(|C|)e,s⟩ →* ⟨e'',s''⟩
    and Red:((e'', s''), e', s') ∈ Red P E
    and final:final e'
    and IH:final e'' ⟹
   ∃ ex sx. P,E ⊢ ⟨e,s⟩ →* ⟨ex,sx⟩ ∧ P,E ⊢ ⟨(|C|)ex,sx⟩ → ⟨e'',s''⟩ ∧ final ex by
fact+
  from Red have red:P,E ⊢ ⟨e'',s''⟩ → ⟨e',s'⟩ by simp
  from step have final e'' ∨ (∃ ex. e'' = (|C|)ex) by(rule Cast-step-Cast-or-fin)
  thus ?case
  proof(rule disjE)
    assume final e''
    with red show ?thesis by(auto simp:final-def)
  next
    assume ∃ ex. e'' = (|C|)ex
    then obtain ex where e'':e'' = (|C|)ex by blast
    with red final have final':final ex
      by(auto elim:red.cases simp:final-def)
    from step e'' have P,E ⊢ ⟨e,s⟩ →* ⟨ex,s''⟩
      by(fastforce intro:Cast-red)
    with e'' red final' show ?thesis by blast
  qed
qed
```

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 −
from *red final* **show** *?thesis*
proof(*auto simp*:*final-def*)
fix *v* **assume** $P,E \vdash \langle (\!| C |\!)(Val\ v),(h,l)\rangle \to \langle e',(h,l)\rangle$
with *final'* **show** $P,E' \vdash \langle (\!| C |\!)(Val\ v),(h,l')\rangle \to \langle e',(h,l')\rangle$
proof(*auto simp*:*final-def*)
fix *v'* **assume** $P,E \vdash \langle (\!| C |\!)(Val\ v),(h,l)\rangle \to \langle Val\ v',(h,l)\rangle$
thus $P,E' \vdash \langle (\!| C |\!)(Val\ v),(h,l')\rangle \to \langle Val\ v',(h,l')\rangle$
by(*auto intro*:*cast-lcl cast-env*)
next
fix *a Cs* **assume** $P,E \vdash \langle (\!| C |\!)(Val\ v),(h,l)\rangle \to \langle Throw\ (a,Cs),(h,l)\rangle$
thus $P,E' \vdash \langle (\!| C |\!)(Val\ v),(h,l')\rangle \to \langle Throw\ (a,Cs),(h,l')\rangle$
by(*auto elim*:*red.cases intro*!:*RedStaticCastFail*)
qed
next
fix *a Cs* **assume** $P,E \vdash \langle (\!| C |\!)(Throw\ (a,Cs)),(h,l)\rangle \to \langle e',(h,l)\rangle$
with *final'* **show** $P,E' \vdash \langle (\!| C |\!)(Throw\ (a,Cs)),(h,l')\rangle \to \langle e',(h,l')\rangle$
proof(*auto simp*:*final-def*)
fix *v* **assume** $P,E \vdash \langle (\!| C |\!)(Throw\ (a,Cs)),(h,l)\rangle \to \langle Val\ v,(h,l)\rangle$
thus $P,E' \vdash \langle (\!| C |\!)(Throw\ (a,Cs)),(h,l')\rangle \to \langle Val\ v,(h,l')\rangle$
by(*auto elim*:*red.cases*)
next
fix *a' Cs'*
assume $P,E \vdash \langle (\!| C |\!)(Throw\ (a,Cs)),(h,l)\rangle \to \langle Throw\ (a',Cs'),(h,l)\rangle$
thus $P,E' \vdash \langle (\!| C |\!)(Throw\ (a,Cs)),(h,l')\rangle \to \langle Throw\ (a',Cs'),(h,l')\rangle$
by(*auto elim*:*red.cases intro*:*red-reds.StaticCastThrow*)
qed
qed
qed

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* | - ⇒ *body*)
and *body*: P,E(*this* ↦ *Class* (*last* Cs'), *pns* [↦] Ts) ⊢ ⟨*new-body*,(h_2,l_2')⟩ →* ⟨*ef*,(h_3,l_3)⟩
and *final*:*final ef*
shows P,E ⊢ ⟨e·M(*es*), s_0⟩ →* ⟨*ef*,(h_3,l_2)⟩
proof −
have *wf*: *size Ts* = *size pns* ∧ *distinct pns* ∧ *this* ∉ *set pns*
and *wt*: *fv body* ⊆ {*this*} ∪ *set pns*
using *assms* **by**(*fastforce dest*!:*select-method-wf-mdecl simp*:*wf-mdecl-def*)+
have *dom* l_3 ⊆ {*this*} ∪ *set pns*
using *Reds-dom-lcl*[*OF wwf body*] *wt* l_2' *set-take-subset body-case*
by (*cases* T') *force*+
hence *eql*$_2$: *override-on* (l_2++l_3) l_2 ({*this*} ∪ *set pns*) = l_2
by(*fastforce simp add*:*map-add-def override-on-def fun-eq-iff*)
from *wwf select* **have** *is-class P* (*last Cs'*)
by (*auto elim*!:*SelectMethodDef.cases intro*:*Subobj-last-isClass*
simp:*LeastMethodDef-def FinalOverriderMethodDef-def*
OverriderMethodDefs-def MinimalMethodDefs-def MethodDefs-def)
hence P ⊢ *Class* (*last Cs'*) *casts Ref*(a,Cs') *to Ref*(a,Cs')
by(*auto intro*!:*casts-ref Subobjs-Base simp*:*path-via-def appendPath-def*)
with *casts*
have *casts'*:P ⊢ *Class* (*last Cs'*)#*Ts Casts Ref*(a,Cs')#*vs to Ref*(a,Cs')#*vs'*
by −(*rule Casts-Cons*)
have *1*:P,E ⊢ ⟨e·M(*es*),s_0⟩ →* ⟨(*ref*(a,*Cs*))·M(*es*),s_1⟩ **by**(*rule CallRedsObj*)(*rule assms*(*2*))
have *2*:P,E ⊢ ⟨(*ref*(a,*Cs*))·M(*es*),s_1⟩ →*
⟨(*ref*(a,*Cs*))·M(*map Val vs*),(h_2,l_2)⟩
by(*rule CallRedsParams*)(*rule assms*(*3*))
from *body*[*THEN Red-lcl-add*, *of* l_2]
have *body'*: P,E(*this* ↦ *Class* (*last Cs'*), *pns* [↦] Ts) ⊢
⟨*new-body*,(h_2,l_2(*this*↦ *Ref*(a,Cs'), *pns*[↦]*vs'*))⟩ →* ⟨*ef*,(h_3,l_2++l_3)⟩
by (*simp add*:l_2')
show *?thesis*
proof(*cases* ∀ C. $T' \neq$ *Class* C)
case *True*
hence P,E ⊢ ⟨(*ref*(a,*Cs*))·M(*map Val vs*), (h_2,l_2)⟩ →
⟨*blocks*(*this*#*pns*,*Class*(*last Cs'*)#*Ts*,*Ref*(a,Cs')#*vs*,*body*), (h_2,l_2)⟩
using *hp method select size wf*
by −(*rule RedCall*,*auto*,*cases* T',*auto*)
hence *3*:P,E ⊢ ⟨(*ref*(a,*Cs*))·M(*map Val vs*), (h_2,l_2)⟩ →*
⟨*blocks*(*this*#*pns*,*Class*(*last Cs'*)#*Ts*,*Ref*(a,Cs')#*vs*,*body*), (h_2,l_2)⟩
by(*simp add*:*r-into-rtrancl*)
have P,E ⊢ ⟨*blocks*(*this*#*pns*,*Class*(*last Cs'*)#*Ts*,*Ref*(a,Cs')#*vs*,*body*),(h_2,l_2)⟩ →*
⟨*ef*,(h_3,*override-on* (l_2++l_3) l_2 ({*this*} ∪ *set pns*))⟩
using *True wf body' wwf size final casts' body-case*
by −(*rule-tac vs'*=*Ref*(a,Cs')#*vs'* **in** *blocksRedsFinal*,*simp-all*,*cases* T',*auto*)
with *1 2 3* **show** *?thesis* **using** *eql*$_2$
by *simp*

next
case *False*
then obtain D **where** $T'{:}T' = Class\ D$ **by** *auto*
with *final body' body-case* **obtain** $s'\ e'$ **where**
$body''{:}P,E(this \mapsto Class\ (last\ Cs'),pns\ [\mapsto]\ Ts) \vdash$
$\langle body,(h_2,l_2(this \mapsto Ref(a,Cs'),\ pns[\mapsto]vs'))\rangle \to* \langle e',s'\rangle$
and *final'*:*final e'*
and $cast{:}P,E(this \mapsto Class\ (last\ Cs'),\ pns\ [\mapsto]\ Ts) \vdash \langle(\!|D|\!)e',s'\rangle \to$
$\langle ef,(h_3,l_2{+}{+}l_3)\rangle$
by(*cases T'*)(*auto dest*:*Cast-final*)
from T' **have** $P,E \vdash \langle(ref(a,Cs))\cdot M(map\ Val\ vs),\ (h_2,l_2)\rangle \to$
$\langle(\!|D|\!)blocks(this\#pns,Class(last\ Cs')\#Ts,Ref(a,Cs')\#vs,body),$
$(h_2,l_2)\rangle$
using *hp method select size wf*
by $-$(*rule RedCall*,*auto*)
hence *3*:$P,E \vdash \langle(ref(a,Cs))\cdot M(map\ Val\ vs),\ (h_2,l_2)\rangle \to*$
$\langle(\!|D|\!)blocks(this\#pns,Class(last\ Cs')\#Ts,Ref(a,Cs')\#vs,body),(h_2,l_2)\rangle$
by(*simp add*:*r-into-rtrancl*)
from *cast final* **have** *eq*:$s' = (h_3,l_2{+}{+}l_3)$
by(*auto elim*:*red.cases simp*:*final-def*)
hence $P,E \vdash \langle blocks(this\#pns,\ Class\ (last\ Cs')\#Ts,\ Ref(a,Cs')\#vs,body),$
$(h_2,l_2)\rangle$
$\to* \langle e',(h_3,override\text{-}on\ (l_2{+}{+}l_3)\ l_2\ (\{this\} \cup set\ pns))\rangle$
using *wf body'' wwf size final' casts'*
by $-$(*rule-tac vs'*=$Ref(a,Cs')\#vs'$ **in** *blocksRedsFinal*,*simp-all*)
hence $P,E \vdash \langle(\!|D|\!)(blocks(this\#pns,Class(last\ Cs')\#Ts,Ref(a,Cs')\#vs,body)),(h_2,l_2)\rangle$
$\to* \langle(\!|D|\!)e',(h_3,override\text{-}on\ (l_2{+}{+}l_3)\ l_2\ (\{this\} \cup set\ pns))\rangle$
by(*rule StaticCastReds*)
moreover
have $P,E \vdash \langle(\!|D|\!)e',(h_3,override\text{-}on\ (l_2{+}{+}l_3)\ l_2\ (\{this\} \cup set\ pns))\rangle \to$
$\langle ef,(h_3,override\text{-}on\ (l_2{+}{+}l_3)\ l_2\ (\{this\} \cup set\ pns))\rangle$
using *eq cast final final'*
by(*fastforce intro*:*Cast-final-eq*)
ultimately
have $P,E \vdash \langle(\!|D|\!)(blocks(this\#pns,\ Class\ (last\ Cs')\#Ts,\ Ref(a,Cs')\#vs,body)),$
$(h_2,l_2)\rangle \to* \langle ef,(h_3,override\text{-}on\ (l_2{+}{+}l_3)\ l_2\ (\{this\} \cup set\ pns))\rangle$
by(*rule-tac b*=$(\!|D|\!)e',(h_3,override\text{-}on\ (l_2{+}{+}l_3)\ l_2\ (\{this\} \cup set\ pns))$)
in *rtrancl-into-rtrancl*,*simp-all*)
with *1 2 3* **show** *?thesis* **using** eql_2
by *simp*
qed
qed

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*: $Ds = (Cs @_p Cs'') @_p Cs'$
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 −
have *wf*: $size\ Ts = size\ pns \wedge distinct\ pns \wedge this \notin set\ pns \wedge$
$(\forall\ T \in set\ Ts.\ is\text{-}type\ P\ T)$
and *wt*: $fv\ body \subseteq \{this\} \cup set\ pns$
using *assms* **by**(*fastforce dest*!:*has-least-wf-mdecl simp*:*wf-mdecl-def*)+
have $dom\ l_3 \subseteq \{this\} \cup set\ pns$
using *Reds-dom-lcl*[*OF wwf body*] *wt* l_2' *set-take-subset*
by *force*
hence *eql*$_2$: $override\text{-}on\ (l_2{+}{+}l_3)\ l_2\ (\{this\} \cup set\ pns) = l_2$
by(*fastforce simp add*:*map-add-def override-on-def fun-eq-iff*)
from *wwf least* **have** $Cs' \neq []$
by (*auto elim*!:*Subobjs-nonempty simp*:*LeastMethodDef-def MethodDefs-def*)
with *Ds* **have** $last\ Cs' = last\ Ds$ **by**(*fastforce intro*:*appendPath-last*)
with *wwf least* **have** $is\text{-}class\ P\ (last\ Ds)$
by(*auto dest*:*Subobj-last-isClass simp*:*LeastMethodDef-def MethodDefs-def*)
hence $P \vdash Class\ (last\ Ds)\ casts\ Ref(a,Ds)\ to\ Ref(a,Ds)$
by(*auto intro*!:*casts-ref Subobjs-Base simp*:*path-via-def appendPath-def*)
with *casts*
have *casts'*:$P \vdash Class\ (last\ Ds)\#Ts\ Casts\ Ref(a,Ds)\#vs\ to\ Ref(a,Ds)\#vs'$
by −(*rule Casts-Cons*)
have *1*:$P,E \vdash \langle e\cdot(C::)M(es),s_0\rangle \to* \langle (ref(a,Cs))\cdot(C::)M(es),s_1\rangle$
by(*rule CallRedsObj*)(*rule assms*(*2*))
have *2*:$P,E \vdash \langle (ref(a,Cs))\cdot(C::)M(es),s_1\rangle \to*$
$\langle (ref(a,Cs))\cdot(C::)M(map\ Val\ vs),(h_2,l_2)\rangle$
by(*rule CallRedsParams*)(*rule assms*(*3*))
from *body*[*THEN Red-lcl-add*, *of* l_2]
have *body'*: $P,E(this \mapsto Class(last\ Ds),\ pns[\mapsto]Ts) \vdash$
$\langle body,(h_2,l_2(this \mapsto Ref(a,Ds),\ pns[\mapsto]vs'))\rangle \to* \langle ef,(h_3,l_2{+}{+}l_3)\rangle$
by (*simp add*:l_2')
have $P,E \vdash \langle (ref(a,Cs))\cdot(C::)M(map\ Val\ vs),\ (h_2,l_2)\rangle \to$
$\langle blocks(this\#pns,\ Class\ (last\ Ds)\#Ts,\ Ref(a,Ds)\#vs,\ body),\ (h_2,l_2)\rangle$
using *path-unique path-via least size wf Ds*
by −(*rule RedStaticCall*,*auto*)
hence *3*:$P,E \vdash \langle (ref(a,Cs))\cdot(C::)M(map\ Val\ vs),\ (h_2,l_2)\rangle \to*$
$\langle blocks(this\#pns,Class(last\ Ds)\#Ts,Ref(a,Ds)\#vs,body),\ (h_2,l_2)\rangle$
by(*simp add*:*r-into-rtrancl*)
have $P,E \vdash \langle blocks(this\#pns,Class(last\ Ds)\#Ts,Ref(a,Ds)\#vs,body),(h_2,l_2)\rangle$
$\to*$
$\langle ef,(h_3,override\text{-}on\ (l_2{+}{+}l_3)\ l_2\ (\{this\} \cup set\ pns))\rangle$

using *wf body′ wwf size final casts′*
by −(*rule-tac vs′=Ref*(*a*,*Ds*)#*vs′* **in** *blocksRedsFinal*,*simp-all*)
with *1 2 3* **show** *?thesis* **using** eql_2
by *simp*
qed

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

apply(*rule rtrancl-trans*)
apply(*erule CallRedsObj*)
apply(*rule rtrancl-into-rtrancl*)
apply(*erule CallRedsParams*)
apply(*simp add*:*CallThrowParams*)
done

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

apply(*rule rtrancl-into-rtrancl*)
apply(*erule CallRedsObj*)
apply(*simp add*:*CallThrowObj*)
done

lemma *CallRedsNull*:
⟦ P,$E \vdash \langle e,s_0\rangle \to* \langle null,s_1\rangle$; P,$E \vdash \langle es,s_1\rangle\ [\to]* \langle map\ Val\ vs,s_2\rangle$ ⟧
⟹ P,$E \vdash \langle Call\ e\ Copt\ M\ es,s_0\rangle \to* \langle THROW\ NullPointer,s_2\rangle$

apply(*rule rtrancl-trans*)
apply(*erule CallRedsObj*)
apply(*rule rtrancl-into-rtrancl*)
apply(*erule CallRedsParams*)
apply(*simp add*:*RedCallNull*)
done

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 (*induct rule*: *eval-evals.inducts*)
case *New* **thus** *?case* **by** (*auto simp*:*RedNew*)
next
case *NewFail* **thus** *?case* **by** (*auto simp*:*RedNewFail*)
next
case *StaticUpCast* **thus** *?case* **by**(*simp add*:*StaticUpCastReds*)
next
case *StaticDownCast* **thus** *?case* **by**(*simp add*:*StaticDownCastReds*)
next
case *StaticCastNull* **thus** *?case* **by**(*simp add*:*StaticCastRedsNull*)
next
case *StaticCastFail* **thus** *?case* **by**(*simp add*:*StaticCastRedsFail*)
next
case *StaticCastThrow* **thus** *?case* **by**(*auto dest*!:*eval-final simp*:*StaticCastRedsThrow*)
next
case *StaticUpDynCast* **thus** *?case* **by**(*simp add*:*StaticUpDynCastReds*)
next
case *StaticDownDynCast* **thus** *?case* **by**(*simp add*:*StaticDownDynCastReds*)
next
case *DynCast* **thus** *?case* **by**(*fastforce intro*:*DynCastRedsRef*)
next
case *DynCastNull* **thus** *?case* **by**(*simp add*:*DynCastRedsNull*)
next
case *DynCastFail* **thus** *?case* **by**(*fastforce intro*!:*DynCastRedsFail*)
next
case *DynCastThrow* **thus** *?case* **by**(*auto dest*!:*eval-final simp*:*DynCastRedsThrow*)
next
case *Val* **thus** *?case* **by** *simp*
next
case *BinOp* **thus** *?case* **by**(*fastforce simp*:*BinOpRedsVal*)
next
case *BinOpThrow1* **thus** *?case* **by**(*fastforce dest*!:*eval-final simp*: *BinOpRedsThrow1*)
next
case *BinOpThrow2* **thus** *?case* **by**(*fastforce dest*!:*eval-final simp*: *BinOpRedsThrow2*)
next
case *Var* **thus** *?case* **by** (*fastforce simp*:*RedVar*)
next
case *LAss* **thus** *?case* **by**(*fastforce simp*: *LAssRedsVal*)
next
case *LAssThrow* **thus** *?case* **by**(*fastforce dest*!:*eval-final simp*: *LAssRedsThrow*)
next
case *FAcc* **thus** *?case* **by**(*fastforce intro*:*FAccRedsVal*)
next
case *FAccNull* **thus** *?case* **by**(*simp add*:*FAccRedsNull*)
next
case *FAccThrow* **thus** *?case* **by**(*fastforce dest*!:*eval-final simp*:*FAccRedsThrow*)

next
case *FAss* **thus** *?case* **by**(*fastforce simp*:*FAssRedsVal*)
next
case *FAssNull* **thus** *?case* **by**(*fastforce simp*:*FAssRedsNull*)
next
case *FAssThrow1* **thus** *?case* **by**(*fastforce dest*!:*eval-final simp*:*FAssRedsThrow1*)
next
case *FAssThrow2* **thus** *?case* **by**(*fastforce dest*!:*eval-final simp*:*FAssRedsThrow2*)
next
case *CallObjThrow* **thus** *?case* **by**(*fastforce dest*!:*eval-final simp*:*CallRedsThrowObj*)
next
case *CallNull* **thus** *?case* **thm** *CallRedsNull* **by**(*simp add*:*CallRedsNull*)
next
case *CallParamsThrow* **thus** *?case*
by(*fastforce dest*!:*evals-final simp*:*CallRedsThrowParams*)
next
case (*Call 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-body e'* h_3 l_3)
have *IHe*: $P,E \vdash \langle e,s_0\rangle \rightarrow* \langle ref(a,Cs),s_1\rangle$
and *IHes*: $P,E \vdash \langle ps,s_1\rangle\ [\rightarrow]*\ \langle map\ Val\ vs,(h_2,l_2)\rangle$
and h_2a: $h_2\ a = Some(C,S)$
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 *same-length*: $length\ vs = length\ pns$
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*: $new\text{-}body = (case\ T'\ of\ Class\ D \Rightarrow (\!|D|\!)body \mid \text{-} \Rightarrow body)$
and *eval-body*: $P,E(this \mapsto Class\ (last\ Cs'),\ pns\ [\mapsto]\ Ts) \vdash$
$\langle new\text{-}body,(h_2,\ l_2')\rangle \Rightarrow \langle e',(h_3,\ l_3)\rangle$
and *IHbody*: $P,E(this \mapsto Class\ (last\ Cs'),\ pns\ [\mapsto]\ Ts) \vdash$
$\langle new\text{-}body,(h_2,\ l_2')\rangle \rightarrow* \langle e',(h_3,\ l_3)\rangle$ **by** *fact+*
from *wwf select same-length* **have** *lengthTs*:$length\ Ts = length\ vs$
by (*fastforce dest*!:*select-method-wf-mdecl simp*:*wf-mdecl-def*)
show $P,E \vdash \langle e{\bullet}M(ps),s_0\rangle \rightarrow* \langle e',(h_3,\ l_2)\rangle$
using *method select same-length* l_2' h_2a *casts body-case*
IHbody eval-final[*OF eval-body*]
by(*fastforce intro*!:*CallRedsFinal*[*OF wwf IHe IHes*])
next
case (*StaticCall 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)
have *IHe*: $P,E \vdash \langle e,s_0\rangle \rightarrow* \langle ref(a,Cs),s_1\rangle$
and *IHes*: $P,E \vdash \langle ps,s_1\rangle\ [\rightarrow]*\ \langle map\ Val\ vs,(h_2,l_2)\rangle$
and *path-unique*: $P \vdash Path\ last\ Cs\ to\ C\ unique$
and *path-via*: $P \vdash Path\ last\ Cs\ to\ C\ via\ Cs''$
and *least*: $P \vdash C\ has\ least\ M = (Ts,\ T,\ pns,\ body)\ via\ Cs'$
and *Ds*: $Ds = (Cs\ @_p\ Cs'')\ @_p\ Cs'$
and *same-length*: $length\ vs = length\ pns$
and *casts*: $P \vdash Ts\ Casts\ vs\ to\ vs'$
and l_2': $l_2' = [this \mapsto Ref\ (a,Ds),\ pns[\mapsto]vs']$

and *eval-body*: $P,E(this \mapsto Class\ (last\ Ds),\ pns\ [\mapsto]\ Ts) \vdash \langle body,(h_2,\ l_2')\rangle \Rightarrow \langle e',(h_3,\ l_3)\rangle$
and *IHbody*: $P,E(this \mapsto Class\ (last\ Ds),\ pns\ [\mapsto]\ Ts) \vdash \langle body,(h_2,\ l_2')\rangle \rightarrow* \langle e',(h_3,\ l_3)\rangle$ **by** *fact+*
from *wwf least same-length* **have** *lengthTs*:*length Ts* = *length vs*
by (*fastforce dest*!:*has-least-wf-mdecl simp*:*wf-mdecl-def*)
show $P,E \vdash \langle e\bullet(C::)M(ps),s_0\rangle \rightarrow* \langle e',(h_3,\ l_2)\rangle$
using *path-unique path-via least Ds same-length* l_2' *casts IHbody eval-final*[*OF eval-body*]
by(*fastforce intro*!:*StaticCallRedsFinal*[*OF wwf IHe IHes*])
next
case *Block* **with** *wwf* **show** *?case* **by**(*fastforce simp*: *BlockRedsFinal dest*:*eval-final*)
next
case *Seq* **thus** *?case* **by**(*fastforce simp*:*SeqReds2*)
next
case *SeqThrow* **thus** *?case* **by**(*fastforce dest*!:*eval-final simp*: *SeqRedsThrow*)
next
case *CondT* **thus** *?case* **by**(*fastforce simp*:*CondReds2T*)
next
case *CondF* **thus** *?case* **by**(*fastforce simp*:*CondReds2F*)
next
case *CondThrow* **thus** *?case* **by**(*fastforce dest*!:*eval-final simp*:*CondRedsThrow*)
next
case *WhileF* **thus** *?case* **by**(*fastforce simp*:*WhileFReds*)
next
case *WhileT* **thus** *?case* **by**(*fastforce simp*: *WhileTReds*)
next
case *WhileCondThrow* **thus** *?case* **by**(*fastforce dest*!:*eval-final simp*: *WhileRedsThrow*)
next
case *WhileBodyThrow* **thus** *?case* **by**(*fastforce dest*!:*eval-final simp*: *WhileTRedsThrow*)
next
case *Throw* **thus** *?case* **by**(*fastforce simp*:*ThrowReds*)
next
case *ThrowNull* **thus** *?case* **by**(*fastforce simp*:*ThrowRedsNull*)
next
case *ThrowThrow* **thus** *?case* **by**(*fastforce dest*!:*eval-final simp*:*ThrowRedsThrow*)
next
case *Nil* **thus** *?case* **by** *simp*
next
case *Cons* **thus** *?case*
by(*fastforce intro*!:*Cons-eq-appendI*[*OF refl refl*] *ListRedsVal*)
next
case *ConsThrow* **thus** *?case* **by**(*fastforce elim*: *ListReds1*)
qed

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
assume $P,E \vdash \langle while\ (b)\ c,s\rangle \Rightarrow \langle e',s'\rangle$
thus $P,E \vdash \langle if\ (b)\ (c;;\ while\ (b)\ c)\ else\ unit,s\rangle \Rightarrow \langle e',s'\rangle$
by *cases* (*fastforce intro*: *eval-evals.intros*)+
next
assume $P,E \vdash \langle if\ (b)\ (c;;\ while\ (b)\ c)\ else\ unit,s\rangle \Rightarrow \langle e',s'\rangle$
thus $P,E \vdash \langle while\ (b)\ c,s\rangle \Rightarrow \langle e',s'\rangle$
proof (*cases*)
fix *ex*
assume *e'*: $e' = throw\ ex$
assume $P,E \vdash \langle b,s\rangle \Rightarrow \langle throw\ ex,s'\rangle$
hence $P,E \vdash \langle while(b)\ c,s\rangle \Rightarrow \langle throw\ ex,s'\rangle$ **by** (*rule WhileCondThrow*)
with e' **show** *?thesis* **by** *simp*
next
fix s_1
assume *eval-false*: $P,E \vdash \langle b,s\rangle \Rightarrow \langle false,s_1\rangle$
and *eval-unit*: $P,E \vdash \langle unit,s_1\rangle \Rightarrow \langle e',s'\rangle$
with *eval-unit* **have** $s' = s_1\ e' = unit$ **by** (*auto elim*: *eval-cases*)
moreover from *eval-false* **have** $P,E \vdash \langle while\ (b)\ c,s\rangle \Rightarrow \langle unit,s_1\rangle$
by − (*rule WhileF*, *simp*)
ultimately show *?thesis* **by** *simp*
next
fix s_1
assume *eval-true*: $P,E \vdash \langle b,s\rangle \Rightarrow \langle true,s_1\rangle$
and *eval-rest*: $P,E \vdash \langle c;;\ while\ (b)\ c,s_1\rangle \Rightarrow \langle e',s'\rangle$
from *eval-rest* **show** *?thesis*
proof (*cases*)
fix $s_2\ v_1$
assume $P,E \vdash \langle c,s_1\rangle \Rightarrow \langle Val\ v_1,s_2\rangle\ P,E \vdash \langle while\ (b)\ c,s_2\rangle \Rightarrow \langle e',s'\rangle$
with *eval-true* **show** $P,E \vdash \langle while(b)\ c,s\rangle \Rightarrow \langle e',s'\rangle$ **by** (*rule WhileT*)
next
fix *ex*
assume $P,E \vdash \langle c,s_1\rangle \Rightarrow \langle throw\ ex,s'\rangle\ e' = throw\ ex$
with *eval-true* **show** $P,E \vdash \langle while(b)\ c,s\rangle \Rightarrow \langle e',s'\rangle$
by (*iprover intro*: *WhileBodyThrow*)
qed
qed
qed

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 (*induct ps*)
case *Nil* **then show** *?case* **by**(*fastforce intro*:*Casts-Nil*)
next
case (*Cons p ps'*)
have *length-eqs*: *length* (*p* # *ps'*) = *length Ts*
length (*p* # *ps'*) = *length vs*
and *IH*:$\bigwedge Ts\ vs\ l\ l'\ E.\ [\![length\ ps' = length\ Ts;\ length\ ps' = length\ vs;$
$P,E \vdash \langle blocks\ (ps',Ts,vs,e),(h,l)\rangle \Rightarrow \langle e',(h',l')\rangle]\!]$
$\Longrightarrow \exists l''\ vs'.\ P,E(ps'\ [\mapsto]\ Ts) \vdash \langle e,(h,l(ps'\ [\mapsto]\ vs'))\rangle \Rightarrow \langle e',(h',\ l'')\rangle \wedge$
$P \vdash Ts\ Casts\ vs\ to\ vs' \wedge length\ vs' = length\ vs$ **by** *fact+*
then obtain *T Ts'* **where** *Ts*: *Ts* = *T*#*Ts'* **by** (*cases Ts*) *simp*
obtain *v vs'* **where** *vs*: *vs* = *v*#*vs'* **using** *length-eqs* **by** (*cases vs*) *simp*
with *length-eqs Ts* **have** *length1*:*length ps'* = *length Ts'*
and *length2*:*length ps'* = *length vs'* **by** *simp-all*
have *P*,*E* ⊢ ⟨*blocks* (*p* # *ps'*, *Ts*, *vs*, *e*),(*h*,*l*)⟩ ⇒ ⟨*e'*,(*h'*, *l'*)⟩ **by** *fact*
with *Ts vs*
have *blocks*:*P*,*E* ⊢ ⟨{*p*:*T* := *Val v*; *blocks* (*ps'*,*Ts'*,*vs'*,*e*)},(*h*,*l*)⟩ ⇒ ⟨*e'*,(*h'*,*l'*)⟩
by *simp*
then obtain *l'''* *v'* **where**
eval-ps': *P*,*E*(*p* ↦ *T*) ⊢ ⟨*blocks* (*ps'*,*Ts'*,*vs'*,*e*),(*h*,*l*(*p*↦*v'*))⟩ ⇒ ⟨*e'*,(*h'*,*l'''*)⟩
and *l'''*: *l'*=*l'''*(*p*:=*l p*)
and *casts*:*P* ⊢ *T casts v to v'*
by(*auto elim*!: *eval-cases simp*:*fun-upd-same*)
from *IH*[*OF length1 length2 eval-ps'*] **obtain** *l''* *vs''* **where**
P,*E*(*p* ↦ *T*)(*ps'* [↦] *Ts'*) ⊢ ⟨*e*,(*h*, *l*(*p*↦*v'*)(*ps'*[↦]*vs''*))⟩ ⇒
⟨*e'*,(*h'*,*l''*)⟩
and *P* ⊢ *Ts'* *Casts vs' to vs''*
and *length vs''* = *length vs'* **by** *auto*
with *Ts vs casts* **show** *?case*
by −(*rule-tac x*=*l''* **in** *exI*,*rule-tac x*=*v'*#*vs''* **in** *exI*,*simp*,
rule Casts-Cons)
qed

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 (*induct ps*)
case *Nil* **then show** *?case* **by**(*fastforce intro*:*Casts-Nil*)
next

case (*Cons p ps'*)
have *length-eqs*: *length* (*p # ps'*) = *length Ts*
length (*p # ps'*) = *length vs* **by** *fact+*
then obtain *T Ts'* **where** *Ts*: *Ts* = *T*#*Ts'* **by** (*cases Ts*) *simp*
obtain *v vs'* **where** *vs*: *vs* = *v*#*vs'* **using** *length-eqs* **by** (*cases vs*) *simp*
with *length-eqs Ts* **have** *length1*:*length ps'* = *length Ts'*
and *length2*:*length ps'* = *length vs'* **by** *simp-all*
have *P*,*E* ⊢ ⟨(| *C'* |)(*blocks* (*p # ps'*, *Ts*, *vs*, *e*)),(*h*,*l*)⟩ ⇒ ⟨*e'*,(*h'*, *l'*)⟩ **by** *fact*
moreover
{ fix *a Cs Cs'*
assume *blocks*:*P*,*E* ⊢ ⟨*blocks*(*p*#*ps'*,*Ts*,*vs*,*e*),(*h*,*l*)⟩ ⇒ ⟨*ref* (*a*,*Cs*),(*h'*,*l'*)⟩
and *path-via*:*P* ⊢ *Path last Cs to C' via Cs'*
and *e'*:*e'* = *ref*(*a*,*Cs*@$_p$*Cs'*)
from *blocks length-eqs* **obtain** *l''* *vs''*
where *eval*:*P*,*E*(*p*#*ps'* [↦] *Ts*) ⊢ ⟨*e*,(*h*,*l*(*p*#*ps'*[↦]*vs''*))⟩ ⇒
⟨*ref* (*a*,*Cs*),(*h'*,*l''*)⟩
and *casts*:*P* ⊢ *Ts Casts vs to vs''*
and *length*:*length vs''* = *length vs*
by −(*drule blocksEval*,*auto*)
from *eval path-via* **have**
P,*E*(*p*#*ps'*[↦]*Ts*) ⊢ ⟨(| *C'* |)*e*,(*h*,*l*(*p*#*ps'*[↦]*vs''*))⟩ ⇒ ⟨*ref*(*a*,*Cs*@$_p$*Cs'*),(*h'*,*l''*)⟩
by(*auto intro*:*StaticUpCast*)
with *e' casts length* **have** *?case* **by** *simp blast* **}**
moreover
{ fix *a Cs Cs'*
assume *blocks*:*P*,*E* ⊢ ⟨*blocks*(*p*#*ps'*,*Ts*,*vs*,*e*),(*h*,*l*)⟩ ⇒
⟨*ref* (*a*,*Cs*@*C'*# *Cs'*),(*h'*,*l'*)⟩
and *e'*:*e'* = *ref* (*a*,*Cs*@[*C'*])
from *blocks length-eqs* **obtain** *l''* *vs''*
where *eval*:*P*,*E*(*p*#*ps'* [↦] *Ts*) ⊢ ⟨*e*,(*h*,*l*(*p*#*ps'*[↦]*vs''*))⟩ ⇒
⟨*ref* (*a*,*Cs*@*C'*# *Cs'*),(*h'*,*l''*)⟩
and *casts*:*P* ⊢ *Ts Casts vs to vs''*
and *length*:*length vs''* = *length vs*
by −(*drule blocksEval*,*auto*)
from *eval* **have** *P*,*E*(*p*#*ps'*[↦]*Ts*) ⊢ ⟨(| *C'* |)*e*,(*h*,*l*(*p*#*ps'*[↦]*vs''*))⟩ ⇒
⟨*ref*(*a*,*Cs*@[*C'*]),(*h'*,*l''*)⟩
by(*auto intro*:*StaticDownCast*)
with *e' casts length* **have** *?case* **by** *simp blast* **}**
moreover
{ assume *P*,*E* ⊢ ⟨*blocks*(*p*#*ps'*,*Ts*,*vs*,*e*),(*h*,*l*)⟩ ⇒ ⟨*null*,(*h'*,*l'*)⟩
and *e'*:*e'* = *null*
with *length-eqs* **obtain** *l''* *vs''*
where *eval*:*P*,*E*(*p*#*ps'* [↦] *Ts*) ⊢ ⟨*e*,(*h*,*l*(*p*#*ps'*[↦]*vs''*))⟩ ⇒
⟨*null*,(*h'*,*l''*)⟩
and *casts*:*P* ⊢ *Ts Casts vs to vs''*
and *length*:*length vs''* = *length vs*
by −(*drule blocksEval*,*auto*)
from *eval* **have** *P*,*E*(*p*#*ps'* [↦] *Ts*) ⊢ ⟨(| *C'* |)*e*,(*h*,*l*(*p*#*ps'*[↦]*vs''*))⟩ ⇒
⟨*null*,(*h'*,*l''*)⟩

```
    by(auto intro:StaticCastNull)
  with e' casts length have ?case by simp blast }
 moreover
 { fix a Cs
  assume blocks:P,E ⊢ ⟨blocks(p#ps',Ts,vs,e),(h,l)⟩ ⇒ ⟨ref (a,Cs),(h',l')⟩
    and notin:C' ∉ set Cs and leq:¬ P ⊢ (last Cs) ⪯* C'
    and  e':e' = THROW ClassCast
  from blocks length-eqs obtain l'' vs''
    where eval:P,E(p#ps' [↦] Ts) ⊢ ⟨e,(h,l(p#ps'[↦]vs''))⟩ ⇒
                                ⟨ref (a,Cs),(h',l'')⟩
    and casts:P ⊢ Ts Casts vs to vs''
    and length:length vs'' = length vs
    by −(drule blocksEval,auto)
  from eval notin leq have
    P,E(p#ps'[↦]Ts) ⊢ ⟨(⟦C'⟧)e,(h,l(p#ps'[↦]vs''))⟩ ⇒
                        ⟨THROW ClassCast,(h',l'')⟩
    by(auto intro:StaticCastFail)
  with e' casts length have ?case by simp blast }
 moreover
 { fix r assume P,E ⊢ ⟨blocks(p#ps',Ts,vs,e),(h,l)⟩ ⇒ ⟨throw r,(h',l')⟩
  and e':e' = throw r
   with length-eqs obtain l'' vs''
    where eval:P,E(p#ps' [↦] Ts) ⊢ ⟨e,(h,l(p#ps'[↦]vs''))⟩ ⇒
                                ⟨throw r,(h',l'')⟩
    and casts:P ⊢ Ts Casts vs to vs''
    and length:length vs'' = length vs
    by −(drule blocksEval,auto)
  from eval have
    P,E(p#ps'[↦]Ts) ⊢ ⟨(⟦C'⟧)e,(h,l(p#ps'[↦]vs''))⟩ ⇒
                        ⟨throw r,(h',l'')⟩
    by(auto intro:eval-evals.StaticCastThrow)
  with e' casts length have ?case by simp blast }
 ultimately show ?case
  by −(erule eval-cases,fastforce+)
qed



lemma
assumes wf: wwf-prog P
shows eval-restrict-lcl:
  P,E ⊢ ⟨e,(h,l)⟩ ⇒ ⟨e',(h',l')⟩ ⟹ (⋀W. fv e ⊆ W ⟹ P,E ⊢ ⟨e,(h,l|`W)⟩ ⇒
⟨e',(h',l'|`W)⟩)
and P,E ⊢ ⟨es,(h,l)⟩ [⇒] ⟨es',(h',l')⟩ ⟹ (⋀W. fvs es ⊆ W ⟹ P,E ⊢ ⟨es,(h,l|`W)⟩
[⇒] ⟨es',(h',l'|`W)⟩)

proof(induct rule:eval-evals-inducts)
 case (Block E V T e0 h0 l0 e1 h1 l1)
 have IH: ⋀W. fv e0 ⊆ W ⟹
```

$P,E(V \mapsto T) \vdash \langle e_0,(h_0,l_0(V{:=}None)|`W)\rangle \Rightarrow \langle e_1,(h_1,l_1|`W)\rangle$ **by** *fact*

have $fv(\{V{:}T;\ e_0\}) \subseteq W$ **by** *fact*
hence $fv\ e_0 - \{V\} \subseteq W$ **by** *simp-all*
hence $fv\ e_0 \subseteq insert\ V\ W$ **by** *fast*
with *IH*[*OF this*]
have $P,E(V \mapsto T) \vdash \langle e_0,(h_0,\ (l_0|`W)(V\ :=\ None))\rangle \Rightarrow \langle e_1,(h_1,\ l_1|`insert\ V\ W)\rangle$
by *fastforce*
from *eval-evals.Block*[*OF this*] **show** *?case* **by** *fastforce*
next
case *Seq* **thus** *?case* **by** *simp* (*blast intro*:*eval-evals.Seq*)
next
case *New* **thus** *?case* **by**(*simp add*:*eval-evals.intros*)
next
case *NewFail* **thus** *?case* **by**(*simp add*:*eval-evals.intros*)
next
case *StaticUpCast* **thus** *?case* **by** *simp* (*blast intro*:*eval-evals.StaticUpCast*)
next
case (*StaticDownCast E e h l a Cs C Cs' h' l'*)
have *IH*:$\bigwedge W.\ fv\ e \subseteq W \Longrightarrow$
$P,E \vdash \langle e,(h,l\ |`\ W)\rangle \Rightarrow \langle ref(a,Cs@[C]@Cs'),(h',l'\ |`\ W)\rangle$ **by** *fact*
have $fv\ (\!|C|\!)e) \subseteq W$ **by** *fact*
hence $fv\ e \subseteq W$ **by** *simp*
from *IH*[*OF this*] **show** *?case* **by**(*rule eval-evals.StaticDownCast*)
next
case *StaticCastNull* **thus** *?case* **by** *simp* (*blast intro*:*eval-evals.StaticCastNull*)
next
case *StaticCastFail* **thus** *?case* **by** *simp* (*blast intro*:*eval-evals.StaticCastFail*)
next
case *StaticCastThrow* **thus** *?case* **by**(*simp add*:*eval-evals.intros*)
next
case *DynCast* **thus** *?case* **by** *simp* (*blast intro*:*eval-evals.DynCast*)
next
case *StaticUpDynCast* **thus** *?case* **by** *simp* (*blast intro*:*eval-evals.StaticUpDynCast*)
next
case (*StaticDownDynCast E e h l a Cs C Cs' h' l'*)
have *IH*:$\bigwedge W.\ fv\ e \subseteq W \Longrightarrow$
$P,E \vdash \langle e,(h,l\ |`\ W)\rangle \Rightarrow \langle ref(a,Cs@[C]@Cs'),(h',l'\ |`\ W)\rangle$ **by** *fact*
have $fv\ (Cast\ C\ e) \subseteq W$ **by** *fact*
hence $fv\ e \subseteq W$ **by** *simp*
from *IH*[*OF this*] **show** *?case* **by**(*rule eval-evals.StaticDownDynCast*)
next
case *DynCastNull* **thus** *?case* **by** *simp* (*blast intro*:*eval-evals.DynCastNull*)
next
case *DynCastFail* **thus** *?case* **by** *simp* (*blast intro*:*eval-evals.DynCastFail*)
next
case *DynCastThrow* **thus** *?case* **by**(*simp add*:*eval-evals.intros*)

next
case *Val* **thus** *?case* **by**(*simp add*:*eval-evals.intros*)
next
case *BinOp* **thus** *?case* **by** *simp* (*blast intro*:*eval-evals.BinOp*)
next
case *BinOpThrow1* **thus** *?case* **by** *simp* (*blast intro*:*eval-evals.BinOpThrow1*)
next
case *BinOpThrow2* **thus** *?case* **by** *simp* (*blast intro*:*eval-evals.BinOpThrow2*)
next
case *Var* **thus** *?case* **by**(*simp add*:*eval-evals.intros*)
next
case (*LAss E e* h_0 l_0 *v h l V T v' l'*)
have *IH*: $\bigwedge W$. *fv e* $\subseteq$ *W* $\Longrightarrow$ *P*,*E* $\vdash$ $\langle e,(h_0,l_0|`W)\rangle \Rightarrow \langle Val\ v,(h,l|`W)\rangle$
and *env*:*E V* = $\lfloor T \rfloor$ **and** *casts*:*P* $\vdash$ *T casts v to v'*
and [*simp*]: *l'* = *l*(*V* $\mapsto$ *v'*) **by** *fact+*
have *fv* (*V*:=*e*) $\subseteq$ *W* **by** *fact*
hence *fv*: *fv e* $\subseteq$ *W* **and** *VinW*: *V* $\in$ *W* **by** *auto*
from *eval-evals.LAss*[*OF IH*[*OF fv*] - *casts*] *env VinW*
show *?case* **by** *fastforce*
next
case *LAssThrow* **thus** *?case* **by**(*fastforce intro*: *eval-evals.LAssThrow*)
next
case *FAcc* **thus** *?case* **by** *simp* (*blast intro*: *eval-evals.FAcc*)
next
case *FAccNull* **thus** *?case* **by**(*fastforce intro*: *eval-evals.FAccNull*)
next
case *FAccThrow* **thus** *?case* **by**(*fastforce intro*: *eval-evals.FAccThrow*)
next
case (*FAss E* e_1 *h l a Cs' h' l'* e_2 *v* h_2 l_2 *D S F T Cs v' Ds fs fs' S'* h_2' *W*)
have *IH1*: $\bigwedge W$. *fv* e_1 $\subseteq$ *W* $\Longrightarrow$ *P*,*E* $\vdash$ $\langle e_1,(h,\ l|`W)\rangle \Rightarrow \langle ref\ (a,\ Cs'),(h',\ l'|`W)\rangle$
and *IH2*: $\bigwedge W$. *fv* e_2 $\subseteq$ *W* $\Longrightarrow$ *P*,*E* $\vdash$ $\langle e_2,(h',\ l'|`W)\rangle \Rightarrow \langle Val\ v,(h_2,\ l_2|`W)\rangle$
and *fv*:*fv* (e_1·*F*{*Cs*} := e_2) $\subseteq$ *W*
and *h*:h_2 *a* = *Some*(*D*,*S*) **and** *Ds*:*Ds* = *Cs'* $@_p$ *Cs*
and *S*:(*Ds*,*fs*) $\in$ *S* **and** *fs'*:*fs'* = *fs*(*F* $\mapsto$ *v'*)
and *S'*:*S'* = *S* − {(*Ds*, *fs*)} ∪ {(*Ds*, *fs'*)}
and *h'*:h_2' = h_2(*a* $\mapsto$ (*D*, *S'*))
and *field*:*P* $\vdash$ *last Cs' has least F*:*T via Cs*
and *casts*:*P* $\vdash$ *T casts v to v'* **by** *fact+*
from *fv* **have** *fv1*:*fv* e_1 $\subseteq$ *W* **and** *fv2*:*fv* e_2 $\subseteq$ *W* **by** *auto*
from *eval-evals.FAss*[*OF IH1*[*OF fv1*] *IH2*[*OF fv2*] - *field casts*] *h Ds S fs' S' h'*
show *?case* **by** *simp*
next
case *FAssNull* **thus** *?case* **by** *simp* (*blast intro*: *eval-evals.FAssNull*)
next
case *FAssThrow1* **thus** *?case* **by** *simp* (*blast intro*: *eval-evals.FAssThrow1*)
next
case *FAssThrow2* **thus** *?case* **by** *simp* (*blast intro*: *eval-evals.FAssThrow2*)
next

case *CallObjThrow* **thus** *?case* **by** *simp* (*blast intro*: *eval-evals.intros*)
next
case *CallNull* **thus** *?case* **by** *simp* (*blast intro*: *eval-evals.CallNull*)
next
case *CallParamsThrow* **thus** *?case*
by *simp* (*blast intro*: *eval-evals.CallParamsThrow*)
next
case (*Call E e* h_0 l_0 *a Cs* h_1 l_1 *ps vs* h_2 l_2 *C S M Ts' T' pns'*
body' Ds Ts T pns body Cs' vs' l_2' *new-body e'* h_3 l_3 *W*)
have *IHe*: $\bigwedge W$. *fv e* $\subseteq$ *W* $\Longrightarrow$ *P*,*E* $\vdash$ $\langle e,(h_0,l_0|`W)\rangle \Rightarrow \langle ref(a,Cs),(h_1,l_1|`W)\rangle$
and *IHps*: $\bigwedge W$. *fvs ps* $\subseteq$ *W* $\Longrightarrow$ *P*,*E* $\vdash$ $\langle ps,(h_1,l_1|`W)\rangle$ [$\Rightarrow$] $\langle map\ Val\ vs,(h_2,l_2|`W)\rangle$
and *IHbd*: $\bigwedge W$. *fv new-body* $\subseteq$ *W* $\Longrightarrow$ *P*,*E*(*this* $\mapsto$ *Class* (*last Cs'*), *pns* [$\mapsto$] *Ts*) $\vdash$
$\langle new\text{-}body,(h_2,l_2'|`W)\rangle \Rightarrow \langle e',(h_3,l_3|`W)\rangle$
and $h_2 a$: 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*@$_p$*Ds*) *selects M* = (*Ts*,*T*,*pns*,*body*) *via Cs'*
and *same-len*: *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*) **by** *fact*+
have *fv* (*e*·*M*(*ps*)) $\subseteq$ *W* **by** *fact*
hence *fve*: *fv e* $\subseteq$ *W* **and** *fvps*: *fvs*(*ps*) $\subseteq$ *W* **by** *auto*
have *wfmethod*: *size Ts* = *size pns* $\wedge$ *this* $\notin$ *set pns* **and**
fvbd: *fv body* $\subseteq$ {*this*} $\cup$ *set pns*
using *select wf* **by**(*fastforce dest*!:*select-method-wf-mdecl simp*:*wf-mdecl-def*)+
from *fvbd body-case* **have** *fvbd'*:*fv new-body* $\subseteq$ {*this*} $\cup$ *set pns*
by(*cases T'*) *auto*
from l_2' **have** l_2' |` ({*this*} $\cup$ *set pns*) = [*this* $\mapsto$ *Ref* (*a*, *Cs'*), *pns* [$\mapsto$] *vs'*]
by (*auto intro*!:*ext simp*:*restrict-map-def fun-upd-def*)
with *eval-evals.Call*[*OF IHe*[*OF fve*] *IHps*[*OF fvps*] - *method select same-len*
casts - *body-case IHbd*[*OF fvbd'*]] $h_2 a$
show *?case* **by** *simp*
next
case (*StaticCall E e* h_0 l_0 *a Cs* h_1 l_1 *ps vs* h_2 l_2 *C Cs'' M Ts T pns body*
Cs' Ds vs' l_2' *e'* h_3 l_3 *W*)
have *IHe*: $\bigwedge W$. *fv e* $\subseteq$ *W* $\Longrightarrow$ *P*,*E* $\vdash$ $\langle e,(h_0,l_0|`W)\rangle \Rightarrow \langle ref(a,Cs),(h_1,l_1|`W)\rangle$
and *IHps*: $\bigwedge W$. *fvs ps* $\subseteq$ *W* $\Longrightarrow$ *P*,*E* $\vdash$ $\langle ps,(h_1,l_1|`W)\rangle$ [$\Rightarrow$] $\langle map\ Val\ vs,(h_2,l_2|`W)\rangle$
and *IHbd*: $\bigwedge W$. *fv body* $\subseteq$ *W* $\Longrightarrow$ *P*,*E*(*this* $\mapsto$ *Class* (*last Ds*), *pns* [$\mapsto$] *Ts*) $\vdash$
$\langle body,(h_2,l_2'|`W)\rangle \Rightarrow \langle e',(h_3,l_3|`W)\rangle$
and *path-unique*: *P* $\vdash$ *Path last Cs to C unique*
and *path-via*: *P* $\vdash$ *Path last Cs to C via Cs''*
and *least*: *P* $\vdash$ *C has least M* = (*Ts*, *T*, *pns*, *body*) *via Cs'*
and *Ds*: *Ds* = (*Cs* @$_p$ *Cs''*) @$_p$ *Cs'*
and *same-len*: *size vs* = *size pns*

```
    and casts:P ⊢ Ts Casts vs to vs'
    and l2': l2' = [this ↦ Ref(a,Ds), pns [↦] vs'] by fact+
  have fv (e•(C::)M(ps)) ⊆ W by fact
  hence fve: fv e ⊆ W and fvps: fvs(ps) ⊆ W by auto
  have wfmethod: size Ts = size pns ∧ this ∉ set pns and
      fvbd: fv body ⊆ {this} ∪ set pns
    using least wf by(fastforce dest!:has-least-wf-mdecl simp:wf-mdecl-def)+
  from fvbd have fvbd':fv body ⊆ {this} ∪ set pns
    by auto
  from l2' have l2' |` ({this} ∪ set pns) = [this ↦ Ref(a,Ds), pns [↦] vs']
    by (auto intro!:ext simp:restrict-map-def fun-upd-def)
  with eval-evals.StaticCall[OF IHe[OF fve] IHps[OF fvps] path-unique path-via
                                least Ds same-len casts - IHbd[OF fvbd']]
  show ?case by simp
next
  case SeqThrow thus ?case by simp (blast intro: eval-evals.SeqThrow)
next
  case CondT thus ?case by simp (blast intro: eval-evals.CondT)
next
  case CondF thus ?case by simp (blast intro: eval-evals.CondF)
next
  case CondThrow thus ?case by simp (blast intro: eval-evals.CondThrow)
next
  case WhileF thus ?case by simp (blast intro: eval-evals.WhileF)
next
  case WhileT thus ?case by simp (blast intro: eval-evals.WhileT)
next
 case WhileCondThrow thus ?case by simp (blast intro: eval-evals.WhileCondThrow)
next
 case WhileBodyThrow thus ?case by simp (blast intro: eval-evals.WhileBodyThrow)
next
  case Throw thus ?case by simp (blast intro: eval-evals.Throw)
next
  case ThrowNull thus ?case by simp (blast intro: eval-evals.ThrowNull)
next
  case ThrowThrow thus ?case by simp (blast intro: eval-evals.ThrowThrow)
next
  case Nil thus ?case by (simp add: eval-evals.Nil)
next
  case Cons thus ?case by simp (blast intro: eval-evals.Cons)
next
  case ConsThrow thus ?case by simp (blast intro: eval-evals.ConsThrow)
qed



lemma eval-notfree-unchanged:
assumes wf:wwf-prog P
shows P,E ⊢ ⟨e,(h,l)⟩ ⇒ ⟨e',(h',l')⟩ ⟹ (⋀V. V ∉ fv e ⟹ 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(*induct rule*:*eval-evals-inducts*)
case *LAss* **thus** *?case* **by**(*simp add*:*fun-upd-apply*)
next
case *Block* **thus** *?case*
by (*simp only*:*fun-upd-apply split*:*if-splits*) *fastforce*
qed *simp-all*

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 −
from *wf eval* **have** $\bigwedge V.\ V \notin fv\ e \Longrightarrow l'\ V = l\ V$ **by** (*rule eval-notfree-unchanged*)
with *fv* **have** $\bigwedge V.\ l'\ V = l\ V$ **by** *simp*
thus *?thesis* **by**(*simp add*:*fun-eq-iff*)
qed

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

declaration ‹*K* (*Simplifier.map-ss* (*fn ss => ss delloop split-all-tac*))›
setup ‹*map-theory-claset* (*fn ctxt => ctxt delSWrapper split-all-tac*)›

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 −
from *eval-e*
obtain *a* **where** *e'*: $e' = Throw\ a$
by (*cases*) (*auto dest*!: *eval-final*)
{
fix *es*
have $\bigwedge vs.\ es = map\ Val\ vs\ @\ throw\ x\ \#\ es'$
$\Longrightarrow P,E \vdash \langle es,s\rangle[\Rightarrow]\langle map\ Val\ vs\ @\ e'\ \#\ es',s'\rangle$
proof (*induct es type*: *list*)
case *Nil* **thus** *?case* **by** *simp*
next
case (*Cons e es vs*)

```
    have e-es: e # es = map Val vs @ throw x # es' by fact
    show P,E ⊢ ⟨e # es,s⟩ [⇒] ⟨map Val vs @ e' # es',s'⟩
    proof (cases vs)
      case Nil
      with e-es obtain e=throw x es=es' by simp
      moreover from eval-e e'
      have P,E ⊢ ⟨throw x # es,s⟩ [⇒] ⟨Throw a # es,s'⟩
        by (iprover intro: ConsThrow)
      ultimately show ?thesis using Nil e' by simp
    next
      case (Cons v vs')
      have vs: vs = v # vs' by fact
      with e-es obtain
        e: e=Val v and es:es= map Val vs' @ throw x # es'
        by simp
      from e
      have P,E ⊢ ⟨e,s⟩ ⇒ ⟨Val v,s⟩
        by (iprover intro: eval-evals.Val)
      moreover from es
      have P,E ⊢ ⟨es,s⟩ [⇒] ⟨map Val vs' @ e' # es',s'⟩
        by (rule Cons.hyps)
      ultimately show
        P,E ⊢ ⟨e#es,s⟩ [⇒] ⟨map Val vs @ e' # es',s'⟩
        using vs by (auto intro: eval-evals.Cons)
    qed
  qed
 }
 thus ?thesis
  by simp
qed
```

The key lemma:

```
lemma
assumes wf: wwf-prog P
shows extend-1-eval:
  P,E ⊢ ⟨e,s⟩ → ⟨e'',s''⟩ ⟹ (⋀s' e'. P,E ⊢ ⟨e'',s''⟩ ⇒ ⟨e',s'⟩ ⟹ P,E ⊢ ⟨e,s⟩
⇒ ⟨e',s'⟩)
and extend-1-evals:
  P,E ⊢ ⟨es,t⟩ [→] ⟨es'',t''⟩ ⟹ (⋀t' es'. P,E ⊢ ⟨es'',t''⟩ [⇒] ⟨es',t'⟩ ⟹ P,E ⊢
⟨es,t⟩ [⇒] ⟨es',t'⟩)

proof (induct rule: red-reds.inducts)
 case RedNew thus ?case by (iprover elim: eval-cases intro: eval-evals.intros)
next
 case RedNewFail thus ?case by (fastforce elim: eval-cases intro: eval-evals.intros)
next
 case (StaticCastRed E e s e'' s'' C s' e') thus ?case
  by -(erule eval-cases,auto intro:eval-evals.intros,
       subgoal-tac P,E ⊢ ⟨e'',s''⟩ ⇒ ⟨ref(a,Cs@[C]@Cs'),s'⟩,
```

```
        rule-tac Cs'=Cs' in StaticDownCast,auto)
next
  case RedStaticCastNull thus ?case
    by (fastforce elim: eval-cases intro: eval-evals.intros)
next
  case RedStaticUpCast thus ?case
    by (fastforce elim: eval-cases intro: eval-evals.intros)
next
  case RedStaticDownCast thus ?case
    by (fastforce elim: eval-cases intro: eval-evals.intros)
next
  case RedStaticCastFail thus ?case
    by (fastforce elim: eval-cases intro: eval-evals.intros)
next
  case RedStaticUpDynCast thus ?case
    by (fastforce elim: eval-cases intro: eval-evals.intros)
next
  case RedStaticDownDynCast thus ?case
    by (fastforce elim: eval-cases intro: eval-evals.intros)
next
  case (DynCastRed E e s e'' s'' C s' e')
  have eval:P,E ⊢ ⟨Cast C e'',s''⟩ ⇒ ⟨e',s'⟩
   and IH:⋀ex sx. P,E ⊢ ⟨e'',s''⟩ ⇒ ⟨ex,sx⟩ ⟹ P,E ⊢ ⟨e,s⟩ ⇒ ⟨ex,sx⟩ by fact+
  moreover
  { fix Cs Cs' a
    assume P,E ⊢ ⟨e'',s''⟩ ⇒ ⟨ref (a, Cs @ C # Cs'),s'⟩
    from IH[OF this] have P,E ⊢ ⟨e,s⟩ ⇒ ⟨ref (a, Cs@[C]@Cs'),s'⟩ by simp
   hence P,E ⊢ ⟨Cast C e,s⟩ ⇒ ⟨ref (a, Cs@[C]),s'⟩ by(rule StaticDownDynCast)
}
  ultimately show ?case by −(erule eval-cases,auto intro: eval-evals.intros)
next
 case RedDynCastNull thus ?case by (iprover elim:eval-cases intro:eval-evals.intros)
next
  case (RedDynCast s a D S C Cs' E Cs s' e')
  thus ?case by (cases s)(auto elim!:eval-cases intro:eval-evals.intros)
next
  case (RedDynCastFail s a D S C Cs E s'' e'')
  thus ?case by (cases s)(auto elim!: eval-cases intro: eval-evals.intros)
next
  case BinOpRed1 thus ?case by −(erule eval-cases,auto intro: eval-evals.intros)
next
  case BinOpRed2
  thus ?case by (fastforce elim!:eval-cases intro:eval-evals.intros eval-finalId)
next
  case RedBinOp thus ?case by (iprover elim:eval-cases intro:eval-evals.intros)
next
  case (RedVar s V v E s' e')
  thus ?case by (cases s)(fastforce elim:eval-cases intro:eval-evals.intros)
next
```

case *LAssRed* **thus** *?case* **by** −(*erule eval-cases*,*auto intro*:*eval-evals.intros*)
next
case *RedLAss*
thus *?case* **by** (*fastforce elim*:*eval-cases intro*:*eval-evals.intros*)
next
case *FAccRed* **thus** *?case* **by** −(*erule eval-cases*,*auto intro*:*eval-evals.intros*)
next
case (*RedFAcc s a D S Ds Cs′ Cs fs F v E s′ e′*)
thus *?case* **by** (*cases s*)(*fastforce elim*:*eval-cases intro*:*eval-evals.intros*)
next
case *RedFAccNull* **thus** *?case* **by** (*fastforce elim*!:*eval-cases intro*:*eval-evals.intros*)
next
case (*FAssRed1 E* e_1 *s* $e_1{}'$ *s″ F Cs* e_2 *s′ e′*)
have *eval*:$P,E \vdash \langle e_1{}',s''\rangle \Rightarrow \langle e',s'\rangle$
and *IH*:$\bigwedge ex\ sx.\ P,E \vdash \langle e_1{}',s''\rangle \Rightarrow \langle ex,sx\rangle \Longrightarrow P,E \vdash \langle e_1,s\rangle \Rightarrow \langle ex,sx\rangle$ **by** *fact*+
{ **fix** *Cs′ D S T a fs* h_2 l_2 s_1 *v v′*
assume *ref*:$P,E \vdash \langle e_1{}',s''\rangle \Rightarrow \langle ref\ (a,\ Cs'),s_1\rangle$
and *rest*:$P,E \vdash \langle e_2,s_1\rangle \Rightarrow \langle Val\ v,(h_2,\ l_2)\rangle$ $h_2\ a = \lfloor (D,\ S)\rfloor$
$P \vdash last\ Cs'\ has\ least\ F{:}T\ via\ Cs\ P \vdash T\ casts\ v\ to\ v'$
$(Cs' @_p Cs,\ fs) \in S$
from *IH*[*OF ref*] **have** $P,E \vdash \langle e_1,s\rangle \Rightarrow \langle ref\ (a,\ Cs'),s_1\rangle$.
with *rest* **have** $P,E \vdash \langle e_1{\cdot}F\{Cs\} := e_2,s\rangle \Rightarrow$
$\langle Val\ v',(h_2(a \mapsto (D,insert\ (Cs'@_pCs,fs(F \mapsto v'))(S - \{(Cs'@_pCs,fs)\}))),l_2)\rangle$
by−(*rule FAss*,*simp-all*) }
moreover
{ **fix** s_1 *v*
assume *null*:$P,E \vdash \langle e_1{}',s''\rangle \Rightarrow \langle null,s_1\rangle$
and *rest*:$P,E \vdash \langle e_2,s_1\rangle \Rightarrow \langle Val\ v,s'\rangle$
from *IH*[*OF null*] **have** $P,E \vdash \langle e_1,s\rangle \Rightarrow \langle null,s_1\rangle$.
with *rest* **have** $P,E \vdash \langle e_1{\cdot}F\{Cs\} := e_2,s\rangle \Rightarrow \langle THROW\ NullPointer,s'\rangle$
by−(*rule FAssNull*,*simp-all*) }
moreover
{ **fix** *e′* **assume** *throw*:$P,E \vdash \langle e_1{}',s''\rangle \Rightarrow \langle throw\ e',s'\rangle$
from *IH*[*OF throw*] **have** $P,E \vdash \langle e_1,s\rangle \Rightarrow \langle throw\ e',s'\rangle$.
hence $P,E \vdash \langle e_1{\cdot}F\{Cs\} := e_2,s\rangle \Rightarrow \langle throw\ e',s'\rangle$
by−(*rule eval-evals.FAssThrow1*,*simp-all*) }
moreover
{ **fix** *e′* s_1 *v*
assume *val*:$P,E \vdash \langle e_1{}',s''\rangle \Rightarrow \langle Val\ v,s_1\rangle$
and *rest*:$P,E \vdash \langle e_2,s_1\rangle \Rightarrow \langle throw\ e',s'\rangle$
from *IH*[*OF val*] **have** $P,E \vdash \langle e_1,s\rangle \Rightarrow \langle Val\ v,s_1\rangle$.
with *rest* **have** $P,E \vdash \langle e_1{\cdot}F\{Cs\} := e_2,s\rangle \Rightarrow \langle throw\ e',s'\rangle$
by−(*rule eval-evals.FAssThrow2*,*simp-all*) }
ultimately show *?case* **using** *eval*
by −(*erule eval-cases*,*auto*)
next
case (*FAssRed2 E* e_2 *s* $e_2{}'$ *s″ v F Cs s′ e′*)
have *eval*:$P,E \vdash \langle Val\ v{\cdot}F\{Cs\} := e_2{}',s''\rangle \Rightarrow \langle e',s'\rangle$

and *IH*:$\bigwedge ex\ sx.\ P,E \vdash \langle e_2',s''\rangle \Rightarrow \langle ex,sx\rangle \Longrightarrow P,E \vdash \langle e_2,s\rangle \Rightarrow \langle ex,sx\rangle$ **by** *fact+*
{ fix $Cs'\ D\ S\ T\ a\ fs\ h_2\ l_2\ s_1\ v'\ v''$
assume *val1*:$P,E \vdash \langle Val\ v,s''\rangle \Rightarrow \langle ref\ (a,Cs'),s_1\rangle$
and *val2*:$P,E \vdash \langle e_2',s_1\rangle \Rightarrow \langle Val\ v',(h_2,\ l_2)\rangle$
and *rest*:$h_2\ a = \lfloor(D,\ S)\rfloor\ P \vdash$ *last* Cs' *has least* F:T *via* Cs
$P \vdash T$ *casts* v' *to* $v''\ (Cs'@_pCs,fs) \in S$
from *val1* **have** s'':$s_1 = s''$ **by** $-$(*erule eval-cases*)
with *val1* **have** $P,E \vdash \langle Val\ v,s\rangle \Rightarrow \langle ref\ (a,Cs'),s\rangle$
by(*fastforce elim*:*eval-cases intro*:*eval-finalId*)
also from *IH*[*OF val2*[*simplified* s'']] **have** $P,E \vdash \langle e_2,s\rangle \Rightarrow \langle Val\ v',(h_2,\ l_2)\rangle$ **.**
ultimately have $P,E \vdash \langle Val\ v{\cdot}F\{Cs\} := e_2,s\rangle \Rightarrow$
$\langle Val\ v'',(h_2(a{\mapsto}(D,insert(Cs'@_pCs,fs(F \mapsto v''))(S - \{(Cs'@_pCs,fs)\}))),l_2)\rangle$
using *rest* **by** $-$(*rule FAss*,*simp-all*) **}**
moreover
{ fix $s_1\ v'$
assume *val1*:$P,E \vdash \langle Val\ v,s''\rangle \Rightarrow \langle null,s_1\rangle$
and *val2*:$P,E \vdash \langle e_2',s_1\rangle \Rightarrow \langle Val\ v',s'\rangle$
from *val1* **have** s'':$s_1 = s''$ **by** $-$(*erule eval-cases*)
with *val1* **have** $P,E \vdash \langle Val\ v,s\rangle \Rightarrow \langle null,s\rangle$
by(*fastforce elim*:*eval-cases intro*:*eval-finalId*)
also from *IH*[*OF val2*[*simplified* s'']] **have** $P,E \vdash \langle e_2,s\rangle \Rightarrow \langle Val\ v',s'\rangle$ **.**
ultimately have $P,E \vdash \langle Val\ v{\cdot}F\{Cs\} := e_2,s\rangle \Rightarrow \langle THROW\ NullPointer,s'\rangle$
by $-$(*rule FAssNull*,*simp-all*) **}**
moreover
{ fix r **assume** *val*:$P,E \vdash \langle Val\ v,s''\rangle \Rightarrow \langle throw\ r,s'\rangle$
hence s'':$s'' = s'$ **by** $-$(*erule eval-cases*,*simp*)
with *val* **have** $P,E \vdash \langle Val\ v{\cdot}F\{Cs\} := e_2,s\rangle \Rightarrow \langle throw\ r,s'\rangle$
by $-$(*rule eval-evals.FAssThrow1*,*erule eval-cases*,*simp*) **}**
moreover
{ fix $r\ s_1\ v'$
assume *val1*:$P,E \vdash \langle Val\ v,s''\rangle \Rightarrow \langle Val\ v',s_1\rangle$
and *val2*:$P,E \vdash \langle e_2',s_1\rangle \Rightarrow \langle throw\ r,s'\rangle$
from *val1* **have** s'':$s_1 = s''$ **by** $-$(*erule eval-cases*)
with *val1* **have** $P,E \vdash \langle Val\ v,s\rangle \Rightarrow \langle Val\ v',s\rangle$
by(*fastforce elim*:*eval-cases intro*:*eval-finalId*)
also from *IH*[*OF val2*[*simplified* s'']] **have** $P,E \vdash \langle e_2,s\rangle \Rightarrow \langle throw\ r,s'\rangle$ **.**
ultimately have $P,E \vdash \langle Val\ v{\cdot}F\{Cs\} := e_2,s\rangle \Rightarrow \langle throw\ r,s'\rangle$
by $-$(*rule eval-evals.FAssThrow2*,*simp-all*) **}**
ultimately show *?case* **using** *eval*
by $-$(*erule eval-cases*,*auto*)
next
case (*RedFAss h a D S Cs' F T Cs v v' Ds fs E l s' e'*)
have *val*:$P,E \vdash \langle Val\ v',(h(a \mapsto (D,insert\ (Ds,fs(F \mapsto v'))(S - \{(Ds,fs)\}))),l)\rangle$
$\Rightarrow$
$\langle e',s'\rangle$
and *rest*:$h\ a = \lfloor(D,\ S)\rfloor\ P \vdash$ *last* Cs' *has least* F:T *via* Cs
$P \vdash T$ *casts* v *to* $v'\ Ds = Cs'\ @_p\ Cs\ (Ds,\ fs) \in S$ **by** *fact+*
from *val* **have** $s' = (h(a \mapsto (D,insert\ (Ds,fs(F \mapsto v'))\ (S - \{(Ds,fs)\}))),l)$

and $e' = Val\ v'$ **by** $-(erule\ eval\text{-}cases, simp\text{-}all)+$
with *rest* **show** *?case* **apply** *simp*
by(*rule FAss*,*simp-all*)(*rule eval-finalId*,*simp*)+
next
case *RedFAssNull*
thus *?case* **by** (*fastforce elim*!: *eval-cases intro*: *eval-evals.intros*)
next
case (*CallObj E e s e' s' Copt M es s'' e''*)
thus *?case*
apply −
apply(*cases Copt*,*simp*)
by(*erule eval-cases*,*auto intro*:*eval-evals.intros*)+
next
case (*CallParams E es s es' s'' v Copt M s' e'*)
have *call*:$P,E \vdash \langle Call\ (Val\ v)\ Copt\ M\ es',s''\rangle \Rightarrow \langle e',s'\rangle$
and *IH*:$\bigwedge esx\ sx.\ P,E \vdash \langle es',s''\rangle\ [\Rightarrow]\ \langle esx,sx\rangle \Longrightarrow P,E \vdash \langle es,s\rangle\ [\Rightarrow]\ \langle esx,sx\rangle$
by *fact*+
show *?case*
proof(*cases Copt*)
case *None* **with** *call* **have** *eval*:$P,E \vdash \langle Val\ v{\cdot}M(es'),s''\rangle \Rightarrow \langle e',s'\rangle$ **by** *simp*
from *eval* **show** *?thesis*
proof(*rule eval-cases*)
fix *r* **assume** $P,E \vdash \langle Val\ v,s''\rangle \Rightarrow \langle throw\ r,s'\rangle\ e' = throw\ r$
with *None* **show** $P,E \vdash \langle Call\ (Val\ v)\ Copt\ M\ es,s\rangle \Rightarrow \langle e',s'\rangle$
by(*fastforce elim*:*eval-cases*)
next
fix *es'' r sx v' vs*
assume *val*:$P,E \vdash \langle Val\ v,s''\rangle \Rightarrow \langle Val\ v',sx\rangle$
and *evals*:$P,E \vdash \langle es',sx\rangle\ [\Rightarrow]\ \langle map\ Val\ vs\ @\ throw\ r\ \#\ es'',s'\rangle$
and *e'*:$e' = throw\ r$
have *val'*:$P,E \vdash \langle Val\ v,s\rangle \Rightarrow \langle Val\ v,s\rangle$ **by**(*rule Val*)
from *val* **have** *eq*:$v' = v \wedge s'' = sx$ **by** $-(erule\ eval\text{-}cases, simp)$
with *IH evals* **have** $P,E \vdash \langle es,s\rangle\ [\Rightarrow]\ \langle map\ Val\ vs\ @\ throw\ r\ \#\ es'',s'\rangle$
by *simp*
with *eq CallParamsThrow*$[OF\ val']\ e'\ None$
show $P,E \vdash \langle Call\ (Val\ v)\ Copt\ M\ es,s\rangle \Rightarrow \langle e',s'\rangle$
by *fastforce*
next
fix *C Cs Cs' Ds S T T' Ts Ts' a body body' h*$_2$ *h*$_3$ *l*$_2$ *l*$_3$ *pns pns' s*$_1$ *vs vs'*
assume *val*:$P,E \vdash \langle Val\ v,s''\rangle \Rightarrow \langle ref(a,Cs),s_1\rangle$
and *evals*:$P,E \vdash \langle es',s_1\rangle\ [\Rightarrow]\ \langle map\ Val\ vs,(h_2,l_2)\rangle$
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 *length*:$length\ vs = length\ pns$
and *casts*:$P \vdash Ts\ Casts\ vs\ to\ vs'$
and *body*:$P,E(this \mapsto Class\ (last\ Cs'),\ pns\ [\mapsto]\ Ts) \vdash$
$\langle case\ T'\ of\ Class\ D \Rightarrow (\!|D|\!)body \mid \text{-} \Rightarrow body,(h_2,[this \mapsto Ref(a,Cs'),pns\ [\mapsto]\ vs'])\rangle$

```
      ⇒ ⟨e',(h3, l3)⟩
      and s':s' = (h3, l2)
    from val have val':P,E ⊢ ⟨Val v,s⟩ ⇒ ⟨ref(a,Cs),s⟩
      and eq:s'' = s1 ∧ v = Ref(a,Cs)
      by(auto elim:eval-cases intro:Val)
    from body obtain new-body
      where body-case:new-body = (case T' of Class D ⇒ (⌊D⌋body | - ⇒ body)
      and body':P,E(this ↦ Class (last Cs'), pns [↦] Ts) ⊢
      ⟨new-body,(h2,[this ↦ Ref(a,Cs'),pns [↦] vs'])⟩ ⇒ ⟨e',(h3, l3)⟩
      by simp
    from eq IH evals have P,E ⊢ ⟨es,s⟩ [⇒] ⟨map Val vs,(h2,l2)⟩ by simp
    with eq Call[OF val' - - method select length casts - body-case]
        hp body' s' None
    show P,E ⊢ ⟨Call (Val v) Copt M es,s⟩ ⇒ ⟨e',s'⟩ by fastforce
  next
    fix s1 vs
    assume val:P,E ⊢ ⟨Val v,s''⟩ ⇒ ⟨null,s1⟩
      and evals:P,E ⊢ ⟨es',s1⟩ [⇒] ⟨map Val vs,s'⟩
      and e':e' = THROW NullPointer
    from val have val':P,E ⊢ ⟨Val v,s⟩ ⇒ ⟨null,s⟩
      and eq:s'' = s1 ∧ v = Null
      by(auto elim:eval-cases intro:Val)
    from eq IH evals have P,E ⊢ ⟨es,s⟩ [⇒] ⟨map Val vs,s'⟩ by simp
    with eq CallNull[OF val'] e' None
    show P,E ⊢ ⟨Call (Val v) Copt M es,s⟩ ⇒ ⟨e',s'⟩ by fastforce
  qed
next
  case (Some C) with call have eval:P,E ⊢ ⟨Val v•(C::)M(es'),s''⟩ ⇒ ⟨e',s'⟩
    by simp
  from eval show ?thesis
  proof(rule eval-cases)
    fix r assume P,E ⊢ ⟨Val v,s''⟩ ⇒ ⟨throw r,s'⟩ e' = throw r
    with Some show P,E ⊢ ⟨Call (Val v) Copt M es,s⟩ ⇒ ⟨e',s'⟩
      by(fastforce elim:eval-cases)
  next
    fix es'' r sx v' vs
    assume val:P,E ⊢ ⟨Val v,s''⟩ ⇒ ⟨Val v',sx⟩
      and evals:P,E ⊢ ⟨es',sx⟩ [⇒] ⟨map Val vs @ throw r # es'',s'⟩
      and e':e' = throw r
    have val':P,E ⊢ ⟨Val v,s⟩ ⇒ ⟨Val v,s⟩ by(rule Val)
    from val have eq:v' = v ∧ s'' = sx by −(erule eval-cases,simp)
    with IH evals have P,E ⊢ ⟨es,s⟩ [⇒] ⟨map Val vs @ throw r # es'',s'⟩
      by simp
    with eq CallParamsThrow[OF val'] e' Some
    show P,E ⊢ ⟨Call (Val v) Copt M es,s⟩ ⇒ ⟨e',s'⟩
      by fastforce
  next
    fix Cs Cs' Cs'' T Ts a body h2 h3 l2 l3 pns s1 vs vs'
    assume val:P,E ⊢ ⟨Val v,s''⟩ ⇒ ⟨ref (a,Cs),s1⟩
```

```
      and evals:P,E ⊢ ⟨es',s1⟩ [⇒] ⟨map Val vs,(h2,l2)⟩
      and path-unique:P ⊢ Path last Cs to C unique
      and path-via:P ⊢ Path last Cs to C via Cs''
      and least:P ⊢ C has least M = (Ts, T, pns, body) via Cs'
      and length:length vs = length pns
      and casts:P ⊢ Ts Casts vs to vs'
      and body:P,E(this ↦ Class (last ((Cs @p Cs'') @p Cs')), pns [↦] Ts) ⊢
        ⟨body,(h2,[this ↦ Ref(a,(Cs@pCs'')@pCs'),pns [↦] vs'])⟩ ⇒ ⟨e',(h3,l3)⟩
      and s':s' = (h3,l2)
    from val have val':P,E ⊢ ⟨Val v,s⟩ ⇒ ⟨ref(a,Cs),s⟩
      and eq:s'' = s1 ∧ v = Ref(a,Cs)
      by(auto elim:eval-cases intro:Val)
    from eq IH evals have P,E ⊢ ⟨es,s⟩ [⇒] ⟨map Val vs,(h2,l2)⟩ by simp
    with eq StaticCall[OF val' - path-unique path-via least - - casts - body]
        length s' Some
    show P,E ⊢ ⟨Call (Val v) Copt M es,s⟩ ⇒ ⟨e',s'⟩ by fastforce
  next
    fix s1 vs
    assume val:P,E ⊢ ⟨Val v,s''⟩ ⇒ ⟨null,s1⟩
      and evals:P,E ⊢ ⟨es',s1⟩ [⇒] ⟨map Val vs,s'⟩
      and e':e' = THROW NullPointer
    from val have val':P,E ⊢ ⟨Val v,s⟩ ⇒ ⟨null,s⟩
      and eq:s'' = s1 ∧ v = Null
      by(auto elim:eval-cases intro:Val)
    from eq IH evals have P,E ⊢ ⟨es,s⟩ [⇒] ⟨map Val vs,s'⟩ by simp
    with eq CallNull[OF val'] e' Some
    show P,E ⊢ ⟨Call (Val v) Copt M es,s⟩ ⇒ ⟨e',s'⟩
      by fastforce
  qed
 qed
next
 case (RedCall s a C S Cs M Ts' T' pns' body' Ds Ts T pns body Cs' vs
          bs new-body E s' e')
 obtain h l where s' = (h,l) by(cases s') auto
 have P,E ⊢ ⟨ref(a,Cs),s⟩ ⇒ ⟨ref(a,Cs),s⟩ by (rule eval-evals.intros)
 moreover
 have finals: finals(map Val vs) by simp
 obtain h2 l2 where s: s = (h2,l2) by (cases s)
 with finals have P,E ⊢ ⟨map Val vs,s⟩ [⇒] ⟨map Val vs,(h2,l2)⟩
   by (iprover intro: eval-finalsId)
 moreover from s have h2a:h2 a = Some (C,S) using RedCall by simp
 moreover have method: P ⊢ last Cs has least M = (Ts',T',pns',body') via Ds
by fact
 moreover have select:P ⊢ (C,Cs@pDs) selects M = (Ts,T,pns,body) via Cs'
by fact
 moreover have blocks:bs = blocks(this#pns,Class(last Cs')#Ts,Ref(a,Cs')#vs,body)
by fact
 moreover have body-case:new-body = (case T' of Class D ⇒ (⌊D⌋)bs | - ⇒ bs)
by fact
```

moreover have *same-len*$_1$: *length Ts* = *length pns*
and *this-distinct*: *this* ∉ *set pns* **and** *fv*: *fv body* ⊆ {*this*} ∪ *set pns*
using *select wf* **by** (*fastforce dest*!:*select-method-wf-mdecl simp*:*wf-mdecl-def*)+
have *same-len*: *length vs* = *length pns* **by** *fact*
moreover
obtain h_3 l_3 **where** *s'*: *s'* = (h_3,l_3) **by** (*cases s'*)
have *eval-blocks*:*P*,*E* ⊢ ⟨*new-body*,*s*⟩ ⇒ ⟨*e'*,*s'*⟩ **by** *fact*
hence *id*: l_3 = l_2 **using** *fv s s' same-len*$_1$ *same-len wf blocks body-case*
by(*cases T'*)(*auto elim*!: *eval-closed-lcl-unchanged*)
from *same-len*$_1$ **have** *same-len'*:*length*(*this*#*pns*) = *length*(*Class* (*last Cs'*)#*Ts*)

by *simp*
from *same-len*$_1$ *same-len*
have *same-len*$_2$:*length*(*this*#*pns*) = *length*(*Ref*(*a*,*Cs'*)#*vs*) **by** *simp*
from *eval-blocks*
have *eval-blocks'*:*P*,*E* ⊢ ⟨*new-body*,(h_2,l_2)⟩ ⇒ ⟨*e'*,(h_3,l_3)⟩ **using** *s s'* **by** *simp*
have *casts-unique*:⋀*vs'*. *P* ⊢ *Class* (*last Cs'*)#*Ts Casts Ref*(*a*,*Cs'*)#*vs to vs'*
⟹ *vs'* = *Ref*(*a*,*Cs'*)#*tl vs'*
using *wf*
by −(*erule Casts-to.cases*,*auto elim*!:*casts-to.cases dest*!:*mdc-eq-last*
simp:*path-via-def appendPath-def*)
have ∃ *l'' vs' new-body'*. *P*,*E*(*this*↦*Class*(*last Cs'*), *pns*[↦]*Ts*) ⊢
⟨*new-body'*,(h_2,l_2(*this* # *pns*[↦]*Ref*(*a*,*Cs'*)#*vs'*))⟩ ⇒ ⟨*e'*,(h_3, *l''*)⟩ ∧
P ⊢ *Class*(*last Cs'*)#*Ts Casts Ref*(*a*,*Cs'*)#*vs to Ref*(*a*,*Cs'*)#*vs'* ∧
length vs' = *length vs* ∧ *fv new-body'* ⊆ {*this*} ∪ *set pns* ∧
new-body' = (*case T' of Class D* ⇒ (|*D*|)*body* | - ⇒ *body*)
proof(*cases* ∀ *C*. *T'* ≠ *Class C*)
case *True*
with *same-len' same-len*$_2$ *eval-blocks' casts-unique body-case blocks*
obtain *l'' vs'*
where *body*:*P*,*E*(*this*↦*Class*(*last Cs'*), *pns*[↦]*Ts*) ⊢
⟨*body*,(h_2,l_2(*this* # *pns*[↦]*Ref*(*a*,*Cs'*)#*vs'*))⟩ ⇒ ⟨*e'*,(h_3, *l''*)⟩
and *casts*:*P* ⊢ *Class*(*last Cs'*)#*Ts Casts Ref*(*a*,*Cs'*)#*vs to Ref*(*a*,*Cs'*)#*vs'*
and *lengthvs'*:*length vs'* = *length vs*
by −(*drule-tac vs*=*Ref*(*a*,*Cs'*)#*vs* **in** *blocksEval*,*assumption*,*cases T'*,
auto simp:*length-Suc-conv*,*blast*)
with *fv True* **show** *?thesis* **by**(*cases T'*) *auto*
next
case *False*
then obtain *D* **where** *T'*:*T'* = *Class D* **by** *auto*
with *same-len' same-len*$_2$ *eval-blocks' casts-unique body-case blocks*
obtain *l'' vs'*
where *body*:*P*,*E*(*this*↦*Class*(*last Cs'*), *pns*[↦]*Ts*) ⊢
⟨(|*D*|)*body*,(h_2,l_2(*this* # *pns*[↦]*Ref*(*a*,*Cs'*)#*vs'*))⟩ ⇒
⟨*e'*,(h_3, *l''*)⟩
and *casts*:*P* ⊢ *Class*(*last Cs'*)#*Ts Casts Ref*(*a*,*Cs'*)#*vs to Ref*(*a*,*Cs'*)#*vs'*
and *lengthvs'*:*length vs'* = *length vs*
by − (*drule-tac vs*=*Ref*(*a*,*Cs'*)#*vs* **in** *CastblocksEval*,
assumption,*simp*,*clarsimp simp*:*length-Suc-conv*,*auto*)

```
  from fv have fv (⦇D⦈body) ⊆ {this} ∪ set pns
    by simp
  with body casts lengthvs' T' show ?thesis by auto
 qed
 then obtain l'' vs' new-body'
  where body:P,E(this↦Class(last Cs'), pns[↦]Ts) ⊢
          ⟨new-body',(h2,l2(this # pns[↦]Ref(a,Cs')#vs'))⟩ ⇒ ⟨e',(h3, l'')⟩
  and casts:P ⊢ Class(last Cs')#Ts Casts Ref(a,Cs')#vs to Ref(a,Cs')#vs'
  and lengthvs':length vs' = length vs
  and body-case':new-body' = (case T' of Class D ⇒ (⦇D⦈body | - ⇒ body)
  and fv':fv new-body' ⊆ {this} ∪ set pns
  by auto
 from same-len2 lengthvs'
 have same-len3:length (this # pns) = length (Ref (a, Cs') # vs') by simp
 from restrict-map-upds[OF same-len3,of set(this#pns) l2]
 have l2(this # pns[↦]Ref(a,Cs')#vs')|`(set(this#pns)) =
       [this # pns[↦]Ref(a,Cs')#vs'] by simp
 with eval-restrict-lcl[OF wf body fv'] this-distinct same-len1 same-len
 have P,E(this↦Class(last Cs'), pns[↦]Ts) ⊢
 ⟨new-body',(h2,[this # pns[↦]Ref(a,Cs')#vs'])⟩ ⇒ ⟨e',(h3, l''|`(set(this#pns)))⟩
  by simp
 with casts obtain l2' l3' vs' where
     P ⊢ Ts Casts vs to vs'
  and P,E(this↦Class(last Cs'), pns[↦]Ts) ⊢
                          ⟨new-body',(h2,l2')⟩ ⇒ ⟨e',(h3,l3')⟩
  and l2' = [this↦Ref(a,Cs'),pns[↦]vs']
  by(auto elim:Casts-to.cases)
 ultimately have P,E ⊢ ⟨(ref(a,Cs))·M(map Val vs),s⟩ ⇒ ⟨e',(h3,l2)⟩
  using body-case'
  by −(rule Call,simp-all)
 with s' id show ?case by simp
next
 case (RedStaticCall Cs C Cs'' M Ts T pns body Cs' Ds vs E a s s' e')
 have P,E ⊢ ⟨ref(a,Cs),s⟩ ⇒ ⟨ref(a,Cs),s⟩ by (rule eval-evals.intros)
 moreover
 have finals: finals(map Val vs) by simp
 obtain h2 l2 where s: s = (h2,l2) by (cases s)
 with finals have P,E ⊢ ⟨map Val vs,s⟩ [⇒] ⟨map Val vs,(h2,l2)⟩
  by (iprover intro: eval-finalsId)
 moreover have path-unique:P ⊢ Path last Cs to C unique by fact
 moreover have path-via:P ⊢ Path last Cs to C via Cs'' by fact
 moreover have least:P ⊢ C has least M = (Ts, T, pns, body) via Cs' by fact
 moreover have same-len1: length Ts = length pns
  and this-distinct: this ∉ set pns and fv: fv body ⊆ {this} ∪ set pns
  using least wf by (fastforce dest!:has-least-wf-mdecl simp:wf-mdecl-def)+
 moreover have same-len:length vs = length pns by fact
 moreover have Ds:Ds = (Cs @p Cs'') @p Cs' by fact
 moreover
 obtain h3 l3 where s': s' = (h3,l3) by (cases s')
```

have *eval-blocks*:$P,E \vdash \langle blocks(this\#pns,Class(last\ Ds)\#Ts,Ref(a,Ds)\#vs,body),s\rangle$
$\Rightarrow \langle e',s'\rangle$ **by** *fact*
hence *id*: $l_3 = l_2$ **using** *fv s s' same-len*$_1$ *same-len wf*
by(*auto elim*!: *eval-closed-lcl-unchanged*)
from *same-len*$_1$ **have** *same-len'*:$length(this\#pns) = length(Class\ (last\ Ds)\#Ts)$
by *simp*
from *same-len*$_1$ *same-len*
have *same-len*$_2$:$length(this\#pns) = length(Ref(a,Ds)\#vs)$ **by** *simp*
from *eval-blocks*
have *eval-blocks'*:$P,E \vdash \langle blocks(this\#pns,Class(last\ Ds)\#Ts,Ref(a,Ds)\#vs,body),$
$(h_2,l_2)\rangle \Rightarrow \langle e',(h_3,l_3)\rangle$ **using** *s s'* **by** *simp*
have *casts-unique*:$\bigwedge vs'.\ P \vdash Class\ (last\ Ds)\#Ts\ Casts\ Ref(a,Ds)\#vs\ to\ vs'$
$\Longrightarrow vs' = Ref(a,Ds)\#tl\ vs'$
using *wf*
by $-$(*erule Casts-to.cases*,*auto elim*!:*casts-to.cases dest*!:*mdc-eq-last*
simp:*path-via-def appendPath-def*)
from *same-len' same-len*$_2$ *eval-blocks' casts-unique*
obtain *l'' vs'* **where** *body*:$P,E(this\mapsto Class(last\ Ds),\ pns[\mapsto]Ts) \vdash$
$\langle body,(h_2,l_2(this\ \#\ pns[\mapsto]Ref(a,Ds)\#vs'))\rangle \Rightarrow \langle e',(h_3,\ l'')\rangle$
and *casts*:$P \vdash Class(last\ Ds)\#Ts\ Casts\ Ref(a,Ds)\#vs\ to\ Ref(a,Ds)\#vs'$
and *lengthvs'*:$length\ vs' = length\ vs$
by $-$(*drule-tac vs*=$Ref(a,Ds)\#vs$ **in** *blocksEval*,*auto simp*:*length-Suc-conv*,*blast*)
from *same-len*$_2$ *lengthvs'*
have *same-len*$_3$:$length\ (this\ \#\ pns) = length\ (Ref(a,Ds)\ \#\ vs')$ **by** *simp*
from *restrict-map-upds*[*OF same-len*$_3$,*of* $set(this\#pns)\ l_2$]
have $l_2(this\ \#\ pns[\mapsto]Ref(a,Ds)\#vs')|`(set(this\#pns)) =$
$[this\ \#\ pns[\mapsto]Ref(a,Ds)\#vs']$ **by** *simp*
with *eval-restrict-lcl*[*OF wf body fv*] *this-distinct same-len*$_1$ *same-len*
have $P,E(this\mapsto Class(last\ Ds),\ pns[\mapsto]Ts) \vdash$
$\langle body,(h_2,[this\#pns\ [\mapsto]\ Ref(a,Ds)\#vs'])\rangle \Rightarrow \langle e',(h_3,\ l''|`(set(this\#pns)))\rangle$
by *simp*
with *casts* **obtain** $l_2'\ l_3'\ vs'$ **where**
$P \vdash Ts\ Casts\ vs\ to\ vs'$
and $P,E(this\mapsto Class(last\ Ds),pns\ [\mapsto]\ Ts) \vdash \langle body,(h_2,l_2')\rangle \Rightarrow \langle e',(h_3,l_3')\rangle$
and $l_2' = [this \mapsto Ref(a,Ds),pns\ [\mapsto]\ vs']$
by(*auto elim*:*Casts-to.cases*)
ultimately have $P,E \vdash \langle(ref(a,Cs))\cdot(C::)M(map\ Val\ vs),s\rangle \Rightarrow \langle e',(h_3,l_2)\rangle$
by $-$(*rule StaticCall*,*simp-all*)
with *s' id* **show** *?case* **by** *simp*
next
case *RedCallNull*
thus *?case*
by (*fastforce elim*: *eval-cases intro*: *eval-evals.intros eval-finalsId*)
next
case *BlockRedNone*
thus *?case*
by (*fastforce elim*!: *eval-cases intro*: *eval-evals.intros*
simp add: *fun-upd-same fun-upd-idem*)
next

case (*BlockRedSome E V T e h l e'' h' l' v s' e'*)
have *eval*:$P,E \vdash \langle\{V{:}T{:=}Val\ v;\ e''\},(h',\ l'(V := l\ V))\rangle \Rightarrow \langle e',s'\rangle$
and *red*:$P,E(V \mapsto T) \vdash \langle e,(h,\ l(V := None))\rangle \rightarrow \langle e'',(h',\ l')\rangle$
and *notassigned*:$\neg$ *assigned V e* **and** l':$l'\ V = Some\ v$
and *IH*:$\bigwedge ex\ sx.\ P,E(V \mapsto T) \vdash \langle e'',(h',\ l')\rangle \Rightarrow \langle ex,sx\rangle \Longrightarrow$
$P,E(V \mapsto T) \vdash \langle e,(h,\ l(V := None))\rangle \Rightarrow \langle ex,sx\rangle$ **by** *fact+*
from l' **have** $l'upd$:$l'(V \mapsto v) = l'$ **by** (*rule map-upd-triv*)
from *wf red* l' **have** *casts*:$P \vdash T\ casts\ v\ to\ v$
apply −
apply(*erule-tac* $V{=}V$ **in** *None-lcl-casts-values*)
by(*simp add*:*fun-upd-same*)+
from *eval* **obtain** $h''\ l''$
where $P,E(V \mapsto T) \vdash \langle V{:=}Val\ v;;\ e'',(h',l'(V{:=}None))\rangle \Rightarrow \langle e',(h'',l'')\rangle \wedge$
$s' = (h'',l''(V{:=}l\ V))$
by (*fastforce elim*:*eval-cases simp*:*fun-upd-same fun-upd-idem*)
moreover
{ fix $T'\ h_0\ l_0\ v'\ v''$
assume *eval'*:$P,E(V \mapsto T) \vdash \langle e'',(h_0,l_0(V \mapsto v''))\rangle \Rightarrow \langle e',(h'',\ l'')\rangle$
and *val*:$P,E(V \mapsto T) \vdash \langle Val\ v,(h',\ l'(V := None))\rangle \Rightarrow \langle Val\ v',(h_0,l_0)\rangle$
and *env*:$(E(V \mapsto T))\ V = Some\ T'$ **and** *casts'*:$P \vdash T'\ casts\ v'\ to\ v''$
from *env* **have** *TeqT'*:$T = T'$ **by** (*simp add*:*fun-upd-same*)
from *val* **have** *eq*:$v = v' \wedge h' = h_0 \wedge l'(V := None) = l_0$
by −(*erule eval-cases*,*simp*)
with *casts casts' wf TeqT'* **have** $v = v''$
by *clarsimp*(*rule casts-casts-eq*)
with *eq eval'*
have $P,E(V \mapsto T) \vdash \langle e'',(h',\ l'(V \mapsto v))\rangle \Rightarrow \langle e',(h'',\ l'')\rangle$
by *clarsimp* **}**
ultimately have $P,E(V \mapsto T) \vdash \langle e'',(h',l'(V \mapsto v))\rangle \Rightarrow \langle e',(h'',l'')\rangle$
and s':$s' = (h'',l''(V{:=}l\ V))$
apply *auto*
apply(*erule eval-cases*)
apply(*erule eval-cases*) **apply** *auto*
apply(*erule eval-cases*) **apply** *auto*
apply(*erule eval-cases*) **apply** *auto*
done
with $l'upd$ **have** *eval''*:$P,E(V \mapsto T) \vdash \langle e'',(h',l')\rangle \Rightarrow \langle e',(h'',l'')\rangle$
by *simp*
from *IH*[*OF eval''*] **have** $P,E(V \mapsto T) \vdash \langle e,(h,\ l(V := None))\rangle \Rightarrow \langle e',(h'',\ l'')\rangle$
.
with s' **show** *?case* **by**(*fastforce intro*:*Block*)
next
case (*InitBlockRed E V T e h l v' e'' h' l' v'' v s' e'*)
have *eval*: $P,E \vdash \langle\{V{:}T{:=}Val\ v'';\ e''\},(h',\ l'(V := l\ V))\rangle \Rightarrow \langle e',s'\rangle$
and *red*:$P,E(V \mapsto T) \vdash \langle e,(h,\ l(V \mapsto v'))\rangle \rightarrow \langle e'',(h',\ l')\rangle$
and *casts*:$P \vdash T\ casts\ v\ to\ v'$ **and** l':$l'\ V = Some\ v''$
and *IH*:$\bigwedge ex\ sx.\ P,E(V \mapsto T) \vdash \langle e'',(h',\ l')\rangle \Rightarrow \langle ex,sx\rangle \Longrightarrow$
$P,E(V \mapsto T) \vdash \langle e,(h,\ l(V \mapsto v'))\rangle \Rightarrow \langle ex,sx\rangle$ **by** *fact+*
from l' **have** $l'upd$:$l'(V \mapsto v'') = l'$ **by** (*rule map-upd-triv*)

from *wf casts* **have** $P \vdash T$ *casts* v' *to* v' **by**(*rule casts-casts*)
with *wf red* l' **have** *casts'*:$P \vdash T$ *casts* v'' *to* v''
apply −
apply(*erule-tac* $V{=}V$ **in** *Some-lcl-casts-values*)
by(*simp add*:*fun-upd-same*)+
from *eval* **obtain** h'' l''
where $P{,}E(V \mapsto T) \vdash \langle V{:=}\mathit{Val}\ v'';;\ e'',(h',l'(V{:=}\mathit{None}))\rangle \Rightarrow \langle e',(h'',l'')\rangle \wedge$
$s' = (h'',l''(V{:=}l\ V))$
by (*fastforce elim*:*eval-cases simp*:*fun-upd-same fun-upd-idem*)
moreover
{ fix T' v'''
assume *eval'*:$P{,}E(V \mapsto T) \vdash \langle e'',(h',l'(V \mapsto v'''))\rangle \Rightarrow \langle e',(h'',\ l'')\rangle$
and *env*:$(E(V \mapsto T))\ V = \mathit{Some}\ T'$ **and** *casts''*:$P \vdash T'$ *casts* v'' *to* v'''
from *env* **have** $T = T'$ **by** (*simp add*:*fun-upd-same*)
with *casts' casts'' wf* **have** $v'' = v'''$ **by** *simp*(*rule casts-casts-eq*)
with *eval'* **have** $P{,}E(V \mapsto T) \vdash \langle e'',(h',\ l'(V \mapsto v''))\rangle \Rightarrow \langle e',(h'',\ l'')\rangle$ **by** *simp* **}**
ultimately have $P{,}E(V \mapsto T) \vdash \langle e'',(h',l'(V \mapsto v''))\rangle \Rightarrow \langle e',(h'',l'')\rangle$
and *s'*:$s' = (h'',l''(V{:=}l\ V))$
by(*auto elim*!:*eval-cases*)
with *l'upd* **have** *eval''*:$P{,}E(V \mapsto T) \vdash \langle e'',(h',l')\rangle \Rightarrow \langle e',(h'',l'')\rangle$
by *simp*
from *IH*[*OF eval''*]
have *evale*:$P{,}E(V \mapsto T) \vdash \langle e,(h,\ l(V \mapsto v'))\rangle \Rightarrow \langle e',(h'',\ l'')\rangle$ **.**
from *casts*
have $P{,}E(V \mapsto T) \vdash \langle V{:=}\mathit{Val}\ v,(h,l(V{:=}\mathit{None}))\rangle \Rightarrow \langle \mathit{Val}\ v',(h,l(V \mapsto v'))\rangle$
by −(*rule-tac* $l{=}l(V{:=}\mathit{None})$ **in** *LAss*,
auto intro:*eval-evals.intros simp*:*fun-upd-same*)
with *evale* s' **show** *?case* **by**(*fastforce intro*:*Block Seq*)
next
case (*RedBlock E V T v s s' e'*)
have $P{,}E \vdash \langle \mathit{Val}\ v,s\rangle \Rightarrow \langle e',s'\rangle$ **by** *fact*
then obtain *s'*: $s'{=}s$ **and** *e'*: $e'{=}\mathit{Val}\ v$
by *cases simp*
obtain *h l* **where** *s*: $s{=}(h,l)$ **by** (*cases s*)
have $P{,}E(V \mapsto T) \vdash \langle \mathit{Val}\ v,(h,l(V{:=}\mathit{None}))\rangle \Rightarrow \langle \mathit{Val}\ v,(h,l(V{:=}\mathit{None}))\rangle$
by (*rule eval-evals.intros*)
hence $P{,}E \vdash \langle\{V{:}T;\mathit{Val}\ v\},(h,l)\rangle \Rightarrow \langle \mathit{Val}\ v,(h,(l(V{:=}\mathit{None}))(V{:=}l\ V))\rangle$
by (*rule eval-evals.Block*)
thus $P{,}E \vdash \langle\{V{:}T;\ \mathit{Val}\ v\},s\rangle \Rightarrow \langle e',s'\rangle$
using *s s' e'*
by *simp*
next
case (*RedInitBlock T v v' E V u s s' e'*)
have $P{,}E \vdash \langle \mathit{Val}\ u,s\rangle \Rightarrow \langle e',s'\rangle$ **and** *casts*:$P \vdash T$ *casts* v *to* v' **by** *fact*+
then obtain *s'*: $s' = s$ **and** *e'*: $e'{=}\mathit{Val}\ u$ **by** *cases simp*
obtain *h l* **where** *s*: $s{=}(h,l)$ **by** (*cases s*)
have *val*:$P{,}E(V \mapsto T) \vdash \langle \mathit{Val}\ v,(h,l(V{:=}\mathit{None}))\rangle \Rightarrow \langle \mathit{Val}\ v,(h,l(V{:=}\mathit{None}))\rangle$
by (*rule eval-evals.intros*)

```
  with casts
  have P,E(V ↦ T) ⊢ ⟨V:=Val v,(h,l(V:=None))⟩ ⇒ ⟨Val v',(h,l(V ↦ v'))⟩
    by −(rule-tac l=l(V:=None) in LAss,auto simp:fun-upd-same)
  hence P,E ⊢ ⟨{V:T := Val v; Val u},(h,l)⟩ ⇒ ⟨Val u,(h, (l(V↦v'))(V:=l V))⟩
    by (fastforce intro!: eval-evals.intros)
  thus ?case using s s' e' by simp
next
  case SeqRed thus ?case by (fastforce elim: eval-cases intro: eval-evals.intros)
next
  case RedSeq thus ?case by (fastforce elim: eval-cases intro: eval-evals.intros)
next
  case CondRed thus ?case by (fastforce elim: eval-cases intro: eval-evals.intros)
next
  case RedCondT thus ?case by (fastforce elim: eval-cases intro: eval-evals.intros)
next
  case RedCondF thus ?case by (fastforce elim: eval-cases intro: eval-evals.intros)
next
  case RedWhile
 thus ?case by (auto simp add: unfold-while intro:eval-evals.intros elim:eval-cases)
next
  case ThrowRed thus ?case by (fastforce elim: eval-cases intro: eval-evals.intros)
next
  case RedThrowNull
  thus ?case by −(auto elim!:eval-cases intro!:eval-evals.ThrowNull eval-finalId)
next
  case ListRed1 thus ?case by (fastforce elim: evals-cases intro: eval-evals.intros)
next
  case ListRed2
  thus ?case by (fastforce elim!: evals-cases eval-cases
                       intro: eval-evals.intros eval-finalId)
next
  case StaticCastThrow
  thus ?case by (fastforce elim: eval-cases intro: eval-evals.intros)
next
  case DynCastThrow
  thus ?case by (fastforce elim: eval-cases intro: eval-evals.intros)
next
 case BinOpThrow1 thus ?case by (fastforce elim: eval-cases intro: eval-evals.intros)
next
 case BinOpThrow2 thus ?case by (fastforce elim: eval-cases intro: eval-evals.intros)
next
 case LAssThrow thus ?case by (fastforce elim: eval-cases intro: eval-evals.intros)
next
 case FAccThrow thus ?case by (fastforce elim: eval-cases intro: eval-evals.intros)
next
 case FAssThrow1 thus ?case by (fastforce elim: eval-cases intro: eval-evals.intros)
next
 case FAssThrow2 thus ?case by (fastforce elim: eval-cases intro: eval-evals.intros)
next
```

case *CallThrowObj* **thus** *?case* **by** (*fastforce elim*: *eval-cases intro*: *eval-evals.intros*)
next
case (*CallThrowParams es vs r es' E v Copt M s s' e'*)
have $P,E \vdash \langle Val\ v,s\rangle \Rightarrow \langle Val\ v,s\rangle$ **by** (*rule eval-evals.intros*)
moreover
have *es*: $es = map\ Val\ vs\ @\ Throw\ r\ \#\ es'$ **by** *fact*
have *eval-e*: $P,E \vdash \langle Throw\ r,s\rangle \Rightarrow \langle e',s'\rangle$ **by** *fact*
then obtain s': $s' = s$ **and** e': $e' = Throw\ r$
by *cases* (*auto elim*!:*eval-cases*)
with *list-eval-Throw* [*OF eval-e*] *es*
have $P,E \vdash \langle es,s\rangle\ [\Rightarrow]\ \langle map\ Val\ vs\ @\ Throw\ r\ \#\ es',s'\rangle$ **by** *simp*
ultimately have $P,E \vdash \langle Call\ (Val\ v)\ Copt\ M\ es,s\rangle \Rightarrow \langle Throw\ r,s'\rangle$
by (*rule eval-evals.CallParamsThrow*)
thus *?case* **using** e' **by** *simp*
next
case (*BlockThrow E V T r s s' e'*)
have $P,E \vdash \langle Throw\ r,\ s\rangle \Rightarrow \langle e',s'\rangle$ **by** *fact*
then obtain s': $s' = s$ **and** e': $e' = Throw\ r$
by *cases* (*auto elim*!:*eval-cases*)
obtain *h l* **where** *s*: $s=(h,l)$ **by** (*cases s*)
have $P,E(V \mapsto T) \vdash \langle Throw\ r,\ (h,l(V:=None))\rangle \Rightarrow \langle Throw\ r,\ (h,l(V:=None))\rangle$
by (*simp add*:*eval-evals.intros eval-finalId*)
hence $P,E \vdash \langle\{V:T;Throw\ r\},(h,l)\rangle \Rightarrow \langle Throw\ r,\ (h,(l(V:=None))(V:=l\ V))\rangle$
by (*rule eval-evals.Block*)
thus $P,E \vdash \langle\{V:T;\ Throw\ r\},s\rangle \Rightarrow \langle e',s'\rangle$ **using** *s s' e'* **by** *simp*
next
case (*InitBlockThrow T v v' E V r s s' e'*)
have $P,E \vdash \langle Throw\ r,s\rangle \Rightarrow \langle e',s'\rangle$ **and** *casts*:$P \vdash T\ casts\ v\ to\ v'$ **by** *fact+*
then obtain s': $s' = s$ **and** e': $e' = Throw\ r$
by *cases* (*auto elim*!:*eval-cases*)
obtain *h l* **where** *s*: $s = (h,l)$ **by** (*cases s*)
have $P,E(V \mapsto T) \vdash \langle Val\ v,(h,l(V:=None))\rangle \Rightarrow \langle Val\ v,(h,l(V:=None))\rangle$
by (*rule eval-evals.intros*)
with *casts*
have $P,E(V \mapsto T) \vdash \langle V:=Val\ v,(h,l(V := None))\rangle \Rightarrow \langle Val\ v',(h,l(V \mapsto v'))\rangle$
by −(*rule-tac l*=$l(V:=None)$ **in** *LAss*,*auto simp*:*fun-upd-same*)
hence $P,E \vdash \langle\{V:T := Val\ v;\ Throw\ r\},(h,l)\rangle \Rightarrow \langle Throw\ r,\ (h,\ (l(V\mapsto v'))(V:=l\ V))\rangle$
by(*fastforce intro*:*eval-evals.intros*)
thus $P,E \vdash \langle\{V:T := Val\ v;\ Throw\ r\},s\rangle \Rightarrow \langle e',s'\rangle$ **using** *s s' e'* **by** *simp*
next
case *SeqThrow* **thus** *?case* **by** (*fastforce elim*: *eval-cases intro*: *eval-evals.intros*)
next
case *CondThrow* **thus** *?case* **by** (*fastforce elim*: *eval-cases intro*: *eval-evals.intros*)
next
case *ThrowThrow* **thus** *?case* **by** (*fastforce elim*: *eval-cases intro*: *eval-evals.intros*)
qed

declare *split-paired-All* [*simp*] *split-paired-Ex* [*simp*]
setup ‹*map-theory-claset* (*fn ctxt => ctxt addSbefore* (*split-all-tac*, *split-all-tac*))›
setup ‹*map-theory-simpset* (*fn ctxt => ctxt addloop* (*split-all-tac*, *split-all-tac*))›

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 $P,E \vdash \langle e,s\rangle \Rightarrow \langle e',s'\rangle$

using *reds eval-rest*
apply (*induct rule*: *converse-rtrancl-induct2*)
apply *simp*
apply *simp*
apply (*rule extend-1-eval*)
apply (*rule wf*)
apply *assumption+*
done

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

using *reds eval-rest*
apply (*induct rule*: *converse-rtrancl-induct2*)
apply *simp*
apply *simp*
apply (*rule extend-1-evals*)
apply (*rule wf*)
apply *assumption+*
done

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 −
note *wf*
moreover assume $P,E \vdash \langle e,s\rangle \rightarrow* \langle e',s'\rangle$
moreover assume *final e'*
then have $P,E \vdash \langle e',s'\rangle \Rightarrow \langle e',s'\rangle$
by (*rule eval-finalId*)
ultimately show $P,E \vdash \langle e,s\rangle \Rightarrow \langle e',s'\rangle$

by (*rule extend-eval*)
next
note *wf*
moreover assume $P,E \vdash \langle es,s\rangle\ [\rightarrow]*\ \langle es',s'\rangle$
moreover assume *finals es'*
then have $P,E \vdash \langle es',s'\rangle\ [\Rightarrow]\ \langle es',s'\rangle$
by (*rule eval-finalsId*)
ultimately show $P,E \vdash \langle es,s\rangle\ [\Rightarrow]\ \langle es',s'\rangle$
by (*rule extend-evals*)
qed

19.18 Equivalence

And now, the crowning achievement:

corollary *big-iff-small*:
wwf-prog P $\Longrightarrow$
$P,E \vdash \langle e,s\rangle \Rightarrow \langle e',s'\rangle\ =\ (P,E \vdash \langle e,s\rangle \rightarrow* \langle e',s'\rangle \wedge$ *final* $e')$
by(*blast dest*: *big-by-small eval-final small-by-big*)

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**
$A \sqcup B \equiv$ *case A of None* ⇒ *None*
| ⌊A⌋ ⇒ (*case B of None* ⇒ *None* | ⌊B⌋ ⇒ ⌊$A \cup B$⌋)

definition *hyperInt* :: *hyperset* ⇒ *hyperset* ⇒ *hyperset* (**infixl** ⊓ *70*) **where**
$A \sqcap B \equiv$ *case A of None* ⇒ *B*
| ⌊A⌋ ⇒ (*case B of None* ⇒ ⌊A⌋ | ⌊B⌋ ⇒ ⌊$A \cap B$⌋)

definition *hyperDiff1* :: *hyperset* ⇒ *vname* ⇒ *hyperset* (**infixl** ⊖ *65*) **where**
$A \ominus a \equiv$ *case A of None* ⇒ *None* | ⌊A⌋ ⇒ ⌊$A - \{a\}$⌋

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

definition *hyper-subset* :: *hyperset* ⇒ *hyperset* ⇒ *bool* (**infix** ⊑ *50*) **where**
$A \sqsubseteq B \equiv$ *case B of None* ⇒ *True*

$$\mid \lfloor B \rfloor \Rightarrow (\mathit{case}\ A\ \mathit{of}\ \mathit{None} \Rightarrow \mathit{False} \mid \lfloor A \rfloor \Rightarrow A \subseteq B)$$

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

lemma [*simp*]: $\lfloor \{\} \rfloor \sqcup A = A \ \wedge \ A \sqcup \lfloor \{\} \rfloor = A$
by(*simp add*:*hyperset-defs*)

lemma [*simp*]: $\lfloor A \rfloor \sqcup \lfloor B \rfloor = \lfloor A \cup B \rfloor \wedge \lfloor A \rfloor \ominus a = \lfloor A - \{a\} \rfloor$
by(*simp add*:*hyperset-defs*)

lemma [*simp*]: $\mathit{None} \sqcup A = \mathit{None} \wedge A \sqcup \mathit{None} = \mathit{None}$
by(*simp add*:*hyperset-defs*)

lemma [*simp*]: $a \in\in \mathit{None} \wedge \mathit{None} \ominus a = \mathit{None}$
by(*simp add*:*hyperset-defs*)

lemma *hyperUn-assoc*: $(A \sqcup B) \sqcup C = A \sqcup (B \sqcup C)$
by(*simp add*:*hyperset-defs Un-assoc*)

lemma *hyper-insert-comm*: $A \sqcup \lfloor \{a\} \rfloor = \lfloor \{a\} \rfloor \sqcup A \wedge A \sqcup (\lfloor \{a\} \rfloor \sqcup B) = \lfloor \{a\} \rfloor \sqcup (A \sqcup B)$
by(*simp add*:*hyperset-defs*)

20.2 Definite assignment

primrec $\mathcal{A} :: \mathit{expr} \Rightarrow \mathit{hyperset}$ **and** $\mathcal{A}s :: \mathit{expr\ list} \Rightarrow \mathit{hyperset}$ **where**
$\mathcal{A}\ (\mathit{new}\ C) = \lfloor \{\} \rfloor \mid$
$\mathcal{A}\ (\mathit{Cast}\ C\ e) = \mathcal{A}\ e \mid$
$\mathcal{A}\ (\langle\!|C|\!\rangle e) = \mathcal{A}\ e \mid$
$\mathcal{A}\ (\mathit{Val}\ v) = \lfloor \{\} \rfloor \mid$
$\mathcal{A}\ (e_1 \ll \mathit{bop} \gg e_2) = \mathcal{A}\ e_1 \sqcup \mathcal{A}\ e_2 \mid$
$\mathcal{A}\ (\mathit{Var}\ V) = \lfloor \{\} \rfloor \mid$
$\mathcal{A}\ (\mathit{LAss}\ V\ e) = \lfloor \{V\} \rfloor \sqcup \mathcal{A}\ e \mid$
$\mathcal{A}\ (e{\cdot}F\{Cs\}) = \mathcal{A}\ e \mid$
$\mathcal{A}\ (e_1{\cdot}F\{Cs\}{:=}e_2) = \mathcal{A}\ e_1 \sqcup \mathcal{A}\ e_2 \mid$
$\mathcal{A}\ (\mathit{Call}\ e\ \mathit{Copt}\ M\ es) = \mathcal{A}\ e \sqcup \mathcal{A}s\ es \mid$
$\mathcal{A}\ (\{V{:}T;\ e\}) = \mathcal{A}\ e \ominus V \mid$
$\mathcal{A}\ (e_1;;e_2) = \mathcal{A}\ e_1 \sqcup \mathcal{A}\ e_2 \mid$
$\mathcal{A}\ (\mathit{if}\ (e)\ e_1\ \mathit{else}\ e_2) = \ \mathcal{A}\ e \sqcup (\mathcal{A}\ e_1 \sqcap \mathcal{A}\ e_2) \mid$
$\mathcal{A}\ (\mathit{while}\ (b)\ e) = \mathcal{A}\ b \mid$
$\mathcal{A}\ (\mathit{throw}\ e) = \mathit{None} \mid$

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

primrec $\mathcal{D} :: \mathit{expr} \Rightarrow \mathit{hyperset} \Rightarrow \mathit{bool}$ **and** $\mathcal{D}s :: \mathit{expr\ list} \Rightarrow \mathit{hyperset} \Rightarrow \mathit{bool}$ **where**
$\mathcal{D}\ (\mathit{new}\ C)\ A = \mathit{True} \mid$

$\mathcal{D}\ (Cast\ C\ e)\ A = \mathcal{D}\ e\ A\ |$
$\mathcal{D}\ (\llparenthesis C \rrparenthesis e)\ A = \mathcal{D}\ e\ A\ |$
$\mathcal{D}\ (Val\ v)\ A = True\ |$
$\mathcal{D}\ (e_1 \ll bop \gg e_2)\ A = (\mathcal{D}\ e_1\ A \wedge \mathcal{D}\ e_2\ (A \sqcup \mathcal{A}\ e_1))\ |$
$\mathcal{D}\ (Var\ V)\ A = (V \in\in A)\ |$
$\mathcal{D}\ (LAss\ V\ e)\ A = \mathcal{D}\ e\ A\ |$
$\mathcal{D}\ (e{\cdot}F\{Cs\})\ A = \mathcal{D}\ e\ A\ |$
$\mathcal{D}\ (e_1{\cdot}F\{Cs\}{:=}e_2)\ A = (\mathcal{D}\ e_1\ A \wedge \mathcal{D}\ e_2\ (A \sqcup \mathcal{A}\ e_1))\ |$
$\mathcal{D}\ (Call\ e\ Copt\ M\ es)\ A = (\mathcal{D}\ e\ A \wedge \mathcal{D}s\ es\ (A \sqcup \mathcal{A}\ e))\ |$
$\mathcal{D}\ (\{V{:}T;\ e\})\ A = \mathcal{D}\ e\ (A \ominus V)\ |$
$\mathcal{D}\ (e_1;;e_2)\ A = (\mathcal{D}\ e_1\ A \wedge \mathcal{D}\ e_2\ (A \sqcup \mathcal{A}\ e_1))\ |$
$\mathcal{D}\ (if\ (e)\ e_1\ else\ e_2)\ A =$
$\quad(\mathcal{D}\ e\ A \wedge \mathcal{D}\ e_1\ (A \sqcup \mathcal{A}\ e) \wedge \mathcal{D}\ e_2\ (A \sqcup \mathcal{A}\ e))\ |$
$\mathcal{D}\ (while\ (e)\ c)\ A = (\mathcal{D}\ e\ A \wedge \mathcal{D}\ c\ (A \sqcup \mathcal{A}\ e))\ |$
$\mathcal{D}\ (throw\ e)\ A = \mathcal{D}\ e\ A\ |$

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

lemma *As-map-Val*[*simp*]: $\mathcal{A}s\ (map\ Val\ vs) = \lfloor\{\}\rfloor$
by (*induct vs*) *simp-all*

lemma *D-append*[*iff*]: $\bigwedge A.\ \mathcal{D}s\ (es\ @\ es')\ A = (\mathcal{D}s\ es\ A \wedge \mathcal{D}s\ es'\ (A \sqcup \mathcal{A}s\ es))$
by (*induct es type*:*list*) (*auto simp*:*hyperUn-assoc*)

lemma *A-fv*: $\bigwedge A.\ \mathcal{A}\ e = \lfloor A \rfloor \Longrightarrow A \subseteq fv\ e$
and $\bigwedge A.\ \mathcal{A}s\ es = \lfloor A \rfloor \Longrightarrow A \subseteq fvs\ es$

apply(*induct e* **and** *es rule*: $\mathcal{A}$.*induct* $\mathcal{A}s$.*induct*)
apply (*simp-all add*:*hyperset-defs*)
apply *blast*+
done

lemma *sqUn-lem*: $A \sqsubseteq A' \Longrightarrow A \sqcup B \sqsubseteq A' \sqcup B$
by(*simp add*:*hyperset-defs*) *blast*

lemma *diff-lem*: $A \sqsubseteq A' \Longrightarrow A \ominus b \sqsubseteq A' \ominus b$
by(*simp add*:*hyperset-defs*) *blast*

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

apply(*induct e* **and** *es rule*: $\mathcal{D}$.*induct* $\mathcal{D}s$.*induct*)
apply *simp*
apply *simp*

apply *simp*
apply *simp*
apply *simp* **apply** (*iprover dest*:*sqUn-lem*)
apply (*fastforce simp add*:*hyperset-defs*)
apply *simp*
apply *simp*
apply *simp* **apply** (*iprover dest*:*sqUn-lem*)
apply *simp* **apply** (*iprover dest*:*sqUn-lem*)
apply *simp* **apply** (*iprover dest*:*diff-lem*)
apply *simp* **apply** (*iprover dest*:*sqUn-lem*)
apply *simp* **apply** (*iprover dest*:*sqUn-lem*)
apply *simp* **apply** (*iprover dest*:*sqUn-lem*)
apply *simp*
apply *simp*
apply *simp*
apply (*iprover dest*:*sqUn-lem*)
done

lemma *D-mono'*: $\mathcal{D}\ e\ A \Longrightarrow A \sqsubseteq A' \Longrightarrow \mathcal{D}\ e\ A'$
and *Ds-mono'*: $\mathcal{D}s\ es\ A \Longrightarrow A \sqsubseteq A' \Longrightarrow \mathcal{D}s\ es\ A'$
by(*blast intro*:*D-mono*, *blast intro*:*Ds-mono*)

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**
  $P \vdash typeof_h\ Unit = Some\ Void$
| $P \vdash typeof_h\ Null = Some\ NT$
| $P \vdash typeof_h\ (Bool\ b) = Some\ Boolean$
| $P \vdash typeof_h\ (Intg\ i) = Some\ Integer$
| $P \vdash typeof_h\ (Ref\ r) = (case\ h\ (the\text{-}addr\ (Ref\ r))\ of\ None \Rightarrow None$
  $\mid Some(C,S) \Rightarrow (if\ Subobjs\ P\ C\ (the\text{-}path(Ref\ r))\ then$
  $Some(Class(last(the\text{-}path(Ref\ r))))$
  $else\ None))$

lemma *type-eq-type*: *typeof v* = *Some T* ⟹ $P \vdash typeof_h\ v = Some\ T$
by(*induct v*)*auto*

lemma *typeof-Void* [*simp*]: $P \vdash typeof_h\ v = Some\ Void \Longrightarrow v = Unit$
by(*induct v*,*auto split*:*if-split-asm*)

lemma *typeof-NT* [*simp*]: $P \vdash typeof_h\ v = Some\ NT \Longrightarrow v = Null$
by(*induct v*,*auto split*:*if-split-asm*)

lemma *typeof-Boolean* [*simp*]: $P \vdash typeof_h\ v = Some\ Boolean \Longrightarrow \exists\, b.\ v = Bool\ b$
by(*induct v*,*auto split*:*if-split-asm*)

lemma *typeof-Integer* [*simp*]: $P \vdash typeof_h\ v = Some\ Integer \Longrightarrow \exists\, i.\ v = Intg\ i$
by(*induct v*,*auto split*:*if-split-asm*)

lemma *typeof-Class-Subo*:
$P \vdash typeof_h\ v = Some\ (Class\ C) \Longrightarrow$
$\exists\, a\ Cs\ D\ S.\ v = Ref(a,Cs) \wedge h\ a = Some(D,S) \wedge Subobjs\ P\ D\ Cs \wedge last\ Cs = C$
by(*induct v*,*auto split*:*if-split-asm*)

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* ⟧

$\Longrightarrow P,E,h \vdash e_1 \ll bop \gg e_2 : T$

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

| *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*:
$[\![P,E,h \vdash e : Boolean;\ \ P,E,h \vdash c : T]\!]$

$\Longrightarrow$ $P,E,h \vdash \mathit{while}(e)\ c : \mathit{Void}$

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

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

lemma [*simp*]: $\forall\, Ts.\ (P,E,h \vdash es_1\ @\ es_2\ [:]\ Ts) =$
$(\exists\, Ts_1\ Ts_2.\ Ts = Ts_1\ @\ Ts_2 \wedge P,E,h \vdash es_1\ [:]\ Ts_1\ \&\ P,E,h \vdash es_2\ [:]\ Ts_2)$

apply(*induct-tac* es_1)
apply *simp*
apply *clarsimp*
apply(*erule thin-rl*)
apply (*rule iffI*)
apply *clarsimp*
apply(*rule exI*)+

apply(*rule conjI*)
prefer *2* **apply** *blast*
apply *simp*
apply *fastforce*
done

inductive-cases *WTrt-elim-cases*[*elim*!]:
$P,E,h \vdash new\ C : T$
$P,E,h \vdash Cast\ C\ e : T$
$P,E,h \vdash (\!|C|\!)e : T$
$P,E,h \vdash e_1 \ll bop \gg e_2 : T$
$P,E,h \vdash V{:=}e : T$
$P,E,h \vdash e{\cdot}F\{Cs\} : T$
$P,E,h \vdash e{\cdot}F\{Cs\} := v : T$
$P,E,h \vdash e{\cdot}M(es) : T$
$P,E,h \vdash e{\cdot}(C::)M(es) : T$
$P,E,h \vdash if\ (e)\ e_1\ else\ e_2 : T$
$P,E,h \vdash while(e)\ c : T$
$P,E,h \vdash throw\ e : T$

21.4 Some interesting lemmas

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

apply(*induct vs*)
apply *fastforce*
apply(*case-tac Ts*)
apply *simp*
apply *simp*
done

lemma *WTrts-same-length*: $\bigwedge Ts.\ P,E,h \vdash es\ [:]\ Ts \Longrightarrow length\ es = length\ Ts$
by(*induct es type*:*list*)*auto*

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**
$P,E,h \vdash es\ [:]\ Ts \Longrightarrow (\bigwedge E'.\ E \subseteq_m E' \Longrightarrow P,E',h \vdash es\ [:]\ Ts)$

apply(*induct rule*: *WTrt-inducts*)
apply(*simp add*: *WTrtNew*)
apply(*fastforce simp*: *WTrtDynCast*)
apply(*fastforce simp*: *WTrtStaticCast*)

apply(*fastforce simp*: *WTrtVal*)
apply(*simp add*: *WTrtVar map-le-def dom-def*)
apply(*fastforce simp add*: *WTrtBinOp*)
apply (*force simp*:*map-le-def*)
apply(*fastforce simp*: *WTrtFAcc*)
apply(*simp add*: *WTrtFAccNT*)
apply(*fastforce simp*: *WTrtFAss*)
apply(*fastforce simp*: *WTrtFAssNT*)
apply(*fastforce simp*: *WTrtCall*)
apply(*fastforce simp*: *WTrtStaticCall*)
apply(*fastforce simp*: *WTrtCallNT*)
apply(*fastforce simp*: *map-le-def*)
apply(*fastforce*)
apply(*fastforce simp*: *WTrtCond*)
apply(*fastforce simp*: *WTrtWhile*)
apply(*fastforce simp*: *WTrtThrow*)
apply(*simp add*: *WTrtNil*)
apply(*simp add*: *WTrtCons*)
done

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(*induct rule*: *WT-WTs-inducts*)
  case *WTVal* **thus** *?case* **by** (*fastforce dest*:*type-eq-type*)
next
  case *WTBinOp* **thus** *?case* **by** (*fastforce split*:*bop.splits*)
next
  case *WTFAcc* **thus** *?case*
    by(*fastforce intro*!:*WTrtFAcc dest*:*Subobjs-nonempty*
                simp:*LeastFieldDecl-def FieldDecls-def*)
next
  case *WTFAss* **thus** *?case*
    by(*fastforce intro*!:*WTrtFAss dest*:*Subobjs-nonempty*
                simp:*LeastFieldDecl-def FieldDecls-def*)
next
  case *WTCall* **thus** *?case* **by** (*fastforce intro*:*WTrtCall*)
qed (*auto simp del*:*fun-upd-apply*)

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* ⊢ *v* :≤ *Void* = (*P* ⊢ *typeof*$_h$ *v* = *Some Void*)
| *P*,*h* ⊢ *v* :≤ *Boolean* = (*P* ⊢ *typeof*$_h$ *v* = *Some Boolean*)
| *P*,*h* ⊢ *v* :≤ *Integer* = (*P* ⊢ *typeof*$_h$ *v* = *Some Integer*)
| *P*,*h* ⊢ *v* :≤ *NT* = (*P* ⊢ *typeof*$_h$ *v* = *Some NT*)
| *P*,*h* ⊢ *v* :≤ (*Class C*) = (*P* ⊢ *typeof*$_h$ *v* = *Some*(*Class C*) ∨ *P* ⊢ *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 (:≤)$_w$ 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*
by(*cases T*) *simp-all*

lemma *typeof-conf*[*simp*]: *P* ⊢ *typeof*$_h$ *v* = *Some T* ⟹ *P*,*h* ⊢ *v* :≤ *T*
by (*cases T*) *auto*

lemma *typeof-lit-conf*[*simp*]: *typeof v* = *Some T* ⟹ *P*,*h* ⊢ *v* :≤ *T*
by (*rule typeof-conf*[*OF type-eq-type*])

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

by(*cases T*) *auto*

lemma *typeof-notclass-heap*:
$\forall C.\ T \neq Class\ C \Longrightarrow (P \vdash typeof_h\ v = Some\ T) = (P \vdash typeof_{h'}\ v = Some\ T)$
by(*cases T*)(*auto dest*:*typeof-Void typeof-NT typeof-Boolean typeof-Integer*)

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(*cases T*)
case *Void*
hence $(P \vdash typeof_{h(a{\mapsto}(C,S'))}\ v = Some\ T) = (P \vdash typeof_h\ v = Some\ T)$
by(*fastforce intro*!:*typeof-notclass-heap*)
with *Void* **show** *?thesis* **by** *simp*
next
case *Boolean*
hence $(P \vdash typeof_{h(a{\mapsto}(C,S'))}\ v = Some\ T) = (P \vdash typeof_h\ v = Some\ T)$
by(*fastforce intro*!:*typeof-notclass-heap*)
with *Boolean* **show** *?thesis* **by** *simp*
next
case *Integer*
hence $(P \vdash typeof_{h(a{\mapsto}(C,S'))}\ v = Some\ T) = (P \vdash typeof_h\ v = Some\ T)$
by(*fastforce intro*!:*typeof-notclass-heap*)
with *Integer* **show** *?thesis* **by** *simp*
next
case *NT*
hence $(P \vdash typeof_{h(a{\mapsto}(C,S'))}\ v = Some\ T) = (P \vdash typeof_h\ v = Some\ T)$
by(*fastforce intro*!:*typeof-notclass-heap*)
with *NT* **show** *?thesis* **by** *simp*
next
case (*Class C'*)
{ assume $P \vdash typeof_{h(a \mapsto (C,\ S'))}\ v = Some(Class\ C')$
with *h* **have** $P \vdash typeof_h\ v = Some(Class\ C')$
by (*cases v*) (*auto split*:*if-split-asm*) **}**
hence *1*:$P \vdash typeof_{h(a \mapsto (C,\ S'))}\ v = Some(Class\ C') \Longrightarrow$
$P \vdash typeof_h\ v = Some(Class\ C')$ **by** *simp*
{ assume *type*:$P \vdash typeof_{h(a \mapsto (C,\ S'))}\ v = Some\ NT$
and *typenot*:$P \vdash typeof_h\ v \neq Some\ NT$
have $\forall C.\ NT \neq Class\ C$ **by** *simp*
with *type* **have** $P \vdash typeof_h\ v = Some\ NT$ **by**(*fastforce dest*:*typeof-notclass-heap*)
with *typenot* **have** $P \vdash typeof_h\ v = Some(Class\ C')$ **by** *simp* **}**
hence *2*:$[\![P \vdash typeof_{h(a \mapsto (C,\ S'))}\ v = Some\ NT;\ P \vdash typeof_h\ v \neq Some\ NT]\!]$
$\Longrightarrow$
$P \vdash typeof_h\ v = Some(Class\ C')$ **by** *simp*
{ assume $P \vdash typeof_h\ v = Some(Class\ C')$
with *h* **have** $P \vdash typeof_{h(a \mapsto (C,\ S'))}\ v = Some(Class\ C')$

by (*cases v*) (*auto split*:*if-split-asm*) **}**
hence *3*:$P \vdash typeof_h\ v = Some(Class\ C') \Longrightarrow$
$P \vdash typeof_{h(a \mapsto (C,\ S'))}\ v = Some(Class\ C')$ **by** *simp*
{ assume *typenot*:$P \vdash typeof_{h(a \mapsto (C,\ S'))}\ v \neq Some\ NT$
and *type*:$P \vdash typeof_h\ v = Some\ NT$
have $\forall\ C.\ NT \neq Class\ C$ **by** *simp*
with *type* **have** $P \vdash typeof_{h(a \mapsto (C,\ S'))}\ v = Some\ NT$
by(*fastforce dest*:*typeof-notclass-heap*)
with *typenot* **have** $P \vdash typeof_{h(a \mapsto (C,\ S'))}\ v = Some(Class\ C')$ **by** *simp* **}**
hence *4*:$[\![P \vdash typeof_{h(a \mapsto (C,\ S'))}\ v \neq Some\ NT;\ P \vdash typeof_h\ v = Some\ NT]\!]$
$\Longrightarrow$
$P \vdash typeof_{h(a \mapsto (C,\ S'))}\ v = Some(Class\ C')$ **by** *simp*
from *Class* **show** *?thesis* **by** (*auto intro*:*1 2 3 4*)
qed

lemma *conf-NT* [*iff*]: $P,h \vdash v :\leq NT = (v = Null)$
by *fastforce*

22.2 Value list conformance [:≤]

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

apply *rule*
apply (*rule subst* [*OF list-all2-rev*])
apply *simp*
apply (*rule subst* [*OF list-all2-rev*])
apply *simp*
done

lemma *confs-Cons2*: $P,h \vdash xs\ [:\leq]\ y\#ys = (\exists\ z\ zs.\ xs = z\#zs \wedge P,h \vdash z :\leq y \wedge P,h \vdash zs\ [:\leq]\ ys)$
by (*rule list-all2-Cons2*)

22.3 Field conformance (:≤)

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

apply(*unfold fconf-def init-class-fieldmap-def*)
apply *clarsimp*
apply (*rule exI*)
apply (*rule conjI*)
apply (*simp add*:*map-of-map*)
apply(*case-tac T*)
apply *simp-all*
done

22.4 Heap conformance

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

apply (*unfold hconf-def*)
apply (*fast*)
done

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

apply (*unfold hconf-def*)
apply *clarsimp*
apply (*erule-tac x=a* **in** *allE*)
apply (*erule-tac x=C* **in** *allE*)
apply (*erule-tac x=S* **in** *allE*)
apply *clarsimp*
apply (*unfold oconf-def*)
apply *fastforce*
done

22.5 Local variable conformance

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

apply (*unfold lconf-def*)
apply *auto*
done

lemma *lconf-empty*[*iff*]: $P,h \vdash Map.empty\ (:\leq)_w\ E$
by(*simp add*:*lconf-def*)

lemma *lconf-upd2*: $[\![P,h \vdash l\ (:\leq)_w\ E; P,h \vdash v :\leq T]\!] \Longrightarrow P,h \vdash l(V \mapsto v)\ (:\leq)_w\ E(V \mapsto T)$
by(*simp add*:*lconf-def*)

22.6 Environment conformance

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

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 ⊢ e :_NT Void    ⟷ (P,E,h ⊢ e : Void)
  | type-conf-Boolean: P,E,h ⊢ e :_NT Boolean ⟷ (P,E,h ⊢ e : Boolean)
  | type-conf-Integer: P,E,h ⊢ e :_NT Integer ⟷ (P,E,h ⊢ e : Integer)
  | type-conf-NT:      P,E,h ⊢ e :_NT NT      ⟷ (P,E,h ⊢ e : NT)
  | type-conf-Class:   P,E,h ⊢ e :_NT Class C ⟷
                         (P,E,h ⊢ e : Class C ∨ P,E,h ⊢ e : NT)

fun
  types-conf :: prog ⇒ env ⇒ heap ⇒ expr list ⇒ ty list ⇒ bool
    (-,-,- ⊢ - [:]_NT -   [51,51,51]50)
where
  P,E,h ⊢ [] [:]_NT [] ⟷ True
  | P,E,h ⊢ (e#es) [:]_NT (T#Ts) ⟷
      (P,E,h ⊢ e:_NT T ∧ P,E,h ⊢ es [:]_NT Ts)
  | P,E,h ⊢ es [:]_NT Ts ⟷ False

lemma wt-same-type-typeconf:
P,E,h ⊢ e : T ⟹ P,E,h ⊢ e :_NT T
by(cases T) auto

lemma wts-same-types-typesconf:
  P,E,h ⊢ es [:] Ts ⟹ types-conf P E h es Ts
proof(induct Ts arbitrary: es)
  case Nil thus ?case by (auto elim:WTrts.cases)
next
  case (Cons T' Ts')
  have wtes:P,E,h ⊢ es [:] T'#Ts'
    and IH:⋀es. P,E,h ⊢ es [:] Ts' ⟹ types-conf P E h es Ts' by fact+
  from wtes obtain e' es' where es:es = e'#es' by(cases es) auto
  with wtes have wte':P,E,h ⊢ e' : T' and wtes':P,E,h ⊢ es' [:] Ts'
    by simp-all
 from IH[OF wtes'] wte' es show ?case by (fastforce intro:wt-same-type-typeconf)
qed



lemma types-conf-smaller-types:
⋀es Ts. ⟦length es = length Ts'; types-conf P E h es Ts'; P ⊢ Ts' [≤] Ts ⟧
  ⟹ ∃ Ts''. P,E,h ⊢ es [:] Ts'' ∧ P ⊢ Ts'' [≤] Ts

proof(induct Ts')
  case Nil thus ?case by simp
next
  case (Cons S Ss)
  have length:length es = length(S#Ss)
    and types-conf:types-conf P E h es (S#Ss)
    and subs:P ⊢ (S#Ss) [≤] Ts
    and IH:⋀es Ts. ⟦length es = length Ss; types-conf P E h es Ss; P ⊢ Ss [≤] Ts⟧
    ⟹ ∃ Ts''. P,E,h ⊢ es [:] Ts'' ∧ P ⊢ Ts'' [≤] Ts by fact+
```

from *subs* **obtain** *U Us* **where** *Ts*:$Ts = U\#Us$ **by**(*cases Ts*) *auto*
from *length* **obtain** e' es' **where** *es*:$es = e'\#es'$ **by**(*cases es*) *auto*
with *types-conf* **have** *type*:$P,E,h \vdash e' :_{NT} S$
and *type'*:*types-conf P E h es' Ss* **by** *simp-all*
from *subs Ts* **have** *subs'*:$P \vdash Ss\ [\leq]\ Us$ **and** *sub*:$P \vdash S \leq U$
by (*simp-all add*:*fun-of-def*)
from *sub type* **obtain** T'' **where** *step*:$P,E,h \vdash e' : T'' \wedge P \vdash T'' \leq U$
by(*cases S*,*auto*,*cases U*,*auto*)
from *length es* **have** *length es'* = *length Ss* **by** *simp*
from *IH*[*OF this type' subs'*] **obtain** Ts''
where $P,E,h \vdash es'\ [:]\ Ts'' \wedge P \vdash Ts''\ [\leq]\ Us$
by *auto*
with *step* **have** $P,E,h \vdash (e'\#es')\ [:]\ (T''\#Ts'') \wedge P \vdash (T''\#Ts'')\ [\leq]\ (U\#Us)$
by (*auto simp*:*fun-of-def*)
with *es Ts* **show** *?case* **by** *blast*
qed

end

23 Progress of Small Step Semantics

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

23.1 Some pre-definitions

lemma *final-refE*:
$[\![P,E,h \vdash e : Class\ C;\ final\ e;$
$\bigwedge r.\ e = ref\ r \Longrightarrow Q;$
$\bigwedge r.\ e = Throw\ r \Longrightarrow Q\ ]\!] \Longrightarrow Q$
by (*simp add*:*final-def*,*auto*,*case-tac v*,*auto*)

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

apply (*cases T*)
apply (*simp add*:*is-refT-def*)+
apply (*simp add*:*final-def*)
apply (*erule disjE*)
apply *clarsimp*
apply (*erule exE*)+
apply *fastforce*
apply (*auto simp*:*final-def is-refT-def*)
apply (*case-tac v*)

apply *auto*
done

lemma *subE*:
⟦ $P \vdash T \leq T'$; *is-type* P T'; *wf-prog wf-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 ⟧ $\Longrightarrow Q$ ⟧ $\Longrightarrow Q$

apply(*cases* T')
apply *auto*
apply(*drule-tac* $T = T$ **in** *widen-Class*)
apply *auto*
done

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(*erule subE*)
from *type* **show** *is-type* P T **.**
next
from *wf* **show** *wf-prog wf-md* P **.**
next
assume $T' = T$ **and** $\forall C.\ T' \neq$ *Class* C
thus $\exists v'.\ P \vdash T$ *casts* v *to* v' **by**(*fastforce intro*:*casts-prim*)
next
fix C D
assume *T'*:$T' =$ *Class* C **and** T:$T =$ *Class* D
and *path-unique*:$P \vdash$ *Path* C *to* D *unique*
from T' *typeof* **obtain** a *Cs* **where** v:$v =$ *Ref*(a,*Cs*) **and** *last*:*last Cs* $= C$
by(*auto dest*!:*typeof-Class-Subo*)
from *last path-unique* **obtain** *Cs'* **where** $P \vdash$ *Path last Cs to* D *via Cs'*
by(*auto simp*:*path-unique-def path-via-def*)
hence $P \vdash$ *Class* D *casts Ref*(a,*Cs*) *to Ref*(a,*Cs*@$_p$*Cs'*)
by −(*rule casts-ref*,*simp-all*)
with T v **show** $\exists v'.\ P \vdash T$ *casts* v *to* v' **by** *auto*
next
fix C
assume $T' = NT$ **and** T:$T =$ *Class* C
with *typeof* **have** $v =$ *Null* **by** *simp*
with T **show** $\exists v'.\ P \vdash T$ *casts* v *to* v' **by**(*fastforce intro*:*casts-null*)
qed

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; $P,E,h \vdash e$:′ T; *is-refT* T⟧
⟹ $P,E,h \vdash$ *Cast* C e :′ *Class* C
| ⟦*is-class* P C; $P,E,h \vdash e$:′ T; *is-refT* T⟧
⟹ $P,E,h \vdash (\!|C|\!)e$:′ *Class* C
| $P \vdash$ *typeof*$_h$ v = *Some* T ⟹ $P,E,h \vdash$ *Val* v :′ T
| E V = *Some* T ⟹ $P,E,h \vdash$ *Var* V :′ T
| ⟦ $P,E,h \vdash e_1$:′ T_1; $P,E,h \vdash e_2$:′ T_2;
case bop of Eq ⇒ T = *Boolean*
| *Add* ⇒ T_1 = *Integer* ∧ T_2 = *Integer* ∧ T = *Integer* ⟧
⟹ $P,E,h \vdash e_1 \ll bop \gg e_2$:′ T
| ⟦ $P,E,h \vdash$ *Var* V :′ T; $P,E,h \vdash e$:′ T'~~////////~~; $P \vdash T' \leq T$ ⟧
⟹ $P,E,h \vdash V{:=}e$:′ T
| ⟦$P,E,h \vdash e$:′ *Class* C; *Cs* ≠ []; $P \vdash C$ *has least* $F{:}T$ *via Cs*⟧
⟹ $P,E,h \vdash e{\cdot}F\{Cs\}$:′ T
| $P,E,h \vdash e$:′ *NT* ⟹ $P,E,h \vdash e{\cdot}F\{Cs\}$:′ T
| ⟦$P,E,h \vdash e_1$:′ *Class* C; *Cs* ≠ []; $P \vdash C$ *has least* $F{:}T$ *via Cs*;
$P,E,h \vdash e_2$:′ T'; $P \vdash T' \leq T$ ⟧
⟹ $P,E,h \vdash e_1{\cdot}F\{Cs\}{:=}e_2$:′ T
| ⟦ $P,E,h \vdash e_1{:}'NT$; $P,E,h \vdash e_2$:′ T'; $P \vdash T' \leq T$ ⟧
⟹ $P,E,h \vdash e_1{\cdot}F\{Cs\}{:=}e_2$:′ T
| ⟦ $P,E,h \vdash e$:′ *Class* C; $P \vdash C$ *has least* $M = (Ts,T,m)$ *via Cs*;
$P,E,h \vdash es$ [:′] Ts'; $P \vdash Ts'$ [≤] Ts ⟧
⟹ $P,E,h \vdash e{\cdot}M(es)$:′ T
| ⟦ $P,E,h \vdash e$:′ *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'; $P \vdash Ts'$ [≤] Ts ⟧
⟹ $P,E,h \vdash e{\cdot}(C::)M(es)$:′ T
| ⟦$P,E,h \vdash e$:′ *NT*; $P,E,h \vdash es$ [:′] Ts⟧ ⟹ $P,E,h \vdash$ *Call* e *Copt* M es :′ T
| ⟦ $P \vdash$ *typeof*$_h$ v = *Some* T'; $P,E(V{\mapsto}T),h \vdash e_2$:′ T_2; $P \vdash T' \leq T$; *is-type* P T ⟧
⟹ $P,E,h \vdash \{V{:}T := Val\ v;\ e_2\}$:′ T_2
| ⟦ $P,E(V{\mapsto}T),h \vdash e$:′ T'; ¬ *assigned* V e; *is-type* P T ⟧
⟹ $P,E,h \vdash \{V{:}T;\ e\}$:′ T'
| ⟦ $P,E,h \vdash e_1$:′ T_1; $P,E,h \vdash e_2$:′ T_2 ⟧ ⟹ $P,E,h \vdash e_1;;e_2$:′ T_2
| ⟦ $P,E,h \vdash e$:′ *Boolean*; $P,E,h \vdash e_1$:′ T; $P,E,h \vdash e_2$:′ T ⟧
⟹ $P,E,h \vdash$ *if* (e) e_1 *else* e_2 :′ T
| ⟦ $P,E,h \vdash e$:′ *Boolean*; $P,E,h \vdash c$:′ T ⟧
⟹ $P,E,h \vdash$ *while*(e) c :′ *Void*
| ⟦ $P,E,h \vdash e$:′ T'; *is-refT* T'⟧ ⟹ $P,E,h \vdash$ *throw* e :′ T

| $P,E,h \vdash$ [] [:′] []
| ⟦ $P,E,h \vdash e$:′ T; $P,E,h \vdash es$ [:′] Ts ⟧ ⟹ $P,E,h \vdash e\#es$ [:′] $T\#Ts$

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

apply(*rule iffI*)
apply (*auto elim*: *WTrt'.cases intro*!: *WTrt'-WTrts'.intros*)
done

lemma [*iff*]: $P,E,h \vdash Val\ v :' T = (P \vdash typeof_h\ v = Some\ T)$

apply(*rule iffI*)
apply (*auto elim*: *WTrt'.cases intro*!: *WTrt'-WTrts'.intros*)
done

lemma [*iff*]: $P,E,h \vdash Var\ V :' T = (E\ V = Some\ T)$

apply(*rule iffI*)
apply (*auto elim*: *WTrt'.cases intro*!: *WTrt'-WTrts'.intros*)
done

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 (*induct rule*:*WTrt-inducts*)
case (*WTrtBlock E V T h e T'*)
thus *?case*
apply(*case-tac assigned V e*)
apply(*auto intro*:*WTrt'-WTrts'.intros*
simp add:*fun-upd-same assigned-def simp del*:*fun-upd-apply*)
done
qed(*auto intro*:*WTrt'-WTrts'.intros simp del*:*fun-upd-apply*)

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

apply (*induct rule*:*WTrt'-inducts*)

apply (*fastforce intro*: *WTrt-WTrts.intros*)+
done

corollary *wt'-iff-wt*: $(P,E,h \vdash e :' T) = (P,E,h \vdash e : T)$
by(*blast intro*:*wt-wt' wt'-wt*)

corollary *wts'-iff-wts*: $(P,E,h \vdash es\ [:']\ Ts) = (P,E,h \vdash es\ [:]\ Ts)$
by(*blast intro*:*wts-wts' wts'-wts*)

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 (*induct rule*:*WTrt-inducts2*)
case (*WTrtVal h v T E*)
have *type*:$P \vdash typeof_h\ v = Some\ T$ **by** *fact*
{ fix *C a Cs D S*
assume $T = Class\ C$ **and** $Val\ v = ref(a,Cs)$ **and** $h\ a = Some(D,S)$
with *type* **have** *Subobjs P D Cs* **and** $C = last\ Cs$ **by** (*auto split*:*if-split-asm*)
hence $P \vdash D \preceq^* C$ **by** *simp* (*rule Subobjs-subclass*) **}**
thus *?case* **by** *blast*
qed *auto*

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

proof −
have $last\ Cs = hd\ Ds \Longrightarrow last\ (Cs\ @\ tl\ Ds) = last\ Ds$

```
  proof(cases tl Ds = [])
    case True
    assume last:last Cs = hd Ds
    with True notempty have Ds = [last Cs] by (fastforce dest:hd-Cons-tl)
    hence last Ds = last Cs by simp
    with True show ?thesis by simp
  next
    case False
    assume last:last Cs = hd Ds
    from notempty False have last (tl Ds) = last Ds
      by -(drule hd-Cons-tl,drule-tac x=hd Ds in last-ConsR,simp)
    with False show ?thesis by simp
  qed
  thus ?thesis by(simp add:appendPath-def)
qed
```

```
theorem assumes wf: wwf-prog P
shows progress: P,E,h ⊢ e : T ⟹
 (⋀l. ⟦ P ⊢ h √; P ⊢ E √; 𝒟 e ⌊dom l⌋; ¬ final e ⟧ ⟹ ∃ e' s'. P,E ⊢ ⟨e,(h,l)⟩
→ ⟨e',s'⟩)
and P,E,h ⊢ es [:] Ts ⟹
 (⋀l. ⟦ P ⊢ h √; P ⊢ E √; 𝒟s es ⌊dom l⌋; ¬ finals es ⟧ ⟹ ∃ es' s'. P,E ⊢
⟨es,(h,l)⟩ [→] ⟨es',s'⟩)
proof (induct rule:WTrt-inducts2)
  case (WTrtNew C E h)
  show ?case
  proof cases
    assume ∃ a. h a = None
    with WTrtNew show ?thesis
      by (fastforce del:exE intro!:RedNew simp:new-Addr-def)
  next
    assume ¬(∃ a. h a = None)
    with WTrtNew show ?thesis
      by(fastforce intro:RedNewFail simp add:new-Addr-def)
  qed
next
  case (WTrtDynCast C E h e T)
  have wte: P,E,h ⊢ e : T and refT: is-refT T and class: is-class P C
    and IH: ⋀l. ⟦P ⊢ h √; P ⊢ E √; 𝒟 e ⌊dom l⌋; ¬ final e⟧
              ⟹ ∃ e' s'. P,E ⊢ ⟨e,(h,l)⟩ → ⟨e',s'⟩
    and D: 𝒟 (Cast C e) ⌊dom l⌋
    and hconf: P ⊢ h √ and envconf:P ⊢ E √ by fact+
  from D have De: 𝒟 e ⌊dom l⌋ by auto
  show ?case
  proof cases
    assume final e
```

```
    with wte refT show ?thesis
    proof (rule finalRefE)
      assume e = null thus ?case by(fastforce intro:RedDynCastNull)
    next
      fix r assume e = ref r
      then obtain a Cs where ref:e = ref(a,Cs) by (cases r) auto
      with wte obtain D S where h:h a = Some(D,S) by auto
      show ?thesis
      proof (cases P ⊢ Path D to C unique)
        case True
        then obtain Cs' where path:P ⊢ Path D to C via Cs'
          by (fastforce simp:path-via-def path-unique-def)
        then obtain Ds where Ds = appendPath Cs Cs' by simp
        with h path True ref show ?thesis by (fastforce intro:RedDynCast)
      next
        case False
        hence path-not-unique:¬ P ⊢ Path D to C unique .
        show ?thesis
        proof(cases P ⊢ Path last Cs to C unique)
          case True
          then obtain Cs' where P ⊢ Path last Cs to C via Cs'
            by(auto simp:path-via-def path-unique-def)
          with True ref show ?thesis by(fastforce intro:RedStaticUpDynCast)
        next
          case False
          hence path-not-unique':¬ P ⊢ Path last Cs to C unique .
          thus ?thesis
          proof(cases C ∉ set Cs)
            case False
            then obtain Ds Ds' where Cs = Ds@[C]@Ds'
              by (auto simp:in-set-conv-decomp)
            with ref show ?thesis by(fastforce intro:RedStaticDownDynCast)
          next
            case True
            with path-not-unique path-not-unique' h ref
            show ?thesis by (fastforce intro:RedDynCastFail)
          qed
        qed
      qed
    next
      fix r assume e = Throw r
      thus ?thesis by(blast intro!:red-reds.DynCastThrow)
    qed
  next
    assume nf: ¬ final e
    from IH[OF hconf envconf De nf] show ?thesis by (blast intro:DynCastRed)
  qed
next
  case (WTrtStaticCast C E h e T)
```

```
  have wte: P,E,h ⊢ e : T and refT: is-refT T and class: is-class P C
   and IH: ⋀l. ⟦P ⊢ h √; P ⊢ E √; 𝒟 e ⌊dom l⌋; ¬ final e⟧
               ⟹ ∃ e' s'. P,E ⊢ ⟨e,(h,l)⟩ → ⟨e',s'⟩
   and D: 𝒟 (⦇C⦈e) ⌊dom l⌋
    and hconf: P ⊢ h √ and envconf:P ⊢ E √ by fact+
  from D have De: 𝒟 e ⌊dom l⌋ by auto
  show ?case
  proof cases
    assume final e
    with wte refT show ?thesis
    proof (rule finalRefE)
     assume e = null with class show ?case by(fastforce intro:RedStaticCastNull)
    next
      fix r assume e = ref r
      then obtain a Cs where ref:e = ref(a,Cs) by (cases r) auto
      with wte wf have class:is-class P (last Cs)
        by (auto intro:Subobj-last-isClass split:if-split-asm)
      show ?thesis
      proof(cases P ⊢ (last Cs) ⪯* C)
        case True
        with class wf obtain Cs' where P ⊢ Path last Cs to C via Cs'
          by(fastforce dest:leq-implies-path)
        with True ref show ?thesis by(fastforce intro:RedStaticUpCast)
      next
        case False
        have notleq:¬ P ⊢ last Cs ⪯* C by fact
        thus ?thesis
        proof(cases C ∉ set Cs)
          case False
          then obtain Ds Ds' where Cs = Ds@[C]@Ds'
            by (auto simp:in-set-conv-decomp)
          with ref show ?thesis
            by(fastforce intro:RedStaticDownCast)
        next
          case True
          with ref notleq show ?thesis by (fastforce intro:RedStaticCastFail)
        qed
      qed
    next
      fix r assume e = Throw r
      thus ?thesis by(blast intro!:red-reds.StaticCastThrow)
    qed
  next
    assume nf: ¬ final e
   from IH[OF hconf envconf De nf] show ?thesis by (blast intro:StaticCastRed)
  qed
next
  case WTrtVal thus ?case by(simp add:final-def)
next
```

```
  case WTrtVar thus ?case by(fastforce intro:RedVar simp:hyper-isin-def)
next
  case (WTrtBinOp E h e1 T1 e2 T2 bop T')
  have bop:case bop of Eq ⇒ T' = Boolean
                      | Add ⇒ T1 = Integer ∧ T2 = Integer ∧ T' = Integer
    and wte1:P,E,h ⊢ e1 : T1 and wte2:P,E,h ⊢ e2 : T2 by fact+
  show ?case
  proof cases
    assume final e1
    thus ?thesis
    proof (rule finalE)
      fix v1 assume e1 [simp]:e1 = Val v1
      show ?thesis
      proof cases
        assume final e2
        thus ?thesis
        proof (rule finalE)
          fix v2 assume e2 [simp]:e2 = Val v2
          show ?thesis
          proof (cases bop)
            assume bop = Eq
            thus ?thesis using WTrtBinOp by(fastforce intro:RedBinOp)
          next
            assume Add:bop = Add
            with e1 e2 wte1 wte2 bop obtain i1 i2
              where v1 = Intg i1 and v2 = Intg i2
              by (auto dest!:typeof-Integer)
            with Add obtain v where binop(bop,v1,v2) = Some v by simp
            with e1 e2 show ?thesis by (fastforce intro:RedBinOp)
          qed
        next
          fix a assume e2 = Throw a
          thus ?thesis by(auto intro:red-reds.BinOpThrow2)
        qed
      next
        assume ¬ final e2 with WTrtBinOp show ?thesis
          by simp (fast intro!:BinOpRed2)
      qed
    next
      fix r assume e1 = Throw r
      thus ?thesis by simp (fast intro:red-reds.BinOpThrow1)
    qed
  next
    assume ¬ final e1 with WTrtBinOp show ?thesis
      by simp (fast intro:BinOpRed1)
  qed
next
  case (WTrtLAss E h V T e T')
  have wte:P,E,h ⊢ e : T'
```

```
    and wtvar:P,E,h ⊢ Var V : T
    and sub:P ⊢ T' ≤ T
    and envconf:P ⊢ E √ by fact+
  from envconf wtvar have type:is-type P T by(auto simp:envconf-def)
  show ?case
  proof cases
    assume fin:final e
    from fin show ?case
    proof (rule finalE)
      fix v assume e:e = Val v
      from sub type wf show ?case
      proof(rule subE)
        assume eq:T' = T and ∀ C. T' ≠ Class C
        hence P ⊢ T casts v to v
          by simp(rule casts-prim)
        with wte wtvar eq e show ?thesis
          by(auto intro!:RedLAss)
      next
        fix C D
        assume T':T' = Class C and T:T = Class D
          and path-unique:P ⊢ Path C to D unique
        from wte e T' obtain a Cs where ref:e = ref(a,Cs)
          and last:last Cs = C
          by (auto dest!:typeof-Class-Subo)
        from path-unique obtain Cs' where path-via:P ⊢ Path C to D via Cs'
          by(auto simp:path-unique-def path-via-def)
        with last have P ⊢ Class D casts Ref(a,Cs) to Ref(a,Cs@_p Cs')
          by (fastforce intro:casts-ref simp:path-via-def)
        with wte wtvar T ref show ?thesis
          by(auto intro!:RedLAss)
      next
        fix C
        assume T':T' = NT and T:T = Class C
        with wte e have null:e = null by auto
        have P ⊢ Class C casts Null to Null
          by −(rule casts-null)
        with wte wtvar T null show ?thesis
          by(auto intro!:RedLAss)
      qed
    next
      fix r assume e = Throw r
      thus ?thesis by(fastforce intro:red-reds.LAssThrow)
    qed
  next
    assume ¬ final e with WTrtLAss show ?thesis
      by simp (fast intro:LAssRed)
  qed
next
  case (WTrtFAcc E h e C Cs F T)
```

```
  have wte: P,E,h ⊢ e : Class C
    and field: P ⊢ C has least F:T via Cs
    and notemptyCs:Cs ≠ []
    and hconf: P ⊢ h √ by fact+
  show ?case
  proof cases
    assume final e
    with wte show ?thesis
    proof (rule final-refE)
      fix r assume e: e = ref r
      then obtain a Cs' where ref:e = ref(a,Cs') by (cases r) auto
       with wte obtain D S where h:h a = Some(D,S) and suboD:Subobjs P D
Cs'
        and last:last Cs' = C
        by (fastforce split:if-split-asm)
      from field obtain Bs fs ms
        where class: class P (last Cs) = Some(Bs,fs,ms)
        and fs:map-of fs F = Some T
        by (fastforce simp:LeastFieldDecl-def FieldDecls-def)
      obtain Ds where Ds:Ds = Cs'@_pCs by simp
      with notemptyCs class have class':class P (last Ds) = Some(Bs,fs,ms)
        by (drule-tac Cs'=Cs' in appendPath-last) simp
      from field suboD last Ds wf have subo:Subobjs P D Ds
      by(fastforce intro:Subobjs-appendPath simp:LeastFieldDecl-def FieldDecls-def)
      with hconf h have P,h ⊢ (D,S) √ by (auto simp:hconf-def)
      with class' subo obtain fs' where S:(Ds,fs') ∈ S
        and P,h ⊢ fs' (:≤) map-of fs
        apply (auto simp:oconf-def)
        apply (erule-tac x=Ds in allE)
        apply auto
        apply (erule-tac x=Ds in allE)
        apply (erule-tac x=fs' in allE)
        apply auto
        done
      with fs obtain v where fs' F = Some v
        by (fastforce simp:fconf-def)
      with h last Ds S
      have P,E ⊢ ⟨(ref (a,Cs'))·F{Cs}, (h,l)⟩ → ⟨Val v,(h,l)⟩
        by (fastforce intro:RedFAcc)
      with ref show ?thesis by blast
    next
      fix r assume e = Throw r
      thus ?thesis by(fastforce intro:red-reds.FAccThrow)
    qed
  next
    assume ¬ final e with WTrtFAcc show ?thesis
      by(fastforce intro!:FAccRed)
  qed
next
```

```
  case (WTrtFAccNT E h e F Cs T)
  show ?case
  proof cases
    assume final e  — e is null or throw
    with WTrtFAccNT show ?thesis
      by(fastforce simp:final-def intro: RedFAccNull red-reds.FAccThrow
                dest!:typeof-NT)
  next
    assume ¬ final e — e reduces by IH
    with WTrtFAccNT show ?thesis by simp (fast intro:FAccRed)
  qed
next
  case (WTrtFAss E h e₁ C Cs F T e₂ T')
  have wte1:P,E,h ⊢ e₁ : Class C
    and wte2:P,E,h ⊢ e₂ : T'
    and field:P ⊢ C has least F:T via Cs
    and notemptyCs:Cs ≠ []
    and sub:P ⊢ T' ≤ T
    and hconf:P ⊢ h √ by fact+
  from field wf have type:is-type P T by(rule least-field-is-type)
  show ?case
  proof cases
    assume final e₁
    with wte1 show ?thesis
    proof (rule final-refE)
      fix r assume e1: e₁ = ref r
      show ?thesis
      proof cases
        assume final e₂
        thus ?thesis
        proof (rule finalE)
          fix v assume e2:e₂ = Val v
          from e1 obtain a Cs' where ref:e₁ = ref(a,Cs') by (cases r) auto
          with wte1 obtain D S where h:h a = Some(D,S)
            and suboD:Subobjs P D Cs' and last:last Cs' = C
            by (fastforce split:if-split-asm)
          from field obtain Bs fs ms
            where class: class P (last Cs) = Some(Bs,fs,ms)
            and fs:map-of fs F = Some T
            by (fastforce simp:LeastFieldDecl-def FieldDecls-def)
          obtain Ds where Ds:Ds = Cs'@_pCs by simp
          with notemptyCs class have class':class P (last Ds) = Some(Bs,fs,ms)
            by (drule-tac Cs'=Cs' in appendPath-last) simp
          from field suboD last Ds wf have subo:Subobjs P D Ds
            by(fastforce intro:Subobjs-appendPath
              simp:LeastFieldDecl-def FieldDecls-def)
          with hconf h have P,h ⊢ (D,S) √ by (auto simp:hconf-def)
          with class' subo obtain fs' where S:(Ds,fs') ∈ S
            by (auto simp:oconf-def)
```

```
    from sub type wf show ?thesis
    proof(rule subE)
      assume eq:T' = T and ∀ C. T' ≠ Class C
      hence P ⊢ T casts v to v
        by simp(rule casts-prim)
      with h last field Ds notemptyCs S eq
      have P,E ⊢ ⟨(ref (a,Cs'))·F{Cs}:=(Val v), (h,l)⟩ →
        ⟨Val v, (h(a ↦ (D,insert (Ds,fs'(F↦v)) (S − {(Ds,fs')}))),l)⟩
        by (fastforce intro:RedFAss)
      with ref e2 show ?thesis by blast
    next
      fix C' D'
      assume T':T' = Class C' and T:T = Class D'
      and path-unique:P ⊢ Path C' to D' unique
      from wte2 e2 T' obtain a' Cs'' where ref2:e2 = ref(a',Cs'')
        and last':last Cs'' = C'
        by (auto dest!:typeof-Class-Subo)
      from path-unique obtain Ds' where P ⊢ Path C' to D' via Ds'
        by(auto simp:path-via-def path-unique-def)
      with last'
      have casts:P ⊢ Class D' casts Ref(a',Cs'') to Ref(a',Cs''@_pDs')
        by (fastforce intro:casts-ref simp:path-via-def)
      obtain v' where v' = Ref(a',Cs''@_pDs') by simp
      with h last field Ds notemptyCs S ref e2 ref2 T casts
      have P,E ⊢ ⟨(ref (a,Cs'))·F{Cs}:=(Val v), (h,l)⟩ →
                ⟨Val v',(h(a ↦ (D,insert (Ds,fs'(F↦v'))(S−{(Ds,fs')}))),l)⟩
        by (fastforce intro:RedFAss)
      with ref e2 show ?thesis by blast
    next
      fix C'
      assume T':T' = NT and T:T = Class C'
      from e2 wte2 T' have null:e2 = null by auto
      have casts:P ⊢ Class C' casts Null to Null
        by −(rule casts-null)
      obtain v' where v' = Null by simp
      with h last field Ds notemptyCs S ref e2 null T casts
      have P,E ⊢ ⟨(ref (a,Cs'))·F{Cs}:=(Val v), (h,l)⟩ →
           ⟨Val v', (h(a ↦ (D,insert (Ds,fs'(F↦v')) (S − {(Ds,fs')}))),l)⟩
        by (fastforce intro:RedFAss)
      with ref e2 show ?thesis by blast
    qed
  next
    fix r assume e2 = Throw r
    thus ?thesis using e1 by(fastforce intro:red-reds.FAssThrow2)
  qed
next
  assume ¬ final e2 with WTrtFAss e1 show ?thesis
    by simp (fast intro!:FAssRed2)
qed
```

```
    next
      fix r assume e1 = Throw r
      thus ?thesis by(fastforce intro:red-reds.FAssThrow1)
    qed
  next
    assume ¬ final e1 with WTrtFAss show ?thesis
      by simp (blast intro!:FAssRed1)
  qed
next
  case (WTrtFAssNT E h e1 e2 T' T F Cs)
  show ?case
  proof cases
    assume e1: final e1  — e1 is null or throw
    show ?thesis
    proof cases
      assume final e2  — e2 is Val or throw
      with WTrtFAssNT e1 show ?thesis
        by(fastforce simp:final-def intro:RedFAssNull red-reds.FAssThrow1
                                          red-reds.FAssThrow2 dest!:typeof-NT)
    next
      assume  ¬ final e2 — e2 reduces by IH
      with WTrtFAssNT e1 show ?thesis
        by (fastforce simp:final-def intro!:red-reds.FAssRed2 red-reds.FAssThrow1)
    qed
  next
    assume ¬ final e1 — e1 reduces by IH
    with WTrtFAssNT show ?thesis by (fastforce intro:FAssRed1)
  qed
next
  case (WTrtCall E h e C M Ts T pns body Cs es Ts')
  have wte: P,E,h ⊢ e : Class C
    and method:P ⊢ C has least M = (Ts, T, pns, body) via Cs
    and wtes: P,E,h ⊢ es [:] Ts'and sub: P ⊢ Ts' [≤] Ts
    and IHes: ⋀l. ⟦P ⊢ h √; P ⊢ E √; 𝒟s es ⌊dom l⌋; ¬ finals es⟧
          ⟹ ∃es' s'. P,E ⊢ ⟨es,(h,l)⟩ [→] ⟨es',s'⟩
    and hconf: P ⊢ h √ and envconf:P ⊢ E √
    and D: 𝒟 (e•M(es)) ⌊dom l⌋ by fact+
  show ?case
  proof cases
    assume final:final e
    with wte show ?thesis
    proof (rule final-refE)
      fix r assume ref: e = ref r
      show ?thesis
      proof cases
        assume es: ∃vs. es = map Val vs
        from ref obtain a Cs' where ref:e = ref(a,Cs') by (cases r) auto
        with wte obtain D S where h:h a = Some(D,S) and suboD:Subobjs P D
Cs'
```

and *last*:*last Cs'* = *C*
by (*fastforce split*:*if-split-asm*)
from *wte ref h* **have** *subcls*:$P \vdash D \preceq^* C$ **by** −(*drule mdc-leq-dyn-type*,*auto*)
from *method* **have** *has*:$P \vdash C$ *has M* = (*Ts*,*T*,*pns*,*body*) *via Cs*
by(*rule has-least-method-has-method*)
from *es* **obtain** *vs* **where** *vs*:*es* = *map Val vs* **by** *auto*
obtain *Cs''* *Ts''* *T'* *pns'* *body'* **where**
ass:$P \vdash (D,Cs'@_p Cs)$ *selects M* = (*Ts''*,*T'*,*pns'*,*body'*) *via Cs''* ∧
length Ts'' = *length pns'* ∧ *length vs* = *length pns'* ∧ $P \vdash T' \leq T$
proof (*cases* ∃ *Ts''* *T'* *pns'* *body'* *Ds*. $P \vdash D$ *has least M* = (*Ts''*,*T'*,*pns'*,*body'*) *via Ds*)
case *True*
then obtain *Ts''* *T'* *pns'* *body'* *Cs''*
where *least*:$P \vdash D$ *has least M* = (*Ts''*,*T'*,*pns'*,*body'*) *via Cs''*
by *auto*
hence *select*:$P \vdash (D,Cs'@_p Cs)$ *selects M* = (*Ts''*,*T'*,*pns'*,*body'*) *via Cs''*
by(*rule dyn-unique*)
from *subcls least wf has* **have** *Ts* = *Ts''* **and** *leq*:$P \vdash T' \leq T$
by −(*drule leq-method-subtypes*,*simp-all*,*blast*)+
hence *length Ts* = *length Ts''* **by** (*simp add*:*list-all2-iff*)
with *sub* **have** *length Ts'* = *length Ts''* **by** (*simp add*:*list-all2-iff*)
with *WTrts-same-length*[*OF wtes*] *vs* **have** *length*:*length vs* = *length Ts''*
by *simp*
from *has-least-wf-mdecl*[*OF wf least*]
have *lengthParams*:*length Ts''* = *length pns'* **by** (*simp add*:*wf-mdecl-def*)
with *length* **have** *length vs* = *length pns'* **by** *simp*
with *select lengthParams leq* **show** *?thesis* **using** *that* **by** *blast*
next
case *False*
hence *non-dyn*:∀ *Ts''* *T'* *pns'* *body'* *Ds*.
¬ $P \vdash D$ *has least M* = (*Ts''*,*T'*,*pns'*,*body'*) *via Ds* **by** *auto*
from *suboD last* **have** *path*:$P \vdash$ *Path D to C via Cs'*
by(*simp add*:*path-via-def*)
from *method* **have** *notempty*:*Cs* ≠ []
by(*fastforce intro*!:*Subobjs-nonempty*
simp:*LeastMethodDef-def MethodDefs-def*)
from *suboD* **have** *class*: *is-class P D* **by**(*rule Subobjs-isClass*)
from *suboD last* **have** *path*:$P \vdash$ *Path D to C via Cs'*
by(*simp add*:*path-via-def*)
with *method wf* **have** $P \vdash D$ *has M* = (*Ts*,*T*,*pns*,*body*) *via* $Cs'@_p Cs$
by(*auto intro*:*has-path-has has-least-method-has-method*)
with *class wf* **obtain** *Cs''* *Ts''* *T'* *pns'* *body'* **where** *overrider*:
$P \vdash (D,Cs'@_p Cs)$ *has overrider M* = (*Ts''*,*T'*,*pns'*,*body'*) *via Cs''*
by(*auto dest*!:*class-wf simp*:*is-class-def wf-cdecl-def*,*blast*)
with *non-dyn*
have *select*:$P \vdash (D,Cs'@_p Cs)$ *selects M* = (*Ts''*,*T'*,*pns'*,*body'*) *via Cs''*
by−(*rule dyn-ambiguous*,*simp-all*)
from *notempty* **have** *eq*:(*Cs'* $@_p$ *Cs*) $@_p$ [*last Cs*] = (*Cs'* $@_p$ *Cs*)
by(*rule appendPath-append-last*)

```
    from method wf
    have P ⊢ last Cs has least M = (Ts,T,pns,body) via [last Cs]
      by(auto dest:Subobj-last-isClass intro:Subobjs-Base subobjs-rel
            simp:LeastMethodDef-def MethodDefs-def)
    with notempty
    have P ⊢ last(Cs'@_pCs) has least M = (Ts,T,pns,body) via [last Cs]
      by −(drule-tac Cs'=Cs' in appendPath-last,simp)
    with overrider wf eq
    have (Cs'',(Ts'',T',pns',body')) ∈ MinimalMethodDefs P D M
      and P,D ⊢ Cs'' ⊑ Cs'@_pCs
      by(auto simp:FinalOverriderMethodDef-def OverriderMethodDefs-def)
        (drule wf-sees-method-fun,auto)
    with subcls wf notempty has path have Ts = Ts'' and leq:P ⊢ T' ≤ T
      by −(drule leq-methods-subtypes,simp-all,blast)+
    hence length Ts = length Ts'' by (simp add:list-all2-iff)
    with sub have length Ts' = length Ts'' by (simp add:list-all2-iff)
    with WTrts-same-length[OF wtes] vs have length:length vs = length Ts''
      by simp
    from select-method-wf-mdecl[OF wf select]
    have lengthParams:length Ts'' = length pns' by (simp add:wf-mdecl-def)
    with length have length vs = length pns' by simp
    with select lengthParams leq show ?thesis using that by blast
  qed
  obtain new-body where case T of Class D ⇒
  new-body = (⌊D⌋)blocks(this#pns',Class(last Cs'')#Ts'',Ref(a,Cs'')#vs,body')
| - ⇒ new-body = blocks(this#pns',Class(last Cs'')#Ts'',Ref(a,Cs'')#vs,body')
    by(cases T) auto
  with h method last ass ref vs
    show ?thesis by (auto intro!:exI RedCall)
next
  assume ¬(∃ vs. es = map Val vs)
  hence not-all-Val: ¬(∀ e ∈ set es. ∃ v. e = Val v)
    by(simp add:ex-map-conv)
  let ?ves = takeWhile (λe. ∃ v. e = Val v) es
  let ?rest = dropWhile (λe. ∃ v. e = Val v) es
  let ?ex = hd ?rest let ?rst = tl ?rest
  from not-all-Val have nonempty: ?rest ≠ [] by auto
  hence es: es = ?ves @ ?ex # ?rst by simp
  have ∀ e ∈ set ?ves. ∃ v. e = Val v by(fastforce dest:set-takeWhileD)
  then obtain vs where ves: ?ves = map Val vs
    using ex-map-conv by blast
  show ?thesis
  proof cases
    assume final ?ex
    moreover from nonempty have ¬(∃ v. ?ex = Val v)
      by(auto simp:neq-Nil-conv simp del:dropWhile-eq-Nil-conv)
        (simp add:dropWhile-eq-Cons-conv)
    ultimately obtain r' where ex-Throw: ?ex = Throw r'
      by(fast elim!:finalE)
```

```
        show ?thesis using ref es ex-Throw ves
          by(fastforce intro:red-reds.CallThrowParams)
      next
        assume not-fin: ¬ final ?ex
        have finals es = finals(?ves @ ?ex # ?rst) using es
          by(rule arg-cong)
        also have ... = finals(?ex # ?rst) using ves by simp
        finally have finals es = finals(?ex # ?rst) .
        hence ¬ finals es using not-finals-ConsI[OF not-fin] by blast
        thus ?thesis using ref D IHes[OF hconf envconf]
          by(fastforce intro!:CallParams)
      qed
    qed
  next
    fix r assume e = Throw r
    with WTrtCall.prems show ?thesis by(fast intro!:red-reds.CallThrowObj)
  qed
next
  assume ¬ final e
  with WTrtCall show ?thesis by simp (blast intro!:CallObj)
qed
next
  case (WTrtStaticCall E h e C' C M Ts T pns body Cs es Ts')
  have wte: P,E,h ⊢ e : Class C'
    and path-unique:P ⊢ Path C' to C unique
    and method:P ⊢ C has least M = (Ts, T, pns, body) via Cs
    and wtes: P,E,h ⊢ es [:] Ts'and sub: P ⊢ Ts' [≤] Ts
    and IHes: ⋀l.
          ⟦P ⊢ h √; envconf P E; 𝒟s es ⌊dom l⌋; ¬ finals es⟧
          ⟹ ∃es' s'. P,E ⊢ ⟨es,(h,l)⟩ [→] ⟨es',s'⟩
    and hconf: P ⊢ h √ and envconf:envconf P E
    and D: 𝒟 (e•(C::)M(es)) ⌊dom l⌋ by fact+
  show ?case
  proof cases
    assume final:final e
    with wte show ?thesis
    proof (rule final-refE)
      fix r assume ref: e = ref r
      show ?thesis
      proof cases
        assume es: ∃vs. es = map Val vs
        from ref obtain a Cs' where ref:e = ref(a,Cs') by (cases r) auto
        with wte have last:last Cs' = C'
          by (fastforce split:if-split-asm)
        with path-unique obtain Cs''
          where path-via:P ⊢ Path (last Cs') to C via Cs''
          by (auto simp add:path-via-def path-unique-def)
        obtain Ds where Ds:Ds = (Cs'@_p Cs'')@_p Cs by simp
        from es obtain vs where vs:es = map Val vs by auto
```

```
      from sub have length Ts' = length Ts by (simp add:list-all2-iff)
      with WTrts-same-length[OF wtes] vs have length:length vs = length Ts
        by simp
      from has-least-wf-mdecl[OF wf method]
      have lengthParams:length Ts = length pns by (simp add:wf-mdecl-def)
      with method last path-unique path-via Ds length ref vs show ?thesis
        by (auto intro!:exI RedStaticCall)
    next
      assume ¬(∃ vs. es = map Val vs)
      hence not-all-Val: ¬(∀ e ∈ set es. ∃ v. e = Val v)
        by(simp add:ex-map-conv)
      let ?ves = takeWhile (λe. ∃ v. e = Val v) es
      let ?rest = dropWhile (λe. ∃ v. e = Val v) es
      let ?ex = hd ?rest let ?rst = tl ?rest
      from not-all-Val have nonempty: ?rest ≠ [] by auto
      hence es: es = ?ves @ ?ex # ?rst by simp
      have ∀ e ∈ set ?ves. ∃ v. e = Val v by(fastforce dest:set-takeWhileD)
      then obtain vs where ves: ?ves = map Val vs
        using ex-map-conv by blast
      show ?thesis
      proof cases
        assume final ?ex
        moreover from nonempty have ¬(∃ v. ?ex = Val v)
          by(auto simp:neq-Nil-conv simp del:dropWhile-eq-Nil-conv)
            (simp add:dropWhile-eq-Cons-conv)
        ultimately obtain r' where ex-Throw: ?ex = Throw r'
          by(fast elim!:finalE)
        show ?thesis using ref es ex-Throw ves
          by(fastforce intro:red-reds.CallThrowParams)
      next
        assume not-fin: ¬ final ?ex
        have finals es = finals(?ves @ ?ex # ?rst) using es
          by(rule arg-cong)
        also have ... = finals(?ex # ?rst) using ves by simp
        finally have finals es = finals(?ex # ?rst) .
        hence ¬ finals es using not-finals-ConsI[OF not-fin] by blast
        thus ?thesis using ref D IHes[OF hconf envconf]
          by(fastforce intro!:CallParams)
      qed
    qed
  next
    fix r assume e = Throw r
  with WTrtStaticCall.prems show ?thesis by(fast intro!:red-reds.CallThrowObj)
  qed
next
  assume ¬ final e
  with WTrtStaticCall show ?thesis by simp (blast intro!:CallObj)
qed
next
```

```
  case (WTrtCallNT E h e es Ts Copt M T)
  show ?case
  proof cases
    assume final e
    moreover
    { fix v assume e: e = Val v
      hence e = null using WTrtCallNT by simp
      have ?case
      proof cases
        assume finals es
        moreover
        { fix vs assume es = map Val vs
       with WTrtCallNT e have ?thesis by(fastforce intro: RedCallNull dest!:typeof-NT)
}
        moreover
        { fix vs a es' assume es = map Val vs @ Throw a # es'
          with WTrtCallNT e have ?thesis by(fastforce intro: CallThrowParams)
}
        ultimately show ?thesis by(fastforce simp:finals-def)
      next
        assume ¬ finals es — es reduces by IH
        with WTrtCallNT e show ?thesis by(fastforce intro: CallParams)
      qed
    }
    moreover
    { fix r assume e = Throw r
      with WTrtCallNT have ?case by(fastforce intro: CallThrowObj) }
    ultimately show ?thesis by(fastforce simp:final-def)
  next
    assume ¬ final e — e reduces by IH
    with WTrtCallNT show ?thesis by (fastforce intro:CallObj)
  qed
next
  case (WTrtInitBlock h v T' E V T e2 T2)
  have IH2: ⋀l. ⟦P ⊢ h √; P ⊢ E(V ↦ T) √; 𝒟 e2 ⌊dom l⌋; ¬ final e2⟧
                ⟹ ∃e' s'. P,E(V ↦ T) ⊢ ⟨e2,(h,l)⟩ → ⟨e',s'⟩
    and typeof:P ⊢ typeof_h v = Some T'
    and type:is-type P T and sub:P ⊢ T' ≤ T
    and hconf: P ⊢ h √ and envconf:P ⊢ E √
    and D: 𝒟 {V:T := Val v; e2} ⌊dom l⌋ by fact+
  from wf typeof type sub obtain v' where casts:P ⊢ T casts v to v'
    by(auto dest:sub-casts)
  show ?case
  proof cases
    assume fin:final e2
    with casts show ?thesis
      by(fastforce elim:finalE intro:RedInitBlock red-reds.InitBlockThrow)
  next
    assume not-fin2: ¬ final e2
```

```
    from D have D2: 𝒟 e2 ⌊dom(l(V↦v'))⌋ by (auto simp:hyperset-defs)
    from envconf type have P ⊢ E(V ↦ T) √ by(auto simp:envconf-def)
    from IH2[OF hconf this D2 not-fin2]
    obtain h' l' e' where red2: P,E(V ↦ T) ⊢ ⟨e2,(h, l(V↦v'))⟩ → ⟨e',(h', l')⟩
      by auto
    from red-lcl-incr[OF red2] have V ∈ dom l' by auto
    with red2 casts show ?thesis by(fastforce intro:InitBlockRed)
  qed
next
  case (WTrtBlock E V T h e T')
  have IH: ⋀l. ⟦P ⊢ h √; P ⊢ E(V ↦ T) √; 𝒟 e ⌊dom l⌋; ¬ final e⟧
              ⟹ ∃e' s'. P,E(V ↦ T) ⊢ ⟨e,(h,l)⟩ → ⟨e',s'⟩
   and unass: ¬ assigned V e and type:is-type P T
   and hconf: P ⊢ h √ and envconf:P ⊢ E √
   and D: 𝒟 {V:T; e} ⌊dom l⌋ by fact+
  show ?case
  proof cases
    assume final e
    thus ?thesis
    proof (rule finalE)
      fix v assume e = Val v with type show ?thesis by(fast intro:RedBlock)
    next
      fix r assume e = Throw r
      with type show ?thesis by(fast intro:red-reds.BlockThrow)
    qed
  next
    assume not-fin: ¬ final e
    from D have De: 𝒟 e ⌊dom(l(V:=None))⌋ by(simp add:hyperset-defs)
    from envconf type have P ⊢ E(V ↦ T) √ by(auto simp:envconf-def)
    from IH[OF hconf this De not-fin]
    obtain h' l' e' where red: P,E(V ↦ T) ⊢ ⟨e,(h,l(V:=None))⟩ → ⟨e',(h',l')⟩
      by auto
    show ?thesis
    proof (cases l' V)
      assume l' V = None
      with red unass show ?thesis by(blast intro: BlockRedNone)
    next
      fix v assume l' V = Some v
      with red unass type show ?thesis by(blast intro: BlockRedSome)
    qed
  qed
next
  case (WTrtSeq E h e1 T1 e2 T2)
  show ?case
  proof cases
    assume final e1
    thus ?thesis
      by(fast elim:finalE intro:intro:RedSeq red-reds.SeqThrow)
  next
```

```
    assume ¬ final e1 with WTrtSeq show ?thesis
      by simp (blast intro:SeqRed)
  qed
next
  case (WTrtCond E h e e1 T e2)
  have wt: P,E,h ⊢ e : Boolean by fact
  show ?case
  proof cases
    assume final e
    thus ?thesis
    proof (rule finalE)
      fix v assume val: e = Val v
    then obtain b where v: v = Bool b using wt by (fastforce dest:typeof-Boolean)
      show ?thesis
      proof (cases b)
        case True with val v show ?thesis by(auto intro:RedCondT)
      next
        case False with val v show ?thesis by(auto intro:RedCondF)
      qed
    next
      fix r assume e = Throw r
      thus ?thesis by(fast intro:red-reds.CondThrow)
    qed
  next
    assume ¬ final e with WTrtCond show ?thesis
      by simp (fast intro:CondRed)
  qed
next
  case WTrtWhile show ?case by(fast intro:RedWhile)
next
  case (WTrtThrow E h e T' T)
  show ?case
  proof cases
    assume final e — Then e must be throw or null
    with WTrtThrow show ?thesis
      by(fastforce simp:final-def is-refT-def
                    intro:red-reds.ThrowThrow red-reds.RedThrowNull
                    dest!:typeof-NT typeof-Class-Subo)
  next
    assume ¬ final e — Then e must reduce
    with WTrtThrow show ?thesis by simp (blast intro:ThrowRed)
  qed
next
  case WTrtNil thus ?case by simp
next
  case (WTrtCons E h e T es Ts)
  have IHe: ⋀l. ⟦P ⊢ h √; P ⊢ E √; 𝒟 e ⌊dom l⌋; ¬ final e⟧
              ⟹ ∃e' s'. P,E ⊢ ⟨e,(h,l)⟩ → ⟨e',s'⟩
   and IHes: ⋀l. ⟦P ⊢ h √; P ⊢ E √; 𝒟s es ⌊dom l⌋; ¬ finals es⟧
```

$\Longrightarrow \exists\, es'\ s'.\ P,E \vdash \langle es,(h,l)\rangle\ [\rightarrow]\ \langle es',s'\rangle$
and *hconf*: $P \vdash h\ \surd$ **and** *envconf*:$P \vdash E\ \surd$
and *D*: $\mathcal{D}s\ (e\#es)\ \lfloor dom\ l \rfloor$
and *not-fins*: $\neg$ *finals*$(e\ \#\ es)$ **by** *fact*+
have *De*: $\mathcal{D}\ e\ \lfloor dom\ l \rfloor$ **and** *Des*: $\mathcal{D}s\ es\ (\lfloor dom\ l \rfloor \sqcup \mathcal{A}\ e)$
using *D* **by** *auto*
show *?case*
proof *cases*
assume *final e*
thus *?thesis*
proof (*rule finalE*)
fix *v* **assume** *e*: $e = Val\ v$
hence *Des'*: $\mathcal{D}s\ es\ \lfloor dom\ l \rfloor$ **using** *De Des* **by** *auto*
have *not-fins-tl*: $\neg$ *finals es* **using** *not-fins e* **by** *simp*
show *?thesis* **using** *e IHes*[*OF hconf envconf Des' not-fins-tl*]
by (*blast intro*!:*ListRed2*)
next
fix *r* **assume** $e = Throw\ r$
hence *False* **using** *not-fins* **by** *simp*
thus *?thesis* **..**
qed
next
assume $\neg$ *final e*
from *IHe*[*OF hconf envconf De this*] **show** *?thesis* **by**(*fast intro*!:*ListRed1*)
qed
qed

end

24 Heap Extension

theory *HeapExtension*
imports *Progress*
begin

24.1 The Heap Extension

definition *hext* :: *heap* $\Rightarrow$ *heap* $\Rightarrow$ *bool* (- $\trianglelefteq$ - [*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*: $\forall\, a\ C\ S.\ h\ a = Some(C,S) \longrightarrow (\exists\, S'.\ h'\ a = Some(C,S')) \Longrightarrow h \trianglelefteq h'$

apply (*unfold hext-def*)
apply *auto*
done

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

apply (*unfold hext-def*)
apply (*force*)
done

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

apply (*rule hextI*)
apply (*fast*)
done

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

apply (*rule hextI*)
apply (*auto simp*:*fun-upd-apply*)
done

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

apply (*rule hextI*)
apply (*fast dest*: *hext-objD*)
done

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

apply (*rule hextI*)
apply (*auto simp*:*fun-upd-apply*)
done

24.2 $\trianglelefteq$ and preallocated

lemma *preallocated-hext*:
$[\![$ *preallocated h*; $h \trianglelefteq h'\ ]\!] \Longrightarrow$ *preallocated h'*
by (*simp add*: *preallocated-def hext-def*)

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

proof(*induct rule*:*red-reds-inducts*)

```
  case RedNew thus ?case
    by(fastforce dest:new-Addr-SomeD simp:hext-def split:if-splits)
next
  case RedFAss thus ?case by(simp add:hext-def split:if-splits)
qed simp-all


lemma step-hext-incr: P,E ⊢ ⟨e,s⟩ →* ⟨e',s'⟩ ⟹ hp s ⊴ hp s'

proof(induct rule:converse-rtrancl-induct2)
  case refl thus ?case by(rule hext-refl)
next
  case (step e s e'' s'')
  have Red:((e, s), e'', s'') ∈ Red P E
    and hext:hp s'' ⊴ hp s' by fact+
  from Red have P,E ⊢ ⟨e,s⟩ → ⟨e'',s''⟩ by simp
  hence hp s ⊴ hp s''
    by(cases s,cases s'')(auto dest:red-hext-incr)
  with hext show ?case by−(rule hext-trans)
qed


lemma steps-hext-incr: P,E ⊢ ⟨es,s⟩ [→]* ⟨es',s'⟩ ⟹ hp s ⊴ hp s'

proof(induct rule:converse-rtrancl-induct2)
  case refl thus ?case by(rule hext-refl)
next
  case (step es s es'' s'')
  have Reds:((es, s), es'', s'') ∈ Reds P E
    and hext:hp s'' ⊴ hp s' by fact+
  from Reds have P,E ⊢ ⟨es,s⟩ [→] ⟨es'',s''⟩ by simp
  hence hp s ⊴ hp s''
    by(cases s,cases s'',auto dest:reds-hext-incr)
  with hext show ?case by−(rule hext-trans)
qed



lemma eval-hext: P,E ⊢ ⟨e,(h,l)⟩ ⇒ ⟨e',(h',l')⟩ ⟹ h ⊴ h'
and evals-hext:  P,E ⊢ ⟨es,(h,l)⟩ [⇒] ⟨es',(h',l')⟩ ⟹ h ⊴ h'

proof (induct rule:eval-evals-inducts)
  case New thus ?case
    by(fastforce intro!: hext-new intro:someI simp:new-Addr-def
              split:if-split-asm simp del:fun-upd-apply)
next
  case FAss thus ?case
    by(auto simp:sym[THEN hext-upd-obj] simp del:fun-upd-apply
          elim!: hext-trans)
```

qed (*auto elim*!: *hext-trans*)

24.4 $\trianglelefteq$ and conformance

lemma *conf-hext*: $h \trianglelefteq h' \Longrightarrow P,h \vdash v :\leq T \Longrightarrow P,h' \vdash v :\leq T$
by(*cases T*)(*induct v*,*auto dest*: *hext-objD split*:*if-split-asm*)+

lemma *confs-hext*: $P,h \vdash vs\ [:\leq]\ Ts \Longrightarrow h \trianglelefteq h' \Longrightarrow P,h' \vdash vs\ [:\leq]\ Ts$
by (*erule list-all2-mono*, *erule conf-hext*, *assumption*)

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

apply (*unfold fconf-def*)
apply (*fast elim*: *conf-hext*)
done

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

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

apply (*auto simp*:*oconf-def*)
apply (*erule allE*)
apply (*erule-tac x*=*Cs* **in** *allE*)
apply (*erule-tac x*=*fs'* **in** *allE*)
apply (*fastforce elim*:*fconf-hext*)
done

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

lemma *hconf-new*: $[\![\ P \vdash h\ \surd;\ h\ a = None;\ P,h \vdash obj\ \surd\ ]\!] \Longrightarrow P \vdash h(a \mapsto obj)\ \surd$
by (*unfold hconf-def*) (*auto intro*: *oconf-new preallocated-new*)

lemma $[\![P \vdash h\ \surd;\ h' = h(a \mapsto (C,\ Collect\ (init\text{-}obj\ P\ C)));\ h\ a = None;\ wf\text{-}prog\ wf\text{-}md\ P]\!]$
$\Longrightarrow P \vdash h'\ \surd$
apply (*simp add*:*hconf-def oconf-def*)
apply *auto*
apply (*rule-tac x*=*init-class-fieldmap P* (*last Cs*) **in** *exI*)
apply (*rule init-obj.intros*)
apply *assumption*
apply (*erule init-obj.cases*)

apply *clarsimp*
apply (*erule init-obj.cases*)
apply *clarsimp*
apply (*erule-tac x=a* **in** *allE*)
apply *clarsimp*
apply (*erule init-obj.cases*)
apply *simp*
apply (*erule-tac x=a* **in** *allE*)
apply *clarsimp*
apply (*erule init-obj.cases*)
apply *clarsimp*
apply (*drule Subobj-last-isClass*)
apply *simp*
apply (*auto simp:is-class-def*)
apply (*rule fconf-init-fields*)
apply *auto*
apply (*erule-tac x=aa* **in** *allE*)
apply (*erule-tac x=aaa* **in** *allE*)
apply (*erule-tac x=b* **in** *allE*)
apply *clarsimp*
apply (*rotate-tac −1*)
apply (*erule-tac x=Cs* **in** *allE*)
apply (*erule-tac x=fs′* **in** *allE*)
apply *clarsimp* **thm** *fconf-new*
apply (*erule fconf-new*)
apply *simp*
apply (*rule preallocated-new*)
apply *simp-all*
done

lemma *hconf-upd-obj*:
$[\![P \vdash h\surd;\ h\ a = Some(C,S);\ P,h \vdash (C,S')\surd]\!] \Longrightarrow P \vdash h(a{\mapsto}(C,S'))\surd$
by (*unfold hconf-def*) (*auto intro*: *oconf-upd-obj preallocated-upd-obj*)

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

apply (*unfold lconf-def*)
apply (*fast elim*: *conf-hext*)
done

24.5 $\trianglelefteq$ in the runtime type system

lemma *hext-typeof-mono*: $[\![h \trianglelefteq h';\ P \vdash typeof_h\ v = Some\ T]\!] \Longrightarrow P \vdash typeof_{h'}\ v = Some\ T$

apply(*cases v*)
apply *simp*

apply *simp*
apply *simp*
apply *simp*
apply(*fastforce simp*:*hext-def*)
done

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*: $P,E,h \vdash es\ [:]\ Ts \Longrightarrow (\bigwedge h'.\ h \trianglelefteq h' \Longrightarrow P,E,h' \vdash es\ [:]\ Ts)$

apply(*induct rule*: *WTrt-inducts*)
apply(*simp add*: *WTrtNew*)
apply(*fastforce intro*: *WTrtDynCast*)
apply(*fastforce intro*: *WTrtStaticCast*)
apply(*fastforce simp*: *WTrtVal dest*:*hext-typeof-mono*)
apply(*simp add*: *WTrtVar*)
apply(*fastforce simp add*: *WTrtBinOp*)
apply(*fastforce simp add*: *WTrtLAss*)
apply(*fastforce simp*: *WTrtFAcc del*:*WTrt-WTrts.intros WTrt-elim-cases*)
apply(*simp add*: *WTrtFAccNT*)
apply(*fastforce simp*: *WTrtFAss del*:*WTrt-WTrts.intros WTrt-elim-cases*)
apply(*fastforce simp*: *WTrtFAssNT del*:*WTrt-WTrts.intros WTrt-elim-cases*)
apply(*fastforce simp*: *WTrtCall del*:*WTrt-WTrts.intros WTrt-elim-cases*)
apply(*fastforce simp*: *WTrtStaticCall del*:*WTrt-WTrts.intros WTrt-elim-cases*)
apply(*fastforce simp*: *WTrtCallNT del*:*WTrt-WTrts.intros WTrt-elim-cases*)
apply(*fastforce*)
apply(*fastforce simp add*: *WTrtSeq*)
apply(*fastforce simp add*: *WTrtCond*)
apply(*fastforce simp add*: *WTrtWhile*)
apply(*fastforce simp add*: *WTrtThrow*)
apply(*simp add*: *WTrtNil*)
apply(*simp add*: *WTrtCons*)
done

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*⌋)
by(*simp add*:*wf-C-mdecl-def*)

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*

apply (*simp add*: *wf-prog-def*)
apply (*simp add*: *wf-cdecl-def*)
apply (*erule conjE*)+
apply (*drule bspec*, *assumption*)
apply *simp*
apply (*erule conjE*)+
apply (*drule bspec*, *assumption*)
apply (*simp add*: *wf-mdecl-def split-beta*)
done

lemma *wf-mdecl-wwf-mdecl*: *wf-C-mdecl P C Md* ⟹ *wwf-mdecl P C Md*
by(*fastforce simp*:*wwf-mdecl-def dest*!:*WT-fv*)

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

apply(*simp add*:*wf-prog-def wf-cdecl-def wf-mdecl-def*)
apply(*fast intro*:*wf-mdecl-wwf-mdecl*)
done

end

26 Type Safety Proof

theory *TypeSafe*
imports *HeapExtension CWellForm*
begin

26.1 Basic preservation lemmas

lemma assumes *wf*:*wwf-prog P* **and** *casts*:$P \vdash T$ *casts v to* v'
and *typeof*:$P \vdash$ *typeof*$_h$ v = *Some* T' **and** *leq*:$P \vdash T' \leq T$
shows *casts-conf*:$P,h \vdash v' :\leq T$

proof −
{ fix a' *C Cs* S'
assume *leq*:$P \vdash$ *Class* (*last Cs*) $\leq T$ **and** *subo*:*Subobjs P C Cs*
and *casts'*:$P \vdash T$ *casts Ref* (a',*Cs*) *to* v' **and** *h*:$h\ a' =$ *Some*(C,S')
from *subo wf* **have** *is-class P* (*last Cs*) **by**(*fastforce intro*:*Subobj-last-isClass*)
with *leq wf* **obtain** C' **where** T:T = *Class* C'
and *path-unique*:$P \vdash$ *Path* (*last Cs*) *to* C' *unique*
by(*auto dest*:*Class-widen*)
from *path-unique* **obtain** Cs' **where** *path-via*:$P \vdash$ *Path* (*last Cs*) *to* C' *via* Cs'
by(*auto simp*:*path-via-def path-unique-def*)
with T *path-unique casts'* **have** v':v' = *Ref* (a',$Cs@_p Cs'$)
by −(*erule casts-to.cases*,*auto simp*:*path-unique-def path-via-def*)
from *subo path-via wf* **have** *Subobjs P C* ($Cs@_p Cs'$)
and *last* ($Cs@_p Cs'$) = C'
apply(*auto intro*:*Subobjs-appendPath simp*:*path-via-def*)
apply(*drule-tac Cs*=Cs' **in** *Subobjs-nonempty*)
by(*rule sym*[*OF appendPath-last*])
with T h v' **have** *?thesis* **by** *auto* **}**
with *casts typeof wf typeof leq* **show** *?thesis*
by(*cases v*,*auto elim*:*casts-to.cases split*:*if-split-asm*)
qed

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

proof (*induct rule*:*red-reds-inducts*)
case (*RedNew h a h' C E l*)
have *new*: *new-Addr h* = *Some a* **and** *h'*:$h' = h(a \mapsto (C,$ *Collect* (*init-obj P C*)))
and *hconf*:$P \vdash h\ \surd$ **and** *wt-New*:$P,E,h \vdash$ *new* $C : T$ **by** *fact+*
from *new* **have** *None*: $h\ a$ = *None* **by**(*rule new-Addr-SomeD*)

with *wf* **have** *oconf*:*P*,*h* ⊢ (*C*, *Collect* (*init-obj P C*)) √
apply (*auto simp*:*oconf-def*)
apply (*rule-tac x*=*init-class-fieldmap P* (*last Cs*) **in** *exI*)
by (*fastforce intro*:*init-obj.intros fconf-init-fields*
elim: *init-obj.cases dest*!:*Subobj-last-isClass simp*:*is-class-def*)+
thus *?case* **using** *h′ None* **by**(*fast intro*: *hconf-new*[*OF hconf*])
next
case (*RedFAss h a D S Cs′ F T Cs v v′ Ds fs′ E l T′*)
let *?fs′* = *fs′*(*F* ↦ *v′*)
let *?S′* = *insert* (*Ds*, *?fs′*) (*S* − {(*Ds*, *fs′*)})
have *ha*:*h a* = *Some*(*D*,*S*) **and** *hconf*:*P* ⊢ *h* √
and *field*:*P* ⊢ *last Cs′ has least F*:*T via Cs*
and *casts*:*P* ⊢ *T casts v to v′*
and *Ds*:*Ds* = *Cs′* @$_p$ *Cs* **and** *S*:(*Ds*,*fs′*) ∈ *S*
and *wte*:*P*,*E*,*h* ⊢ *ref*(*a*,*Cs′*)·*F*{*Cs*} := *Val v* : *T′* **by** *fact*+
from *wte* **have** *P* ⊢ *last Cs′ has least F*:*T′ via Cs* **by** (*auto split*:*if-split-asm*)
with *field* **have** *eq*:*T* = *T′* **by** (*rule sees-field-fun*)
with *casts wte wf* **have** *conf*:*P*,*h* ⊢ *v′* :≤ *T′*
by(*auto intro*:*casts-conf*)
from *hconf ha* **have** *oconf*:*P*,*h* ⊢ (*D*,*S*) √ **by** (*fastforce simp*:*hconf-def*)
with *S* **have** *suboD*:*Subobjs P D Ds* **by** (*fastforce simp*:*oconf-def*)
from *field* **obtain** *Bs fs ms*
where *subo*:*Subobjs P* (*last Cs′*) *Cs*
and *class*: *class P* (*last Cs*) = *Some*(*Bs*,*fs*,*ms*)
and *map*:*map-of fs F* = *Some T*
by (*auto simp*:*LeastFieldDecl-def FieldDecls-def*)
from *Ds subo* **have** *last*:*last Cs* = *last Ds*
by(*fastforce dest*:*Subobjs-nonempty intro*:*appendPath-last simp*:*appendPath-last*)
with *class* **have** *classDs*:*class P* (*last Ds*) = *Some*(*Bs*,*fs*,*ms*) **by** *simp*
with *S suboD oconf* **have** *P*,*h* ⊢ *fs′* (:≤) *map-of fs*
apply (*auto simp*:*oconf-def*)
apply (*erule allE*)
apply (*erule-tac x*=*Ds* **in** *allE*)
apply (*erule-tac x*=*fs′* **in** *allE*)
apply *clarsimp*
done
with *map conf eq* **have** *fconf*:*P*,*h* ⊢ *fs′*(*F* ↦ *v′*) (:≤) *map-of fs*
by (*simp add*:*fconf-def*)
from *oconf* **have** ∀ *Cs fs′*. (*Cs*,*fs′*) ∈ *S* ⟶ *Subobjs P D Cs* ∧
(∃ *fs Bs ms*. *class P* (*last Cs*) = *Some* (*Bs*,*fs*,*ms*) ∧
P,*h* ⊢ *fs′* (:≤) *map-of fs*)
by(*simp add*:*oconf-def*)
with *suboD classDs fconf*
have *oconf′*:∀ *Cs fs′*. (*Cs*,*fs′*) ∈ *?S′* ⟶ *Subobjs P D Cs* ∧
(∃ *fs Bs ms*. *class P* (*last Cs*) = *Some* (*Bs*,*fs*,*ms*) ∧
P,*h* ⊢ *fs′* (:≤) *map-of fs*)
by *auto*
from *oconf* **have** *all*:∀ *Cs*. *Subobjs P D Cs* ⟶ (∃!*fs′*. (*Cs*,*fs′*) ∈ *S*)
by(*simp add*:*oconf-def*)

with S **have** $\forall\, Cs.\ Subobjs\ P\ D\ Cs \longrightarrow (\exists!fs'.\ (Cs,fs') \in\ ?S')$ **by** *blast*
with *oconf'* **have** *oconf'*:$P,h \vdash (D,?S')\ \surd$
by (*simp add*:*oconf-def*)
with *hconf ha* **show** *?case* **by** (*rule hconf-upd-obj*)
next
case (*CallObj E e h l e' h' l' Copt M es*) **thus** *?case* **by** (*cases Copt*) *auto*
next
case (*CallParams E es h l es' h' l' v Copt M*) **thus** *?case* **by** (*cases Copt*) *auto*
next
case (*RedCallNull E Copt M vs h l*) **thus** *?case* **by** (*cases Copt*) *auto*
qed *auto*

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(*induct rule*:*red-reds-inducts*)
case *RedNew* **thus** *?case*
by(*fast intro*:*lconf-hext red-hext-incr*[*OF red-reds.RedNew*])
next
case (*RedLAss E V T v v' h l T'*)
have *casts*:$P \vdash T$ *casts* v *to* v' **and** *env*:$E\ V = Some\ T$
and *wt*:$P,E,h \vdash V{:=}Val\ v : T'$ **and** *lconf*:$P,h \vdash l\ (:\leq)_w\ E$ **by** *fact+*
from *wt env* **have** *eq*:$T = T'$ **by** *auto*
with *casts wt wf* **have** *conf*:$P,h \vdash v' :\leq T'$
by(*auto intro*:*casts-conf*)
with *lconf env eq* **show** *?case*
by (*simp del*:*fun-upd-apply*)(*erule lconf-upd*,*simp-all*)
next
case *RedFAss* **thus** *?case*
by(*auto intro*:*lconf-hext red-hext-incr*[*OF red-reds.RedFAss*]
simp del:*fun-upd-apply*)
next
case (*BlockRedNone E V T e h l e' h' l' T'*)
have *red*:$P,E(V \mapsto T) \vdash \langle e,(h,\ l(V :=\ None))\rangle \to \langle e',(h',\ l')\rangle$
and *IH*: $\bigwedge T''.\ [\![\ P,E(V \mapsto T),h \vdash e : T'';\ P,h \vdash l(V{:=}None)\ (:\leq)_w\ E(V \mapsto T);$
$envconf\ P\ (E(V \mapsto T))\ ]\!]$
$\Longrightarrow P,h' \vdash l'\ (:\leq)_w\ E(V \mapsto T)$
and *lconf*: $P,h \vdash l\ (:\leq)_w\ E$ **and** *wte*: $P,E,h \vdash \{V{:}T;\ e\} : T'$
and *envconf*:*envconf P E* **by** *fact+*
from *lconf-hext*[*OF lconf red-hext-incr*[*OF red*]]

have *lconf'*:$P,h' \vdash l\ (:\leq)_w\ E$ **.**
from *wte* **have** *wte'*:$P,E(V\mapsto T),h \vdash e : T'$ **and** *type*:*is-type P T*
by (*auto elim*:*WTrt.cases*)
from *envconf type* **have** *envconf'*:*envconf P* $(E(V \mapsto T))$
by(*auto simp*:*envconf-def*)
from *lconf* **have** $P,h \vdash (l(V := None))\ (:\leq)_w\ E(V\mapsto T)$
by (*simp add*:*lconf-def fun-upd-apply*)
from *IH*[*OF wte' this envconf'*] **have** $P,h' \vdash l'\ (:\leq)_w\ E(V\mapsto T)$ **.**
with *lconf'* **show** *?case*
by (*fastforce simp*:*lconf-def fun-upd-apply split*:*if-split-asm*)
next
case (*BlockRedSome E V T e h l e' h' l' v T'*)
have *red*:$P,E(V \mapsto T) \vdash \langle e,(h,\ l(V := None))\rangle \to \langle e',(h',\ l')\rangle$
and *IH*: $\bigwedge T''.\ [\![\ P,E(V \mapsto T),h \vdash e : T''; P,h \vdash l(V{:=}None)\ (:\leq)_w\ E(V \mapsto T);$
envconf P $(E(V \mapsto T))\ ]\!]$
$\Longrightarrow P,h' \vdash l'\ (:\leq)_w\ E(V \mapsto T)$
and *lconf*: $P,h \vdash l\ (:\leq)_w\ E$ **and** *wte*: $P,E,h \vdash \{V{:}T;\ e\} : T'$
and *envconf*:*envconf P E* **by** *fact+*
from *lconf-hext*[*OF lconf red-hext-incr*[*OF red*]]
have *lconf'*:$P,h' \vdash l\ (:\leq)_w\ E$ **.**
from *wte* **have** *wte'*:$P,E(V\mapsto T),h \vdash e : T'$ **and** *type*:*is-type P T*
by (*auto elim*:*WTrt.cases*)
from *envconf type* **have** *envconf'*:*envconf P* $(E(V \mapsto T))$
by(*auto simp*:*envconf-def*)
from *lconf* **have** $P,h \vdash (l(V := None))\ (:\leq)_w\ E(V\mapsto T)$
by (*simp add*:*lconf-def fun-upd-apply*)
from *IH*[*OF wte' this envconf'*] **have** $P,h' \vdash l'\ (:\leq)_w\ E(V\mapsto T)$ **.**
with *lconf'* **show** *?case*
by (*fastforce simp*:*lconf-def fun-upd-apply split*:*if-split-asm*)
next
case (*InitBlockRed E V T e h l v' e' h' l' v'' v T'*)
have *red*: $P,E(V \mapsto T) \vdash \langle e,\ (h,\ l(V{\mapsto}v'))\rangle \to \langle e',(h',\ l')\rangle$
and *IH*: $\bigwedge T''.\ [\![\ P,E(V \mapsto T),h \vdash e : T''; P,h \vdash l(V \mapsto v')\ (:\leq)_w\ E(V \mapsto T);$
envconf P $(E(V \mapsto T))\ ]\!]$
$\Longrightarrow P,h' \vdash l'\ (:\leq)_w\ E(V \mapsto T)$
and *lconf*:$P,h \vdash l\ (:\leq)_w\ E$ **and** *l'*:$l'\ V = Some\ v''$
and *wte*:$P,E,h \vdash \{V{:}T;\ V{:=}Val\ v;;\ e\} : T'$
and *casts*:$P \vdash T$ *casts v to v'* **and** *envconf*:*envconf P E* **by** *fact+*
from *lconf-hext*[*OF lconf red-hext-incr*[*OF red*]]
have *lconf'*:$P,h' \vdash l\ (:\leq)_w\ E$ **.**
from *wte* **obtain** T'' **where** *wte'*:$P,E(V\mapsto T),h \vdash e : T'$
and *wt*:$P,E(V \mapsto T),h \vdash V{:=}Val\ v : T''$
and *type*:*is-type P T*
by (*auto elim*:*WTrt.cases*)
from *envconf type* **have** *envconf'*:*envconf P* $(E(V \mapsto T))$
by(*auto simp*:*envconf-def*)
from *wt* **have** $T'' = T$ **by** *auto*

with *wf casts wt* **have** $P,h \vdash v' :\leq T$
by(*auto intro*:*casts-conf*)
with *lconf* **have** $P,h \vdash l(V \mapsto v')\ (:\leq)_w\ E(V \mapsto T)$
by −(*rule lconf-upd2*)
from *IH*[*OF wte' this envconf'*] **have** $P,h' \vdash l'\ (:\leq)_w\ E(V \mapsto T)$ **.**
with *lconf'* **show** *?case*
by (*fastforce simp*:*lconf-def fun-upd-apply split*:*if-split-asm*)
next
case (*CallObj E e h l e' h' l' Copt M es*) **thus** *?case* **by** (*cases Copt*) *auto*
next
case (*CallParams E es h l es' h' l' v Copt M*) **thus** *?case* **by** (*cases Copt*) *auto*
next
case (*RedCallNull E Copt M vs h l*) **thus** *?case* **by** (*cases Copt*) *auto*
qed *auto*

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

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

apply(*induct Vs Ts vs e rule*:*blocks-old-induct*)
apply(*simp-all add*:*hyperset-defs*)
done

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*: $P,E \vdash \langle es,(h,l)\rangle\ [\to]\ \langle es',(h',l')\rangle \Longrightarrow \lfloor dom\ l \rfloor \sqcup \mathcal{A}s\ es \sqsubseteq \lfloor dom\ l' \rfloor \sqcup \mathcal{A}s\ es'$
apply (*induct rule*:*red-reds-inducts*)
apply (*simp-all del*: *fun-upd-apply add*: *hyperset-defs*)
apply *blast*
apply *blast*
apply *blast*
apply *blast*
apply *blast*
apply *blast*
apply *blast*
apply *auto*
done

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 (*induct rule*:*red-reds-inducts*)
case *BinOpRed1* **thus** *?case* **by** (*auto elim*!: *D-mono*[*OF red-lA-incr*])

next
case *FAssRed1* **thus** *?case* **by** (*auto elim*!: *D-mono*[*OF red-lA-incr*])
next
case *CallObj* **thus** *?case* **by** (*auto elim*!: *Ds-mono*[*OF red-lA-incr*])
next
case (*RedCall h l a C S Cs M Ts' T' pns' body' Ds Ts T pns body Cs' vs bs new-body E*)
thus *?case*
apply (*auto dest*!:*select-method-wf-mdecl*[*OF wf*] *simp*:*wf-mdecl-def elim*!:*D-mono'*)
apply(*cases T'*) **apply** *auto*
by(*rule-tac A*=⌊*insert this* (*set pns*)⌋ **in** *D-mono*,*clarsimp simp*:*hyperset-defs*, *assumption*)+
next
case *RedStaticCall* **thus** *?case*
apply (*auto dest*!:*has-least-wf-mdecl*[*OF wf*] *simp*:*wf-mdecl-def elim*!:*D-mono'*)
by(*auto simp*:*hyperset-defs*)
next
case *InitBlockRed* **thus** *?case*
by(*auto simp*:*hyperset-defs elim*!:*D-mono' simp del*:*fun-upd-apply*)
next
case *BlockRedNone* **thus** *?case*
by(*auto simp*:*hyperset-defs elim*!:*D-mono' simp del*:*fun-upd-apply*)
next
case *BlockRedSome* **thus** *?case*
by(*auto simp*:*hyperset-defs elim*!:*D-mono' simp del*:*fun-upd-apply*)
next
case *SeqRed* **thus** *?case* **by** (*auto elim*!: *D-mono*[*OF red-lA-incr*])
next
case *CondRed* **thus** *?case* **by** (*auto elim*!: *D-mono*[*OF red-lA-incr*])
next
case *RedWhile* **thus** *?case* **by**(*auto simp*:*hyperset-defs elim*!:*D-mono'*)
next
case *ListRed1* **thus** *?case* **by** (*auto elim*!: *Ds-mono*[*OF red-lA-incr*])
qed (*auto simp*:*hyperset-defs*)

Combining conformance of heap and local variables:

definition *sconf* :: *prog* ⇒ *env* ⇒ *state* ⇒ *bool* (-,- ⊢ - √ [*51*,*51*,*51*]*50*) **where**
P,*E* ⊢ *s* √ ≡ *let* (*h*,*l*) = *s in P* ⊢ *h* √ ∧ *P*,*h* ⊢ *l* (:≤)$_w$ *E* ∧ *P* ⊢ *E* √

lemma *red-preserves-sconf*:
⟦ *P*,*E* ⊢ ⟨*e*,*s*⟩ → ⟨*e'*,*s'*⟩; *P*,*E*,*hp s* ⊢ *e* : *T*; *P*,*E* ⊢ *s* √; *wwf-prog P*⟧
⟹ *P*,*E* ⊢ *s'* √

by(*fastforce intro*:*red-preserves-hconf red-preserves-lconf simp add*:*sconf-def*)

lemma *reds-preserves-sconf*:
⟦ *P*,*E* ⊢ ⟨*es*,*s*⟩ [→] ⟨*es'*,*s'*⟩; *P*,*E*,*hp s* ⊢ *es* [:] *Ts*; *P*,*E* ⊢ *s* √; *wwf-prog P*⟧

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

by(*fastforce intro*:*reds-preserves-hconf reds-preserves-lconf*
simp add:*sconf-def*)

26.2 Subject reduction

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

proof(*induct Vs Ts vs e rule*:*blocks-old-induct*)
case (*5 V Vs T' Ts v vs e*)
have *length*:*length* $(V\#Vs) = length\ (T'\#Ts)$ *length* $(v\#vs) = length\ (T'\#Ts)$
and *type*:$\forall\, S \in set\ (T'\#Ts).\ is\text{-}type\ P\ S$
and *IH*:$\bigwedge E.\ [\![length\ Vs = length\ Ts;\ length\ vs = length\ Ts;$
$\forall\, S \in set\ Ts.\ is\text{-}type\ P\ S\ ]\!]$
$\Longrightarrow (P,E,h \vdash blocks\ (Vs,\ Ts,\ vs,\ e) : T) =$
$(P,E(Vs\ [\mapsto]\ Ts),h \vdash e : T \wedge$
$(\exists\, Ts'.\ map\ P \vdash typeof_h\ vs = map\ Some\ Ts' \wedge P \vdash Ts'\ [\leq]\ Ts))$ **by** *fact+*
from *type* **have** *typeT'*:*is-type P T'* **and** *type'*:$\forall\, S \in set\ Ts.\ is\text{-}type\ P\ S$
by *simp-all*
from *length* **have** *length Vs = length Ts length vs = length Ts*
by *simp-all*
from *IH*[*OF this type'*] **have** *eq*:$(P,E(V \mapsto T'),h \vdash blocks\ (Vs,Ts,vs,e) : T) =$
$(P,E(V \mapsto T')(Vs\ [\mapsto]\ Ts),h \vdash e : T \wedge$
$(\exists\, Ts'.\ map\ P \vdash typeof_h\ vs = map\ Some\ Ts' \wedge P \vdash Ts'\ [\leq]\ Ts))$ **.**
show *?case*
proof(*rule iffI*)
assume $P,E,h \vdash blocks\ (V\#Vs,T'\#Ts,v\#vs,e) : T$
then have *wt*:$P,E(V \mapsto T'),h \vdash V{:=}Val\ v : T'$
and *blocks*:$P,E(V \mapsto T'),h \vdash blocks\ (Vs,Ts,vs,e) : T$ **by** *auto*
from *blocks eq* **obtain** Ts' **where** *wte*:$P,E(V \mapsto T')(Vs\ [\mapsto]\ Ts),h \vdash e : T$
and *typeof*:*map* $P \vdash typeof_h\ vs = map\ Some\ Ts'$ **and** *subs*:$P \vdash Ts'\ [\leq]\ Ts$
by *auto*
from *wt* **obtain** T'' **where** $P \vdash typeof_h\ v = Some\ T''$ **and** $P \vdash T'' \leq T'$
by *auto*
with *wte typeof subs*
show $P,E(V\ \#\ Vs\ [\mapsto]\ T'\ \#\ Ts),h \vdash e : T \wedge$
$(\exists\, Ts'.\ map\ P \vdash typeof_h\ (v\ \#\ vs) = map\ Some\ Ts' \wedge P \vdash Ts'\ [\leq]\ (T'\ \#\ Ts))$
by *auto*
next
assume $P,E(V\ \#\ Vs\ [\mapsto]\ T'\ \#\ Ts),h \vdash e : T \wedge$
$(\exists\, Ts'.\ map\ P \vdash typeof_h\ (v\ \#\ vs) = map\ Some\ Ts' \wedge P \vdash Ts'\ [\leq]\ (T'\ \#$

Ts))
then obtain *Ts'* **where** *wte*:*P*,*E*(*V* # *Vs* [↦] *T'* # *Ts*),*h* ⊢ *e* : *T*
and *typeof*:*map P* ⊢ *typeof*$_h$ (*v* # *vs*) = *map Some Ts'*
and *subs*:*P* ⊢ *Ts'* [≤] (*T'*#*Ts*) **by** *auto*
from *subs* **obtain** *U Us* **where** *Ts'*:*Ts'* = *U*#*Us* **by**(*cases Ts'*) *auto*
with *wte typeof subs eq* **have** *blocks*:*P*,*E*(*V* ↦ *T'*),*h* ⊢ *blocks* (*Vs*,*Ts*,*vs*,*e*) : *T*
by *auto*
from *Ts' typeof subs* **have** *P* ⊢ *typeof*$_h$ *v* = *Some U*
and *P* ⊢ *U* ≤ *T'* **by** (*auto simp*:*fun-of-def*)
hence *wtval*:*P*,*E*(*V* ↦ *T'*),*h* ⊢ *V*:=*Val v* : *T'* **by** *auto*
with *blocks typeT'* **show** *P*,*E*,*h* ⊢ *blocks* (*V*#*Vs*,*T'*#*Ts*,*v*#*vs*,*e*) : *T* **by** *auto*
qed
qed *auto*

theorem assumes *wf*: *wf-C-prog P*
shows *subject-reduction2*: *P*,*E* ⊢ ⟨*e*,(*h*,*l*)⟩ → ⟨*e'*,(*h'*,*l'*)⟩ ⟹
(⋀*T*. ⟦ *P*,*E* ⊢ (*h*,*l*) √; *P*,*E*,*h* ⊢ *e* : *T* ⟧ ⟹ *P*,*E*,*h'* ⊢ *e'* :$_{NT}$ *T*)
and *subjects-reduction2*: *P*,*E* ⊢ ⟨*es*,(*h*,*l*)⟩ [→] ⟨*es'*,(*h'*,*l'*)⟩ ⟹
(⋀*Ts*.⟦ *P*,*E* ⊢ (*h*,*l*) √; *P*,*E*,*h* ⊢ *es* [:] *Ts* ⟧ ⟹ *types-conf P E h' es' Ts*)

proof (*induct rule*:*red-reds-inducts*)
case (*RedNew h a h' C E l*)
have *new*:*new-Addr h* = *Some a* **and** *h'*:*h'* = *h*(*a* ↦ (*C*, *Collect* (*init-obj P C*)))

and *wt*:*P*,*E*,*h* ⊢ *new C* : *T* **by** *fact*+
from *wt* **have** *eq*:*T* = *Class C* **and** *class*: *is-class P C* **by** *auto*
from *class* **have** *subo*:*Subobjs P C* [*C*] **by**(*rule Subobjs-Base*)
from *h'* **have** *h' a* = *Some*(*C*, *Collect* (*init-obj P C*)) **by**(*simp add*:*map-upd-Some-unfold*)
with *subo* **have** *P*,*E*,*h'* ⊢ *ref*(*a*,[*C*]) : *Class C* **by** *auto*
with *eq* **show** *?case* **by** *auto*
next
case (*RedNewFail h E C l*)
have *sconf*:*P*,*E* ⊢ (*h*, *l*) √ **by** *fact*
from *wf* **have** *is-class P OutOfMemory*
by (*fastforce intro*:*is-class-xcpt wf-prog-wwf-prog*)
hence *preallocated h* ⟹ *P* ⊢ *typeof*$_h$ (*Ref* (*addr-of-sys-xcpt OutOfMemory*,[*OutOfMemory*]))
= *Some*(*Class OutOfMemory*)
by (*auto elim*: *preallocatedE dest*!:*preallocatedD Subobjs-Base*)
with *sconf* **have** *P*,*E*,*h* ⊢ *THROW OutOfMemory* : *T* **by**(*auto simp*:*sconf-def*
hconf-def)
thus *?case* **by** (*fastforce intro*:*wt-same-type-typeconf*)
next
case (*StaticCastRed E e h l e' h' l' C*)
have *wt*:*P*,*E*,*h* ⊢ (|*C*|)*e* : *T*
and *IH*:⋀*T'*. ⟦*P*,*E* ⊢ (*h*,*l*) √; *P*,*E*,*h* ⊢ *e* : *T'*⟧
⟹ *P*,*E*,*h'* ⊢ *e'* :$_{NT}$ *T'*

and *sconf*:P,$E \vdash (h,\ l) \surd$ **by** *fact*+
from *wt* **obtain** T' **where** *wte*:P,E,$h \vdash e : T'$ **and** *isref*:*is-refT* T'
and *class*: *is-class* P C **and** T:T = *Class* C
by *auto*
from *isref* **have** P,E,$h' \vdash (\!|C|\!)e' :$ *Class* C
proof(*rule refTE*)
assume $T' = NT$
with *IH*[*OF sconf wte*] *isref class* **show** *?thesis* **by** *auto*
next
fix D **assume** $T' =$ *Class* D
with *IH*[*OF sconf wte*] *isref class* **show** *?thesis* **by** *auto*
qed
with T **show** *?case* **by** (*fastforce intro*:*wt-same-type-typeconf*)
next
case *RedStaticCastNull*
thus *?case* **by** (*auto elim*:*WTrt.cases*)
next
case (*RedStaticUpCast Cs C Cs' Ds E a h l*)
have *wt*:P,E,$h \vdash (\!|C|\!)$*ref* (a,*Cs*) : T
and *path-via*:$P \vdash$ *Path last Cs to C via Cs'*
and *Ds*:*Ds* = *Cs* @$_p$ *Cs'* **by** *fact*+
from *wt* **have** *typeof*:$P \vdash$ *typeof*$_h$ (*Ref*(a,*Cs*)) = *Some*(*Class*(*last Cs*))
and *class*: *is-class* P C **and** T:T = *Class* C
by *auto*
from *typeof* **obtain** D S **where** *h*:h a = *Some*(D,S) **and** *subo*:*Subobjs* P D *Cs*
by (*auto dest*:*typeof-Class-Subo split*:*if-split-asm*)
from *path-via subo wf Ds* **have** *Subobjs* P D *Ds* **and** *last*:*last Ds* = C
by(*auto intro*!:*Subobjs-appendPath appendPath-last*[*THEN sym*] *Subobjs-nonempty*
simp:*path-via-def*)
with h **have** P,E,$h \vdash$ *ref* (a,*Ds*) : *Class* C **by** *auto*
with T **show** *?case* **by** (*fastforce intro*:*wt-same-type-typeconf*)
next
case (*RedStaticDownCast E C a Cs Cs' h l*)
have P,E,$h \vdash (\!|C|\!)$*ref* (a,*Cs*@[C]@*Cs'*) : T **by** *fact*
hence *typeof*:$P \vdash$ *typeof*$_h$ (*Ref*(a,*Cs*@[C]@*Cs'*)) = *Some*(*Class*(*last*(*Cs*@[C]@*Cs'*)))
and *class*: *is-class* P C **and** T:T = *Class* C
by *auto*
from *typeof* **obtain** D S **where** *h*:h a = *Some*(D,S)
and *subo*:*Subobjs* P D (*Cs*@[C]@*Cs'*)
by (*auto dest*:*typeof-Class-Subo split*:*if-split-asm*)
from *subo* **have** *Subobjs* P D (*Cs*@[C]) **by**(*fastforce intro*:*appendSubobj*)
with h **have** P,E,$h \vdash$ *ref* (a,*Cs*@[C]) : *Class* C **by** *auto*
with T **show** *?case* **by** (*fastforce intro*:*wt-same-type-typeconf*)
next
case (*RedStaticCastFail C Cs E a h l*)
have *sconf*:P,$E \vdash (h,\ l) \surd$ **by** *fact*
from *wf* **have** *is-class* P *ClassCast*
by (*fastforce intro*:*is-class-xcpt wf-prog-wwf-prog*)
hence *preallocated* $h \Longrightarrow P \vdash$ *typeof*$_h$ (*Ref* (*addr-of-sys-xcpt ClassCast*,[*ClassCast*]))

$= Some(Class\ ClassCast)$
by (*auto elim*: *preallocatedE dest*!:*preallocatedD Subobjs-Base*)
with *sconf* **have** $P,E,h \vdash THROW\ ClassCast : T$ **by**(*auto simp*:*sconf-def hconf-def*)
thus *?case* **by** (*fastforce intro*:*wt-same-type-typeconf*)
next
case (*DynCastRed E e h l e' h' l' C*)
have *wt*:$P,E,h \vdash Cast\ C\ e : T$
and *IH*:$\bigwedge T'.\ [\![P,E \vdash (h,l)\ \surd;\ P,E,h \vdash e : T']\!] \Longrightarrow P,E,h' \vdash e' :_{NT} T'$
and *sconf*:$P,E \vdash (h,l)\ \surd$ **by** *fact+*
from *wt* **obtain** T' **where** *wte*:$P,E,h \vdash e : T'$ **and** *isref*:*is-refT* T'
and *class*: *is-class P C* **and** T:$T = Class\ C$
by *auto*
from *isref* **have** $P,E,h' \vdash Cast\ C\ e' : Class\ C$
proof(*rule refTE*)
assume $T' = NT$
with *IH*[*OF sconf wte*] *isref class* **show** *?thesis* **by** *auto*
next
fix D **assume** $T' = Class\ D$
with *IH*[*OF sconf wte*] *isref class* **show** *?thesis* **by** *auto*
qed
with T **show** *?case* **by** (*fastforce intro*:*wt-same-type-typeconf*)
next
case *RedDynCastNull*
thus *?case* **by** (*auto elim*:*WTrt.cases*)
next
case (*RedDynCast h l a D S C Cs' E Cs*)
have *wt*:$P,E,h \vdash Cast\ C\ (ref\ (a,Cs)) : T$
and *path-via*:$P \vdash Path\ D\ to\ C\ via\ Cs'$
and *hp*:*hp* $(h,l)\ a = Some(D,S)$ **by** *fact+*
from *wt* **have** *typeof*:$P \vdash typeof_h\ (Ref(a,Cs)) = Some(Class(last\ Cs))$
and *class*: *is-class P C* **and** T:$T = Class\ C$
by *auto*
from *typeof hp* **have** *subo*:*Subobjs P D Cs*
by (*auto dest*:*typeof-Class-Subo split*:*if-split-asm*)
from *path-via subo* **have** *Subobjs P D Cs'*
and *last*:*last Cs'* $= C$ **by** (*auto simp*:*path-via-def*)
with *hp* **have** $P,E,h \vdash ref\ (a,Cs') : Class\ C$ **by** *auto*
with T **show** *?case* **by** (*fastforce intro*:*wt-same-type-typeconf*)
next
case (*RedStaticUpDynCast Cs C Cs' Ds E a h l*)
have *wt*:$P,E,h \vdash Cast\ C\ (ref\ (a,Cs)) : T$
and *path-via*:$P \vdash Path\ last\ Cs\ to\ C\ via\ Cs'$
and *Ds*:$Ds = Cs\ @_p\ Cs'$ **by** *fact+*
from *wt* **have** *typeof*:$P \vdash typeof_h\ (Ref(a,Cs)) = Some(Class(last\ Cs))$
and *class*: *is-class P C* **and** T:$T = Class\ C$
by *auto*
from *typeof* **obtain** $D\ S$ **where** *h*:$h\ a = Some(D,S)$ **and** *subo*:*Subobjs P D Cs*
by (*auto dest*:*typeof-Class-Subo split*:*if-split-asm*)

from *path-via subo wf Ds* **have** *Subobjs P D Ds* **and** *last*:*last Ds* = *C*
by(*auto intro*!:*Subobjs-appendPath appendPath-last*[*THEN sym*] *Subobjs-nonempty*
simp:*path-via-def*)
with *h* **have** *P*,*E*,*h* ⊢ *ref* (*a*,*Ds*) : *Class C* **by** *auto*
with *T* **show** *?case* **by** (*fastforce intro*:*wt-same-type-typeconf*)
next
case (*RedStaticDownDynCast E C a Cs Cs*′ *h l*)
have *P*,*E*,*h* ⊢ *Cast C* (*ref* (*a*,*Cs*@[*C*]@*Cs*′)) : *T* **by** *fact*
hence *typeof*:*P* ⊢ *typeof*$_h$ (*Ref*(*a*,*Cs*@[*C*]@*Cs*′)) = *Some*(*Class*(*last*(*Cs*@[*C*]@*Cs*′)))
and *class*: *is-class P C* **and** *T*:*T* = *Class C*
by *auto*
from *typeof* **obtain** *D S* **where** *h*:*h a* = *Some*(*D*,*S*)
and *subo*:*Subobjs P D* (*Cs*@[*C*]@*Cs*′)
by (*auto dest*:*typeof-Class-Subo split*:*if-split-asm*)
from *subo* **have** *Subobjs P D* (*Cs*@[*C*]) **by**(*fastforce intro*:*appendSubobj*)
with *h* **have** *P*,*E*,*h* ⊢ *ref* (*a*,*Cs*@[*C*]) : *Class C* **by** *auto*
with *T* **show** *?case* **by** (*fastforce intro*:*wt-same-type-typeconf*)
next
case *RedDynCastFail* **thus** *?case* **by** *fastforce*
next
case (*BinOpRed1 E e h l e*′ *h*′ *l*′ *bop e*$_2$)
have *red*:*P*,*E* ⊢ ⟨*e*,(*h*, *l*)⟩ → ⟨*e*′,(*h*′, *l*′)⟩
and *wt*:*P*,*E*,*h* ⊢ *e* ≪*bop*≫ *e*$_2$: *T*
and *IH*:⋀*T*′. ⟦*P*,*E* ⊢ (*h*,*l*) √; *P*,*E*,*h* ⊢ *e* : *T*′⟧
⟹ *P*,*E*,*h*′ ⊢ *e*′ :$_{NT}$ *T*′
and *sconf*:*P*,*E* ⊢ (*h*,*l*) √ **by** *fact*+
from *wt* **obtain** *T*$_1$ *T*$_2$ **where** *wte*:*P*,*E*,*h* ⊢ *e* : *T*$_1$ **and** *wte2*:*P*,*E*,*h* ⊢ *e*$_2$: *T*$_2$
and *binop*:*case bop of Eq* ⇒ *T* = *Boolean*
| *Add* ⇒ *T*$_1$ = *Integer* ∧ *T*$_2$ = *Integer* ∧ *T* = *Integer*
by *auto*
from *WTrt-hext-mono*[*OF wte2 red-hext-incr*[*OF red*]] **have** *wte2*′:*P*,*E*,*h*′ ⊢ *e*$_2$
: *T*$_2$ **.**
have *P*,*E*,*h*′ ⊢ *e*′ ≪*bop*≫ *e*$_2$: *T*
proof (*cases bop*)
assume *Eq*:*bop* = *Eq*
from *IH*[*OF sconf wte*] **obtain** *T*′ **where** *P*,*E*,*h*′ ⊢ *e*′ : *T*′
by (*cases T*$_1$) *auto*
with *wte2*′ *binop Eq* **show** *?thesis* **by**(*cases bop*) *auto*
next
assume *Add*:*bop* = *Add*
with *binop* **have** *Intg*:*T*$_1$ = *Integer* **by** *simp*
with *IH*[*OF sconf wte*] **have** *P*,*E*,*h*′ ⊢ *e*′ : *Integer* **by** *simp*
with *wte2*′ *binop Add* **show** *?thesis* **by**(*cases bop*) *auto*
qed
with *binop* **show** *?case* **by**(*cases bop*) *simp-all*
next
case (*BinOpRed2 E e h l e*′ *h*′ *l*′ *v*$_1$ *bop*)
have *red*:*P*,*E* ⊢ ⟨*e*,(*h*,*l*)⟩ → ⟨*e*′,(*h*′,*l*′)⟩
and *wt*:*P*,*E*,*h* ⊢ *Val v*$_1$ ≪*bop*≫ *e* : *T*

and *IH*:$\bigwedge T'$. $[\![P,E \vdash (h,l)\ \surd;\ P,E,h \vdash e : T']\!]$
$\Longrightarrow P,E,h' \vdash e' :_{NT} T'$
and *sconf*:$P,E \vdash (h,l)\ \surd$ **by** *fact+*
from *wt* **obtain** T_1 T_2 **where** *wtval*:$P,E,h \vdash Val\ v_1 : T_1$ **and** *wte*:$P,E,h \vdash e : T_2$
and *binop*:*case bop of Eq* $\Rightarrow T = Boolean$
$\mid Add \Rightarrow T_1 = Integer \wedge T_2 = Integer \wedge T = Integer$
by *auto*
from *WTrt-hext-mono*[*OF wtval red-hext-incr*[*OF red*]]
have *wtval'*:$P,E,h' \vdash Val\ v_1 : T_1$ **.**
have $P,E,h' \vdash Val\ v_1 \ll bop \gg e' : T$
proof (*cases bop*)
assume *Eq*:*bop* = *Eq*
from *IH*[*OF sconf wte*] **obtain** T' **where** $P,E,h' \vdash e' : T'$
by (*cases* T_2) *auto*
with *wtval' binop Eq* **show** *?thesis* **by**(*cases bop*) *auto*
next
assume *Add*:*bop* = *Add*
with *binop* **have** *Intg*:T_2 = *Integer* **by** *simp*
with *IH*[*OF sconf wte*] **have** $P,E,h' \vdash e'$: *Integer* **by** *simp*
with *wtval' binop Add* **show** *?thesis* **by**(*cases bop*) *auto*
qed
with *binop* **show** *?case* **by**(*cases bop*) *simp-all*
next
case (*RedBinOp bop* v_1 v_2 *v E a b*) **thus** *?case*
proof (*cases bop*)
case *Eq* **thus** *?thesis* **using** *RedBinOp* **by** *auto*
next
case *Add* **thus** *?thesis* **using** *RedBinOp* **by** *auto*
qed
next
case (*RedVar h l V v E*)
have *l*:*lcl* $(h,\ l)$ V = *Some v* **and** *sconf*:$P,E \vdash (h,\ l)\ \surd$
and *wt*:$P,E,h \vdash Var\ V : T$ **by** *fact+*
hence *conf*:$P,h \vdash v :\leq T$ **by**(*force simp*:*sconf-def lconf-def*)
show *?case*
proof(*cases* $\forall C.\ T \neq Class\ C$)
case *True*
with *conf* **have** $P \vdash typeof_h\ v = Some\ T$ **by**(*cases T*) *auto*
hence $P,E,h \vdash Val\ v : T$ **by** *auto*
thus *?thesis* **by**(*rule wt-same-type-typeconf*)
next
case *False*
then obtain *C* **where** *T*:$T = Class\ C$ **by** *auto*
with *conf* **have** $P \vdash typeof_h\ v = Some(Class\ C) \vee P \vdash typeof_h\ v = Some\ NT$
by *simp*
with *T* **show** *?thesis* **by** *simp*
qed
next

```
  case (LAssRed E e h l e' h' l' V)
  have wt:P,E,h ⊢ V:=e : T and sconf:P,E ⊢ (h, l) √
    and IH:⋀T'. ⟦P,E ⊢ (h, l) √; P,E,h ⊢ e : T'⟧ ⟹ P,E,h' ⊢ e' :_NT T' by
fact+
  from wt obtain T' where wte:P,E,h ⊢ e : T' and env:E V = Some T
    and sub:P ⊢ T' ≤ T by auto
  from sconf env have is-type P T by(auto simp:sconf-def envconf-def)
  from sub this wf show ?case
  proof(rule subE)
    assume eq:T' = T and notclass:∀ C. T' ≠ Class C
    with IH[OF sconf wte] have P,E,h' ⊢ e' : T by(cases T) auto
    with eq env have P,E,h' ⊢ V:=e' : T by auto
    with eq show ?thesis by(cases T) auto
  next
    fix C D
    assume T':T' = Class C and T:T = Class D
      and path-unique:P ⊢ Path C to D unique
    with IH[OF sconf wte] have P,E,h' ⊢ e' : Class C ∨ P,E,h' ⊢ e' : NT
      by simp
    hence P,E,h' ⊢ V:=e' : T
    proof(rule disjE)
      assume P,E,h' ⊢ e' : Class C
      with env T' sub show ?thesis by (fastforce intro:WTrtLAss)
    next
      assume P,E,h' ⊢ e' : NT
      with env T show ?thesis by (fastforce intro:WTrtLAss)
    qed
    with T show ?thesis by(cases T) auto
  next
    fix C
    assume T':T' = NT and T:T = Class C
    with IH[OF sconf wte] have P,E,h' ⊢ e' : NT by simp
    with env T show ?thesis by (fastforce intro:WTrtLAss)
  qed
next
  case (RedLAss E V T v v' h l T')
  have env:E V = Some T and casts:P ⊢ T casts v to v'
    and sconf:P,E ⊢ (h, l) √ and wt:P,E,h ⊢ V:=Val v : T' by fact+
  show ?case
  proof(cases ∀ C. T ≠ Class C)
    case True
    with casts wt env show ?thesis
      by(cases T',auto elim!:casts-to.cases)
  next
    case False
    then obtain C where T = Class C by auto
    with casts wt env wf show ?thesis
      by(auto elim!:casts-to.cases,
        auto intro!:sym[OF appendPath-last] Subobjs-nonempty split:if-split-asm
```

```
        simp:path-via-def,drule-tac Cs=Cs in Subobjs-appendPath,auto)
  qed
next
  case (FAccRed E e h l e' h' l' F Cs)
  have red:P,E ⊢ ⟨e,(h,l)⟩ → ⟨e',(h',l')⟩
    and wt:P,E,h ⊢ e•F{Cs} : T
    and IH:⋀T'. ⟦P,E ⊢ (h,l) √; P,E,h ⊢ e : T'⟧
          ⟹ P,E,h' ⊢ e' :_NT T'
    and sconf:P,E ⊢ (h,l) √ by fact+
  from wt have P,E,h' ⊢ e'•F{Cs} : T
  proof(rule WTrt-elim-cases)
    fix C assume wte: P,E,h ⊢ e : Class C
      and field:P ⊢ C has least F:T via Cs
      and notemptyCs:Cs ≠ []
    from field have class: is-class P C
     by (fastforce intro:Subobjs-isClass simp add:LeastFieldDecl-def FieldDecls-def)
    from IH[OF sconf wte] have P,E,h' ⊢ e' : NT ∨ P,E,h' ⊢ e' : Class C by
auto
    thus ?thesis
    proof(rule disjE)
      assume P,E,h' ⊢ e' : NT
      thus ?thesis by (fastforce intro!:WTrtFAccNT)
    next
      assume wte':P,E,h' ⊢ e' : Class C
      from wte' notemptyCs field show ?thesis by(rule WTrtFAcc)
    qed
  next
    assume wte: P,E,h ⊢ e : NT
    from IH[OF sconf wte] have P,E,h' ⊢ e' : NT by auto
    thus ?thesis by (rule WTrtFAccNT)
  qed
  thus ?case by(rule wt-same-type-typeconf)
next
  case (RedFAcc h l a D S Ds Cs' Cs fs' F v E)
  have h:hp (h,l) a = Some(D,S)
    and Ds:Ds = Cs'@_pCs and S:(Ds,fs') ∈ S
    and fs':fs' F = Some v and sconf:P,E ⊢ (h,l) √
    and wte:P,E,h ⊢ ref (a,Cs')•F{Cs} : T by fact+
  from wte have field:P ⊢ last Cs' has least F:T via Cs
    and notemptyCs:Cs ≠ []
    by (auto split:if-split-asm)
  from h S sconf obtain Bs fs ms where classDs:class P (last Ds) = Some
(Bs,fs,ms)
    and fconf:P,h ⊢ fs' (:≤) map-of fs
    by (simp add:sconf-def hconf-def oconf-def) blast
  from field Ds have last Cs = last Ds
    by (fastforce intro!:appendPath-last Subobjs-nonempty
                  simp:LeastFieldDecl-def FieldDecls-def)
  with field classDs have map:map-of fs F =  Some T
```

by (*simp add*:*LeastFieldDecl-def FieldDecls-def*)
with *fconf fs'* **have** *conf*:$P,h \vdash v :\leq T$
by (*simp add*:*fconf-def*,*erule-tac x*=*F* **in** *allE*,*fastforce*)
thus *?case* **by** (*cases T*) *auto*
next
case (*RedFAccNull E F Cs h l*)
have *sconf*:$P,E \vdash (h,\ l)\ \surd$ **by** *fact*
from *wf* **have** *is-class P NullPointer*
by (*fastforce intro*:*is-class-xcpt wf-prog-wwf-prog*)
hence *preallocated h* $\Longrightarrow P \vdash typeof_h\ (Ref\ (addr\text{-}of\text{-}sys\text{-}xcpt\ NullPointer,[NullPointer]))$
$= Some(Class\ NullPointer)$
by (*auto elim*: *preallocatedE dest*!:*preallocatedD Subobjs-Base*)
with *sconf* **have** $P,E,h \vdash THROW\ NullPointer : T$ **by**(*auto simp*:*sconf-def hconf-def*)
thus *?case* **by** (*fastforce intro*:*wt-same-type-typeconf wf-prog-wwf-prog*)
next
case (*FAssRed1 E e h l e' h' l' F Cs* e_2)
have *red*:$P,E \vdash \langle e,(h,l)\rangle \rightarrow \langle e',(h',l')\rangle$
and *wt*:$P,E,h \vdash e{\cdot}F\{Cs\} := e_2 : T$
and *IH*:$\bigwedge T'.\ [\![P,E \vdash (h,l)\ \surd;\ P,E,h \vdash e : T']\!]$
$\Longrightarrow P,E,h' \vdash e' :_{NT} T'$
and *sconf*:$P,E \vdash (h,l)\ \surd$ **by** *fact+*
from *wt* **have** $P,E,h' \vdash e'{\cdot}F\{Cs\} := e_2 : T$
proof (*rule WTrt-elim-cases*)
fix *C T'* **assume** *wte*: $P,E,h \vdash e : Class\ C$
and *field*:$P \vdash C$ *has least* F:T *via Cs*
and *notemptyCs*:$Cs \neq []$
and *wte2*:$P,E,h \vdash e_2 : T'$ **and** *sub*:$P \vdash T' \leq T$
have *wte2'*: $P,E,h' \vdash e_2 : T'$
by(*rule WTrt-hext-mono*[*OF wte2 red-hext-incr*[*OF red*]])
from *IH*[*OF sconf wte*] **have** $P,E,h' \vdash e' : Class\ C \vee P,E,h' \vdash e' : NT$
by *simp*
thus *?thesis*
proof(*rule disjE*)
assume *wte'*:$P,E,h' \vdash e' : Class\ C$
from *wte' notemptyCs field wte2' sub* **show** *?thesis* **by** (*rule WTrtFAss*)
next
assume *wte'*:$P,E,h' \vdash e' : NT$
from *wte' wte2' sub* **show** *?thesis* **by** (*rule WTrtFAssNT*)
qed
next
fix *T'* **assume** *wte*:$P,E,h \vdash e : NT$
and *wte2*:$P,E,h \vdash e_2 : T'$ **and** *sub*:$P \vdash T' \leq T$
have *wte2'*: $P,E,h' \vdash e_2 : T'$
by(*rule WTrt-hext-mono*[*OF wte2 red-hext-incr*[*OF red*]])
from *IH*[*OF sconf wte*] **have** *wte'*:$P,E,h' \vdash e' : NT$ **by** *simp*
from *wte' wte2' sub* **show** *?thesis* **by** (*rule WTrtFAssNT*)
qed
thus *?case* **by**(*rule wt-same-type-typeconf*)

next
case (*FAssRed2 E e h l e′ h′ l′ v F Cs*)
have *red*:$P,E \vdash \langle e,(h,l)\rangle \to \langle e',(h',l')\rangle$
and *wt*:$P,E,h \vdash Val\ v{\cdot}F\{Cs\} := e : T$
and *IH*:$\bigwedge T'.\ [\![P,E \vdash (h,l)\ \surd;\ P,E,h \vdash e : T']\!] \Longrightarrow P,E,h' \vdash e' :_{NT} T'$
and *sconf*:$P,E \vdash (h,l)\ \surd$ **by** *fact+*
from *wt* **have** $P,E,h' \vdash Val\ v{\cdot}F\{Cs\}:=e' : T$
proof (*rule WTrt-elim-cases*)
fix $C\ T'$ **assume** *wtval*:$P,E,h \vdash Val\ v : Class\ C$
and *field*:$P \vdash C$ *has least* $F{:}T$ *via Cs*
and *notemptyCs*:$Cs \neq []$
and *wte*:$P,E,h \vdash e : T'$
and *sub*:$P \vdash T' \leq T$
have *wtval′*:$P,E,h' \vdash Val\ v : Class\ C$
by(*rule WTrt-hext-mono*[*OF wtval red-hext-incr*[*OF red*]])
from *field wf* **have** *type*:*is-type P T* **by**(*rule least-field-is-type*)
from *sub type wf* **show** *?thesis*
proof(*rule subE*)
assume $T' = T$ **and** *notclass*:$\forall C.\ T' \neq Class\ C$
from *IH*[*OF sconf wte*] *notclass* **have** *wte′*:$P,E,h' \vdash e' : T'$
by(*cases T′*) *auto*
from *wtval′ notemptyCs field wte′ sub* **show** *?thesis*
by(*rule WTrtFAss*)
next
fix $C'\ D$ **assume** $T'{:}T' = Class\ C'$ **and** $T{:}T = Class\ D$
and *path-unique*:$P \vdash Path\ C'$ *to* D *unique*
from *IH*[*OF sconf wte*] T' **have** $P,E,h' \vdash e' : Class\ C' \vee P,E,h' \vdash e' : NT$
by *simp*
thus *?thesis*
proof(*rule disjE*)
assume *wte′*:$P,E,h' \vdash e' : Class\ C'$
from *wtval′ notemptyCs field wte′ sub* T' **show** *?thesis*
by (*fastforce intro*: *WTrtFAss*)
next
assume *wte′*:$P,E,h' \vdash e' : NT$
from *wtval′ notemptyCs field wte′ sub* T **show** *?thesis*
by (*fastforce intro*: *WTrtFAss*)
qed
next
fix C' **assume** $T'{:}T' = NT$ **and** $T{:}T = Class\ C'$
from *IH*[*OF sconf wte*] T' **have** *wte′*:$P,E,h' \vdash e' : NT$ **by** *simp*
from *wtval′ notemptyCs field wte′ sub* T **show** *?thesis*
by (*fastforce intro*: *WTrtFAss*)
qed
next
fix T' **assume** *wtval*:$P,E,h \vdash Val\ v : NT$
and *wte*:$P,E,h \vdash e : T'$
and *sub*:$P \vdash T' \leq T$

```
    have wtval':P,E,h' ⊢ Val v : NT
      by(rule WTrt-hext-mono[OF wtval red-hext-incr[OF red]])
    from IH[OF sconf wte] sub obtain T'' where wte':P,E,h' ⊢ e' : T''
      and sub':P ⊢ T'' ≤ T by (cases T',auto,cases T,auto)
    from wtval' wte' sub' show ?thesis
      by(rule WTrtFAssNT)
  qed
  thus ?case by(rule wt-same-type-typeconf)
next
  case (RedFAss h a D S Cs' F T Cs v v' Ds fs E l T')
  let ?fs' = fs(F ↦ v')
  let ?S' = insert (Ds, ?fs') (S − {(Ds, fs)})
  let ?h' = h(a ↦ (D,?S'))
  have h:h a = Some(D,S) and casts:P ⊢ T casts v to v'
    and field:P ⊢ last Cs' has least F:T via Cs
    and wt:P,E,h ⊢ ref (a,Cs')·F{Cs} := Val v : T' by fact+
  from wt wf have type:is-type P T'
    by (auto dest:least-field-is-type split:if-split-asm)
  from wt field obtain T'' where wtval:P,E,h ⊢ Val v : T'' and eq:T = T'
    and leq:P ⊢ T'' ≤ T'
    by (auto dest:sees-field-fun split:if-split-asm)
  from casts eq wtval show ?case
  proof(induct rule:casts-to.induct)
    case (casts-prim T₀ w)
    have T₀ = T' and ∀ C. T₀ ≠ Class C and wtval':P,E,h ⊢ Val w : T'' by
fact+
    with leq have T' = T'' by(cases T',auto)
    with wtval' have P,E,h ⊢ Val w : T' by simp
    with h have P,E,(h(a↦(D,insert(Ds,fs(F ↦ w))(S−{(Ds,fs)})))) ⊢ Val w :
T'
      by(cases w,auto split:if-split-asm)
    thus P,E,(h(a↦(D,insert(Ds,fs(F ↦ w))(S−{(Ds,fs)})))) ⊢ (Val w) :_NT T'
      by(rule wt-same-type-typeconf)
  next
    case (casts-null C'')
    have T':Class C'' = T' by fact
    have P,E,(h(a↦(D,insert(Ds,fs(F ↦ Null))(S−{(Ds,fs)})))) ⊢ null : NT
      by simp
    with sym[OF T']
    show P,E,(h(a↦(D,insert(Ds,fs(F ↦ Null))(S−{(Ds,fs)})))) ⊢ null :_NT T'
      by simp
  next
    case (casts-ref Xs C'' Xs' Ds'' a')
    have Class C'' = T' and Ds'' = Xs @_p Xs'
      and P ⊢ Path last Xs to C'' via Xs'
      and P,E,h ⊢ ref (a', Xs) : T'' by fact+
    with wf have P,E,h ⊢ ref (a',Ds'') : T'
      by (auto intro!:appendPath-last[THEN sym] Subobjs-nonempty
        split:if-split-asm simp:path-via-def,
```

```
      drule-tac Cs=Xs in Subobjs-appendPath,auto)
    with h have P,E,(h(a↦(D,insert(Ds,fs(F ↦ Ref(a',Ds'')))(S−{(Ds,fs)}))))
⊢
     ref (a',Ds'') : T'
     by auto
    thus P,E,(h(a↦(D,insert(Ds,fs(F ↦ Ref(a',Ds'')))(S−{(Ds,fs)})))) ⊢
     ref (a',Ds'') :NT T'
     by(rule wt-same-type-typeconf)
  qed
next
  case (RedFAssNull E F Cs v h l)
  have sconf:P,E ⊢ (h, l) √ by fact
  from wf have is-class P NullPointer
    by (fastforce intro:is-class-xcpt wf-prog-wwf-prog)
 hence preallocated h ⟹ P ⊢ typeof_h (Ref (addr-of-sys-xcpt NullPointer,[NullPointer]))
= Some(Class NullPointer)
    by (auto elim: preallocatedE dest!:preallocatedD Subobjs-Base)
   with sconf have P,E,h ⊢ THROW NullPointer : T by(auto simp:sconf-def
hconf-def)
  thus ?case by (fastforce intro:wt-same-type-typeconf wf-prog-wwf-prog)
next
  case (CallObj E e h l e' h' l' Copt M es)
  have red: P,E ⊢ ⟨e,(h,l)⟩ → ⟨e',(h',l')⟩
   and IH: ⋀T'. ⟦P,E ⊢ (h,l) √; P,E,h ⊢ e : T'⟧
              ⟹ P,E,h' ⊢ e' :NT T'
   and sconf: P,E ⊢ (h,l) √ and wt: P,E,h ⊢ Call e Copt M es : T by fact+
  from wt have P,E,h' ⊢ Call e' Copt M es : T
  proof(cases Copt)
    case None
    with wt have P,E,h ⊢ e·M(es) : T by simp
    hence P,E,h' ⊢ e'·M(es) : T
    proof(rule WTrt-elim-cases)
      fix C Cs Ts Ts' m
      assume wte:P,E,h ⊢ e : Class C
        and method:P ⊢ C has least M = (Ts, T, m) via Cs
        and wtes:P,E,h ⊢ es [:] Ts' and subs: P ⊢ Ts' [≤] Ts
      from IH[OF sconf wte] have P,E,h' ⊢ e' : NT ∨ P,E,h' ⊢ e' : Class C by
auto
      thus ?thesis
      proof(rule disjE)
        assume wte':P,E,h' ⊢ e' : NT
        have P,E,h' ⊢ es [:] Ts'
          by(rule WTrts-hext-mono[OF wtes red-hext-incr[OF red]])
        with wte' show ?thesis by(rule WTrtCallNT)
      next
        assume wte':P,E,h' ⊢ e' : Class C
        have wtes':P,E,h' ⊢ es [:] Ts'
          by(rule WTrts-hext-mono[OF wtes red-hext-incr[OF red]])
        from wte' method wtes' subs show ?thesis by(rule WTrtCall)
```

```
    qed
  next
    fix Ts
    assume wte:P,E,h ⊢ e : NT and wtes:P,E,h ⊢ es [:] Ts
    from IH[OF sconf wte] have wte':P,E,h' ⊢ e' : NT by simp
    have P,E,h' ⊢ es [:] Ts
      by(rule WTrts-hext-mono[OF wtes red-hext-incr[OF red]])
    with wte' show ?thesis by(rule WTrtCallNT)
  qed
  with None show ?thesis by simp
 next
  case (Some C)
  with wt have P,E,h ⊢ e•(C::)M(es) : T by simp
  hence P,E,h' ⊢ e'•(C::)M(es) : T
  proof(rule WTrt-elim-cases)
    fix C' Cs Ts Ts' m
   assume wte:P,E,h ⊢ e : Class C' and path-unique:P ⊢ Path C' to C unique
     and method:P ⊢ C has least M = (Ts, T, m) via Cs
     and wtes:P,E,h ⊢ es [:] Ts' and subs: P ⊢ Ts' [≤] Ts
    from IH[OF sconf wte] have P,E,h' ⊢ e' : NT ∨ P,E,h' ⊢ e' : Class C' by
auto
    thus ?thesis
    proof(rule disjE)
      assume wte':P,E,h' ⊢ e' : NT
      have P,E,h' ⊢ es [:] Ts'
        by(rule WTrts-hext-mono[OF wtes red-hext-incr[OF red]])
      with wte' show ?thesis by(rule WTrtCallNT)
    next
      assume wte':P,E,h' ⊢ e' : Class C'
      have wtes':P,E,h' ⊢ es [:] Ts'
        by(rule WTrts-hext-mono[OF wtes red-hext-incr[OF red]])
      from wte' path-unique method wtes' subs show ?thesis by(rule WTrtStat-
icCall)
    qed
  next
    fix Ts
    assume wte:P,E,h ⊢ e : NT and wtes:P,E,h ⊢ es [:] Ts
    from IH[OF sconf wte] have wte':P,E,h' ⊢ e' : NT by simp
    have P,E,h' ⊢ es [:] Ts
      by(rule WTrts-hext-mono[OF wtes red-hext-incr[OF red]])
    with wte' show ?thesis by(rule WTrtCallNT)
  qed
  with Some show ?thesis by simp
 qed
 thus ?case by (rule wt-same-type-typeconf)
next
 case (CallParams E es h l es' h' l' v Copt M)
 have reds: P,E ⊢ ⟨es,(h,l)⟩ [→] ⟨es',(h',l')⟩
  and IH: ⋀Ts. ⟦P,E ⊢ (h,l) √; P,E,h ⊢ es [:] Ts⟧
```

```
            ⟹ types-conf P E h' es' Ts
  and sconf: P,E ⊢ (h,l) √ and wt: P,E,h ⊢ Call (Val v) Copt M es : T by
fact+
 from wt have P,E,h' ⊢ Call (Val v) Copt M es' : T
 proof(cases Copt)
  case None
  with wt have P,E,h ⊢ (Val v)·M(es) : T by simp
  hence P,E,h' ⊢ Val v·M(es') : T
  proof (rule WTrt-elim-cases)
   fix C Cs Ts Ts' m
   assume wte: P,E,h ⊢ Val v : Class C
    and method:P ⊢ C has least M = (Ts,T,m) via Cs
    and wtes: P,E,h ⊢ es [:] Ts' and subs:P ⊢ Ts' [≤] Ts
   from wtes have length es = length Ts' by(rule WTrts-same-length)
   with reds have length es' = length Ts'
    by −(drule reds-length,simp)
   with IH[OF sconf wtes] subs obtain Ts'' where wtes':P,E,h' ⊢ es' [:] Ts''
    and subs':P ⊢ Ts'' [≤] Ts by(auto dest:types-conf-smaller-types)
   have wte':P,E,h' ⊢ Val v : Class C
    by(rule WTrt-hext-mono[OF wte reds-hext-incr[OF reds]])
   from wte' method wtes' subs' show ?thesis
    by(rule WTrtCall)
  next
   fix Ts
   assume wte:P,E,h ⊢ Val v : NT
    and wtes:P,E,h ⊢ es [:] Ts
   from wtes have length es = length Ts by(rule WTrts-same-length)
   with reds have length es' = length Ts
    by −(drule reds-length,simp)
   with IH[OF sconf wtes] obtain Ts' where wtes':P,E,h' ⊢ es' [:] Ts'
    and P ⊢ Ts' [≤] Ts by(auto dest:types-conf-smaller-types)
   have wte':P,E,h' ⊢ Val v : NT
    by(rule WTrt-hext-mono[OF wte reds-hext-incr[OF reds]])
   from wte' wtes' show ?thesis by(rule WTrtCallNT)
  qed
  with None show ?thesis by simp
 next
  case (Some C)
  with wt have P,E,h ⊢ (Val v)·(C::)M(es) : T by simp
  hence P,E,h' ⊢ (Val v)·(C::)M(es') : T
  proof(rule WTrt-elim-cases)
   fix C' Cs Ts Ts' m
    assume wte:P,E,h ⊢ Val v : Class C' and path-unique:P ⊢ Path C' to C
unique
    and method:P ⊢ C has least M = (Ts,T,m) via Cs
    and wtes:P,E,h ⊢ es [:] Ts' and subs: P ⊢ Ts' [≤] Ts
   from wtes have length es = length Ts' by(rule WTrts-same-length)
   with reds have length es' = length Ts'
    by −(drule reds-length,simp)
```

```
      with IH[OF sconf wtes] subs obtain Ts'' where wtes':P,E,h' ⊢ es' [:] Ts''
        and subs':P ⊢ Ts'' [≤] Ts by(auto dest:types-conf-smaller-types)
      have wte':P,E,h' ⊢ Val v : Class C'
        by(rule WTrt-hext-mono[OF wte reds-hext-incr[OF reds]])
      from wte' path-unique method wtes' subs' show ?thesis
        by(rule WTrtStaticCall)
    next
      fix Ts
      assume wte:P,E,h ⊢ Val v : NT
        and wtes:P,E,h ⊢ es [:] Ts
      from wtes have length es = length Ts by(rule WTrts-same-length)
      with reds have length es' = length Ts
        by −(drule reds-length,simp)
      with IH[OF sconf wtes] obtain Ts' where wtes':P,E,h' ⊢ es' [:] Ts'
        and P ⊢ Ts' [≤] Ts by(auto dest:types-conf-smaller-types)
      have wte':P,E,h' ⊢ Val v : NT
        by(rule WTrt-hext-mono[OF wte reds-hext-incr[OF reds]])
      from wte' wtes' show ?thesis by(rule WTrtCallNT)
    qed
    with Some show ?thesis by simp
  qed
  thus ?case by (rule wt-same-type-typeconf)
next
  case (RedCall h l a C S Cs M Ts' T' pns' body' Ds Ts T pns body Cs'
        vs bs new-body E T'')
  have hp:hp (h,l) a = Some(C,S)
    and method:P ⊢ last Cs has least M = (Ts',T',pns',body') via Ds
    and select:P ⊢ (C,Cs@_p Ds) selects M = (Ts,T,pns,body) via Cs'
    and length1:length vs = length pns and length2:length Ts = length pns
    and bs:bs = blocks(this#pns,Class(last Cs')#Ts,Ref(a,Cs')#vs,body)
    and body-case:new-body = (case T' of Class D ⇒ (⌊D⌋)bs | - ⇒ bs)
    and wt:P,E,h ⊢ ref (a,Cs)·M(map Val vs) : T'' by fact+
  from wt hp method wf obtain Ts''
    where wtref:P,E,h ⊢ ref (a,Cs) : Class (last Cs) and eq:T'' = T'
    and wtes:P,E,h ⊢ map Val vs [:] Ts'' and subs: P ⊢ Ts'' [≤] Ts'
    by(auto dest:wf-sees-method-fun split:if-split-asm)
  from select wf have is-class P (last Cs')
    by(induct rule:SelectMethodDef.induct,
       auto intro:Subobj-last-isClass simp:FinalOverriderMethodDef-def
     OverriderMethodDefs-def MinimalMethodDefs-def LeastMethodDef-def MethodDefs-def)
  with select-method-wf-mdecl[OF wf select]
  have length-pns:length (this#pns) = length (Class(last Cs')#Ts)
    and notNT:T ≠ NT and type:∀ T∈set (Class(last Cs')#Ts). is-type P T
    and wtabody:P,[this↦Class(last Cs'),pns[↦]Ts] ⊢ body :: T
    by(auto simp:wf-mdecl-def)
  from wtes hp select
  have map:map (P ⊢ typeof_h) (Ref(a,Cs')#vs) = map Some (Class(last Cs')#Ts'')
    by(auto elim:SelectMethodDef.cases split:if-split-asm
          simp:FinalOverriderMethodDef-def OverriderMethodDefs-def
```

```
                MinimalMethodDefs-def LeastMethodDef-def MethodDefs-def)
  from wtref hp have P ⊢ Path C to (last Cs) via Cs
    by (auto simp:path-via-def split:if-split-asm)
  with select method wf have Ts' = Ts ∧ P ⊢ T ≤ T'
    by −(rule select-least-methods-subtypes,simp-all)
  hence eqs:Ts' = Ts and sub:P ⊢ T ≤ T' by auto
  from wf wtabody have P,Map.empty(this↦Class(last Cs'),pns[↦]Ts),h ⊢ body
: T
    by −(rule WT-implies-WTrt,simp-all)
  hence wtbody:P,E(this#pns [↦] Class (last Cs')#Ts),h ⊢ body : T
    by(rule WTrt-env-mono) simp
  from wtes have length vs = length Ts''
    by (fastforce dest:WTrts-same-length)
  with eqs subs
  have length-vs:length (Ref(a,Cs')#vs) = length (Class(last Cs')#Ts)
    by (simp add:list-all2-iff)
  from subs eqs have P ⊢ (Class(last Cs')#Ts'') [≤] (Class(last Cs')#Ts)
    by (simp add:fun-of-def)
  with wt-blocks[OF length-pns length-vs type] wtbody map eq
  have blocks:P,E,h ⊢ blocks(this#pns,Class(last Cs')#Ts,Ref(a,Cs')#vs,body) :
T
    by auto
  have P,E,h ⊢ new-body : T'
  proof(cases ∀ C. T' ≠ Class C)
    case True
    with sub notNT have T = T' by (cases T') auto
    with blocks True body-case bs show ?thesis by(cases T') auto
  next
    case False
    then obtain D where T':T' = Class D by auto
    with method sub wf have class: is-class P D
      by (auto elim!:widen.cases dest:least-method-is-type
            intro:Subobj-last-isClass simp:path-unique-def)
    with blocks T' body-case bs class sub show ?thesis
      by(cases T',auto,cases T,auto)
  qed
  with eq show ?case by(fastforce intro:wt-same-type-typeconf)
next
  case (RedStaticCall Cs C Cs'' M Ts T pns body Cs' Ds vs E a h l T')
  have method:P ⊢ C has least M = (Ts, T, pns, body) via Cs'
    and length1:length vs = length pns
    and length2:length Ts = length pns
    and path-unique:P ⊢ Path last Cs to C unique
    and path-via:P ⊢ Path last Cs to C via Cs''
    and Ds:Ds = (Cs @_p Cs'') @_p Cs'
    and wt:P,E,h ⊢ ref (a,Cs)•(C::)M(map Val vs) : T' by fact+
  from wt method wf obtain Ts'
    where wtref:P,E,h ⊢ ref (a,Cs) : Class (last Cs)
    and wtes:P,E,h ⊢ map Val vs [:] Ts' and subs:P ⊢ Ts' [≤] Ts
```

and *TeqT'*:$T = T'$
by(*auto dest*:*wf-sees-method-fun split*:*if-split-asm*)
from *wtref* **obtain** D S **where** *hp*:$h\ a = Some(D,S)$ **and** *subo*:*Subobjs P D Cs*
by (*auto split*:*if-split-asm*)
from *length1 length2*
have *length-vs*: *length* (*Ref*(*a*,*Ds*)#*vs*) = *length* (*Class* (*last Ds*)#*Ts*) **by** *simp*
from *length2* **have** *length-pns*:*length* (*this*#*pns*) = *length* (*Class* (*last Ds*)#*Ts*)
by *simp*
from *method* **have** $Cs' \neq []$
by (*fastforce intro*!:*Subobjs-nonempty simp add*:*LeastMethodDef-def MethodDefs-def*)
with *Ds* **have** *last*:*last Cs'* = *last Ds*
by (*fastforce dest*:*appendPath-last*)
with *method* **have** *is-class P* (*last Ds*)
by(*auto simp*:*LeastMethodDef-def MethodDefs-def is-class-def*)
with *last has-least-wf-mdecl*[*OF wf method*]
have *wtabody*: *P*,[*this*#*pns* [↦] *Class* (*last Ds*)#*Ts*] ⊢ *body* :: *T*
and *type*:∀ *T*∈*set* (*Class*(*last Ds*)#*Ts*). *is-type P T*
by(*auto simp*:*wf-mdecl-def*)
from *path-via* **have** *suboCs''*:*Subobjs P* (*last Cs*) *Cs''*
and *lastCs''*:*last Cs''* = *C*
by (*auto simp add*:*path-via-def*)
with *subo wf* **have** *subo'*:*Subobjs P D* ($Cs@_p Cs''$)
by(*fastforce intro*: *Subobjs-appendPath*)
from *lastCs''* *suboCs''* **have** *lastC*:*C* = *last*($Cs@_p Cs''$)
by (*fastforce dest*:*Subobjs-nonempty intro*:*appendPath-last*)
from *method* **have** *Subobjs P C Cs'*
by (*auto simp*:*LeastMethodDef-def MethodDefs-def*)
with *subo' wf lastC* **have** *Subobjs P D* (($Cs\ @_p\ Cs''$) $@_p$ *Cs'*)
by (*fastforce intro*:*Subobjs-appendPath*)
with *Ds* **have** *suboDs*:*Subobjs P D Ds* **by** *simp*
from *wtabody* **have** *P*,*Map.empty*(*this*#*pns* [↦] *Class* (*last Ds*)#*Ts*),*h* ⊢ *body* : *T*
by(*rule WT-implies-WTrt*)
hence *P*,*E*(*this*#*pns* [↦] *Class* (*last Ds*)#*Ts*),*h* ⊢ *body* : *T*
by(*rule WTrt-env-mono*) *simp*
hence *P*,*E*,*h* ⊢ *blocks*(*this*#*pns*, *Class* (*last Ds*)#*Ts*, *Ref*(*a*,*Ds*)#*vs*, *body*) : *T*
using *wtes subs wt-blocks*[*OF length-pns length-vs type*] *hp suboDs*
by(*auto simp add*:*rel-list-all2-Cons2*)
with *TeqT'* **show** *?case* **by**(*fastforce intro*:*wt-same-type-typeconf*)
next
case (*RedCallNull E Copt M vs h l*)
have *sconf*:*P*,*E* ⊢ (*h*, *l*) √ **by** *fact*
from *wf* **have** *is-class P NullPointer*
by (*fastforce intro*:*is-class-xcpt wf-prog-wwf-prog*)
hence *preallocated h* ⟹ *P* ⊢ *typeof*$_h$ (*Ref* (*addr-of-sys-xcpt NullPointer*,[*NullPointer*])) = *Some*(*Class NullPointer*)
by (*auto elim*: *preallocatedE dest*!:*preallocatedD Subobjs-Base*)
with *sconf* **have** *P*,*E*,*h* ⊢ *THROW NullPointer* : *T* **by**(*auto simp*:*sconf-def hconf-def*)

thus *?case* **by** (*fastforce intro*:*wt-same-type-typeconf*)
next
case (*BlockRedNone E V T e h l e′ h′ l′ T′*)
have *IH*:$\bigwedge T'.\ [\![P,E(V \mapsto T) \vdash (h,\ l(V := None))\ \surd;\ P,E(V \mapsto T),h \vdash e : T']\!] \Longrightarrow P,E(V \mapsto T),h' \vdash e' :_{NT} T'$
and *sconf*:$P,E \vdash (h,\ l)\ \surd$ **and** *wt*:$P,E,h \vdash \{V{:}T;\ e\} : T'$ **by** *fact+*
from *wt* **have** *type*:*is-type P T* **and** *wte*:$P,E(V \mapsto T),h \vdash e : T'$ **by** *auto*
from *sconf type* **have** $P,E(V \mapsto T) \vdash (h,\ l(V := None))\ \surd$
by (*auto simp*:*sconf-def lconf-def envconf-def*)
from *IH*[*OF this wte*] *type* **show** *?case* **by** (*cases T′*) *auto*
next
case (*BlockRedSome E V T e h l e′ h′ l′ v T′*)
have *red*:$P,E(V \mapsto T) \vdash \langle e,(h,\ l(V := None))\rangle \to \langle e',(h',\ l')\rangle$
and *IH*:$\bigwedge T'.\ [\![P,E(V \mapsto T) \vdash (h,\ l(V := None))\ \surd;\ P,E(V \mapsto T),h \vdash e : T']\!] \Longrightarrow P,E(V \mapsto T),h' \vdash e' :_{NT} T'$
and *Some*:$l'\ V = Some\ v$
and *sconf*:$P,E \vdash (h,\ l)\ \surd$ **and** *wt*:$P,E,h \vdash \{V{:}T;\ e\} : T'$ **by** *fact+*
from *wt* **have** *wte*:$P,E(V \mapsto T),h \vdash e : T'$ **and** *type*:*is-type P T* **by** *auto*
with *sconf wf red type* **have** $P,h' \vdash l'\ (:\leq)_w\ E(V \mapsto T)$
by −(*auto simp*:*sconf-def*,*rule red-preserves-lconf*, *auto intro*:*wf-prog-wwf-prog simp*:*envconf-def lconf-def*)
hence *conf*:$P,h' \vdash v :\leq T$ **using** *Some*
by(*auto simp*:*lconf-def*,*erule-tac x*=*V* **in** *allE*,*clarsimp*)
have *wtval*:$P,E(V \mapsto T),h' \vdash V := Val\ v : T$
proof(*cases T*)
case *Void* **with** *conf* **show** *?thesis* **by** *auto*
next
case *Boolean* **with** *conf* **show** *?thesis* **by** *auto*
next
case *Integer* **with** *conf* **show** *?thesis* **by** *auto*
next
case *NT* **with** *conf* **show** *?thesis* **by** *auto*
next
case (*Class C*)
with *conf* **have** $P,E(V \mapsto T),h' \vdash Val\ v : T \vee P,E(V \mapsto T),h' \vdash Val\ v : NT$
by *auto*
with *Class* **show** *?thesis* **by** *auto*
qed
from *sconf type* **have** $P,E(V \mapsto T) \vdash (h,\ l(V := None))\ \surd$
by (*auto simp*:*sconf-def lconf-def envconf-def*)
from *IH*[*OF this wte*] *wtval type* **show** *?case* **by**(*cases T′*) *auto*
next
case (*InitBlockRed E V T e h l v′ e′ h′ l′ v″ v T′*)
have *red*:$P,E(V \mapsto T) \vdash \langle e,(h,\ l(V \mapsto v'))\rangle \to \langle e',(h',\ l')\rangle$
and *IH*:$\bigwedge T'.\ [\![P,E(V \mapsto T) \vdash (h,\ l(V \mapsto v'))\ \surd;\ P,E(V \mapsto T),h \vdash e : T']\!] \Longrightarrow P,E(V \mapsto T),h' \vdash e' :_{NT} T'$
and *Some*:$l'\ V = Some\ v''$ **and** *casts*:$P \vdash T\ casts\ v\ to\ v'$
and *sconf*:$P,E \vdash (h,\ l)\ \surd$ **and** *wt*:$P,E,h \vdash \{V{:}T := Val\ v;\ e\} : T'$ **by** *fact+*

from *wt* **have** *wte*:$P,E(V \mapsto T),h \vdash e : T'$ **and** *wtval*:$P,E(V \mapsto T),h \vdash V{:=}Val\ v : T$
and *type*:*is-type P T*
by *auto*
from *wf casts wtval* **have** $P,h \vdash v' :\leq T$
by(*fastforce intro*!:*casts-conf wf-prog-wwf-prog*)
with *sconf* **have** *lconf*:$P,h \vdash l(V \mapsto v')\ (:\leq)_w\ E(V \mapsto T)$
by (*fastforce intro*!:*lconf-upd2 simp*:*sconf-def*)
from *sconf type* **have** *envconf P* $(E(V \mapsto T))$ **by**(*simp add*:*sconf-def envconf-def*)
from *red-preserves-lconf*[*OF wf-prog-wwf-prog*[*OF wf*] *red wte lconf this*]
have $P,h' \vdash l'\ (:\leq)_w\ E(V \mapsto T)$ **.**
with *Some* **have** $P,h' \vdash v'' :\leq T$
by(*simp add*:*lconf-def*,*erule-tac* $x{=}V$ **in** *allE*,*auto*)
hence *wtval'*:$P,E(V \mapsto T),h' \vdash V{:=}Val\ v'' : T$
by(*cases T*) *auto*
from *lconf sconf type* **have** $P,E(V \mapsto T) \vdash (h,\ l(V \mapsto v'))\ \surd$
by(*auto simp*:*sconf-def envconf-def*)
from *IH*[*OF this wte*] *wtval' type* **show** *?case* **by**(*cases T'*) *auto*
next
case *RedBlock* **thus** *?case* **by** (*fastforce intro*:*wt-same-type-typeconf*)
next
case *RedInitBlock* **thus** *?case* **by** (*fastforce intro*:*wt-same-type-typeconf*)
next
case (*SeqRed E e h l e' h' l' e*$_2$ *T*)
have *red*:$P,E \vdash \langle e,(h,\ l)\rangle \to \langle e',(h',\ l')\rangle$
and *IH*:$\bigwedge T'.\ [\![P,E \vdash (h,\ l)\ \surd;\ P,E,h \vdash e : T']\!] \Longrightarrow P,E,h' \vdash e' :_{NT} T'$
and *sconf*:$P,E \vdash (h,\ l)\ \surd$ **and** *wt*:$P,E,h \vdash e;;\ e_2 : T$ **by** *fact+*
from *wt* **obtain** T' **where** *wte*:$P,E,h \vdash e : T'$ **and** *wte2*:$P,E,h \vdash e_2 : T$ **by** *auto*
from *WTrt-hext-mono*[*OF wte2 red-hext-incr*[*OF red*]] **have** *wte2'*:$P,E,h' \vdash e_2 : T$ **.**
from *IH*[*OF sconf wte*] **obtain** T'' **where** $P,E,h' \vdash e' : T''$ **by**(*cases T'*) *auto*
with *wte2'* **have** $P,E,h' \vdash e';;\ e_2 : T$ **by** *auto*
thus *?case* **by**(*rule wt-same-type-typeconf*)
next
case *RedSeq* **thus** *?case* **by** (*fastforce intro*:*wt-same-type-typeconf*)
next
case (*CondRed E e h l e' h' l' e*$_1$ *e*$_2$)
have *red*:$P,E \vdash \langle e,(h,\ l)\rangle \to \langle e',(h',\ l')\rangle$
and *IH*: $\bigwedge T.\ [\![P,E \vdash (h,l)\ \surd;\ P,E,h \vdash e : T]\!]$
$\Longrightarrow P,E,h' \vdash e' :_{NT} T$
and *wt*:$P,E,h \vdash$ *if* $(e)\ e_1$ *else* $e_2 : T$
and *sconf*:$P,E \vdash (h,l)\ \surd$ **by** *fact+*
from *wt* **have** *wte*:$P,E,h \vdash e$: *Boolean*
and *wte1*:$P,E,h \vdash e_1 : T$ **and** *wte2*:$P,E,h \vdash e_2 : T$ **by** *auto*
from *IH*[*OF sconf wte*] **have** *wte'*:$P,E,h' \vdash e'$: *Boolean* **by** *auto*
from *wte' WTrt-hext-mono*[*OF wte1 red-hext-incr*[*OF red*]]
WTrt-hext-mono[*OF wte2 red-hext-incr*[*OF red*]]
have $P,E,h' \vdash$ *if* $(e')\ e_1$ *else* $e_2 : T$

by (*rule WTrtCond*)
thus *?case* **by**(*rule wt-same-type-typeconf*)
next
case *RedCondT* **thus** *?case* **by** (*fastforce intro*: *wt-same-type-typeconf*)
next
case *RedCondF* **thus** *?case* **by** (*fastforce intro*: *wt-same-type-typeconf*)
next
case *RedWhile* **thus** *?case* **by** (*fastforce intro*: *wt-same-type-typeconf*)
next
case (*ThrowRed E e h l e' h' l' T*)
have *IH*:$\bigwedge T.\ [\![P,E \vdash (h,\ l)\ \surd;\ P,E,h \vdash e : T]\!] \Longrightarrow P,E,h' \vdash e' :_{NT} T$
and *sconf*:$P,E \vdash (h,\ l)\ \surd$ **and** *wt*:$P,E,h \vdash$ *throw e* : *T* **by** *fact+*
from *wt* **obtain** *T'* **where** *wte*:$P,E,h \vdash e : T'$ **and** *ref*:*is-refT T'*
by *auto*
from *ref* **have** $P,E,h' \vdash$ *throw e'* : *T*
proof(*rule refTE*)
assume *T'*:$T' = NT$
with *wte* **have** $P,E,h \vdash e : NT$ **by** *simp*
from *IH*[*OF sconf this*] *ref T'* **show** *?thesis* **by** *auto*

next
fix *C* **assume** *T'*:$T' =$ *Class C*
with *wte* **have** $P,E,h \vdash e :$ *Class C* **by** *simp*
from *IH*[*OF sconf this*] **have** $P,E,h' \vdash e' :$ *Class C* $\vee$ $P,E,h' \vdash e' :$ *NT*
by *simp*
thus *?thesis*
proof(*rule disjE*)
assume *wte'*:$P,E,h' \vdash e' :$ *Class C*
have *is-refT* (*Class C*) **by** *simp*
with *wte'* **show** *?thesis* **by** *auto*
next
assume *wte'*:$P,E,h' \vdash e' :$ *NT*
have *is-refT NT* **by** *simp*
with *wte'* **show** *?thesis* **by** *auto*
qed
qed
thus *?case* **by** (*rule wt-same-type-typeconf*)
next
case (*RedThrowNull E h l*)
have *sconf*:$P,E \vdash (h,\ l)\ \surd$ **by** *fact*
from *wf* **have** *is-class P NullPointer*
by (*fastforce intro*:*is-class-xcpt wf-prog-wwf-prog*)
hence *preallocated h* $\Longrightarrow P \vdash typeof_h$ (*Ref* (*addr-of-sys-xcpt NullPointer*,[*NullPointer*]))
= *Some*(*Class NullPointer*)
by (*auto elim*: *preallocatedE dest*!:*preallocatedD Subobjs-Base*)
with *sconf* **have** $P,E,h \vdash$ *THROW NullPointer* : *T* **by**(*auto simp*:*sconf-def hconf-def*)
thus *?case* **by** (*fastforce intro*:*wt-same-type-typeconf wf-prog-wwf-prog*)
next

case (*ListRed1 E e h l e′ h′ l′ es Ts*)
have *red*:$P,E \vdash \langle e,(h,\ l)\rangle \to \langle e',(h',\ l')\rangle$
and *IH*:$\bigwedge T.\ [\![P,E \vdash (h,\ l)\ \surd;\ P,E,h \vdash e : T]\!] \Longrightarrow P,E,h' \vdash e' :_{NT} T$
and *sconf*:$P,E \vdash (h,\ l)\ \surd$ **and** *wt*:$P,E,h \vdash e\ \#\ es\ [:]\ Ts$ **by** *fact+*
from *wt* **obtain** *U Us* **where** *Ts*:$Ts = U\#Us$ **by**(*cases Ts*) *auto*
with *wt* **have** *wte*:$P,E,h \vdash e : U$ **and** *wtes*:$P,E,h \vdash es\ [:]\ Us$ **by** *simp-all*
from *WTrts-hext-mono*[*OF wtes red-hext-incr*[*OF red*]]
have *wtes′*:$P,E,h' \vdash es\ [:]\ Us$ **.**
hence *length es* = *length Us* **by** (*rule WTrts-same-length*)
with *wtes′* **have** *types-conf P E h′ es Us*
by (*fastforce intro*:*wts-same-types-typesconf*)
with *IH*[*OF sconf wte*] *Ts* **show** *?case* **by** *simp*
next
case (*ListRed2 E es h l es′ h′ l′ v Ts*)
have *reds*:$P,E \vdash \langle es,(h,\ l)\rangle\ [\to]\ \langle es',(h',\ l')\rangle$
and *IH*:$\bigwedge Ts.\ [\![P,E \vdash (h,\ l)\ \surd;\ P,E,h \vdash es\ [:]\ Ts]\!] \Longrightarrow$ *types-conf P E h′ es′ Ts*
and *sconf*:$P,E \vdash (h,\ l)\ \surd$ **and** *wt*:$P,E,h \vdash Val\ v\#es\ [:]\ Ts$ **by** *fact+*
from *wt* **obtain** *U Us* **where** *Ts*:$Ts = U\#Us$ **by**(*cases Ts*) *auto*
with *wt* **have** *wtval*:$P,E,h \vdash Val\ v : U$ **and** *wtes*:$P,E,h \vdash es\ [:]\ Us$ **by** *simp-all*
from *WTrt-hext-mono*[*OF wtval reds-hext-incr*[*OF reds*]]
have $P,E,h' \vdash Val\ v : U$ **.**
hence $P,E,h' \vdash (Val\ v) :_{NT} U$ **by**(*rule wt-same-type-typeconf*)
with *IH*[*OF sconf wtes*] *Ts* **show** *?case* **by** *simp*
next
case (*CallThrowObj E h l Copt M es h′ l′*)
thus *?case* **by**(*cases Copt*)(*auto intro*:*wt-same-type-typeconf*)
next
case (*CallThrowParams es vs h l es′ E v Copt M h′ l′*)
thus *?case* **by**(*cases Copt*)(*auto intro*:*wt-same-type-typeconf*)
qed (*fastforce intro*:*wt-same-type-typeconf*)+

corollary *subject-reduction*:
$[\![$ *wf-C-prog P*; $P,E \vdash \langle e,s\rangle \to \langle e',s'\rangle$; $P,E \vdash s\ \surd$; $P,E,hp\ s \vdash e{:}T\ ]\!]$
$\Longrightarrow P,E,(hp\ s') \vdash e' :_{NT} T$
by(*cases s*, *cases s′*, *fastforce dest*:*subject-reduction2*)

corollary *subjects-reduction*:
$[\![$ *wf-C-prog P*; $P,E \vdash \langle es,s\rangle\ [\to]\ \langle es',s'\rangle$; $P,E \vdash s\ \surd$; $P,E,hp\ s \vdash es[:]Ts\ ]\!]$
$\Longrightarrow$ *types-conf P E* (*hp s′*) *es′ Ts*
by(*cases s*, *cases s′*, *fastforce dest*:*subjects-reduction2*)

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 $\bigwedge T.\ [\![\ P,E,hp\ s \vdash e : T;\ P,E \vdash s\ \surd\ ]\!] \Longrightarrow P,E \vdash s'\ \surd$

using *step*
proof (*induct rule*:*converse-rtrancl-induct2*)
  case *refl* **show** *?case* **by** *fact*
next
  case *step*
  thus *?case* **using** *wf*
    apply *simp*
    apply (*frule subject-reduction*[*OF wf*])
      apply (*rule step.prems*)
      apply (*rule step.prems*)
      apply (*cases T*)
      apply (*auto dest*:*red-preserves-sconf intro*:*wf-prog-wwf-prog*)
      done
qed

lemma *steps-preserves-sconf*:
assumes *wf*: *wf-C-prog P* **and** *step*: $P,E \vdash \langle es,s\rangle\ [\rightarrow]*\ \langle es',s'\rangle$
shows $\bigwedge Ts.\ [\![\ P,E,hp\ s \vdash es\ [:]\ Ts;\ P,E \vdash s\ \surd\ ]\!] \Longrightarrow P,E \vdash s'\ \surd$

using *step*
proof (*induct rule*:*converse-rtrancl-induct2*)
  case *refl* **show** *?case* **by** *fact*
next
  case (*step es s es'' s'' Ts*)
  have *Reds*:$((es,\ s),\ es'',\ s'') \in Reds\ P\ E$
    and *reds*:$P,E \vdash \langle es'',s''\rangle\ [\rightarrow]*\ \langle es',s'\rangle$
    and *wtes*:$P,E,hp\ s \vdash es\ [:]\ Ts$
    and *sconf*:$P,E \vdash s\ \surd$
    and *IH*:$\bigwedge Ts.\ [\![P,E,hp\ s'' \vdash es''\ [:]\ Ts;\ P,E \vdash s''\ \surd]\!] \Longrightarrow P,E \vdash s'\ \surd$ **by** *fact+*
  from *Reds* **have** *reds1*:$P,E \vdash \langle es,s\rangle\ [\rightarrow]\ \langle es'',s''\rangle$ **by** *simp*
  from *subjects-reduction*[*OF wf this sconf wtes*]
  have *type*:*types-conf P E* (*hp s''*) *es'' Ts* **.**
  from *reds1 wtes sconf wf* **have** *sconf'*:$P,E \vdash s''\ \surd$
    by(*fastforce intro*:*wf-prog-wwf-prog reds-preserves-sconf*)
  from *type* **have** $\exists\, Ts'.\ P,E,hp\ s'' \vdash es''\ [:]\ Ts'$
  proof (*induct Ts arbitrary*: *es''*)
    fix *esi*
    assume *types-conf P E* (*hp s''*) *esi* []
    thus $\exists\, Ts'.\ P,E,hp\ s'' \vdash esi\ [:]\ Ts'$
    proof(*induct esi*)
      case *Nil* **thus** $\exists\, Ts'.\ P,E,hp\ s'' \vdash []\ [:]\ Ts'$ **by** *simp*
    next
      fix *ex esx*
      assume *types-conf P E* (*hp s''*) (*ex#esx*) []
      thus $\exists\, Ts'.\ P,E,hp\ s'' \vdash ex\#esx\ [:]\ Ts'$ **by** *simp*
    qed
  next

```
    fix T' Ts' esi
    assume type':types-conf P E (hp s'') esi (T'#Ts')
      and IH:⋀es''. types-conf P E (hp s'') es'' Ts' ⟹
                  ∃ Ts''. P,E,hp s'' ⊢ es'' [:] Ts''
    from type' show ∃ Ts'. P,E,hp s'' ⊢ esi [:] Ts'
    proof(induct esi)
      case Nil thus ∃ Ts'. P,E,hp s'' ⊢ [] [:] Ts' by simp
    next
      fix ex esx
      assume types-conf P E (hp s'') (ex#esx) (T'#Ts')
      hence type':P,E,hp s'' ⊢ ex :_NT T'
        and types':types-conf P E (hp s'') esx Ts' by simp-all
      from type' obtain Tx where type'':P,E,hp s'' ⊢ ex : Tx
        by(cases T') auto
      from IH[OF types'] obtain Tsx where P,E,hp s'' ⊢ esx [:] Tsx by auto
      with type'' show ∃ Ts'. P,E,hp s'' ⊢ ex#esx [:] Ts' by auto
    qed
  qed
  then obtain Ts' where P,E,hp s'' ⊢ es'' [:] Ts' by blast
  from IH[OF this sconf'] show ?case .
qed


lemma step-preserves-defass:
assumes wf: wf-C-prog P and step: P,E ⊢ ⟨e,s⟩ →* ⟨e',s'⟩
shows 𝒟 e ⌊dom(lcl s)⌋ ⟹ 𝒟 e' ⌊dom(lcl s')⌋

using step
proof (induct rule:converse-rtrancl-induct2)
  case refl thus ?case .
next
  case (step e s e' s') thus ?case
    by(cases s,cases s')(auto dest:red-preserves-defass[OF wf])
qed



lemma step-preserves-type:
assumes wf: wf-C-prog P and step: P,E ⊢ ⟨e,s⟩ →* ⟨e',s'⟩
shows ⋀T. ⟦ P,E ⊢ s √; P,E,hp s ⊢ e:T ⟧
    ⟹ P,E,(hp s') ⊢ e' :_NT T

using step
proof (induct rule:converse-rtrancl-induct2)
  case refl thus ?case by −(rule wt-same-type-typeconf)
next
  case (step e s e'' s'' T) thus ?case using wf
    apply simp
    apply (frule subject-reduction[OF wf])
```

apply (*auto dest*!:*red-preserves-sconf intro*:*wf-prog-wwf-prog*)
apply(*cases T*)
apply *fastforce*+
done
qed

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*'
∨ (∃ *C*. *T*'' = *NT* ∧ *T*' = *Class C*)) ∧ *conformable Ts*'' *Ts*'
| *conformable* - - ⟷ *False*

lemma *types-conf-conf-types-conf*:
⟦*types-conf P E h es Ts*; *conformable Ts Ts*'⟧ ⟹ *types-conf P E h es Ts*'
proof (*induct Ts arbitrary*: *Ts*' *es*)
case *Nil* **thus** *?case* **by** (*cases Ts*') (*auto split*: *if-split-asm*)
next
case (*Cons T*'' *Ts*'')
have *type*:*types-conf P E h es* (*T*''#*Ts*'')
and *conf*:*conformable* (*T*''#*Ts*'') *Ts*'
and *IH*:⋀*Ts*' *es*. ⟦*types-conf P E h es Ts*''; *conformable Ts*'' *Ts*'⟧
⟹ *types-conf P E h es Ts*' **by** *fact*+
from *type* **obtain** *e*' *es*' **where** *es*:*es* = *e*'#*es*' **by** (*cases es*) *auto*
with *type* **have** *type*':*P*,*E*,*h* ⊢ *e*' :$_{NT}$ *T*''
and *types*': *types-conf P E h es*' *Ts*''
by *simp-all*
from *conf* **obtain** *U Us* **where** *Ts*': *Ts*' = *U*#*Us* **by** (*cases Ts*') *auto*
with *conf* **have** *disj*:*T*'' = *U* ∨ (∃ *C*. *T*'' = *NT* ∧ *U* = *Class C*)
and *conf*':*conformable Ts*'' *Us*
by *simp-all*
from *type*' *disj* **have** *P*,*E*,*h* ⊢ *e*' :$_{NT}$ *U* **by** *auto*
with *IH*[*OF types*' *conf*'] *Ts*' *es* **show** *?case* **by** *simp*
qed

lemma *types-conf-Wtrt-conf*:
types-conf P E h es Ts ⟹ ∃ *Ts*'. *P*,*E*,*h* ⊢ *es* [:] *Ts*' ∧ *conformable Ts*' *Ts*
proof (*induct Ts arbitrary*: *es*)
case *Nil* **thus** *?case* **by** (*cases es*) (*auto split*:*if-split-asm*)
next
case (*Cons T*'' *Ts*'')
have *type*:*types-conf P E h es* (*T*''#*Ts*'')
and *IH*:⋀*es*. *types-conf P E h es Ts*'' ⟹
∃ *Ts*'. *P*,*E*,*h* ⊢ *es* [:] *Ts*' ∧ *conformable Ts*' *Ts*'' **by** *fact*+
from *type* **obtain** *e*' *es*' **where** *es*:*es* = *e*'#*es*' **by** (*cases es*) *auto*
with *type* **have** *type*':*P*,*E*,*h* ⊢ *e*' :$_{NT}$ *T*''

and *types'*: *types-conf P E h es' Ts''*
by *simp-all*
from *type'* **obtain** T' **where** $P,E,h \vdash e' : T'$ **and**
$T' = T'' \vee (\exists\, C.\ T' = NT \wedge T'' = Class\ C)$ **by**(*cases T''*) *auto*
with *IH*[*OF types'*] *es* **show** *?case*
by(*auto*,*rule-tac x*=*T''*#*Ts'* **in** *exI*,*simp*,*rule-tac x*=*NT*#*Ts'* **in** *exI*,*simp*)
qed

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-conf P E* (*hp s'*) *es' Ts*

using *steps*
proof (*induct rule*:*converse-rtrancl-induct2*)
case *refl* **thus** *?case* **by** −(*rule wts-same-types-typesconf*)
next
case (*step es s es'' s'' Ts*)
have *Reds*:((*es*, *s*), *es''*, *s''*) ∈ *Reds P E*
and *steps*:$P,E \vdash \langle es'',s''\rangle\ [\rightarrow]*\ \langle es',s'\rangle$
and *sconf*:$P,E \vdash s\ \surd$ **and** *wtes*:$P,E,hp\ s \vdash es\ [:]\ Ts$
and *IH*:$\bigwedge Ts.\ [\![P,E \vdash s''\ \surd;\ P,E,hp\ s'' \vdash es''\ [:]\ Ts\ ]\!]$
$\Longrightarrow$ *types-conf P E* (*hp s'*) *es' Ts* **by** *fact*+
from *Reds* **have** *step*:$P,E \vdash \langle es,s\rangle\ [\rightarrow]\ \langle es'',s''\rangle$ **by** *simp*
with *wtes sconf wf* **have** *sconf'*:$P,E \vdash s''\ \surd$
by(*auto intro*:*reds-preserves-sconf wf-prog-wwf-prog*)
from *wtes* **have** *length es* = *length Ts* **by**(*fastforce dest*:*WTrts-same-length*)
from *step sconf wtes*
have *type'*: *types-conf P E* (*hp s''*) *es'' Ts*
by (*rule subjects-reduction*[*OF wf*])
then obtain *Ts'* **where** *wtes''*:$P,E,hp\ s'' \vdash es''\ [:]\ Ts'$
and *conf*:*conformable Ts' Ts* **by** (*auto dest*:*types-conf-Wtrt-conf*)
from *IH*[*OF sconf' wtes''*] **have** *types-conf P E* (*hp s'*) *es' Ts'* **.**
with *conf* **show** *?case* **by**(*fastforce intro*:*types-conf-conf-types-conf*)
qed

26.4 Lifting to ⇒

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

lemma *eval-preserves-sconf*:
$[\![$ *wf-C-prog P*; $P,E \vdash \langle e,s\rangle \Rightarrow \langle e',s'\rangle$; $P,E \vdash e::T$; $P,E \vdash s\ \surd\ ]\!] \Longrightarrow P,E \vdash s'\ \surd$

by(*blast intro*:*step-preserves-sconf big-by-small WT-implies-WTrt wf-prog-wwf-prog*)

lemma *evals-preserves-sconf*:
$[\![$ *wf-C-prog P*; $P,E \vdash \langle es,s\rangle\ [\Rightarrow]\ \langle es',s'\rangle$; $P,E \vdash es\ [::]\ Ts$; $P,E \vdash s\ \surd\ ]\!]$
$\Longrightarrow P,E \vdash s'\ \surd$

by(*blast intro*:*steps-preserves-sconf bigs-by-smalls WTs-implies-WTrts*
wf-prog-wwf-prog)

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*

using *wf*
by (*auto dest*!:*big-by-small*[*OF wf-prog-wwf-prog*[*OF wf*]] *WT-implies-WTrt*
intro:*wf-prog-wwf-prog*
dest!:*step-preserves-type*[*OF wf*])

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*
using *wf*
by (*auto dest*!:*bigs-by-smalls*[*OF wf-prog-wwf-prog*[*OF wf*]] *WTs-implies-WTrts*
intro:*wf-prog-wwf-prog*
dest!:*steps-preserves-types*[*OF wf*])

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*

using *wf*
by (*force elim*!:*red-preserves-sconf intro*:*wf-prog-wwf-prog*
dest:*subject-reduction*[*OF wf*] *simp*:*wf-config-def*)

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*

using *reds*
proof (*induct rule*:*converse-rtrancl-induct2*)
case *refl* **thus** *?case*
by (*fastforce intro*:*wt-same-type-typeconf simp*:*wf-config-def*)
next

case (*step e s e'' s'' T*)
have *Red*:((*e*, *s*), *e''*, *s''*) ∈ *Red P E*
and *IH*:⋀*T*. *P*,*E*,*s''* ⊢ *e''* : *T* √ ⟹ *P*,*E*,(*hp s'*) ⊢ *e'* :$_{NT}$ *T*
and *wte*:*P*,*E*,*s* ⊢ *e* : *T* √ **by** *fact*+
from *Red* **have** *red*:*P*,*E* ⊢ ⟨*e*,*s*⟩ → ⟨*e''*,*s''*⟩ **by** *simp*
from *red-preserves-sconf*[*OF red*] *wte wf* **have** *sconf*:*P*,*E* ⊢ *s''* √
by(*fastforce dest*:*wf-prog-wwf-prog simp*:*wf-config-def*)
from *wf red wte* **have** *type-conf*:*P*,*E*,(*hp s''*) ⊢ *e''* :$_{NT}$ *T*
by(*rule Subject-reduction*)
show *?case*
proof(*cases T*)
case *Void*
with *type-conf* **have** *P*,*E*,*hp s''* ⊢ *e''* : *T* **by** *simp*
with *sconf* **have** *P*,*E*,*s''* ⊢ *e''* : *T* √ **by**(*simp add*:*wf-config-def*)
from *IH*[*OF this*] **show** *?thesis* **.**
next
case *Boolean*
with *type-conf* **have** *P*,*E*,*hp s''* ⊢ *e''* : *T* **by** *simp*
with *sconf* **have** *P*,*E*,*s''* ⊢ *e''* : *T* √ **by**(*simp add*:*wf-config-def*)
from *IH*[*OF this*] **show** *?thesis* **.**
next
case *Integer*
with *type-conf* **have** *P*,*E*,*hp s''* ⊢ *e''* : *T* **by** *simp*
with *sconf* **have** *P*,*E*,*s''* ⊢ *e''* : *T* √ **by**(*simp add*:*wf-config-def*)
from *IH*[*OF this*] **show** *?thesis* **.**
next
case *NT*
with *type-conf* **have** *P*,*E*,*hp s''* ⊢ *e''* : *T* **by** *simp*
with *sconf* **have** *P*,*E*,*s''* ⊢ *e''* : *T* √ **by**(*simp add*:*wf-config-def*)
from *IH*[*OF this*] **show** *?thesis* **.**
next
case (*Class C*)
with *type-conf* **have** *P*,*E*,*hp s''* ⊢ *e''* : *T* ∨ *P*,*E*,*hp s''* ⊢ *e''* : *NT* **by** *simp*
thus *?thesis*
proof(*rule disjE*)
assume *P*,*E*,*hp s''* ⊢ *e''* : *T*
with *sconf* **have** *P*,*E*,*s''* ⊢ *e''* : *T* √ **by**(*simp add*:*wf-config-def*)
from *IH*[*OF this*] **show** *?thesis* **.**
next
assume *P*,*E*,*hp s''* ⊢ *e''* : *NT*
with *sconf* **have** *P*,*E*,*s''* ⊢ *e''* : *NT* √ **by**(*simp add*:*wf-config-def*)
from *IH*[*OF this*] **have** *P*,*E*,*hp s'* ⊢ *e'* : *NT* **by** *simp*
with *Class* **show** *?thesis* **by** *simp*
qed
qed
qed

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′*⟩

using *progress*[*OF wf-prog-wwf-prog*[*OF wf*]]
by(*auto simp*:*wf-config-def sconf-def*)

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 −
from *sconf wte wf* **have** *wf-config*:*P*,*E*,*s* ⊢ *e* : *T* √
by(*fastforce intro*:*WT-implies-WTrt simp*:*wf-config-def*)
with *wf step* **have** *type-conf*:*P*,*E*,(*hp s′*) ⊢ *e′* :$_{NT}$ *T*
by(*rule Subject-reductions*)
from *step-preserves-sconf*[*OF wf step wte*[*THEN WT-implies-WTrt*] *sconf*] *wf*
have *sconf′*:*P*,*E* ⊢ *s′* √ **by** *simp*
from *wf step D* **have** *D′*:𝒟 *e′* ⌊*dom*(*lcl s′*)⌋ **by**(*rule step-preserves-defass*)
show *?thesis*
proof(*cases T*)
case *Void*
with *type-conf* **have** *wte′*:*P*,*E*,*hp s′* ⊢ *e′* : *T* **by** *simp*
with *sconf′* **have** *wf-config′*:*P*,*E*,*s′* ⊢ *e′* : *T* √ **by**(*simp add*:*wf-config-def*)
{ assume ¬ *final e′*
from *Progress*[*OF wf wf-config′ D′ this*] *nored* **have** *False*
by *simp* **}**
hence *final e′* **by** *fast*
with *wte′* **show** *?thesis* **by**(*auto simp*:*final-def*)
next
case *Boolean*
with *type-conf* **have** *wte′*:*P*,*E*,*hp s′* ⊢ *e′* : *T* **by** *simp*
with *sconf′* **have** *wf-config′*:*P*,*E*,*s′* ⊢ *e′* : *T* √ **by**(*simp add*:*wf-config-def*)
{ assume ¬ *final e′*
from *Progress*[*OF wf wf-config′ D′ this*] *nored* **have** *False*
by *simp* **}**
hence *final e′* **by** *fast*
with *wte′* **show** *?thesis* **by**(*auto simp*:*final-def*)
next
case *Integer*
with *type-conf* **have** *wte′*:*P*,*E*,*hp s′* ⊢ *e′* : *T* **by** *simp*
with *sconf′* **have** *wf-config′*:*P*,*E*,*s′* ⊢ *e′* : *T* √ **by**(*simp add*:*wf-config-def*)
{ assume ¬ *final e′*
from *Progress*[*OF wf wf-config′ D′ this*] *nored* **have** *False*
by *simp* **}**

hence *final e′* **by** *fast*
with *wte′* **show** *?thesis* **by**(*auto simp*:*final-def*)
next
case *NT*
with *type-conf* **have** *wte′*:*P*,*E*,*hp s′* ⊢ *e′* : *T* **by** *simp*
with *sconf′* **have** *wf-config′*:*P*,*E*,*s′* ⊢ *e′* : *T* √ **by**(*simp add*:*wf-config-def*)
{ assume ¬ *final e′*
from *Progress*[*OF wf wf-config′ D′ this*] *nored* **have** *False*
by *simp* **}**
hence *final e′* **by** *fast*
with *wte′* **show** *?thesis* **by**(*auto simp*:*final-def*)
next
case (*Class C*)
with *type-conf* **have** *wte′*:*P*,*E*,*hp s′* ⊢ *e′* : *T* ∨ *P*,*E*,*hp s′* ⊢ *e′* : *NT* **by** *simp*
thus *?thesis*
proof(*rule disjE*)
assume *wte′*:*P*,*E*,*hp s′* ⊢ *e′* : *T*
with *sconf′* **have** *wf-config′*:*P*,*E*,*s′* ⊢ *e′* : *T* √ **by**(*simp add*:*wf-config-def*)
{ assume ¬ *final e′*
from *Progress*[*OF wf wf-config′ D′ this*] *nored* **have** *False*
by *simp* **}**
hence *final e′* **by** *fast*
with *wte′* **show** *?thesis* **by**(*auto simp*:*final-def*)
next
assume *wte′*:*P*,*E*,*hp s′* ⊢ *e′* : *NT*
with *sconf′* **have** *wf-config′*:*P*,*E*,*s′* ⊢ *e′* : *NT* √ **by**(*simp add*:*wf-config-def*)
{ assume ¬ *final e′*
from *Progress*[*OF wf wf-config′ D′ this*] *nored* **have** *False*
by *simp* **}**
hence *final e′* **by** *fast*
with *wte′ Class* **show** *?thesis* **by**(*auto simp*:*final-def*)
qed
qed
qed

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 (induct xs arbitrary: ys E)
  case Nil thus ?case by simp
next
  case (Cons x' xs')
  have map:(E(x' # xs' [↦] ys)) x = Some y
    and length:length (x'#xs') = length ys
    and dist:distinct (x'#xs')
    and IH:⋀ys E. ⟦(E(xs' [↦] ys)) x = Some y; length xs' = length ys;
                  distinct xs'⟧
        ⟹ ∀ i. x = xs'!i ∧ i < length xs' ⟶ y = ys!i by fact+
  from length obtain y' ys' where ys:ys = y'#ys' by(cases ys) auto
  { fix i assume x:x = (x'#xs')!i and i:i < length(x'#xs')
    have y = ys!i
    proof(cases i)
      case 0 with x map ys dist show ?thesis by simp
    next
      case (Suc n)
      with x i have x':x = xs'!n and n:n < length xs' by simp-all
      from map ys have map':(E(x' ↦ y')(xs' [↦] ys')) x = Some y by simp
      from length ys have length':length xs' = length ys' by simp
      from dist have dist':distinct xs' by simp
      from IH[OF map' length' dist']
      have ∀ i. x = xs'!i ∧ i < length xs' ⟶ y = ys'!i .
      with x' n have y = ys'!n by simp
      with ys n Suc show ?thesis by simp
    qed }
  thus ?case by simp
qed


lemma nth-maps:⟦length pns = length Ts; distinct pns; i < length Ts⟧
  ⟹ (E(pns [↦] Ts)) (pns!i) = Some (Ts!i)
proof (induct i arbitrary: E pns Ts)
  case 0
  have dist:distinct pns and length:length pns = length Ts
    and i-length:0 < length Ts by fact+
  from i-length obtain T' Ts' where Ts:Ts = T'#Ts' by(cases Ts) auto
  with length obtain p' pns' where pns = p'#pns' by(cases pns) auto
  with Ts dist show ?case by simp
next
  case (Suc n)
  have i-length:Suc n < length Ts and dist:distinct pns
    and length:length pns = length Ts by fact+
  from Suc obtain T' Ts' where Ts:Ts = T'#Ts' by(cases Ts) auto
  with length obtain p' pns' where pns:pns = p'#pns' by(cases pns) auto
  with Ts length dist have length':length pns' = length Ts'
    and dist':distinct pns' and notin:p' ∉ set pns' by simp-all
  from i-length Ts have n-length:n < length Ts' by simp
```

with *length' dist'* **have** *map*:$(E(p' \mapsto T')(pns'\,[\mapsto]\,Ts'))\ (pns'!n) = Some(Ts'!n)$
by *fact*
with *notin* **have** $(E(p' \mapsto T')(pns'\,[\mapsto]\,Ts'))\ p' = Some\ T'$ **by** *simp*
with *pns Ts map* **show** *?case* **by** *simp*
qed

lemma *casts-casts-eq-result*:
fixes $s :: state$
assumes *casts*:$P \vdash T\ casts\ v\ to\ v'$ **and** *casts'*:$P \vdash T\ casts\ v\ to\ w'$
and *type*:*is-type P T* **and** *wte*:$P,E \vdash e :: T'$ **and** *leq*:$P \vdash T' \leq T$
and *eval*:$P,E \vdash \langle e,s\rangle \Rightarrow \langle Val\ v,(h,l)\rangle$ **and** *sconf*:$P,E \vdash s\ \surd$
and *wf*:*wf-C-prog P*
shows $v' = w'$
proof(*cases* $\forall C.\ T \neq Class\ C$)
case *True*
with *casts casts'* **show** *?thesis*
by(*auto elim*:*casts-to.cases*)
next
case *False*
then obtain C **where** T:$T = Class\ C$ **by** *auto*
with *type* **have** *is-class P C* **by** *simp*
with *wf T leq* **have** $T' = NT \vee (\exists D.\ T' = Class\ D \wedge P \vdash Path\ D\ to\ C\ unique)$
by(*simp add*:*widen-Class*)
thus *?thesis*
proof(*rule disjE*)
assume $T' = NT$
with *wf eval sconf wte* **have** $v = Null$
by(*fastforce dest*:*eval-preserves-type*)
with *casts casts'* **show** *?thesis* **by**(*fastforce elim*:*casts-to.cases*)
next
assume $\exists D.\ T' = Class\ D \wedge P \vdash Path\ D\ to\ C\ unique$
then obtain D **where** T':$T' = Class\ D$
and *path-unique*:$P \vdash Path\ D\ to\ C\ unique$ **by** *auto*
with *wf eval sconf wte*
have $P,E,h \vdash Val\ v : T' \vee P,E,h \vdash Val\ v : NT$
by(*fastforce dest*:*eval-preserves-type*)
thus *?thesis*
proof(*rule disjE*)
assume $P,E,h \vdash Val\ v : T'$
with T' **obtain** $a\ Cs\ C'\ S$ **where** h:$h\ a = Some(C',S)$ **and** v:$v = Ref(a,Cs)$
and *last*:*last Cs* $= D$
by(*fastforce dest*:*typeof-Class-Subo*)
from *casts' v last T* **obtain** $Cs'\ Ds$ **where** $P \vdash Path\ D\ to\ C\ via\ Cs'$
and $Ds = Cs@_p Cs'$ **and** $w' = Ref(a,Ds)$
by(*auto elim*:*casts-to.cases*)
with *casts T v last path-unique* **show** *?thesis*
by *auto*(*erule casts-to.cases*,*auto simp*:*path-via-def path-unique-def*)
next
assume $P,E,h \vdash Val\ v : NT$

```
    with wf eval sconf wte have v = Null
      by(fastforce dest:eval-preserves-type)
    with casts casts' show ?thesis by(fastforce elim:casts-to.cases)
   qed
  qed
qed

lemma Casts-Casts-eq-result:
  assumes wf:wf-C-prog P
  shows ⟦P ⊢ Ts Casts vs to vs'; P ⊢ Ts Casts vs to ws'; ∀ T ∈ set Ts. is-type P
T;
         P,E ⊢ es [::] Ts'; P ⊢ Ts' [≤] Ts; P,E ⊢ ⟨es,s⟩ [⇒] ⟨map Val vs,(h,l)⟩;
         P,E ⊢ s √⟧
      ⟹ vs' = ws'
proof (induct vs arbitrary: vs' ws' Ts Ts' es s)
  case Nil thus ?case by (auto elim!:Casts-to.cases)
next
  case (Cons x xs)
  have CastsCons:P ⊢ Ts Casts x # xs to vs'
    and CastsCons':P ⊢ Ts Casts x # xs to ws'
    and type:∀ T ∈ set Ts. is-type P T
    and wtes:P,E ⊢ es [::] Ts' and subs:P ⊢ Ts' [≤] Ts
    and evals:P,E ⊢ ⟨es,s⟩ [⇒] ⟨map Val (x#xs),(h,l)⟩
    and sconf:P,E ⊢ s √
    and IH:⋀vs' ws' Ts Ts' es s.
    ⟦P ⊢ Ts Casts xs to vs'; P ⊢ Ts Casts xs to ws'; ∀ T ∈ set Ts. is-type P T;
     P,E ⊢ es [::] Ts'; P ⊢ Ts' [≤] Ts; P,E ⊢ ⟨es,s⟩ [⇒] ⟨map Val xs,(h,l)⟩;
     P,E ⊢ s √⟧
     ⟹ vs' = ws' by fact+
  from CastsCons obtain y ys S Ss where vs':vs' = y#ys and Ts:Ts = S#Ss
    apply −
    apply(frule length-Casts-vs,cases Ts,auto)
    apply(frule length-Casts-vs',cases vs',auto)
    done
  with CastsCons have casts:P ⊢ S casts x to y and Casts:P ⊢ Ss Casts xs to ys
    by(auto elim:Casts-to.cases)
  from Ts type have type':is-type P S and types':∀ T ∈ set Ss. is-type P T
    by auto
  from Ts CastsCons' obtain z zs where ws':ws' = z#zs
    by simp(frule length-Casts-vs',cases ws',auto)
  with Ts CastsCons' have casts':P ⊢ S casts x to z
    and Casts':P ⊢ Ss Casts xs to zs
    by(auto elim:Casts-to.cases)
  from Ts subs obtain U Us where Ts':Ts' = U#Us and subs':P ⊢ Us [≤] Ss
    and sub:P ⊢ U ≤ S by(cases Ts',auto simp:fun-of-def)
  from wtes Ts' obtain e' es' where es:es = e'#es' and wte':P,E ⊢ e' :: U
    and wtes':P,E ⊢ es' [::] Us by(cases es) auto
  with evals obtain h' l' where eval:P,E ⊢ ⟨e',s⟩ ⇒ ⟨Val x,(h',l')⟩
    and evals':P,E ⊢ ⟨es',(h',l')⟩ [⇒] ⟨map Val xs,(h,l)⟩
```

by (*auto elim*:*evals.cases*)
from *wf eval wte′ sconf* **have** $P,E \vdash (h',l') \surd$ **by**(*rule eval-preserves-sconf*)
from *IH*[*OF Casts Casts′ types′ wtes′ subs′ evals′ this*] **have** *eq*:$ys = zs$ **.**
from *casts casts′ type′ wte′ sub eval sconf wf* **have** $y = z$
by(*rule casts-casts-eq-result*)
with *eq vs′ ws′* **show** *?case* **by** *simp*
qed

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(*induct rule*:*Casts-to.induct*)
case *Casts-Nil* **thus** *?case* **by** *simp*
next
case (*Casts-Cons T v v′ Ts vs vs′*)
have *casts*:$P \vdash T$ *casts v to* v' **and** *wtes*:$P,E \vdash es\ [::]\ Ts'$
and *evals*:$P,E \vdash \langle es,s\rangle\ [\Rightarrow]\ \langle map\ Val\ (v\#vs),(h,l)\rangle$
and *subs*:$P \vdash Ts'\ [\leq]\ (T\#Ts)$ **and** *sconf*:$P,E \vdash s\ \surd$
and *IH*:$\bigwedge es\ s\ Ts'.[\![P,E \vdash es\ [::]\ Ts';\ P,E \vdash \langle es,s\rangle\ [\Rightarrow]\ \langle map\ Val\ vs,(h,l)\rangle;$
$P,E \vdash s\ \surd;\ P \vdash Ts'\ [\leq]\ Ts]\!]$
$\Longrightarrow \forall i{<}length\ Ts.\ P,h \vdash vs'\ !\ i :\leq Ts\ !\ i$ **by** *fact+*
from *subs* **obtain** *U Us* **where** *Ts′*:$Ts' = U\#Us$ **by**(*cases Ts′*) *auto*
with *subs* **have** *sub′*:$P \vdash U \leq T$ **and** *subs′*:$P \vdash Us\ [\leq]\ Ts$
by (*simp-all add*:*fun-of-def*)
from *wtes Ts′* **obtain** *e′ es′* **where** *es*:$es = e'\#es'$ **by**(*cases es*) *auto*
with *Ts′ wtes* **have** *wte′*:$P,E \vdash e' :: U$ **and** *wtes′*:$P,E \vdash es'\ [::]\ Us$ **by** *auto*
from *es evals* **obtain** *s′* **where** *eval′*:$P,E \vdash \langle e',s\rangle \Rightarrow \langle Val\ v,s'\rangle$
and *evals′*:$P,E \vdash \langle es',s'\rangle\ [\Rightarrow]\ \langle map\ Val\ vs,(h,l)\rangle$
by(*auto elim*:*evals.cases*)
from *wf eval′ wte′ sconf* **have** *sconf′*:$P,E \vdash s'\ \surd$ **by**(*rule eval-preserves-sconf*)
from *evals′* **have** *hext*:$hp\ s' \trianglelefteq h$ **by**(*cases s′*,*auto intro*:*evals-hext*)
from *wf eval′ sconf wte′* **have** $P,E,(hp\ s') \vdash Val\ v :_{NT} U$
by(*rule eval-preserves-type*)
with *hext* **have** *wtrt*:$P,E,h \vdash Val\ v :_{NT} U$
by(*cases U*,*auto intro*:*hext-typeof-mono*)
from *casts wtrt sub′* **have** $P,h \vdash v' :\leq T$
proof(*induct rule*:*casts-to.induct*)
case (*casts-prim T″ v″*)
have $\forall C.\ T'' \neq Class\ C$ **and** $P,E,h \vdash Val\ v'' :_{NT} U$ **and** $P \vdash U \leq T''$ **by** *fact+*
thus *?case* **by**(*cases T″*) *auto*
next
case (*casts-null C*) **thus** *?case* **by** *simp*
next
case (*casts-ref Cs C Cs′ Ds a*)

have *path*:$P \vdash$ *Path last Cs to C via Cs'*
and *Ds*:*Ds* = *Cs* $@_p$ *Cs'*
and *wtref*:*P*,*E*,*h* $\vdash$ *ref* (*a*, *Cs*) :$_{NT}$ *U* **by** *fact+*
from *wtref* **obtain** *D S* **where** *subo*:*Subobjs P D Cs* **and** *h*:*h a* = *Some*(*D*,*S*)
by(*cases U*,*auto split*:*if-split-asm*)
from *path Ds* **have** *last*:*C* = *last Ds*
by(*fastforce intro*!:*appendPath-last Subobjs-nonempty simp*:*path-via-def*)
from *subo path Ds wf* **have** *Subobjs P D Ds*
by(*fastforce intro*:*Subobjs-appendPath simp*:*path-via-def*)
with *last h* **show** *?case* **by** *simp*
qed
with *IH*[*OF wtes' evals' sconf' subs'*] **show** *?case*
by(*auto simp*:*nth-Cons*,*case-tac i*,*auto*)
qed

lemma *map-Val-throw-False*:*map Val vs* = *map Val ws* @ *throw ex* # *es* $\Longrightarrow$ *False*
proof (*induct vs arbitrary*: *ws*)
case *Nil* **thus** *?case* **by** *simp*
next
case (*Cons v' vs'*)
have *eq*:*map Val* (*v'*#*vs'*) = *map Val ws* @ *throw ex* # *es*
and *IH*:$\bigwedge$*ws'*. *map Val vs'* = *map Val ws'* @ *throw ex* # *es* $\Longrightarrow$ *False* **by** *fact+*
from *eq* **obtain** *w' ws'* **where** *ws*:*ws* = *w'*#*ws'* **by**(*cases ws*) *auto*
from *eq* **have** *tl*(*map Val* (*v'*#*vs'*)) = *tl*(*map Val ws* @ *throw ex* # *es*) **by** *simp*
hence *map Val vs'* = *tl*(*map Val ws* @ *throw ex* # *es*) **by** *simp*
with *ws* **have** *map Val vs'* = *map Val ws'* @ *throw ex* # *es* **by** *simp*
from *IH*[*OF this*] **show** *?case* **.**
qed

lemma *map-Val-throw-eq*:*map Val vs* @ *throw ex* # *es* = *map Val ws* @ *throw ex'* # *es'*
$\Longrightarrow$ *vs* = *ws* $\wedge$ *ex* = *ex'* $\wedge$ *es* = *es'*
apply(*clarsimp simp*:*append-eq-append-conv2*)
apply(*erule disjE*)
apply(*case-tac us*)
apply(*fastforce elim*:*map-injective simp*:*inj-on-def*)
apply(*fastforce dest*:*map-Val-throw-False*)
apply(*case-tac us*)
apply(*fastforce elim*:*map-injective simp*:*inj-on-def*)
apply(*fastforce dest*:*sym*[*THEN map-Val-throw-False*])
done

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 (*induct rule*:*eval-evals.inducts*)
case *New* **thus** *?case* **by**(*auto elim*: *eval-cases*)
next
case *NewFail* **thus** *?case* **by**(*auto elim*: *eval-cases*)
next
case (*StaticUpCast E e* s_0 *a Cs* s_1 *C Cs' Ds* e_2 s_2)
have *eval*:$P,E \vdash \langle(\!|C|\!)e,s_0\rangle \Rightarrow \langle e_2,s_2\rangle$
and *path-via*:$P \vdash Path\ last\ Cs\ to\ C\ via\ Cs'$ **and** *Ds*:$Ds = Cs\ @_p\ Cs'$
and *wt*:$P,E \vdash (\!|C|\!)e :: T$ **and** *sconf*:$P,E \vdash s_0\ \surd$
and *IH*:$\bigwedge e_2\ s_2\ T.\ [\![P,E \vdash \langle e,s_0\rangle \Rightarrow \langle e_2,s_2\rangle;\ P,E \vdash e :: T;\ P,E \vdash s_0\ \surd]\!]$
$\Longrightarrow ref\ (a,Cs) = e_2 \wedge s_1 = s_2$ **by** *fact+*
from *wt* **obtain** *D* **where** *class*:*is-class P C* **and** *wte*:$P,E \vdash e :: Class\ D$
and *disj*:$P \vdash Path\ D\ to\ C\ unique \vee$
$(P \vdash C \preceq^* D \wedge (\forall Cs.\ P \vdash Path\ C\ to\ D\ via\ Cs \longrightarrow Subobjs_R\ P\ C\ Cs))$
by *auto*
from *eval* **show** *?case*
proof(*rule eval-cases*)
fix *Xs Xs' a'*
assume *eval-ref*:$P,E \vdash \langle e,s_0\rangle \Rightarrow \langle ref\ (a',Xs),s_2\rangle$
and *path-via'*:$P \vdash Path\ last\ Xs\ to\ C\ via\ Xs'$
and *ref*:$e_2 = ref\ (a',Xs@_pXs')$
from *IH*[*OF eval-ref wte sconf*] **have** *eq*:$a = a' \wedge Cs = Xs \wedge s_1 = s_2$ **by** *simp*
with *wf eval-ref sconf wte* **have** *last*:*last Cs* = *D*
by(*auto dest*:*eval-preserves-type split*:*if-split-asm*)
from *disj* **show** *ref* $(a,Ds) = e_2 \wedge s_1 = s_2$
proof (*rule disjE*)
assume $P \vdash Path\ D\ to\ C\ unique$
with *path-via path-via' eq last* **have** $Cs' = Xs'$
by(*fastforce simp add*:*path-via-def path-unique-def*)
with *eq ref Ds* **show** *?thesis* **by** *simp*
next
assume $P \vdash C \preceq^* D \wedge (\forall Cs.\ P \vdash Path\ C\ to\ D\ via\ Cs \longrightarrow Subobjs_R\ P\ C\ Cs)$
with *class wf* **obtain** Cs'' **where** $P \vdash Path\ C\ to\ D\ via\ Cs''$
by(*auto dest*:*leq-implies-path*)
with *path-via path-via' wf eq last* **have** $Cs' = Xs'$
by(*auto dest*:*path-via-reverse*)
with *eq ref Ds* **show** *?thesis* **by** *simp*
qed
next
fix *Xs Xs' a'*
assume *eval-ref*:$P,E \vdash \langle e,s_0\rangle \Rightarrow \langle ref(a',Xs@C\#Xs'),s_2\rangle$
and *ref*:$e_2 = ref\ (a',Xs@[C])$
from *IH*[*OF eval-ref wte sconf*] **have** *eq*:$a = a' \wedge Cs = Xs@C\#Xs' \wedge s_1 = s_2$ **by** *simp*

with *wf eval-ref sconf wte* **obtain** C' **where**
last:*last Cs* = *D* **and** *Subobjs P C'* (*Xs*@*C*#*Xs'*)
by(*auto dest*:*eval-preserves-type split*:*if-split-asm*)
hence *subo*:*Subobjs P C* (*C*#*Xs'*) **by**(*fastforce intro*:*Subobjs-Subobjs*)
with *eq last* **have** *leq*:$P \vdash C \preceq^* D$ **by**(*fastforce dest*:*Subobjs-subclass*)
from *path-via last* **have** $P \vdash D \preceq^* C$
by(*auto dest*:*Subobjs-subclass simp*:*path-via-def*)
with *leq wf* **have** *CeqD*:$C = D$ **by**(*rule subcls-asym2*)
with *last path-via wf* **have** $Cs' = [D]$ **by**(*fastforce intro*:*path-via-C*)
with *Ds last* **have** *Ds'*:$Ds = Cs$ **by**(*simp add*:*appendPath-def*)
from *subo CeqD last eq wf* **have** $Xs' = []$ **by**(*auto dest*:*mdc-eq-last*)
with *eq Ds' ref* **show** *ref* $(a,Ds) = e_2 \wedge s_1 = s_2$ **by** *simp*
next
assume *eval-null*:$P,E \vdash \langle e,s_0\rangle \Rightarrow \langle null,s_2\rangle$
from *IH*[*OF eval-null wte sconf*] **show** *ref* $(a,Ds) = e_2 \wedge s_1 = s_2$ **by** *simp*
next
fix *Xs a'*
assume *eval-ref*:$P,E \vdash \langle e,s_0\rangle \Rightarrow \langle ref(a',Xs),s_2\rangle$ **and** *notin*:$C \notin set\ Xs$
and *notleq*:$\neg\ P \vdash last\ Xs \preceq^* C$ **and** *throw*:$e_2 = THROW\ ClassCast$
from *IH*[*OF eval-ref wte sconf*] **have** *eq*:$a = a' \wedge Cs = Xs \wedge s_1 = s_2$ **by** *simp*
with *wf eval-ref sconf wte* **have** *last*:*last Cs* = *D* **and** *notempty*:$Cs \neq []$
by(*auto dest*!:*eval-preserves-type Subobjs-nonempty split*:*if-split-asm*)
from *disj* **have** $C = D$
proof(*rule disjE*)
assume *path-unique*:$P \vdash Path\ D\ to\ C\ unique$
with *last* **have** $P \vdash D \preceq^* C$
by(*fastforce dest*:*Subobjs-subclass simp*:*path-unique-def*)
with *notleq last eq* **show** *?thesis* **by** *simp*
next
assume *ass*:$P \vdash C \preceq^* D \wedge$
$(\forall Cs.\ P \vdash Path\ C\ to\ D\ via\ Cs \longrightarrow Subobjs_R\ P\ C\ Cs)$
with *class wf* **obtain** *Cs''* **where** *path-via'*:$P \vdash Path\ C\ to\ D\ via\ Cs''$
by(*auto dest*:*leq-implies-path*)
with *path-via wf eq last* **have** $Cs'' = [D]$
by(*fastforce dest*:*path-via-reverse*)
with *ass path-via'* **have** $Subobjs_R\ P\ C\ [D]$ **by** *simp*
thus *?thesis* **by**(*fastforce dest*:*hd-SubobjsR*)
qed
with *last notin eq notempty* **show** *ref* $(a,Ds) = e_2 \wedge s_1 = s_2$
by(*fastforce intro*:*last-in-set*)
next
fix *e'* **assume** *eval-throw*:$P,E \vdash \langle e,s_0\rangle \Rightarrow \langle throw\ e',s_2\rangle$
from *IH*[*OF eval-throw wte sconf*] **show** *ref* $(a,Ds) = e_2 \wedge s_1 = s_2$ **by** *simp*
qed
next
case (*StaticDownCast E e* s_0 *a Cs C Cs'* s_1 e_2 s_2 *T*)
have *eval*:$P,E \vdash \langle (\!|C|\!)e,s_0\rangle \Rightarrow \langle e_2,s_2\rangle$
and *eval'*:$P,E \vdash \langle e,s_0\rangle \Rightarrow \langle ref(a,Cs@[C]@Cs'),s_1\rangle$
and *wt*:$P,E \vdash (\!|C|\!)e :: T$ **and** *sconf*:$P,E \vdash s_0\ \surd$

and *IH*:$\bigwedge e_2\ s_2\ T.\ [\![P,E \vdash \langle e,s_0\rangle \Rightarrow \langle e_2,s_2\rangle;\ P,E \vdash e :: T;\ P,E \vdash s_0\ \surd]\!]$
$\Longrightarrow ref(a,Cs@[C]@Cs') = e_2 \wedge s_1 = s_2$ **by** *fact+*
from *wt* **obtain** *D* **where** *wte*:$P,E \vdash e :: Class\ D$
and *disj*:$P \vdash Path\ D\ to\ C\ unique \vee$
$(P \vdash C \preceq^* D \wedge (\forall Cs.\ P \vdash Path\ C\ to\ D\ via\ Cs \longrightarrow Subobjs_R\ P\ C\ Cs))$
by *auto*
from *eval* **show** *?case*
proof(*rule eval-cases*)
fix *Xs Xs' a'*
assume *eval-ref*:$P,E \vdash \langle e,s_0\rangle \Rightarrow \langle ref(a',Xs),s_2\rangle$
and *path-via*:$P \vdash Path\ last\ Xs\ to\ C\ via\ Xs'$
and *ref*:$e_2 = ref\ (a',Xs@_pXs')$
from *IH*[*OF eval-ref wte sconf*] **have** *eq*:$a = a' \wedge Cs@[C]@Cs' = Xs \wedge s_1 = s_2$
by *simp*
with *wf eval-ref sconf wte* **obtain** *C'* **where**
last:$last(C\#Cs') = D$ **and** $Subobjs\ P\ C'\ (Cs@[C]@Cs')$
by(*auto dest*:*eval-preserves-type split*:*if-split-asm*)
hence $P \vdash Path\ C\ to\ D\ via\ C\#Cs'$
by(*fastforce intro*:*Subobjs-Subobjs simp*:*path-via-def*)
with *eq last path-via wf* **have** $Xs' = [C] \wedge Cs' = [] \wedge C = D$
apply *clarsimp*
apply(*split if-split-asm*)
by(*simp*,*drule path-via-reverse*,*simp*,*simp*)+
with *ref eq* **show** $ref(a,Cs@[C]) = e_2 \wedge s_1 = s_2$ **by**(*fastforce simp*:*appendPath-def*)
next
fix *Xs Xs' a'*
assume *eval-ref*:$P,E \vdash \langle e,s_0\rangle \Rightarrow \langle ref(a',Xs@C\#Xs'),s_2\rangle$
and *ref*:$e_2 = ref\ (a',Xs@[C])$
from *IH*[*OF eval-ref wte sconf*] **have** *eq*:$a = a' \wedge Cs@[C]@Cs' = Xs@C\#Xs' \wedge s_1 = s_2$
by *simp*
with *wf eval-ref sconf wte* **obtain** *C'* **where**
last:$last(C\#Xs') = D$ **and** *subo*:$Subobjs\ P\ C'\ (Cs@[C]@Cs')$
by(*auto dest*:*eval-preserves-type split*:*if-split-asm*)
from *subo wf* **have** *notin*:$C \notin set\ Cs$ **by** $-$(*rule unique2*,*simp*)
from *subo wf* **have** $C \notin set\ Cs'$ **by** $-$(*rule unique1*,*simp*,*simp*)
with *notin eq* **have** $Cs = Xs \wedge Cs' = Xs'$
by $-$(*rule only-one-append*,*simp+*)
with *eq ref* **show** $ref(a,Cs@[C]) = e_2 \wedge s_1 = s_2$ **by** *simp*
next
assume *eval-null*:$P,E \vdash \langle e,s_0\rangle \Rightarrow \langle null,s_2\rangle$
from *IH*[*OF eval-null wte sconf*] **show** $ref\ (a,Cs@[C]) = e_2 \wedge s_1 = s_2$ **by** *simp*
next
fix *Xs a'*
assume *eval-ref*:$P,E \vdash \langle e,s_0\rangle \Rightarrow \langle ref(a',Xs),s_2\rangle$ **and** *notin*:$C \notin set\ Xs$
from *IH*[*OF eval-ref wte sconf*] **have** $a = a' \wedge Cs@[C]@Cs' = Xs \wedge s_1 = s_2$
by *simp*

with *notin* **show** *ref*$(a,Cs@[C]) = e_2 \wedge s_1 = s_2$ **by** *fastforce*
next
fix e' **assume** *eval-throw*:$P,E \vdash \langle e,s_0\rangle \Rightarrow \langle throw\ e',s_2\rangle$
from *IH*[*OF eval-throw wte sconf*] **show** *ref* $(a,Cs@[C]) = e_2 \wedge s_1 = s_2$ **by** *simp*
qed
next
case (*StaticCastNull E e* s_0 s_1 *C* e_2 s_2 *T*)
have *eval*:$P,E \vdash \langle(\!|C|\!)e,s_0\rangle \Rightarrow \langle e_2,s_2\rangle$
and *wt*:$P,E \vdash (\!|C|\!)e :: T$ **and** *sconf*:$P,E \vdash s_0\ \surd$
and *IH*:$\bigwedge e_2\ s_2\ T.\ [\![P,E \vdash \langle e,s_0\rangle \Rightarrow \langle e_2,s_2\rangle;\ P,E \vdash e :: T;\ P,E \vdash s_0\ \surd]\!] \Longrightarrow null = e_2 \wedge s_1 = s_2$ **by** *fact+*
from *wt* **obtain** *D* **where** *wte*:$P,E \vdash e :: Class\ D$ **by** *auto*
from *eval* **show** *?case*
proof(*rule eval-cases*)
fix *Xs Xs' a'*
assume *eval-ref*:$P,E \vdash \langle e,s_0\rangle \Rightarrow \langle ref\ (a',Xs),s_2\rangle$
from *IH*[*OF eval-ref wte sconf*] **show** *null* $= e_2 \wedge s_1 = s_2$ **by** *simp*
next
fix *Xs Xs' a'*
assume *eval-ref*:$P,E \vdash \langle e,s_0\rangle \Rightarrow \langle ref(a',Xs@C\#Xs'),s_2\rangle$
from *IH*[*OF eval-ref wte sconf*] **show** *null* $= e_2 \wedge s_1 = s_2$ **by** *simp*
next
assume *eval-null*:$P,E \vdash \langle e,s_0\rangle \Rightarrow \langle null,s_2\rangle$ **and** $e_2 = null$
with *IH*[*OF eval-null wte sconf*] **show** *null* $= e_2 \wedge s_1 = s_2$ **by** *simp*
next
fix *Xs a'*
assume *eval-ref*:$P,E \vdash \langle e,s_0\rangle \Rightarrow \langle ref(a',Xs),s_2\rangle$
from *IH*[*OF eval-ref wte sconf*] **show** *null* $= e_2 \wedge s_1 = s_2$ **by** *simp*
next
fix e' **assume** *eval-throw*:$P,E \vdash \langle e,s_0\rangle \Rightarrow \langle throw\ e',s_2\rangle$
from *IH*[*OF eval-throw wte sconf*] **show** *null* $= e_2 \wedge s_1 = s_2$ **by** *simp*
qed
next
case (*StaticCastFail E e* s_0 *a Cs* s_1 *C* e_2 s_2 *T*)
have *eval*:$P,E \vdash \langle(\!|C|\!)e,s_0\rangle \Rightarrow \langle e_2,s_2\rangle$
and *notleq*:$\neg\ P \vdash last\ Cs \preceq^* C$ **and** *notin*:$C \notin set\ Cs$
and *wt*:$P,E \vdash (\!|C|\!)e :: T$ **and** *sconf*:$P,E \vdash s_0\ \surd$
and *IH*:$\bigwedge e_2\ s_2\ T.\ [\![P,E \vdash \langle e,s_0\rangle \Rightarrow \langle e_2,s_2\rangle;\ P,E \vdash e :: T;\ P,E \vdash s_0\ \surd]\!] \Longrightarrow ref\ (a,\ Cs) = e_2 \wedge s_1 = s_2$ **by** *fact+*
from *wt* **obtain** *D* **where** *wte*:$P,E \vdash e :: Class\ D$ **by** *auto*
from *eval* **show** *?case*
proof(*rule eval-cases*)
fix *Xs Xs' a'*
assume *eval-ref*:$P,E \vdash \langle e,s_0\rangle \Rightarrow \langle ref\ (a',Xs),s_2\rangle$
and *path-via*:$P \vdash Path\ last\ Xs\ to\ C\ via\ Xs'$
from *IH*[*OF eval-ref wte sconf*] **have** *eq*:$a = a' \wedge Cs = Xs \wedge s_1 = s_2$ **by** *simp*
with *path-via wf* **have** $P \vdash last\ Cs \preceq^* C$
by(*auto dest*:*Subobjs-subclass simp*:*path-via-def*)

with *notleq* **show** *THROW ClassCast* = $e_2 \wedge s_1 = s_2$ **by** *simp*
next
fix *Xs Xs′ a′*
assume *eval-ref*:$P,E \vdash \langle e,s_0\rangle \Rightarrow \langle ref(a',Xs@C\#Xs'),s_2\rangle$
from *IH*[*OF eval-ref wte sconf*] **have** $a = a' \wedge Cs = Xs@C\#Xs' \wedge s_1 = s_2$ **by** *simp*
with *notin* **show** *THROW ClassCast* = $e_2 \wedge s_1 = s_2$ **by** *simp*
next
assume *eval-null*:$P,E \vdash \langle e,s_0\rangle \Rightarrow \langle null,s_2\rangle$
from *IH*[*OF eval-null wte sconf*] **show** *THROW ClassCast* = $e_2 \wedge s_1 = s_2$ **by** *simp*
next
fix *Xs a′*
assume *eval-ref*:$P,E \vdash \langle e,s_0\rangle \Rightarrow \langle ref(a',Xs),s_2\rangle$
and *throw*:e_2 = *THROW ClassCast*
from *IH*[*OF eval-ref wte sconf*] **have** $a = a' \wedge Cs = Xs \wedge s_1 = s_2$
by *simp*
with *throw* **show** *THROW ClassCast* = $e_2 \wedge s_1 = s_2$ **by** *simp*
next
fix *e′* **assume** *eval-throw*:$P,E \vdash \langle e,s_0\rangle \Rightarrow \langle throw\ e',s_2\rangle$
from *IH*[*OF eval-throw wte sconf*] **show** *THROW ClassCast* = $e_2 \wedge s_1 = s_2$ **by** *simp*
qed
next
case (*StaticCastThrow E e* s_0 *e′* s_1 *C* e_2 s_2 *T*)
have *eval*:$P,E \vdash \langle(\!|C|\!)e,s_0\rangle \Rightarrow \langle e_2,s_2\rangle$
and *wt*:$P,E \vdash (\!|C|\!)e :: T$ **and** *sconf*:$P,E \vdash s_0\ \surd$
and *IH*:$\bigwedge e_2\ s_2\ T.\ [\![P,E \vdash \langle e,s_0\rangle \Rightarrow \langle e_2,s_2\rangle;\ P,E \vdash e :: T;\ P,E \vdash s_0\ \surd]\!]$
$\Longrightarrow$ *throw e′* = $e_2 \wedge s_1 = s_2$ **by** *fact+*
from *wt* **obtain** *D* **where** *wte*:$P,E \vdash e :: Class\ D$ **by** *auto*
from *eval* **show** *?case*
proof(*rule eval-cases*)
fix *Xs Xs′ a′*
assume *eval-ref*:$P,E \vdash \langle e,s_0\rangle \Rightarrow \langle ref\ (a',Xs),s_2\rangle$
from *IH*[*OF eval-ref wte sconf*] **show** *throw e′* = $e_2 \wedge s_1 = s_2$ **by** *simp*
next
fix *Xs Xs′ a′*
assume *eval-ref*:$P,E \vdash \langle e,s_0\rangle \Rightarrow \langle ref(a',Xs@C\#Xs'),s_2\rangle$
from *IH*[*OF eval-ref wte sconf*] **show** *throw e′* = $e_2 \wedge s_1 = s_2$ **by** *simp*
next
assume *eval-null*:$P,E \vdash \langle e,s_0\rangle \Rightarrow \langle null,s_2\rangle$
from *IH*[*OF eval-null wte sconf*] **show** *throw e′* = $e_2 \wedge s_1 = s_2$ **by** *simp*
next
fix *Xs a′*
assume *eval-ref*:$P,E \vdash \langle e,s_0\rangle \Rightarrow \langle ref(a',Xs),s_2\rangle$
from *IH*[*OF eval-ref wte sconf*] **show** *throw e′* = $e_2 \wedge s_1 = s_2$ **by** *simp*
next
fix *e″* **assume** *eval-throw*:$P,E \vdash \langle e,s_0\rangle \Rightarrow \langle throw\ e'',s_2\rangle$
and *throw*:e_2 = *throw e″*

from *IH*[*OF eval-throw wte sconf*] *throw* **show** *throw* $e' = e_2 \wedge s_1 = s_2$ **by** *simp*
qed
next
case (*StaticUpDynCast E e* s_0 *a Cs* s_1 *C Cs' Ds* e_2 s_2 *T*)
have *eval*:$P,E \vdash \langle Cast\ C\ e,s_0\rangle \Rightarrow \langle e_2,s_2\rangle$
and *path-via*:$P \vdash Path\ last\ Cs\ to\ C\ via\ Cs'$
and *path-unique*:$P \vdash Path\ last\ Cs\ to\ C\ unique$
and *Ds*:$Ds = Cs@_pCs'$ **and** *wt*:$P,E \vdash Cast\ C\ e :: T$ **and** *sconf*:$P,E \vdash s_0\ \surd$
and *IH*:$\bigwedge e_2\ s_2\ T.\ [\![P,E \vdash \langle e,s_0\rangle \Rightarrow \langle e_2,s_2\rangle;\ P,E \vdash e :: T;\ P,E \vdash s_0\ \surd]\!]$
$\Longrightarrow ref(a,Cs) = e_2 \wedge s_1 = s_2$ **by** *fact+*
from *wt* **obtain** *D* **where** *wte*:$P,E \vdash e :: Class\ D$ **by** *auto*
from *eval* **show** *?case*
proof(*rule eval-cases*)
fix *Xs Xs' a'*
assume *eval-ref*:$P,E \vdash \langle e,s_0\rangle \Rightarrow \langle ref\ (a',Xs),s_2\rangle$
and *path-via'*:$P \vdash Path\ last\ Xs\ to\ C\ via\ Xs'$
and *ref*:$e_2 = ref\ (a',Xs@_pXs')$
from *IH*[*OF eval-ref wte sconf*] **have** *eq*:$a = a' \wedge Cs = Xs \wedge s_1 = s_2$ **by** *simp*
with *wf eval-ref sconf wte* **have** *last*:$last\ Cs = D$
by(*auto dest*:*eval-preserves-type split*:*if-split-asm*)
with *path-unique path-via path-via' eq* **have** $Xs' = Cs'$
by(*fastforce simp*:*path-via-def path-unique-def*)
with *eq Ds ref* **show** *ref* $(a,\ Ds) = e_2 \wedge s_1 = s_2$ **by** *simp*
next
fix *Xs Xs' a'*
assume *eval-ref*:$P,E \vdash \langle e,s_0\rangle \Rightarrow \langle ref(a',Xs@C\#Xs'),s_2\rangle$
and *ref*:$e_2 = ref\ (a',Xs@[C])$
from *IH*[*OF eval-ref wte sconf*] **have** *eq*:$a = a' \wedge Cs = Xs@C\#Xs' \wedge s_1 = s_2$ **by** *simp*
with *wf eval-ref sconf wte* **obtain** C' **where**
last:$last\ Cs = D$ **and** $Subobjs\ P\ C'\ (Xs@C\#Xs')$
by(*auto dest*:*eval-preserves-type split*:*if-split-asm*)
hence $Subobjs\ P\ C\ (C\#Xs')$ **by**(*fastforce intro*:*Subobjs-Subobjs*)
with *last eq* **have** $P \vdash Path\ C\ to\ D\ via\ C\#Xs'$
by(*simp add*:*path-via-def*)
with *path-via wf last* **have** $Xs' = [] \wedge Cs' = [C] \wedge C = D$
by(*fastforce dest*:*path-via-reverse*)
with *eq Ds ref* **show** *ref* $(a,\ Ds) = e_2 \wedge s_1 = s_2$ **by** (*simp add*:*appendPath-def*)
next
fix *Xs Xs' D' S a' h l*
assume *eval-ref*:$P,E \vdash \langle e,s_0\rangle \Rightarrow \langle ref(a',Xs),(h,l)\rangle$
and *h*:$h\ a' = Some(D',S)$ **and** *path-via'*:$P \vdash Path\ D'\ to\ C\ via\ Xs'$
and *path-unique'*:$P \vdash Path\ D'\ to\ C\ unique$ **and** *s2*:$s_2 = (h,l)$
and *ref*:$e_2 = ref(a',Xs')$
from *IH*[*OF eval-ref wte sconf*] *s2* **have** *eq*:$a = a' \wedge Cs = Xs \wedge s_1 = s_2$ **by** *simp*
with *wf eval-ref sconf wte h* **have** $last\ Cs = D$
and $Subobjs\ P\ D'\ Cs$

by(*auto dest*:*eval-preserves-type split*:*if-split-asm*)
with *path-via wf* **have** $P \vdash Path\ D'\ to\ C\ via\ Cs@_p Cs'$
by(*fastforce intro*:*Subobjs-appendPath appendPath-last*[*THEN sym*]
dest:*Subobjs-nonempty simp*:*path-via-def*)
with *path-via'* *path-unique'* *Ds* **have** $Xs' = Ds$
by(*fastforce simp*:*path-via-def path-unique-def*)
with *eq ref* **show** *ref* $(a,\ Ds) = e_2 \wedge s_1 = s_2$ **by** *simp*
next
assume *eval-null*:$P,E \vdash \langle e,s_0\rangle \Rightarrow \langle null,s_2\rangle$
from *IH*[*OF eval-null wte sconf*] **show** *ref* $(a,\ Ds) = e_2 \wedge s_1 = s_2$ **by** *simp*
next
fix *Xs D' S a' h l*
assume *eval-ref*:$P,E \vdash \langle e,s_0\rangle \Rightarrow \langle ref(a',Xs),(h,l)\rangle$
and *not-unique*:$\neg\ P \vdash Path\ last\ Xs\ to\ C\ unique$ **and** *s2*:$s_2 = (h,l)$
from *IH*[*OF eval-ref wte sconf*] *s2* **have** *eq*:$a = a' \wedge Cs = Xs \wedge s_1 = s_2$ **by** *simp*
with *path-unique not-unique* **show** *ref* $(a,\ Ds) = e_2 \wedge s_1 = s_2$ **by** *simp*
next
fix *e'* **assume** *eval-throw*:$P,E \vdash \langle e,s_0\rangle \Rightarrow \langle throw\ e',s_2\rangle$
from *IH*[*OF eval-throw wte sconf*] **show** *ref* $(a,\ Ds) = e_2 \wedge s_1 = s_2$ **by** *simp*
qed
next
case (*StaticDownDynCast E e* s_0 *a Cs C Cs'* s_1 e_2 s_2 *T*)
have *eval*:$P,E \vdash \langle Cast\ C\ e,s_0\rangle \Rightarrow \langle e_2,s_2\rangle$
and *wt*:$P,E \vdash Cast\ C\ e :: T$ **and** *sconf*:$P,E \vdash s_0\ \surd$
and *IH*:$\bigwedge e_2\ s_2\ T.\ [\![P,E \vdash \langle e,s_0\rangle \Rightarrow \langle e_2,s_2\rangle;\ P,E \vdash e :: T;\ P,E \vdash s_0\ \surd]\!]$
$\Longrightarrow ref(a,Cs@[C]@Cs') = e_2 \wedge s_1 = s_2$ **by** *fact*+
from *wt* **obtain** *D* **where** *wte*:$P,E \vdash e :: Class\ D$ **by** *auto*
from *eval* **show** *?case*
proof(*rule eval-cases*)
fix *Xs Xs' a'*
assume *eval-ref*:$P,E \vdash \langle e,s_0\rangle \Rightarrow \langle ref(a',Xs),s_2\rangle$
and *path-via*:$P \vdash Path\ last\ Xs\ to\ C\ via\ Xs'$
and *ref*:$e_2 = ref\ (a',Xs@_p Xs')$
from *IH*[*OF eval-ref wte sconf*] **have** *eq*:$a = a' \wedge Cs@[C]@Cs' = Xs \wedge s_1 = s_2$
by *simp*
with *wf eval-ref sconf wte* **obtain** *C'* **where**
last:$last(C\#Cs') = D$ **and** $Subobjs\ P\ C'\ (Cs@[C]@Cs')$
by(*auto dest*:*eval-preserves-type split*:*if-split-asm*)
hence $P \vdash Path\ C\ to\ D\ via\ C\#Cs'$
by(*fastforce intro*:*Subobjs-Subobjs simp*:*path-via-def*)
with *eq last path-via wf* **have** $Xs' = [C] \wedge Cs' = [] \wedge C = D$
apply *clarsimp*
apply(*split if-split-asm*)
by(*simp*,*drule path-via-reverse*,*simp*,*simp*)+
with *ref eq* **show** $ref(a,Cs@[C]) = e_2 \wedge s_1 = s_2$ **by**(*fastforce simp*:*appendPath-def*)
next
fix *Xs Xs' a'*

assume *eval-ref*:$P,E \vdash \langle e,s_0\rangle \Rightarrow \langle ref(a',Xs@C\#Xs'),s_2\rangle$
and *ref*:$e_2 = ref\ (a',Xs@[C])$
from *IH*[*OF eval-ref wte sconf*] **have** *eq*:$a = a' \wedge Cs@[C]@Cs' = Xs@C\#Xs' \wedge s_1 = s_2$
by *simp*
with *wf eval-ref sconf wte* **obtain** C' **where**
last:$last(C\#Xs') = D$ **and** *subo*:*Subobjs P* C' $(Cs@[C]@Cs')$
by(*auto dest*:*eval-preserves-type split*:*if-split-asm*)
from *subo wf* **have** *notin*:$C \notin set\ Cs$ **by** $-$(*rule unique2*,*simp*)
from *subo wf* **have** $C \notin set\ Cs'$ **by** $-$(*rule unique1*,*simp*,*simp*)
with *notin eq* **have** $Cs = Xs \wedge Cs' = Xs'$
by $-$(*rule only-one-append*,*simp*+)
with *eq ref* **show** $ref(a,Cs@[C]) = e_2 \wedge s_1 = s_2$ **by** *simp*
next
fix *Xs Xs' D' S a' h l*
assume *eval-ref*:$P,E \vdash \langle e,s_0\rangle \Rightarrow \langle ref(a',Xs),(h,l)\rangle$
and *h*:*h* $a' = Some(D',S)$ **and** *path-via*:$P \vdash Path\ D'\ to\ C\ via\ Xs'$
and *path-unique*:$P \vdash Path\ D'\ to\ C\ unique$ **and** *s2*:$s_2 = (h,l)$
and *ref*:$e_2 = ref(a',Xs')$
from *IH*[*OF eval-ref wte sconf*] *s2* **have** *eq*:$a = a' \wedge Cs@[C]@Cs' = Xs \wedge s_1 = s_2$
by *simp*
with *wf eval-ref sconf wte h* **have** *Subobjs P D'* $(Cs@[C]@Cs')$
by(*auto dest*:*eval-preserves-type split*:*if-split-asm*)
hence *Subobjs P D'* $(Cs@[C])$ **by**(*fastforce intro*:*appendSubobj*)
with *path-via path-unique* **have** $Xs' = Cs@[C]$
by(*fastforce simp*:*path-via-def path-unique-def*)
with *eq ref* **show** $ref(a,Cs@[C]) = e_2 \wedge s_1 = s_2$ **by** *simp*
next
assume *eval-null*:$P,E \vdash \langle e,s_0\rangle \Rightarrow \langle null,s_2\rangle$
from *IH*[*OF eval-null wte sconf*] **show** *ref* $(a,Cs@[C]) = e_2 \wedge s_1 = s_2$ **by** *simp*
next
fix *Xs D' S a' h l*
assume *eval-ref*:$P,E \vdash \langle e,s_0\rangle \Rightarrow \langle ref(a',Xs),(h,l)\rangle$
and *notin*:$C \notin set\ Xs$ **and** *s2*:$s_2 = (h,l)$
from *IH*[*OF eval-ref wte sconf*] *s2* **have** $a = a' \wedge Cs@[C]@Cs' = Xs \wedge s_1 = s_2$
by *simp*
with *notin* **show** *ref* $(a,Cs@[C]) = e_2 \wedge s_1 = s_2$ **by** *fastforce*
next
fix e' **assume** *eval-throw*:$P,E \vdash \langle e,s_0\rangle \Rightarrow \langle throw\ e',s_2\rangle$
from *IH*[*OF eval-throw wte sconf*] **show** *ref* $(a,Cs@[C]) = e_2 \wedge s_1 = s_2$ **by** *simp*
qed
next
case (*DynCast E e* s_0 *a Cs h l D S C Cs'* e_2 s_2 *T*)
have *eval*:$P,E \vdash \langle Cast\ C\ e,s_0\rangle \Rightarrow \langle e_2,s_2\rangle$
and *path-via*:$P \vdash Path\ D\ to\ C\ via\ Cs'$ **and** *path-unique*:$P \vdash Path\ D\ to\ C\ unique$

and h:$h\ a = Some(D,S)$ **and** wt:$P,E \vdash Cast\ C\ e :: T$ **and** $sconf$:$P,E \vdash s_0\ \surd$
and IH:$\bigwedge e_2\ s_2\ T.\ [\![P,E \vdash \langle e,s_0\rangle \Rightarrow \langle e_2,s_2\rangle;\ P,E \vdash e :: T;\ P,E \vdash s_0\ \surd]\!]$
$\Longrightarrow ref(a,Cs) = e_2 \wedge (h,l) = s_2$ **by** *fact+*
from wt **obtain** D' **where** wte:$P,E \vdash e :: Class\ D'$ **by** *auto*
from *eval* **show** *?case*
proof(*rule eval-cases*)
fix $Xs\ Xs'\ a'$
assume $eval\text{-}ref$:$P,E \vdash \langle e,s_0\rangle \Rightarrow \langle ref(a',Xs),s_2\rangle$
and $path\text{-}via'$:$P \vdash Path\ last\ Xs\ to\ C\ via\ Xs'$
and ref:$e_2 = ref\ (a',Xs@_pXs')$
from $IH[OF\ eval\text{-}ref\ wte\ sconf]$ **have** eq:$a = a' \wedge Cs = Xs \wedge (h,l) = s_2$ **by** *simp*
with $wf\ eval\text{-}ref\ sconf\ wte\ h$ **have** $last\ Cs = D'$
and $Subobjs\ P\ D\ Cs$
by(*auto dest*:*eval-preserves-type split*:*if-split-asm*)
with $path\text{-}via'\ wf\ eq$ **have** $P \vdash Path\ D\ to\ C\ via\ Xs@_pXs'$
by(*fastforce intro*:*Subobjs-appendPath appendPath-last*[*THEN sym*]
dest:*Subobjs-nonempty simp*:*path-via-def*)
with *path-via path-unique* **have** $Cs' = Xs@_pXs'$
by(*fastforce simp*:*path-via-def path-unique-def*)
with *ref eq* **show** $ref(a,Cs') = e_2 \wedge (h,\ l) = s_2$ **by** *simp*
next
fix $Xs\ Xs'\ a'$
assume $eval\text{-}ref$:$P,E \vdash \langle e,s_0\rangle \Rightarrow \langle ref(a',Xs@C\#Xs'),s_2\rangle$
and ref:$e_2 = ref\ (a',Xs@[C])$
from $IH[OF\ eval\text{-}ref\ wte\ sconf]$ **have** eq:$a = a' \wedge Cs = Xs@C\#Xs' \wedge (h,l) = s_2$
by *simp*
with $wf\ eval\text{-}ref\ sconf\ wte\ h$ **have** $Subobjs\ P\ D\ (Xs@[C]@Xs')$
by(*auto dest*:*eval-preserves-type split*:*if-split-asm*)
hence $Subobjs\ P\ D\ (Xs@[C])$ **by**(*fastforce intro*:*appendSubobj*)
with *path-via path-unique* **have** $Cs' = Xs@[C]$
by(*fastforce simp*:*path-via-def path-unique-def*)
with *eq ref* **show** $ref(a,Cs') = e_2 \wedge (h,\ l) = s_2$ **by** *simp*
next
fix $Xs\ Xs'\ D''\ S'\ a'\ h'\ l'$
assume $eval\text{-}ref$:$P,E \vdash \langle e,s_0\rangle \Rightarrow \langle ref(a',Xs),(h',l')\rangle$
and h':$h'\ a' = Some(D'',S')$ **and** $path\text{-}via'$:$P \vdash Path\ D''\ to\ C\ via\ Xs'$
and $s2$:$s_2 = (h',l')$ **and** ref:$e_2 = ref(a',Xs')$
from $IH[OF\ eval\text{-}ref\ wte\ sconf]$ **have** eq:$a = a' \wedge Cs = Xs \wedge h = h' \wedge l = l'$
by *simp*
with $h\ h'$ *path-via path-via' path-unique s2 ref*
show $ref(a,Cs') = e_2 \wedge (h,l) = s_2$
by(*fastforce simp*:*path-via-def path-unique-def*)
next
assume $eval\text{-}null$:$P,E \vdash \langle e,s_0\rangle \Rightarrow \langle null,s_2\rangle$
from $IH[OF\ eval\text{-}null\ wte\ sconf]$ **show** $ref(a,Cs') = e_2 \wedge (h,l) = s_2$ **by** *simp*
next
fix $Xs\ D''\ S'\ a'\ h'\ l'$

assume *eval-ref*:$P,E \vdash \langle e,s_0\rangle \Rightarrow \langle ref(a',Xs),(h',l')\rangle$
and h':$h'\ a' = Some(D'',S')$ **and** *not-unique*:$\neg\ P \vdash Path\ D''\ to\ C\ unique$
from *IH*[*OF eval-ref wte sconf*] **have** *eq*:$a = a' \wedge Cs = Xs \wedge h = h' \wedge l = l'$
by *simp*
with $h\ h'$ *path-unique not-unique* **show** $ref(a,Cs') = e_2 \wedge (h,l) = s_2$ **by** *simp*
next
fix e' **assume** *eval-throw*:$P,E \vdash \langle e,s_0\rangle \Rightarrow \langle throw\ e',s_2\rangle$
from *IH*[*OF eval-throw wte sconf*] **show** $ref\ (a,Cs') = e_2 \wedge (h,l) = s_2$ **by** *simp*
qed
next
case (*DynCastNull E e* $s_0\ s_1\ C\ e_2\ s_2\ T$)
have *eval*:$P,E \vdash \langle Cast\ C\ e,s_0\rangle \Rightarrow \langle e_2,s_2\rangle$
and *wt*:$P,E \vdash Cast\ C\ e :: T$ **and** *sconf*:$P,E \vdash s_0\ \surd$
and *IH*:$\bigwedge e_2\ s_2\ T.\ [\![P,E \vdash \langle e,s_0\rangle \Rightarrow \langle e_2,s_2\rangle;\ P,E \vdash e :: T;\ P,E \vdash s_0\ \surd]\!]$
$\Longrightarrow null = e_2 \wedge s_1 = s_2$ **by** *fact+*
from *wt* **obtain** *D* **where** *wte*:$P,E \vdash e :: Class\ D$ **by** *auto*
from *eval* **show** *?case*
proof(*rule eval-cases*)
fix *Xs Xs' a'*
assume *eval-ref*:$P,E \vdash \langle e,s_0\rangle \Rightarrow \langle ref\ (a',Xs),s_2\rangle$
from *IH*[*OF eval-ref wte sconf*] **show** $null = e_2 \wedge s_1 = s_2$ **by** *simp*
next
fix *Xs Xs' a'*
assume *eval-ref*:$P,E \vdash \langle e,s_0\rangle \Rightarrow \langle ref(a',Xs@C\#Xs'),s_2\rangle$
from *IH*[*OF eval-ref wte sconf*] **show** $null = e_2 \wedge s_1 = s_2$ **by** *simp*
next
fix *Xs Xs' D' S a' h l*
assume *eval-ref*:$P,E \vdash \langle e,s_0\rangle \Rightarrow \langle ref(a',Xs),(h,l)\rangle$
from *IH*[*OF eval-ref wte sconf*] **show** $null = e_2 \wedge s_1 = s_2$ **by** *simp*
next
assume *eval-null*:$P,E \vdash \langle e,s_0\rangle \Rightarrow \langle null,s_2\rangle$ **and** $e_2 = null$
with *IH*[*OF eval-null wte sconf*] **show** $null = e_2 \wedge s_1 = s_2$ **by** *simp*
next
fix *Xs D' S a' h l*
assume *eval-ref*:$P,E \vdash \langle e,s_0\rangle \Rightarrow \langle ref(a',Xs),(h,l)\rangle$ **and** *s2*:$s_2 = (h,l)$
from *IH*[*OF eval-ref wte sconf*] *s2* **show** $null = e_2 \wedge s_1 = s_2$ **by** *simp*
next
fix e' **assume** *eval-throw*:$P,E \vdash \langle e,s_0\rangle \Rightarrow \langle throw\ e',s_2\rangle$
from *IH*[*OF eval-throw wte sconf*] **show** $null = e_2 \wedge s_1 = s_2$ **by** *simp*
qed
next
case (*DynCastFail E e* $s_0\ a\ Cs\ h\ l\ D\ S\ C\ e_2\ s_2\ T$)
have *eval*:$P,E \vdash \langle Cast\ C\ e,s_0\rangle \Rightarrow \langle e_2,s_2\rangle$
and *h*:$h\ a = Some(D,S)$ **and** *not-unique1*:$\neg\ P \vdash Path\ D\ to\ C\ unique$
and *not-unique2*:$\neg\ P \vdash Path\ last\ Cs\ to\ C\ unique$ **and** *notin*:$C \notin set\ Cs$
and *wt*:$P,E \vdash Cast\ C\ e :: T$ **and** *sconf*:$P,E \vdash s_0\ \surd$
and *IH*:$\bigwedge e_2\ s_2\ T.\ [\![P,E \vdash \langle e,s_0\rangle \Rightarrow \langle e_2,s_2\rangle;\ P,E \vdash e :: T;\ P,E \vdash s_0\ \surd]\!]$
$\Longrightarrow ref\ (a,\ Cs) = e_2 \wedge (h,l) = s_2$ **by** *fact+*

from *wt* **obtain** D' **where** *wte*:$P,E \vdash e :: Class\ D'$ **by** *auto*
from *eval* **show** *?case*
proof(*rule eval-cases*)
fix *Xs Xs' a'*
assume *eval-ref*:$P,E \vdash \langle e,s_0\rangle \Rightarrow \langle ref(a',Xs),s_2\rangle$
and *path-unique*:$P \vdash$ *Path last Xs to C unique*
from *IH*[*OF eval-ref wte sconf*] **have** *eq*:$a = a' \wedge Cs = Xs \wedge (h,l) = s_2$ **by** *simp*
with *path-unique not-unique2* **show** $null = e_2 \wedge (h,l) = s_2$ **by** *simp*
next
fix *Xs Xs' a'*
assume *eval-ref*:$P,E \vdash \langle e,s_0\rangle \Rightarrow \langle ref(a',Xs@C\#Xs'),s_2\rangle$
from *IH*[*OF eval-ref wte sconf*] **have** *eq*:$a = a' \wedge Cs = Xs@C\#Xs' \wedge (h,l) = s_2$
by *simp*
with *notin* **show** $null = e_2 \wedge (h,l) = s_2$ **by** *fastforce*
next
fix *Xs Xs' D'' S' a' h' l'*
assume *eval-ref*:$P,E \vdash \langle e,s_0\rangle \Rightarrow \langle ref(a',Xs),(h',l')\rangle$
and *h'*:$h'\ a' = Some(D'',S')$ **and** *path-unique*:$P \vdash$ *Path D'' to C unique*
from *IH*[*OF eval-ref wte sconf*] **have** $a = a' \wedge Cs = Xs \wedge h = h' \wedge l = l'$
by *simp*
with *h h' not-unique1 path-unique* **show** $null = e_2 \wedge (h,l) = s_2$ **by** *simp*
next
assume *eval-null*:$P,E \vdash \langle e,s_0\rangle \Rightarrow \langle null,s_2\rangle$
from *IH*[*OF eval-null wte sconf*] **show** $null = e_2 \wedge (h,l) = s_2$ **by** *simp*
next
fix *Xs D'' S' a' h' l'*
assume *eval-ref*:$P,E \vdash \langle e,s_0\rangle \Rightarrow \langle ref(a',Xs),(h',l')\rangle$
and *null*:$e_2 = null$ **and** *s2*:$s_2 = (h',l')$
from *IH*[*OF eval-ref wte sconf*] *null s2* **show** $null = e_2 \wedge (h,l) = s_2$ **by** *simp*
next
fix e' **assume** *eval-throw*:$P,E \vdash \langle e,s_0\rangle \Rightarrow \langle throw\ e',s_2\rangle$
from *IH*[*OF eval-throw wte sconf*] **show** $null = e_2 \wedge (h,l) = s_2$ **by** *simp*
qed
next
case (*DynCastThrow E e* s_0 e' s_1 *C* e_2 s_2 *T*)
have *eval*:$P,E \vdash \langle Cast\ C\ e,s_0\rangle \Rightarrow \langle e_2,s_2\rangle$
and *wt*:$P,E \vdash Cast\ C\ e :: T$ **and** *sconf*:$P,E \vdash s_0\ \surd$
and *IH*:$\bigwedge e_2\ s_2\ T.\ [\![P,E \vdash \langle e,s_0\rangle \Rightarrow \langle e_2,s_2\rangle;\ P,E \vdash e :: T;\ P,E \vdash s_0\ \surd]\!] \Longrightarrow throw\ e' = e_2 \wedge s_1 = s_2$ **by** *fact+*
from *wt* **obtain** *D* **where** *wte*:$P,E \vdash e :: Class\ D$ **by** *auto*
from *eval* **show** *?case*
proof(*rule eval-cases*)
fix *Xs Xs' a'*
assume *eval-ref*:$P,E \vdash \langle e,s_0\rangle \Rightarrow \langle ref\ (a',Xs),s_2\rangle$
from *IH*[*OF eval-ref wte sconf*] **show** $throw\ e' = e_2 \wedge s_1 = s_2$ **by** *simp*
next
fix *Xs Xs' a'*

assume *eval-ref*:$P,E \vdash \langle e,s_0\rangle \Rightarrow \langle ref(a',Xs@C\#Xs'),s_2\rangle$
from *IH*[*OF eval-ref wte sconf*] **show** *throw* $e' = e_2 \wedge s_1 = s_2$ **by** *simp*
next
fix *Xs Xs' D'' S' a' h' l'*
assume *eval-ref*:$P,E \vdash \langle e,s_0\rangle \Rightarrow \langle ref(a',Xs),(h',l')\rangle$
from *IH*[*OF eval-ref wte sconf*] **show** *throw* $e' = e_2 \wedge s_1 = s_2$ **by** *simp*
next
assume *eval-null*:$P,E \vdash \langle e,s_0\rangle \Rightarrow \langle null,s_2\rangle$
from *IH*[*OF eval-null wte sconf*] **show** *throw* $e' = e_2 \wedge s_1 = s_2$ **by** *simp*
next
fix *Xs D'' S' a' h' l'*
assume *eval-ref*:$P,E \vdash \langle e,s_0\rangle \Rightarrow \langle ref(a',Xs),(h',l')\rangle$
from *IH*[*OF eval-ref wte sconf*] **show** *throw* $e' = e_2 \wedge s_1 = s_2$ **by** *simp*
next
fix e'' **assume** *eval-throw*:$P,E \vdash \langle e,s_0\rangle \Rightarrow \langle throw\ e'',s_2\rangle$
and *throw*:$e_2 = throw\ e''$
from *IH*[*OF eval-throw wte sconf*] *throw* **show** *throw* $e' = e_2 \wedge s_1 = s_2$ **by** *simp*
qed
next
case *Val* **thus** *?case* **by**(*auto elim*: *eval-cases*)
next
case (*BinOp E* e_1 s_0 v_1 s_1 e_2 v_2 s_2 *bop v* e_2' s_2' *T*)
have *eval*:$P,E \vdash \langle e_1 \ll bop \gg e_2,s_0\rangle \Rightarrow \langle e_2',s_2'\rangle$
and *binop*:*binop* (*bop*, v_1, v_2) = *Some v*
and *wt*:$P,E \vdash e_1 \ll bop \gg e_2 :: T$ **and** *sconf*:$P,E \vdash s_0\ \surd$
and *IH1*:$\bigwedge ei\ si\ T.\ [\![P,E \vdash \langle e_1,s_0\rangle \Rightarrow \langle ei,si\rangle;\ P,E \vdash e_1 :: T;\ P,E \vdash s_0\ \surd]\!]$
$\Longrightarrow$ *Val* $v_1 = ei \wedge s_1 = si$
and *IH2*:$\bigwedge ei\ si\ T.\ [\![P,E \vdash \langle e_2,s_1\rangle \Rightarrow \langle ei,si\rangle;\ P,E \vdash e_2 :: T;\ P,E \vdash s_1\ \surd]\!]$
$\Longrightarrow$ *Val* $v_2 = ei \wedge s_2 = si$ **by** *fact+*
from *wt* **obtain** T_1 T_2 **where** *wte1*:$P,E \vdash e_1 :: T_1$ **and** *wte2*:$P,E \vdash e_2 :: T_2$
by *auto*
from *eval* **show** *?case*
proof(*rule eval-cases*)
fix *s w* w_1 w_2
assume *eval-val1*:$P,E \vdash \langle e_1,s_0\rangle \Rightarrow \langle Val\ w_1,s\rangle$
and *eval-val2*:$P,E \vdash \langle e_2,s\rangle \Rightarrow \langle Val\ w_2,s_2'\rangle$
and *binop'*:*binop*(*bop*,w_1,w_2) = *Some w* **and** *e2'*:$e_2' = Val\ w$
from *IH1*[*OF eval-val1 wte1 sconf*] **have** *w1*:$v_1 = w_1$ **and** *s*:$s = s_1$ **by** *simp-all*
with *wf eval-val1 wte1 sconf* **have** $P,E \vdash s_1\ \surd$
by(*fastforce intro*:*eval-preserves-sconf*)
from *IH2*[*OF eval-val2*[*simplified s*] *wte2 this*] **have** $v_2 = w_2$ **and** *s2*:$s_2 = s_2'$
by *simp-all*
with *w1 binop binop'* **have** $w = v$ **by** *simp*
with *e2' s2* **show** *Val* $v = e_2' \wedge s_2 = s_2'$ **by** *simp*
next
fix *e* **assume** *eval-throw*:$P,E \vdash \langle e_1,s_0\rangle \Rightarrow \langle throw\ e,s_2'\rangle$
from *IH1*[*OF eval-throw wte1 sconf*] **show** *Val* $v = e_2' \wedge s_2 = s_2'$ **by** *simp*
next

fix $e\ s\ w$
assume *eval-val*:$P,E \vdash \langle e_1,s_0\rangle \Rightarrow \langle Val\ w,s\rangle$
and *eval-throw*:$P,E \vdash \langle e_2,s\rangle \Rightarrow \langle throw\ e,s_2{}'\rangle$
from *IH1*[*OF eval-val wte1 sconf*] **have** $s{:}s = s_1$ **by** *simp-all*
with *wf eval-val wte1 sconf* **have** $P,E \vdash s_1\ \surd$
by(*fastforce intro*:*eval-preserves-sconf*)
from *IH2*[*OF eval-throw*[*simplified s*] *wte2 this*] **show** $Val\ v = e_2{}' \wedge s_2 = s_2{}'$
by *simp*
qed
next
case (*BinOpThrow1 E* $e_1\ s_0\ e\ s_1$ *bop* $e_2\ e_2{}'\ s_2\ T$)
have *eval*:$P,E \vdash \langle e_1 \ll bop \gg e_2,s_0\rangle \Rightarrow \langle e_2{}',s_2\rangle$
and *wt*:$P,E \vdash e_1 \ll bop \gg e_2 :: T$ **and** *sconf*:$P,E \vdash s_0\ \surd$
and *IH*:$\bigwedge ei\ si\ T.\ [\![P,E \vdash \langle e_1,s_0\rangle \Rightarrow \langle ei,si\rangle;\ P,E \vdash e_1 :: T;\ P,E \vdash s_0\ \surd]\!]$
$\Longrightarrow throw\ e = ei \wedge s_1 = si$ **by** *fact+*
from *wt* **obtain** $T_1\ T_2$ **where** *wte1*:$P,E \vdash e_1 :: T_1$ **by** *auto*
from *eval* **show** *?case*
proof(*rule eval-cases*)
fix $s\ w\ w_1\ w_2$
assume *eval-val*:$P,E \vdash \langle e_1,s_0\rangle \Rightarrow \langle Val\ w_1,s\rangle$
from *IH*[*OF eval-val wte1 sconf*] **show** $throw\ e = e_2{}' \wedge s_1 = s_2$ **by** *simp*
next
fix e'
assume *eval-throw*:$P,E \vdash \langle e_1,s_0\rangle \Rightarrow \langle throw\ e',s_2\rangle$ **and** *throw*:$e_2{}' = throw\ e'$
from *IH*[*OF eval-throw wte1 sconf*] *throw* **show** $throw\ e = e_2{}' \wedge s_1 = s_2$ **by** *simp*
next
fix $e\ s\ w$
assume *eval-val*:$P,E \vdash \langle e_1,s_0\rangle \Rightarrow \langle Val\ w,s\rangle$
from *IH*[*OF eval-val wte1 sconf*] **show** $throw\ e = e_2{}' \wedge s_1 = s_2$ **by** *simp*
qed
next
case (*BinOpThrow2 E* $e_1\ s_0\ v_1\ s_1\ e_2\ e\ s_2$ *bop* $e_2{}'\ s_2{}'\ T$)
have *eval*:$P,E \vdash \langle e_1 \ll bop \gg e_2,s_0\rangle \Rightarrow \langle e_2{}',s_2{}'\rangle$
and *wt*:$P,E \vdash e_1 \ll bop \gg e_2 :: T$ **and** *sconf*:$P,E \vdash s_0\ \surd$
and *IH1*:$\bigwedge ei\ si\ T.\ [\![P,E \vdash \langle e_1,s_0\rangle \Rightarrow \langle ei,si\rangle;\ P,E \vdash e_1 :: T;\ P,E \vdash s_0\ \surd]\!]$
$\Longrightarrow Val\ v_1 = ei \wedge s_1 = si$
and *IH2*:$\bigwedge ei\ si\ T.\ [\![P,E \vdash \langle e_2,s_1\rangle \Rightarrow \langle ei,si\rangle;\ P,E \vdash e_2 :: T;\ P,E \vdash s_1\ \surd]\!]$
$\Longrightarrow throw\ e = ei \wedge s_2 = si$ **by** *fact+*
from *wt* **obtain** $T_1\ T_2$ **where** *wte1*:$P,E \vdash e_1 :: T_1$ **and** *wte2*:$P,E \vdash e_2 :: T_2$
by *auto*
from *eval* **show** *?case*
proof(*rule eval-cases*)
fix $s\ w\ w_1\ w_2$
assume *eval-val1*:$P,E \vdash \langle e_1,s_0\rangle \Rightarrow \langle Val\ w_1,s\rangle$
and *eval-val2*:$P,E \vdash \langle e_2,s\rangle \Rightarrow \langle Val\ w_2,s_2{}'\rangle$
from *IH1*[*OF eval-val1 wte1 sconf*] **have** $s{:}s = s_1$ **by** *simp-all*
with *wf eval-val1 wte1 sconf* **have** $P,E \vdash s_1\ \surd$
by(*fastforce intro*:*eval-preserves-sconf*)

from *IH2*[*OF eval-val2*[*simplified s*] *wte2 this*] **show** *throw e* = $e_2{}'$ ∧ s_2 = $s_2{}'$
by *simp*
next
fix e'
assume *eval-throw*:$P,E \vdash \langle e_1,s_0\rangle \Rightarrow \langle$*throw* e',$s_2{}'\rangle$
from *IH1*[*OF eval-throw wte1 sconf*] **show** *throw e* = $e_2{}'$ ∧ s_2 = $s_2{}'$ **by** *simp*
next
fix e' *s w*
assume *eval-val*:$P,E \vdash \langle e_1,s_0\rangle \Rightarrow \langle$*Val w*,*s*$\rangle$
and *eval-throw*:$P,E \vdash \langle e_2,s\rangle \Rightarrow \langle$*throw* e',$s_2{}'\rangle$
and *throw*:$e_2{}'$ = *throw* e'
from *IH1*[*OF eval-val wte1 sconf*] **have** *s*:*s* = s_1 **by** *simp-all*
with *wf eval-val wte1 sconf* **have** $P,E \vdash s_1\ \surd$
by(*fastforce intro*:*eval-preserves-sconf*)
from *IH2*[*OF eval-throw*[*simplified s*] *wte2 this*] *throw*
show *throw e* = $e_2{}'$ ∧ s_2 = $s_2{}'$
by *simp*
qed
next
case *Var* **thus** *?case* **by**(*auto elim*: *eval-cases*)
next
case (*LAss E e* s_0 *v h l V T v' l'* e_2 s_2 *T'*)
have *eval*:$P,E \vdash \langle V{:=}e,s_0\rangle \Rightarrow \langle e_2,s_2\rangle$
and *env*:*E V* = *Some T* **and** *casts*:$P \vdash T$ *casts v to v'* **and** *l'*:$l' = l(V \mapsto v')$
and *wt*:$P,E \vdash V{:=}e :: T'$ **and** *sconf*:$P,E \vdash s_0\ \surd$
and *IH*:$\bigwedge e_2\ s_2\ T.\ [\![P,E \vdash \langle e,s_0\rangle \Rightarrow \langle e_2,s_2\rangle;\ P,E \vdash e :: T;\ P,E \vdash s_0\ \surd]\!]$
$\Longrightarrow$ *Val v* = e_2 ∧ $(h,l) = s_2$ **by** *fact+*
from *wt env* **obtain** T'' **where** *wte*:$P,E \vdash e :: T''$ **and** *leq*:$P \vdash T'' \leq T$ **by** *auto*
from *eval* **show** *?case*
proof(*rule eval-cases*)
fix *U h' l''* w w'
assume *eval-val*:$P,E \vdash \langle e,s_0\rangle \Rightarrow \langle$*Val w*,$(h',l'')\rangle$ **and** *env'*:*E V* = *Some U*
and *casts'*:$P \vdash U$ *casts w to w'* **and** *e2*:e_2 = *Val w'*
and *s2*:$s_2 = (h',l''(V \mapsto w'))$
from *env env'* **have** *UeqT*:*U* = *T* **by** *simp*
from *IH*[*OF eval-val wte sconf*] **have** *eq*:$v = w \wedge h = h' \wedge l = l''$ **by** *simp*
from *sconf env* **have** *is-type P T*
by(*clarsimp simp*:*sconf-def envconf-def*)
with *casts casts' eq UeqT wte leq eval-val sconf wf* **have** $v' = w'$
by(*auto intro*:*casts-casts-eq-result*)
with *e2 s2 l' eq* **show** *Val v'* = e_2 ∧ $(h,\ l') = s_2$ **by** *simp*
next
fix e' **assume** *eval-throw*:$P,E \vdash \langle e,s_0\rangle \Rightarrow \langle$*throw* e',$s_2\rangle$
from *IH*[*OF eval-throw wte sconf*] **show** *Val v'* = e_2 ∧ $(h,\ l') = s_2$ **by** *simp*
qed
next
case (*LAssThrow E e* s_0 e' s_1 *V* e_2 s_2 *T*)
have *eval*:$P,E \vdash \langle V{:=}e,s_0\rangle \Rightarrow \langle e_2,s_2\rangle$

and *wt*:$P,E \vdash V{:=}e :: T$ **and** *sconf*:$P,E \vdash s_0\ \surd$
and *IH*:$\bigwedge e_2\ s_2\ T.\ [\![P,E \vdash \langle e,s_0\rangle \Rightarrow \langle e_2,s_2\rangle;\ P,E \vdash e :: T;\ P,E \vdash s_0\ \surd]\!]$
$\Longrightarrow$ *throw* $e' = e_2 \wedge s_1 = s_2$ **by** *fact+*
from *wt* **obtain** T'' **where** *wte*:$P,E \vdash e :: T''$ **by** *auto*
from *eval* **show** *?case*
proof(*rule eval-cases*)
fix $U\ h'\ l''\ w\ w'$
assume *eval-val*:$P,E \vdash \langle e,s_0\rangle \Rightarrow \langle Val\ w,(h',l'')\rangle$
from *IH*[*OF eval-val wte sconf*] **show** *throw* $e' = e_2 \wedge s_1 = s_2$ **by** *simp*
next
fix *ex*
assume *eval-throw*:$P,E \vdash \langle e,s_0\rangle \Rightarrow \langle throw\ ex,s_2\rangle$ **and** *e2*:$e_2 = throw\ ex$
from *IH*[*OF eval-throw wte sconf*] *e2* **show** *throw* $e' = e_2 \wedge s_1 = s_2$ **by** *simp*
qed
next
case (*FAcc E e* s_0 *a Cs' h l D S Ds Cs fs F v* e_2 s_2 *T*)
have *eval*:$P,E \vdash \langle e{\cdot}F\{Cs\},s_0\rangle \Rightarrow \langle e_2,s_2\rangle$
and *eval'*:$P,E \vdash \langle e,s_0\rangle \Rightarrow \langle ref\ (a,\ Cs'),(h,l)\rangle$
and *h*:$h\ a = Some(D,S)$ **and** *Ds*:$Ds = Cs'@_pCs$
and *S*:$(Ds,fs) \in S$ **and** *fs*:$fs\ F = Some\ v$
and *wt*:$P,E \vdash e{\cdot}F\{Cs\} :: T$ **and** *sconf*:$P,E \vdash s_0\ \surd$
and *IH*:$\bigwedge e_2\ s_2\ T.\ [\![P,E \vdash \langle e,s_0\rangle \Rightarrow \langle e_2,s_2\rangle;\ P,E \vdash e :: T;\ P,E \vdash s_0\ \surd]\!]$
$\Longrightarrow$ *ref* $(a,\ Cs') = e_2 \wedge (h,l) = s_2$ **by** *fact+*
from *wt* **obtain** *C* **where** *wte*:$P,E \vdash e :: Class\ C$ **by** *auto*
from *eval-preserves-sconf*[*OF wf eval' wte sconf*] *h* **have** *oconf*:$P,h \vdash (D,S)\ \surd$
by(*simp add*:*sconf-def hconf-def*)
from *eval* **show** *?case*
proof(*rule eval-cases*)
fix $Xs'\ D'\ S'\ a'\ fs'\ h'\ l'\ v'$
assume *eval-ref*:$P,E \vdash \langle e,s_0\rangle \Rightarrow \langle ref(a',Xs'),(h',l')\rangle$
and *h'*:$h'\ a' = Some(D',S')$ **and** *S'*:$(Xs'@_pCs,fs') \in S'$
and *fs'*:$fs'\ F = Some\ v'$ **and** *e2*:$e_2 = Val\ v'$ **and** *s2*:$s_2 = (h',l')$
from *IH*[*OF eval-ref wte sconf*] *h h'*
have *eq*:$a = a' \wedge Cs' = Xs' \wedge h = h' \wedge l = l' \wedge D = D' \wedge S = S'$ **by** *simp*
with *oconf S S' Ds* **have** *fs* = *fs'* **by** (*auto simp*:*oconf-def*)
with *fs fs'* **have** $v = v'$ **by** *simp*
with *e2 s2 eq* **show** $Val\ v = e_2 \wedge (h,l) = s_2$ **by** *simp*
next
assume *eval-null*:$P,E \vdash \langle e,s_0\rangle \Rightarrow \langle null,s_2\rangle$
from *IH*[*OF eval-null wte sconf*] **show** $Val\ v = e_2 \wedge (h,l) = s_2$ **by** *simp*
next
fix e' **assume** *eval-throw*:$P,E \vdash \langle e,s_0\rangle \Rightarrow \langle throw\ e',s_2\rangle$
from *IH*[*OF eval-throw wte sconf*] **show** $Val\ v = e_2 \wedge (h,l) = s_2$ **by** *simp*
qed
next
case (*FAccNull E e* s_0 s_1 *F Cs* e_2 s_2 *T*)
have *eval*:$P,E \vdash \langle e{\cdot}F\{Cs\},s_0\rangle \Rightarrow \langle e_2,s_2\rangle$
and *wt*:$P,E \vdash e{\cdot}F\{Cs\} :: T$ **and** *sconf*:$P,E \vdash s_0\ \surd$
and *IH*:$\bigwedge e_2\ s_2\ T.\ [\![P,E \vdash \langle e,s_0\rangle \Rightarrow \langle e_2,s_2\rangle;\ P,E \vdash e :: T;\ P,E \vdash s_0\ \surd]\!]$

$\Longrightarrow$ *null* $= e_2 \wedge s_1 = s_2$ **by** *fact+*
from *wt* **obtain** *C* **where** *wte*:$P,E \vdash e :: Class\ C$ **by** *auto*
from *eval* **show** *?case*
proof(*rule eval-cases*)
fix $Xs'\ D'\ S'\ a'\ fs'\ h'\ l'\ v'$
assume *eval-ref*:$P,E \vdash \langle e,s_0\rangle \Rightarrow \langle ref(a',Xs'),(h',l')\rangle$
from *IH*[*OF eval-ref wte sconf*] **show** *THROW NullPointer* $= e_2 \wedge s_1 = s_2$ **by** *simp*
next
assume *eval-null*:$P,E \vdash \langle e,s_0\rangle \Rightarrow \langle null,s_2\rangle$ **and** *e2*:$e_2 = THROW\ NullPointer$
from *IH*[*OF eval-null wte sconf*] *e2* **show** *THROW NullPointer* $= e_2 \wedge s_1 = s_2$
by *simp*
next
fix e' **assume** *eval-throw*:$P,E \vdash \langle e,s_0\rangle \Rightarrow \langle throw\ e',s_2\rangle$
from *IH*[*OF eval-throw wte sconf*] **show** *THROW NullPointer* $= e_2 \wedge s_1 = s_2$ **by** *simp*
qed
next
case (*FAccThrow E e* s_0 e' s_1 *F Cs* e_2 s_2 *T*)
have *eval*:$P,E \vdash \langle e{\cdot}F\{Cs\},s_0\rangle \Rightarrow \langle e_2,s_2\rangle$
and *wt*:$P,E \vdash e{\cdot}F\{Cs\} :: T$ **and** *sconf*:$P,E \vdash s_0\ \surd$
and *IH*:$\bigwedge e_2\ s_2\ T.\ [\![P,E \vdash \langle e,s_0\rangle \Rightarrow \langle e_2,s_2\rangle;\ P,E \vdash e :: T;\ P,E \vdash s_0\ \surd]\!]$
$\Longrightarrow$ *throw* $e' = e_2 \wedge s_1 = s_2$ **by** *fact+*
from *wt* **obtain** *C* **where** *wte*:$P,E \vdash e :: Class\ C$ **by** *auto*
from *eval* **show** *?case*
proof(*rule eval-cases*)
fix $Xs'\ D'\ S'\ a'\ fs'\ h'\ l'\ v'$
assume *eval-ref*:$P,E \vdash \langle e,s_0\rangle \Rightarrow \langle ref(a',Xs'),(h',l')\rangle$
from *IH*[*OF eval-ref wte sconf*] **show** *throw* $e' = e_2 \wedge s_1 = s_2$ **by** *simp*
next
assume *eval-null*:$P,E \vdash \langle e,s_0\rangle \Rightarrow \langle null,s_2\rangle$
from *IH*[*OF eval-null wte sconf*] **show** *throw* $e' = e_2 \wedge s_1 = s_2$ **by** *simp*
next
fix *ex*
assume *eval-throw*:$P,E \vdash \langle e,s_0\rangle \Rightarrow \langle throw\ ex,s_2\rangle$ **and** *e2*:$e_2 = throw\ ex$
from *IH*[*OF eval-throw wte sconf*] *e2* **show** *throw* $e' = e_2 \wedge s_1 = s_2$ **by** *simp*
qed
next
case (*FAss E* e_1 s_0 *a* Cs' s_1 e_2 *v* h_2 l_2 *D S F T Cs* v' *Ds fs* fs' S' h_2' e_2' s_2 T')
have *eval*:$P,E \vdash \langle e_1{\cdot}F\{Cs\} := e_2,s_0\rangle \Rightarrow \langle e_2',s_2\rangle$
and *eval'*:$P,E \vdash \langle e_1,s_0\rangle \Rightarrow \langle ref(a,Cs'),s_1\rangle$
and *eval''*:$P,E \vdash \langle e_2,s_1\rangle \Rightarrow \langle Val\ v,(h_2,l_2)\rangle$
and *h2*:$h_2\ a = Some(D,\ S)$
and *has-least*:$P \vdash last\ Cs'\ has\ least\ F{:}T\ via\ Cs$
and *casts*:$P \vdash T\ casts\ v\ to\ v'$ **and** *Ds*:$Ds = Cs'@_pCs$
and $S{:}(Ds,\ fs) \in S$ **and** $fs'{:}fs' = fs(F \mapsto v')$
and $S'{:}S' = S - \{(Ds,\ fs)\} \cup \{(Ds,\ fs')\}$
and *h2'*:$h_2' = h_2(a \mapsto (D,\ S'))$

and *wt*:$P,E \vdash e_1{\cdot}F\{Cs\} := e_2 :: T'$ **and** *sconf*:$P,E \vdash s_0\ \surd$
and *IH1*:$\bigwedge ei\ si\ T.\ [\![P,E \vdash \langle e_1,s_0\rangle \Rightarrow \langle ei,si\rangle;\ P,E \vdash e_1 :: T;\ P,E \vdash s_0\ \surd]\!]$
$\Longrightarrow ref(a,Cs') = ei \wedge s_1 = si$
and *IH2*:$\bigwedge ei\ si\ T.\ [\![P,E \vdash \langle e_2,s_1\rangle \Rightarrow \langle ei,si\rangle;\ P,E \vdash e_2 :: T;\ P,E \vdash s_1\ \surd]\!]$
$\Longrightarrow Val\ v = ei \wedge (h_2,l_2) = si$ **by** *fact+*
from *wt* **obtain** $C\ T''$ **where** *wte1*:$P,E \vdash e_1 :: Class\ C$
and *has-least'*:$P \vdash C$ *has least* $F{:}T'$ *via* Cs
and *wte2*:$P,E \vdash e_2 :: T''$ **and** *leq*:$P \vdash T'' \leq T'$
by *auto*
from *wf eval' wte1 sconf* **have** *last* $Cs' = C$
by(*auto dest*!:*eval-preserves-type split*:*if-split-asm*)
with *has-least has-least'* **have** *TeqT'*:$T = T'$ **by** (*fastforce intro*:*sees-field-fun*)
from *eval* **show** *?case*
proof(*rule eval-cases*)
fix *Xs D' S'' U a' fs'' h l s w w'*
assume *eval-ref*:$P,E \vdash \langle e_1,s_0\rangle \Rightarrow \langle ref(a',Xs),s\rangle$
and *eval-val*:$P,E \vdash \langle e_2,s\rangle \Rightarrow \langle Val\ w,(h,l)\rangle$
and *h*:$h\ a' = Some(D',S'')$
and *has-least''*:$P \vdash last\ Xs$ *has least* $F{:}U$ *via* Cs
and *casts'*:$P \vdash U$ *casts* w *to* w'
and S'':$(Xs@_pCs,fs'') \in S''$ **and** *e2'*:$e_2' = Val\ w'$
and *s2*:$s_2 = (h(a' \mapsto (D',insert\ (Xs@_pCs,fs''(F \mapsto w'))$
$(S''-\{(Xs@_pCs,fs'')\})),l)$
from *IH1*[*OF eval-ref wte1 sconf*] **have** *eq*:$a = a' \wedge Cs' = Xs \wedge s_1 = s$ **by** *simp*
with *wf eval-ref wte1 sconf* **have** *sconf'*:$P,E \vdash s_1\ \surd$
by(*fastforce intro*:*eval-preserves-sconf*)
from *IH2*[*OF - wte2 this*] *eval-val eq* **have** *eq'*:$v = w \wedge h = h_2 \wedge l = l_2$ **by** *auto*
from *has-least'' eq has-least* **have** *UeqT*:$U = T$ **by** (*fastforce intro*:*sees-field-fun*)
from *has-least wf* **have** *is-type P T* **by**(*rule least-field-is-type*)
with *casts casts' eq eq' UeqT TeqT' wte2 leq eval-val sconf' wf* **have** *v'*:$v' = w'$
by(*auto intro*!:*casts-casts-eq-result*)
from *eval-preserves-sconf*[*OF wf eval'' wte2 sconf'*] *h2*
have *oconf*:$P,h_2 \vdash (D,S)\ \surd$
by(*simp add*:*sconf-def hconf-def*)
from *eq eq' h2 h* **have** $S = S''$ **by** *simp*
with *oconf eq S S' S'' Ds* **have** *fs* $= fs''$ **by** (*auto simp*:*oconf-def*)
with *h2' h h2 eq eq' s2 S' Ds fs' v' e2'* **show** $Val\ v' = e_2' \wedge (h_2',l_2) = s_2$
by *simp*
next
fix *s w* **assume** *eval-null*:$P,E \vdash \langle e_1,s_0\rangle \Rightarrow \langle null,s\rangle$
from *IH1*[*OF eval-null wte1 sconf*] **show** $Val\ v' = e_2' \wedge (h_2',l_2) = s_2$ **by** *simp*
next
fix *ex* **assume** *eval-throw*:$P,E \vdash \langle e_1,s_0\rangle \Rightarrow \langle throw\ ex,s_2\rangle$
from *IH1*[*OF eval-throw wte1 sconf*] **show** $Val\ v' = e_2' \wedge (h_2',l_2) = s_2$ **by** *simp*
next

fix *ex s w*
assume *eval-val*:$P,E \vdash \langle e_1,s_0\rangle \Rightarrow \langle Val\ w,s\rangle$
and *eval-throw*:$P,E \vdash \langle e_2,s\rangle \Rightarrow \langle throw\ ex,s_2\rangle$
from *IH1*[*OF eval-val wte1 sconf*] **have** *eq*:$s = s_1$ **by** *simp*
with *wf eval-val wte1 sconf* **have** *sconf'*:$P,E \vdash s_1\ \surd$
by(*fastforce intro*:*eval-preserves-sconf*)
from *IH2*[*OF eval-throw*[*simplified eq*] *wte2 this*]
show $Val\ v' = e_2' \wedge (h_2',l_2) = s_2$ **by** *simp*
qed
next
case (*FAssNull E* $e_1\ s_0\ s_1\ e_2\ v\ s_2\ F\ Cs\ e_2'\ s_2'\ T$)
have *eval*:$P,E \vdash \langle e_1{\cdot}F\{Cs\} := e_2,s_0\rangle \Rightarrow \langle e_2',s_2'\rangle$
and *wt*:$P,E \vdash e_1{\cdot}F\{Cs\} := e_2 :: T$ **and** *sconf*:$P,E \vdash s_0\ \surd$
and *IH1*:$\bigwedge ei\ si\ T.\ [\![P,E \vdash \langle e_1,s_0\rangle \Rightarrow \langle ei,si\rangle;\ P,E \vdash e_1 :: T;\ P,E \vdash s_0\ \surd]\!]$
$\Longrightarrow null = ei \wedge s_1 = si$
and *IH2*:$\bigwedge ei\ si\ T.\ [\![P,E \vdash \langle e_2,s_1\rangle \Rightarrow \langle ei,si\rangle;\ P,E \vdash e_2 :: T;\ P,E \vdash s_1\ \surd]\!]$
$\Longrightarrow Val\ v = ei \wedge s_2 = si$ **by** *fact+*
from *wt* **obtain** $C\ T''$ **where** *wte1*:$P,E \vdash e_1 :: Class\ C$
and *wte2*:$P,E \vdash e_2 :: T''$ **by** *auto*
from *eval* **show** *?case*
proof(*rule eval-cases*)
fix $Xs\ D'\ S''\ U\ a'\ fs''\ h\ l\ s\ w\ w'$
assume *eval-ref*:$P,E \vdash \langle e_1,s_0\rangle \Rightarrow \langle ref(a',Xs),s\rangle$
from *IH1*[*OF eval-ref wte1 sconf*] **show** $THROW\ NullPointer = e_2' \wedge s_2 = s_2'$
by *simp*
next
fix *s w*
assume *eval-null*:$P,E \vdash \langle e_1,s_0\rangle \Rightarrow \langle null,s\rangle$
and *eval-val*:$P,E \vdash \langle e_2,s\rangle \Rightarrow \langle Val\ w,s_2'\rangle$
and *e2'*:$e_2' = THROW\ NullPointer$
from *IH1*[*OF eval-null wte1 sconf*] **have** *eq*:$s = s_1$ **by** *simp*
with *wf eval-null wte1 sconf* **have** *sconf'*:$P,E \vdash s_1\ \surd$
by(*fastforce intro*:*eval-preserves-sconf*)
from *IH2*[*OF eval-val*[*simplified eq*] *wte2 this*] *e2'*
show $THROW\ NullPointer = e_2' \wedge s_2 = s_2'$ **by** *simp*
next
fix *ex* **assume** *eval-throw*:$P,E \vdash \langle e_1,s_0\rangle \Rightarrow \langle throw\ ex,s_2'\rangle$
from *IH1*[*OF eval-throw wte1 sconf*] **show** $THROW\ NullPointer = e_2' \wedge s_2 = s_2'$
by *simp*
next
fix *ex s w*
assume *eval-val*:$P,E \vdash \langle e_1,s_0\rangle \Rightarrow \langle Val\ w,s\rangle$
and *eval-throw*:$P,E \vdash \langle e_2,s\rangle \Rightarrow \langle throw\ ex,s_2'\rangle$
from *IH1*[*OF eval-val wte1 sconf*] **have** *eq*:$s = s_1$ **by** *simp*
with *wf eval-val wte1 sconf* **have** *sconf'*:$P,E \vdash s_1\ \surd$
by(*fastforce intro*:*eval-preserves-sconf*)
from *IH2*[*OF eval-throw*[*simplified eq*] *wte2 this*]

```
    show THROW NullPointer = e2' ∧ s2 = s2' by simp
  qed
next
  case (FAssThrow1 E e1 s0 e' s1 F Cs e2 e2' s2 T)
  have eval:P,E ⊢ ⟨e1·F{Cs} := e2,s0⟩ ⇒ ⟨e2',s2⟩
    and wt:P,E ⊢ e1·F{Cs} := e2 :: T and sconf:P,E ⊢ s0 √
    and IH:⋀ei si T. ⟦P,E ⊢ ⟨e1,s0⟩ ⇒ ⟨ei,si⟩; P,E ⊢ e1 :: T; P,E ⊢ s0 √⟧
                  ⟹ throw e' = ei ∧ s1 = si by fact+
  from wt obtain C T'' where wte1:P,E ⊢ e1 :: Class C by auto
  from eval show ?case
  proof(rule eval-cases)
    fix Xs D' S'' U a' fs'' h l s w w'
    assume eval-ref:P,E ⊢ ⟨e1,s0⟩ ⇒ ⟨ref(a',Xs),s⟩
    from IH[OF eval-ref wte1 sconf] show throw e' = e2' ∧ s1 = s2 by simp
  next
    fix s w
    assume eval-null:P,E ⊢ ⟨e1,s0⟩ ⇒ ⟨null,s⟩
    from IH[OF eval-null wte1 sconf] show throw e' = e2' ∧ s1 = s2 by simp
  next
    fix ex
    assume eval-throw:P,E ⊢ ⟨e1,s0⟩ ⇒ ⟨throw ex,s2⟩ and e2':e2' = throw ex
     from IH[OF eval-throw wte1 sconf] e2' show throw e' = e2' ∧ s1 = s2 by
simp
  next
    fix ex s w assume eval-val:P,E ⊢ ⟨e1,s0⟩ ⇒ ⟨Val w,s⟩
    from IH[OF eval-val wte1 sconf] show throw e' = e2' ∧ s1 = s2 by simp
  qed
next
  case (FAssThrow2 E e1 s0 v s1 e2 e' s2 F Cs e2' s2' T)
  have eval:P,E ⊢ ⟨e1·F{Cs} := e2,s0⟩ ⇒ ⟨e2',s2'⟩
    and wt:P,E ⊢ e1·F{Cs} := e2 :: T and sconf:P,E ⊢ s0 √
    and IH1:⋀ei si T. ⟦P,E ⊢ ⟨e1,s0⟩ ⇒ ⟨ei,si⟩; P,E ⊢ e1 :: T; P,E ⊢ s0 √⟧
                  ⟹ Val v = ei ∧ s1 = si
    and IH2:⋀ei si T. ⟦P,E ⊢ ⟨e2,s1⟩ ⇒ ⟨ei,si⟩; P,E ⊢ e2 :: T; P,E ⊢ s1 √⟧
                  ⟹ throw e' = ei ∧ s2 = si by fact+
  from wt obtain C T'' where wte1:P,E ⊢ e1 :: Class C
    and wte2:P,E ⊢ e2 :: T'' by auto
  from eval show ?case
  proof(rule eval-cases)
    fix Xs D' S'' U a' fs'' h l s w w'
    assume eval-ref:P,E ⊢ ⟨e1,s0⟩ ⇒ ⟨ref(a',Xs),s⟩
      and eval-val:P,E ⊢ ⟨e2,s⟩ ⇒ ⟨Val w,(h,l)⟩
    from IH1[OF eval-ref wte1 sconf] have eq:s = s1 by simp
    with wf eval-ref wte1 sconf have sconf':P,E ⊢ s1 √
      by(fastforce intro:eval-preserves-sconf)
    from IH2[OF eval-val[simplified eq] wte2 this] show throw e' = e2' ∧ s2 = s2'
      by simp
  next
    fix s w
```

assume *eval-null*:$P,E \vdash \langle e_1,s_0\rangle \Rightarrow \langle null,s\rangle$
and *eval-val*:$P,E \vdash \langle e_2,s\rangle \Rightarrow \langle Val\ w,s_2{}'\rangle$
from *IH1*[*OF eval-null wte1 sconf*] **have** *eq*:$s = s_1$ **by** *simp*
with *wf eval-null wte1 sconf* **have** *sconf'*:$P,E \vdash s_1\ \surd$
by(*fastforce intro*:*eval-preserves-sconf*)
from *IH2*[*OF eval-val*[*simplified eq*] *wte2 this*] **show** *throw* $e' = e_2{}' \wedge s_2 = s_2{}'$
by *simp*
next
fix *ex* **assume** *eval-throw*:$P,E \vdash \langle e_1,s_0\rangle \Rightarrow \langle throw\ ex,s_2{}'\rangle$
from *IH1*[*OF eval-throw wte1 sconf*] **show** *throw* $e' = e_2{}' \wedge s_2 = s_2{}'$ **by** *simp*
next
fix *ex s w*
assume *eval-val*:$P,E \vdash \langle e_1,s_0\rangle \Rightarrow \langle Val\ w,s\rangle$
and *eval-throw*:$P,E \vdash \langle e_2,s\rangle \Rightarrow \langle throw\ ex,s_2{}'\rangle$ **and** *e2'*:$e_2{}' = throw\ ex$
from *IH1*[*OF eval-val wte1 sconf*] **have** *eq*:$s = s_1$ **by** *simp*
with *wf eval-val wte1 sconf* **have** *sconf'*:$P,E \vdash s_1\ \surd$
by(*fastforce intro*:*eval-preserves-sconf*)
from *IH2*[*OF eval-throw*[*simplified eq*] *wte2 this*] *e2'*
show *throw* $e' = e_2{}' \wedge s_2 = s_2{}'$ **by** *simp*
qed
next
case (*CallObjThrow E e* s_0 e' s_1 *Copt M es* e_2 s_2 *T*)
have *eval*:$P,E \vdash \langle Call\ e\ Copt\ M\ es,s_0\rangle \Rightarrow \langle e_2,s_2\rangle$
and *wt*:$P,E \vdash Call\ e\ Copt\ M\ es :: T$ **and** *sconf*:$P,E \vdash s_0\ \surd$
and *IH*:$\bigwedge e_2\ s_2\ T.\ [\![P,E \vdash \langle e,s_0\rangle \Rightarrow \langle e_2,s_2\rangle;\ P,E \vdash e :: T;\ P,E \vdash s_0\ \surd]\!]$
$\Longrightarrow$ *throw* $e' = e_2 \wedge s_1 = s_2$ **by** *fact+*
from *wt* **obtain** *C* **where** *wte*:$P,E \vdash e :: Class\ C$ **by**(*cases Copt*)*auto*
show *?case*
proof(*cases Copt*)
assume $Copt = None$
with *eval* **have** $P,E \vdash \langle e{\cdot}M(es),s_0\rangle \Rightarrow \langle e_2,s_2\rangle$ **by** *simp*
thus *?thesis*
proof(*rule eval-cases*)
fix *ex*
assume *eval-throw*:$P,E \vdash \langle e,s_0\rangle \Rightarrow \langle throw\ ex,s_2\rangle$ **and** *e2*:$e_2 = throw\ ex$
from *IH*[*OF eval-throw wte sconf*] *e2* **show** *throw* $e' = e_2 \wedge s_1 = s_2$ **by** *simp*
next
fix es' ex' *s w ws* **assume** *eval-val*:$P,E \vdash \langle e,s_0\rangle \Rightarrow \langle Val\ w,s\rangle$
from *IH*[*OF eval-val wte sconf*] **show** *throw* $e' = e_2 \wedge s_1 = s_2$ **by** *simp*
next
fix C' *Xs* Xs' Ds' S' *U* U' *Us* Us' a' $body''$ $body'''$ *h* h' *l* l' pns'' pns'''
s ws ws'
assume *eval-ref*:$P,E \vdash \langle e,s_0\rangle \Rightarrow \langle ref(a',Xs),s\rangle$
from *IH*[*OF eval-ref wte sconf*] **show** *throw* $e' = e_2 \wedge s_1 = s_2$ **by** *simp*
next
fix *s ws*
assume *eval-null*:$P,E \vdash \langle e,s_0\rangle \Rightarrow \langle null,s\rangle$
from *IH*[*OF eval-null wte sconf*] **show** *throw* $e' = e_2 \wedge s_1 = s_2$ **by** *simp*
qed

```
  next
    fix C' assume Copt = Some C'
    with eval have P,E ⊢ ⟨e•(C'::)M(es),s0⟩ ⇒ ⟨e2,s2⟩ by simp
    thus ?thesis
    proof(rule eval-cases)
      fix ex
      assume eval-throw:P,E ⊢ ⟨e,s0⟩ ⇒ ⟨throw ex,s2⟩ and e2:e2 = throw ex
     from IH[OF eval-throw wte sconf] e2 show throw e' = e2 ∧ s1 = s2 by simp
    next
      fix es' ex' s w ws assume eval-val:P,E ⊢ ⟨e,s0⟩ ⇒ ⟨Val w,s⟩
      from IH[OF eval-val wte sconf] show throw e' = e2 ∧ s1 = s2 by simp
    next
      fix C'' Xs Xs' Ds' S' U U' Us Us' a' body'' body''' h h' l l' pns'' pns'''
          s ws ws'
      assume eval-ref:P,E ⊢ ⟨e,s0⟩ ⇒ ⟨ref(a',Xs),s⟩
      from IH[OF eval-ref wte sconf] show throw e' = e2 ∧ s1 = s2 by simp
    next
      fix s ws
      assume eval-null:P,E ⊢ ⟨e,s0⟩ ⇒ ⟨null,s⟩
      from IH[OF eval-null wte sconf] show throw e' = e2 ∧ s1 = s2 by simp
    qed
  qed
next
  case (CallParamsThrow E e s0 v s1 es vs ex es' s2 Copt M e2 s2' T)
  have eval:P,E ⊢ ⟨Call e Copt M es,s0⟩ ⇒ ⟨e2,s2'⟩
    and wt:P,E ⊢ Call e Copt M es :: T and sconf:P,E ⊢ s0 √
    and IH1:⋀ei si T. ⟦P,E ⊢ ⟨e,s0⟩ ⇒ ⟨ei,si⟩; P,E ⊢ e :: T; P,E ⊢ s0 √⟧
                  ⟹ Val v = ei ∧ s1 = si
    and IH2:⋀esi si Ts. ⟦P,E ⊢ ⟨es,s1⟩ [⇒] ⟨esi,si⟩; P,E ⊢ es [::] Ts; P,E ⊢ s1
√⟧
                  ⟹ map Val vs @ throw ex # es' = esi ∧ s2 = si by fact+
  from wt obtain C Ts where wte:P,E ⊢ e :: Class C and wtes:P,E ⊢ es [::] Ts

    by(cases Copt)auto
  show ?case
  proof(cases Copt)
    assume Copt = None
    with eval have P,E ⊢ ⟨e•M(es),s0⟩ ⇒ ⟨e2,s2'⟩ by simp
    thus ?thesis
    proof(rule eval-cases)
      fix ex' assume eval-throw:P,E ⊢ ⟨e,s0⟩ ⇒ ⟨throw ex',s2'⟩
     from IH1[OF eval-throw wte sconf] show throw ex = e2 ∧ s2 = s2' by simp
    next
      fix es'' ex' s w ws
      assume eval-val:P,E ⊢ ⟨e,s0⟩ ⇒ ⟨Val w,s⟩
        and evals-throw:P,E ⊢ ⟨es,s⟩ [⇒] ⟨map Val ws@throw ex'#es'',s2'⟩
        and e2:e2 = throw ex'
      from IH1[OF eval-val wte sconf] have eq:s = s1 by simp
      with wf eval-val wte sconf have sconf':P,E ⊢ s1 √
```

by(*fastforce intro*:*eval-preserves-sconf*)
from *IH2*[*OF evals-throw*[*simplified eq*] *wtes this*] *e2*
have $vs = ws \wedge ex = ex' \wedge es' = es'' \wedge s_2 = s_2'$
by(*fastforce dest*:*map-Val-throw-eq*)
with *e2* **show** *throw ex* $= e_2 \wedge s_2 = s_2'$ **by** *simp*
next
fix *C' Xs Xs' Ds' S' U U' Us Us' a' body'' body''' h h' l l' pns'' pns'''*
s ws ws'
assume *eval-ref*:$P,E \vdash \langle e,s_0\rangle \Rightarrow \langle ref(a',Xs),s\rangle$
and *evals-vals*:$P,E \vdash \langle es,s\rangle$ [⇒] $\langle map\ Val\ ws,(h,l)\rangle$
from *IH1*[*OF eval-ref wte sconf*] **have** *eq*:$s = s_1$ **by** *simp*
with *wf eval-ref wte sconf* **have** *sconf'*:$P,E \vdash s_1\ \surd$
by(*fastforce intro*:*eval-preserves-sconf*)
from *IH2*[*OF evals-vals*[*simplified eq*] *wtes this*]
show *throw ex* $= e_2 \wedge s_2 = s_2'$
by(*fastforce dest*:*sym*[*THEN map-Val-throw-False*])
next
fix *s ws*
assume *eval-null*:$P,E \vdash \langle e,s_0\rangle \Rightarrow \langle null,s\rangle$
and *evals-vals*:$P,E \vdash \langle es,s\rangle$ [⇒] $\langle map\ Val\ ws,s_2'\rangle$
and *e2*:$e_2 = THROW\ NullPointer$
from *IH1*[*OF eval-null wte sconf*] **have** *eq*:$s = s_1$ **by** *simp*
with *wf eval-null wte sconf* **have** *sconf'*:$P,E \vdash s_1\ \surd$
by(*fastforce intro*:*eval-preserves-sconf*)
from *IH2*[*OF evals-vals*[*simplified eq*] *wtes this*]
show *throw ex* $= e_2 \wedge s_2 = s_2'$
by(*fastforce dest*:*sym*[*THEN map-Val-throw-False*])
qed
next
fix *C'* **assume** *Copt* $= Some\ C'$
with *eval* **have** $P,E \vdash \langle e{\cdot}(C'::)M(es),s_0\rangle \Rightarrow \langle e_2,s_2'\rangle$ **by** *simp*
thus *?thesis*
proof(*rule eval-cases*)
fix *ex'* **assume** *eval-throw*:$P,E \vdash \langle e,s_0\rangle \Rightarrow \langle throw\ ex',s_2'\rangle$
from *IH1*[*OF eval-throw wte sconf*] **show** *throw ex* $= e_2 \wedge s_2 = s_2'$ **by** *simp*
next
fix *es'' ex' s w ws*
assume *eval-val*:$P,E \vdash \langle e,s_0\rangle \Rightarrow \langle Val\ w,s\rangle$
and *evals-throw*:$P,E \vdash \langle es,s\rangle$ [⇒] $\langle map\ Val\ ws@throw\ ex'\#es'',s_2'\rangle$
and *e2*:$e_2 = throw\ ex'$
from *IH1*[*OF eval-val wte sconf*] **have** *eq*:$s = s_1$ **by** *simp*
with *wf eval-val wte sconf* **have** *sconf'*:$P,E \vdash s_1\ \surd$
by(*fastforce intro*:*eval-preserves-sconf*)
from *IH2*[*OF evals-throw*[*simplified eq*] *wtes this*] *e2*
have $vs = ws \wedge ex = ex' \wedge es' = es'' \wedge s_2 = s_2'$
by(*fastforce dest*:*map-Val-throw-eq*)
with *e2* **show** *throw ex* $= e_2 \wedge s_2 = s_2'$ **by** *simp*
next
fix *C' Xs Xs' Ds' S' U U' Us Us' a' body'' body''' h h' l l' pns'' pns'''*

```
      s ws ws'
    assume eval-ref:P,E ⊢ ⟨e,s0⟩ ⇒ ⟨ref(a',Xs),s⟩
      and evals-vals:P,E ⊢ ⟨es,s⟩ [⇒] ⟨map Val ws,(h,l)⟩
    from IH1[OF eval-ref wte sconf] have eq:s = s1 by simp
    with wf eval-ref wte sconf have sconf':P,E ⊢ s1 √
      by(fastforce intro:eval-preserves-sconf)
    from IH2[OF evals-vals[simplified eq] wtes this]
    show throw ex = e2 ∧ s2 = s2'
      by(fastforce dest:sym[THEN map-Val-throw-False])
  next
    fix s ws
    assume eval-null:P,E ⊢ ⟨e,s0⟩ ⇒ ⟨null,s⟩
      and evals-vals:P,E ⊢ ⟨es,s⟩ [⇒] ⟨map Val ws,s2'⟩
      and e2:e2 = THROW NullPointer
    from IH1[OF eval-null wte sconf] have eq:s = s1 by simp
    with wf eval-null wte sconf have sconf':P,E ⊢ s1 √
      by(fastforce intro:eval-preserves-sconf)
    from IH2[OF evals-vals[simplified eq] wtes this]
    show throw ex = e2 ∧ s2 = s2'
      by(fastforce dest:sym[THEN map-Val-throw-False])
  qed
 qed
next
 case (Call E e s0 a Cs s1 es vs h2 l2 C S M Ts' T' pns' body' Ds Ts T pns
        body Cs' vs' l2' new-body e' h3 l3 e2 s2 T'')
 have eval:P,E ⊢ ⟨e•M(es),s0⟩ ⇒ ⟨e2,s2⟩
   and eval':P,E ⊢ ⟨e,s0⟩ ⇒ ⟨ref(a,Cs),s1⟩
   and eval'':P,E ⊢ ⟨es,s1⟩ [⇒] ⟨map Val vs,(h2,l2)⟩ and h2:h2 a = Some(C,S)
   and has-least:P ⊢ last Cs has least M = (Ts',T',pns',body') via Ds
   and selects:P ⊢ (C,Cs@pDs) selects M = (Ts,T,pns,body) via Cs'
   and length:length vs = length pns and Casts:P ⊢ Ts Casts vs to vs'
   and l2':l2' = [this ↦ Ref (a, Cs'), pns [↦] vs']
   and new-body:new-body = (case T' of Class D ⇒ (⌊D⌋)body | - ⇒ body)
   and eval-body:P,E(this ↦ Class (last Cs'), pns [↦] Ts) ⊢
                                   ⟨new-body,(h2,l2')⟩ ⇒ ⟨e',(h3,l3)⟩
   and wt:P,E ⊢ e•M(es) :: T'' and sconf:P,E ⊢ s0 √
   and IH1:⋀ei si T. ⟦P,E ⊢ ⟨e,s0⟩ ⇒ ⟨ei,si⟩; P,E ⊢ e :: T; P,E ⊢ s0 √⟧
                 ⟹ ref (a,Cs) = ei ∧ s1 = si
   and IH2:⋀esi si Ts. ⟦P,E ⊢ ⟨es,s1⟩ [⇒] ⟨esi,si⟩; P,E ⊢ es [::] Ts; P,E ⊢ s1
√⟧
                  ⟹ map Val vs = esi ∧ (h2,l2) = si
   and IH3:⋀ei si T.
   ⟦P,E(this ↦ Class (last Cs'), pns [↦] Ts) ⊢ ⟨new-body,(h2,l2')⟩ ⇒ ⟨ei,si⟩;
    P,E(this ↦ Class (last Cs'), pns [↦] Ts) ⊢ new-body :: T;
    P,E(this ↦ Class (last Cs'), pns [↦] Ts) ⊢ (h2,l2') √⟧
 ⟹ e' = ei ∧ (h3, l3) = si by fact+
 from wt obtain D Ss Ss' m Cs'' where wte:P,E ⊢ e :: Class D
   and has-least':P ⊢ D has least M = (Ss,T'',m) via Cs''
   and wtes:P,E ⊢ es [::] Ss' and subs:P ⊢ Ss' [≤] Ss by auto
```

from *eval-preserves-type*[*OF wf eval′ sconf wte*]
have *last*:*last Cs* = *D* **by** (*auto split*:*if-split-asm*)
with *has-least has-least′ wf*
have *eq*:*Ts′* = *Ss* ∧ *T′* = *T″* ∧ (*pns′*,*body′*) = *m* ∧ *Ds* = *Cs″*
by(*fastforce dest*:*wf-sees-method-fun*)
from *wf selects* **have** *param-type*:∀ *T* ∈ *set Ts*. *is-type P T*
and *return-type*:*is-type P T* **and** *TnotNT*:*T* ≠ *NT*
by(*auto dest*:*select-method-wf-mdecl simp*:*wf-mdecl-def*)
from *selects wf* **have** *subo*:*Subobjs P C Cs′*
by(*induct rule*:*SelectMethodDef*.*induct*,
auto simp:*FinalOverriderMethodDef-def OverriderMethodDefs-def*
MinimalMethodDefs-def LeastMethodDef-def MethodDefs-def)
with *wf* **have** *class*:*is-class P* (*last Cs′*) **by**(*auto intro*!:*Subobj-last-isClass*)
from *eval″* **have** *hext*:*hp s*$_1$ ◁ *h*$_2$ **by** (*cases s*$_1$,*auto intro*: *evals-hext*)
from *wf eval′ sconf wte last* **have** *P*,*E*,(*hp s*$_1$) ⊢ *ref*(*a*,*Cs*) :$_{NT}$ *Class*(*last Cs*)
by −(*rule eval-preserves-type*,*simp-all*)
with *hext* **have** *P*,*E*,*h*$_2$ ⊢ *ref*(*a*,*Cs*) :$_{NT}$ *Class*(*last Cs*)
by(*auto intro*:*WTrt-hext-mono dest*:*hext-objD split*:*if-split-asm*)
with *h2* **have** *Subobjs P C Cs* **by** (*auto split*:*if-split-asm*)
hence *P* ⊢ *Path C to* (*last Cs*) *via Cs*
by (*auto simp*:*path-via-def split*:*if-split-asm*)
with *selects has-least wf* **have** *param-types*:*Ts′* = *Ts* ∧ *P* ⊢ *T* ≤ *T′*
by −(*rule select-least-methods-subtypes*,*simp-all*)
from *wf selects* **have** *wt-body*:*P*,[*this*↦*Class*(*last Cs′*),*pns*[↦]*Ts*] ⊢ *body* :: *T*
and *this-not-pns*:*this* ∉ *set pns* **and** *length*:*length pns* = *length Ts*
and *dist*:*distinct pns*
by(*auto dest*!:*select-method-wf-mdecl simp*:*wf-mdecl-def*)
have *P*,[*this*↦*Class*(*last Cs′*),*pns*[↦]*Ts*] ⊢ *new-body* :: *T′*
proof(*cases* ∃ *C*. *T′* = *Class C*)
case *False* **with** *wt-body new-body param-types* **show** *?thesis* **by**(*cases T′*) *auto*
next
case *True*
then obtain *D′* **where** *T′*:*T′* = *Class D′* **by** *auto*
with *wf has-least* **have** *class*:*is-class P D′*
by(*fastforce dest*:*has-least-wf-mdecl simp*:*wf-mdecl-def*)
with *wf T′ TnotNT param-types* **obtain** *D″* **where** *T*:*T* = *Class D″*
by(*fastforce dest*:*widen-Class*)
with *wf return-type T′ param-types* **have** *P* ⊢ *Path D″ to D′ unique*
by(*simp add*:*Class-widen-Class*)
with *wt-body class T T′ new-body* **show** *?thesis* **by** *auto*
qed
hence *wt-new-body*:*P*,*E*(*this*↦*Class*(*last Cs′*),*pns*[↦]*Ts*) ⊢ *new-body* :: *T′*
by(*fastforce intro*:*wt-env-mono*)
from *eval* **show** *?case*
proof(*rule eval-cases*)
fix *ex′* **assume** *eval-throw*:*P*,*E* ⊢ ⟨*e*,*s*$_0$⟩ ⇒ ⟨*throw ex′*,*s*$_2$⟩
from *IH1*[*OF eval-throw wte sconf*] **show** *e′* = *e*$_2$ ∧ (*h*$_3$, *l*$_2$) = *s*$_2$ **by** *simp*
next
fix *es″ ex′ s w ws*

assume *eval-val*:$P,E \vdash \langle e,s_0\rangle \Rightarrow \langle Val\ w,s\rangle$
and *evals-throw*:$P,E \vdash \langle es,s\rangle\ [\Rightarrow]\ \langle map\ Val\ ws@throw\ ex'\#es'',s_2\rangle$
from *IH1*[*OF eval-val wte sconf*] **have** *eq*:$s = s_1$ **by** *simp*
with *wf eval-val wte sconf* **have** *sconf'*:$P,E \vdash s_1\ \surd$
by(*fastforce intro*:*eval-preserves-sconf*)
from *IH2*[*OF evals-throw*[*simplified eq*] *wtes this*] **show** $e' = e_2 \wedge (h_3,\ l_2) = s_2$
by(*fastforce dest*:*map-Val-throw-False*)
next
fix *C' Xs Xs' Ds' S' U U' Us Us' a' body'' body''' h h' l l' pns'' pns''' s ws ws'*
assume *eval-ref*:$P,E \vdash \langle e,s_0\rangle \Rightarrow \langle ref(a',Xs),s\rangle$
and *evals-vals*:$P,E \vdash \langle es,s\rangle\ [\Rightarrow]\ \langle map\ Val\ ws,(h,l)\rangle$
and *h*:$h\ a' = Some(C',S')$
and *has-least''*:$P \vdash last\ Xs\ has\ least\ M = (Us',U',pns''',body''')\ via\ Ds'$
and *selects'*:$P \vdash (C',Xs@_pDs')\ selects\ M = (Us,U,pns'',body'')\ via\ Xs'$
and *length'*:$length\ ws = length\ pns''$ **and** *Casts'*:$P \vdash Us\ Casts\ ws\ to\ ws'$
and *eval-body'*:$P,E(this \mapsto Class\ (last\ Xs'),\ pns''\ [\mapsto]\ Us) \vdash$
$\langle case\ U'\ of\ Class\ D \Rightarrow (\!|D|\!)body''\ |\ \text{-} \Rightarrow body'',$
$(h,[this \mapsto Ref(a',Xs'),\ pns''\ [\mapsto]\ ws'])\rangle \Rightarrow \langle e_2,(h',l')\rangle$
and *s2*:$s_2 = (h',l)$
from *IH1*[*OF eval-ref wte sconf*] **have** *eq1*:$a = a' \wedge Cs = Xs$ **and** *s*:$s = s_1$
by *simp-all*
with *has-least has-least'' wf* **have** *eq2*:$T' = U' \wedge Ts' = Us' \wedge Ds = Ds'$
by(*fastforce dest*:*wf-sees-method-fun*)
from *s wf eval-ref wte sconf* **have** *sconf'*:$P,E \vdash s_1\ \surd$
by(*fastforce intro*:*eval-preserves-sconf*)
from *IH2*[*OF evals-vals*[*simplified s*] *wtes this*]
have *eq3*:$vs = ws \wedge h_2 = h \wedge l_2 = l$
by(*fastforce elim*:*map-injective simp*:*inj-on-def*)
with *eq1 h2 h* **have** *eq4*:$C = C' \wedge S = S'$ **by** *simp*
with *eq1 eq2 selects selects' wf*
have *eq5*:$Ts = Us \wedge T = U \wedge pns'' = pns \wedge body'' = body \wedge Cs' = Xs'$
by *simp*(*drule-tac mthd'*=$(Us,U,pns'',body'')$ **in** *wf-select-method-fun*,*auto*)
with *subs eq param-types* **have** $P \vdash Ss'\ [\leq]\ Us$ **by** *simp*
with *wf Casts Casts' param-type wtes evals-vals sconf' s eq eq2 eq3 eq5*
have *eq6*:$vs' = ws'$
by(*fastforce intro*:*Casts-Casts-eq-result*)
with *eval-body' l2' eq1 eq2 eq3 eq5 new-body*
have *eval-body''*:$P,E(this \mapsto Class(last\ Cs'),\ pns\ [\mapsto]\ Ts) \vdash$
$\langle new\text{-}body,(h_2,l_2')\rangle \Rightarrow \langle e_2,(h',l')\rangle$
by *fastforce*
from *wf evals-vals wtes sconf' s eq3* **have** *sconf''*:$P,E \vdash (h_2,l_2)\ \surd$
by(*fastforce intro*:*evals-preserves-sconf*)
have $P,E(this \mapsto Class\ (last\ Cs'),\ pns\ [\mapsto]\ Ts) \vdash (h_2,l_2')\ \surd$
proof(*auto simp*:*sconf-def*)
from *sconf''* **show** $P \vdash h_2\ \surd$ **by**(*simp add*:*sconf-def*)
next
{ fix *V v* **assume** *map*:$[this \mapsto Ref\ (a,Cs'),\ pns\ [\mapsto]\ vs']\ V = Some\ v$

have $\exists\, T.\ (E(this \mapsto Class\ (last\ Cs'),\ pns\ [\mapsto]\ Ts))\ V = Some\ T \wedge$
$P,h_2 \vdash v :\leq T$
proof(*cases* $V \in set\ (this\#pns)$)
case *False* **with** *map* **show** *?thesis* **by** *simp*
next
case *True*
hence $V = this \vee V \in set\ pns$ **by** *simp*
thus *?thesis*
proof(*rule disjE*)
assume $V{:}V = this$
with *map this-not-pns* **have** $v = Ref(a,Cs')$ **by** *simp*
with *V h2 subo this-not-pns* **have**
$(E(this \mapsto Class\ (last\ Cs'),pns\ [\mapsto]\ Ts))\ V = Some(Class\ (last\ Cs'))$
and $P,h_2 \vdash v :\leq Class\ (last\ Cs')$ **by** *simp-all*
thus *?thesis* **by** *simp*
next
assume $V \in set\ pns$
then obtain i **where** $V{:}V = pns!i$ **and** *length-i*$:i < length\ pns$
by(*auto simp*:*in-set-conv-nth*)
from *Casts* **have** $length\ Ts = length\ vs'$
by(*induct rule*:*Casts-to.induct*,*auto*)
with *length* **have** $length\ pns = length\ vs'$ **by** *simp*
with *map dist V length-i* **have** $v{:}v = vs'!i$ **by**(*fastforce dest*:*maps-nth*)
from *length dist length-i*
have *env*$:(E(this \mapsto Class\ (last\ Cs'))(pns\ [\mapsto]\ Ts))\ (pns!i) = Some(Ts!i)$
by(*rule-tac* $E{=}E(this \mapsto Class\ (last\ Cs'))$ **in** *nth-maps*,*simp-all*)
from *wf Casts wtes subs eq param-types eval'' sconf'*
have $\forall\, i < length\ Ts.\ P,h_2 \vdash vs'!i :\leq Ts!i$
by *simp*(*rule Casts-conf*,*auto*)
with *length-i length env V v* **show** *?thesis* **by** *simp*
qed
qed }
thus $P,h_2 \vdash l_2'\ (:\leq)_w\ E(this \mapsto Class\ (last\ Cs'),\ pns\ [\mapsto]\ Ts)$
using *l2'* **by**(*simp add*:*lconf-def*)
next
{ **fix** *V Tx* **assume** *env*$:(E(this \mapsto Class\ (last\ Cs'),\ pns\ [\mapsto]\ Ts))\ V = Some\ Tx$
have *is-type P Tx*
proof(*cases* $V \in set\ (this\#pns)$)
case *False*
with *env sconf''* **show** *?thesis*
by(*clarsimp simp*:*sconf-def envconf-def*)
next
case *True*
hence $V = this \vee V \in set\ pns$ **by** *simp*
thus *?thesis*
proof(*rule disjE*)
assume $V = this$
with *env this-not-pns* **have** $Tx = Class(last\ Cs')$ **by** *simp*

with *class* **show** *?thesis* **by** *simp*
next
assume $V \in$ *set pns*
then obtain *i* **where** V:V = *pns*!*i* **and** *length-i*:$i <$ *length pns*
by(*auto simp*:*in-set-conv-nth*)
with *dist length env* **have** Tx = Ts!i **by**(*fastforce dest*:*maps-nth*)
with *length-i length* **have** $Tx \in$ *set Ts*
by(*fastforce simp*:*in-set-conv-nth*)
with *param-type* **show** *?thesis* **by** *simp*
qed
qed }
thus $P \vdash E(this \mapsto Class\ (last\ Cs'),\ pns\ [\mapsto]\ Ts)\ \surd$ **by** (*simp add*:*envconf-def*)
qed
from *IH3*[*OF eval-body'' wt-new-body this*] **have** $e' = e_2 \wedge (h_3,\ l_3) = (h',l')$ **.**
with *eq3 s2* **show** $e' = e_2 \wedge (h_3,l_2) = s_2$ **by** *simp*
next
fix *s ws*
assume *eval-null*:$P,E \vdash \langle e,s_0\rangle \Rightarrow \langle null,s\rangle$
from *IH1*[*OF eval-null wte sconf*] **show** $e' = e_2 \wedge (h_3,l_2) = s_2$ **by** *simp*
qed
next
case (*StaticCall E e* s_0 *a Cs* s_1 *es vs* h_2 l_2 *C Cs'' M Ts T pns body Cs'*
Ds vs' l_2' e' h_3 l_3 e_2 s_2 T')
have *eval*:$P,E \vdash \langle e\cdot(C::)M(es),s_0\rangle \Rightarrow \langle e_2,s_2\rangle$
and *eval'*:$P,E \vdash \langle e,s_0\rangle \Rightarrow \langle ref(a,Cs),s_1\rangle$
and *eval''*:$P,E \vdash \langle es,s_1\rangle\ [\Rightarrow]\ \langle map\ Val\ vs,(h_2,\ l_2)\rangle$
and *path-unique*:$P \vdash Path\ last\ Cs\ to\ C\ unique$
and *path-via*:$P \vdash Path\ last\ Cs\ to\ C\ via\ Cs''$
and *has-least*:$P \vdash C\ has\ least\ M = (Ts,T,pns,body)\ via\ Cs'$
and *Ds*:$Ds = (Cs@_pCs'')@_pCs'$ **and** *length*:*length vs* = *length pns*
and *Casts*:$P \vdash Ts\ Casts\ vs\ to\ vs'$
and $l2'$:$l_2' = [this \mapsto Ref\ (a,\ Ds),\ pns\ [\mapsto]\ vs']$
and *eval-body*:$P,E(this \mapsto Class\ (last\ Ds),\ pns\ [\mapsto]\ Ts) \vdash$
$\langle body,(h_2,l_2')\rangle \Rightarrow \langle e',(h_3,l_3)\rangle$
and *wt*:$P,E \vdash e\cdot(C::)M(es) :: T'$ **and** *sconf*:$P,E \vdash s_0\ \surd$
and *IH1*:$\bigwedge ei\ si\ T.\ [\![P,E \vdash \langle e,s_0\rangle \Rightarrow \langle ei,si\rangle;\ P,E \vdash e :: T;\ P,E \vdash s_0\ \surd]\!]$
$\Longrightarrow ref\ (a,Cs) = ei \wedge s_1 = si$
and *IH2*:$\bigwedge esi\ si\ Ts.$
$[\![P,E \vdash \langle es,s_1\rangle\ [\Rightarrow]\ \langle esi,si\rangle;\ P,E \vdash es\ [::]\ Ts;\ P,E \vdash s_1\ \surd]\!]$
$\Longrightarrow map\ Val\ vs = esi \wedge (h_2,l_2) = si$
and *IH3*:$\bigwedge ei\ si\ T.$
$[\![P,E(this \mapsto Class\ (last\ Ds),\ pns\ [\mapsto]\ Ts) \vdash \langle body,(h_2,l_2')\rangle \Rightarrow \langle ei,si\rangle;$
$P,E(this \mapsto Class\ (last\ Ds),\ pns\ [\mapsto]\ Ts) \vdash body :: T;$
$P,E(this \mapsto Class\ (last\ Ds),\ pns\ [\mapsto]\ Ts) \vdash (h_2,l_2')\ \surd]\!]$
$\Longrightarrow e' = ei \wedge (h_3,\ l_3) = si$ **by** *fact+*
from *wt has-least wf* **obtain** C' Ts' **where** *wte*:$P,E \vdash e :: Class\ C'$
and *wtes*:$P,E \vdash es\ [::]\ Ts'$ **and** *subs*:$P \vdash Ts'\ [\leq]\ Ts$
by(*auto dest*:*wf-sees-method-fun*)
from *eval-preserves-type*[*OF wf eval' sconf wte*]

have *last*:*last Cs* = *C′* **by** (*auto split*:*if-split-asm*)
from *wf has-least* **have** *param-type*:∀ *T* ∈ *set Ts*. *is-type P T*
and *return-type*:*is-type P T* **and** *TnotNT*:*T* ≠ *NT*
by(*auto dest*:*has-least-wf-mdecl simp*:*wf-mdecl-def*)
from *path-via* **have** *last′*:*last Cs″* = *last*(*Cs*@$_p$*Cs″*)
by(*fastforce intro*!:*appendPath-last Subobjs-nonempty simp*:*path-via-def*)
from *eval″* **have** *hext*:*hp* s_1 ◁ h_2 **by** (*cases* s_1,*auto intro*: *evals-hext*)
from *wf eval′ sconf wte last* **have** *P*,*E*,(*hp* s_1) ⊢ *ref*(*a*,*Cs*) :$_{NT}$ *Class*(*last Cs*)
by −(*rule eval-preserves-type*,*simp-all*)
with *hext* **have** *P*,*E*,h_2 ⊢ *ref*(*a*,*Cs*) :$_{NT}$ *Class*(*last Cs*)
by(*auto intro*:*WTrt-hext-mono dest*:*hext-objD split*:*if-split-asm*)
then obtain *D S* **where** *h2*:h_2 *a* = *Some*(*D*,*S*) **and** *Subobjs P D Cs*
by (*auto split*:*if-split-asm*)
with *path-via wf* **have** *Subobjs P D* (*Cs*@$_p$*Cs″*) **and** *last Cs″* = *C*
by(*auto intro*:*Subobjs-appendPath simp*:*path-via-def*)
with *has-least wf last′ Ds* **have** *subo*:*Subobjs P D Ds*
by(*fastforce intro*:*Subobjs-appendPath simp*:*LeastMethodDef-def MethodDefs-def*)
with *wf* **have** *class*:*is-class P* (*last Ds*) **by**(*auto intro*!:*Subobj-last-isClass*)
from *has-least wf* **obtain** *D′* **where** *Subobjs P D′ Cs′*
by(*auto simp*:*LeastMethodDef-def MethodDefs-def*)
with *Ds* **have** *last-Ds*:*last Cs′* = *last Ds*
by(*fastforce intro*!:*appendPath-last Subobjs-nonempty*)
with *wf has-least* **have** *P*,[*this*↦*Class*(*last Ds*),*pns*[↦]*Ts*] ⊢ *body* :: *T*
and *this-not-pns*:*this* ∉ *set pns* **and** *length*:*length pns* = *length Ts*
and *dist*:*distinct pns*
by(*auto dest*!:*has-least-wf-mdecl simp*:*wf-mdecl-def*)
hence *wt-body*:*P*,*E*(*this*↦*Class*(*last Ds*),*pns*[↦]*Ts*) ⊢ *body* :: *T*
by(*fastforce intro*:*wt-env-mono*)
from *eval* **show** *?case*
proof(*rule eval-cases*)
fix *ex′* **assume** *eval-throw*:*P*,*E* ⊢ ⟨*e*,s_0⟩ ⇒ ⟨*throw ex′*,s_2⟩
from *IH1*[*OF eval-throw wte sconf*] **show** *e′* = e_2 ∧ (h_3, l_2) = s_2 **by** *simp*
next
fix *es″ ex′ s w ws*
assume *eval-val*:*P*,*E* ⊢ ⟨*e*,s_0⟩ ⇒ ⟨*Val w*,*s*⟩
and *evals-throw*:*P*,*E* ⊢ ⟨*es*,*s*⟩ [⇒] ⟨*map Val ws*@*throw ex′*#*es″*,s_2⟩
from *IH1*[*OF eval-val wte sconf*] **have** *eq*:*s* = s_1 **by** *simp*
with *wf eval-val wte sconf* **have** *sconf′*:*P*,*E* ⊢ s_1 √
by(*fastforce intro*:*eval-preserves-sconf*)
from *IH2*[*OF evals-throw*[*simplified eq*] *wtes this*] **show** *e′* = e_2 ∧ (h_3, l_2) = s_2
by(*fastforce dest*:*map-Val-throw-False*)
next
fix *Xs Xs′ Xs″ U Us a′ body′ h h′ l l′ pns′ s ws ws′*
assume *eval-ref*:*P*,*E* ⊢ ⟨*e*,s_0⟩ ⇒ ⟨*ref*(*a′*,*Xs*),*s*⟩
and *evals-vals*:*P*,*E* ⊢ ⟨*es*,*s*⟩ [⇒] ⟨*map Val ws*,(*h*,*l*)⟩
and *path-unique′*:*P* ⊢ *Path last Xs to C unique*
and *path-via′*:*P* ⊢ *Path last Xs to C via Xs″*
and *has-least′*:*P* ⊢ *C has least M* = (*Us*,*U*,*pns′*,*body′*) *via Xs′*

and *length'*:*length ws* = *length pns'*
and *Casts'*:$P \vdash$ *Us Casts ws to ws'*
and *eval-body'*:$P,E(this \mapsto Class(last((Xs@_pXs'')@_pXs')),pns'\ [\mapsto]\ Us) \vdash$
$\langle body',(h,[this \mapsto Ref(a',(Xs@_pXs'')@_pXs'),pns'\ [\mapsto]\ ws'])\rangle \Rightarrow \langle e_2,(h',l')\rangle$
and *s2*:$s_2 = (h',l)$
from *IH1*[*OF eval-ref wte sconf*] **have** *eq1*:$a = a' \wedge Cs = Xs$ **and** *s*:$s = s_1$
by *simp-all*
from *has-least has-least' wf*
have *eq2*:$T = U \wedge Ts = Us \wedge Cs' = Xs' \wedge pns = pns' \wedge body = body'$
by(*fastforce dest*:*wf-sees-method-fun*)
from *s wf eval-ref wte sconf* **have** *sconf'*:$P,E \vdash s_1\ \surd$
by(*fastforce intro*:*eval-preserves-sconf*)
from *IH2*[*OF evals-vals*[*simplified s*] *wtes this*]
have *eq3*:$vs = ws \wedge h_2 = h \wedge l_2 = l$
by(*fastforce elim*:*map-injective simp*:*inj-on-def*)
from *path-unique path-via path-via' eq1* **have** $Cs'' = Xs''$
by(*fastforce simp*:*path-unique-def path-via-def*)
with *Ds eq1 eq2* **have** *Ds'*:$Ds = (Xs@_pXs'')@_pXs'$ **by** *simp*
from *wf Casts Casts' param-type wtes subs evals-vals sconf' s eq2 eq3*
have *eq4*:$vs' = ws'$
by(*fastforce intro*:*Casts-Casts-eq-result*)
with *eval-body' Ds' l2' eq1 eq2 eq3*
have *eval-body''*:$P,E(this \mapsto Class(last\ Ds),pns\ [\mapsto]\ Ts) \vdash$
$\langle body,(h_2,l_2')\rangle \Rightarrow \langle e_2,(h',l')\rangle$
by *simp*
from *wf evals-vals wtes sconf' s eq3* **have** *sconf''*:$P,E \vdash (h_2,l_2)\ \surd$
by(*fastforce intro*:*evals-preserves-sconf*)
have $P,E(this \mapsto Class\ (last\ Ds),\ pns\ [\mapsto]\ Ts) \vdash (h_2,l_2')\ \surd$
proof(*auto simp*:*sconf-def*)
from *sconf''* **show** $P \vdash h_2\ \surd$ **by**(*simp add*:*sconf-def*)
next
{ fix *V v* **assume** *map*:$[this \mapsto Ref\ (a,Ds),\ pns\ [\mapsto]\ vs']\ V = Some\ v$
have $\exists\, T.\ (E(this \mapsto Class\ (last\ Ds),\ pns\ [\mapsto]\ Ts))\ V = Some\ T \wedge$
$P,h_2 \vdash v :\leq T$
proof(*cases* $V \in set\ (this\#pns)$)
case *False* **with** *map* **show** *?thesis* **by** *simp*
next
case *True*
hence $V = this \vee V \in set\ pns$ **by** *simp*
thus *?thesis*
proof(*rule disjE*)
assume *V*:$V = this$
with *map this-not-pns* **have** $v = Ref(a,Ds)$ **by** *simp*
with *V h2 subo this-not-pns* **have**
$(E(this \mapsto Class\ (last\ Ds),pns\ [\mapsto]\ Ts))\ V = Some(Class\ (last\ Ds))$
and $P,h_2 \vdash v :\leq Class\ (last\ Ds)$ **by** *simp-all*
thus *?thesis* **by** *simp*
next
assume $V \in set\ pns$

then obtain i **where** V:$V = pns!i$ **and** *length-i*:$i < length\ pns$
by(*auto simp*:*in-set-conv-nth*)
from *Casts* **have** $length\ Ts = length\ vs'$
by(*induct rule*:*Casts-to.induct*,*auto*)
with *length* **have** $length\ pns = length\ vs'$ **by** *simp*
with *map dist V length-i* **have** v:$v = vs'!i$ **by**(*fastforce dest*:*maps-nth*)
from *length dist length-i*
have *env*:$(E(this \mapsto Class\ (last\ Ds))(pns\ [\mapsto]\ Ts))\ (pns!i) = Some(Ts!i)$
by(*rule-tac* E=$E(this \mapsto Class\ (last\ Ds))$ **in** *nth-maps*,*simp-all*)
from *wf Casts wtes subs eval'' sconf'*
have $\forall\, i < length\ Ts.\ P,h_2 \vdash vs'!i :\leq Ts!i$
by −(*rule Casts-conf*,*auto*)
with *length-i length env V v* **show** *?thesis* **by** *simp*
qed
qed }
thus $P,h_2 \vdash l_2'\ (:\leq)_w\ E(this \mapsto Class\ (last\ Ds),\ pns\ [\mapsto]\ Ts)$
using *l2'* **by**(*simp add*:*lconf-def*)
next
{ **fix** $V\ Tx$ **assume** *env*:$(E(this \mapsto Class\ (last\ Ds),\ pns\ [\mapsto]\ Ts))\ V = Some\ Tx$
have *is-type P Tx*
proof(*cases* $V \in set\ (this\#pns)$)
case *False*
with *env sconf''* **show** *?thesis*
by(*clarsimp simp*:*sconf-def envconf-def*)
next
case *True*
hence $V = this \vee V \in set\ pns$ **by** *simp*
thus *?thesis*
proof(*rule disjE*)
assume $V = this$
with *env this-not-pns* **have** $Tx = Class(last\ Ds)$ **by** *simp*
with *class* **show** *?thesis* **by** *simp*
next
assume $V \in set\ pns$
then obtain i **where** V:$V = pns!i$ **and** *length-i*:$i < length\ pns$
by(*auto simp*:*in-set-conv-nth*)
with *dist length env* **have** $Tx = Ts!i$ **by**(*fastforce dest*:*maps-nth*)
with *length-i length* **have** $Tx \in set\ Ts$
by(*fastforce simp*:*in-set-conv-nth*)
with *param-type* **show** *?thesis* **by** *simp*
qed
qed }
thus $P \vdash E(this \mapsto Class\ (last\ Ds),\ pns\ [\mapsto]\ Ts)\ \surd$ **by** (*simp add*:*envconf-def*)
qed
from *IH3*[*OF eval-body'' wt-body this*] **have** $e' = e_2 \wedge (h_3,\ l_3) = (h',l')$.
with *eq3 s2* **show** $e' = e_2 \wedge (h_3,\ l_2) = s_2$ **by** *simp*
next
fix *s ws*

```
    assume eval-null:P,E ⊢ ⟨e,s0⟩ ⇒ ⟨null,s⟩
    from IH1[OF eval-null wte sconf] show e' = e2 ∧ (h3,l2) = s2 by simp
  qed
next
  case (CallNull E e s0 s1 es vs s2 Copt M e2 s2' T)
  have eval:P,E ⊢ ⟨Call e Copt M es,s0⟩ ⇒ ⟨e2,s2'⟩
    and wt:P,E ⊢ Call e Copt M es :: T and sconf:P,E ⊢ s0 √
    and IH1:⋀ei si T. ⟦P,E ⊢ ⟨e,s0⟩ ⇒ ⟨ei,si⟩; P,E ⊢ e :: T; P,E ⊢ s0 √⟧
    ⟹ null = ei ∧ s1 = si
    and IH2:⋀esi si Ts. ⟦P,E ⊢ ⟨es,s1⟩ [⇒] ⟨esi,si⟩; P,E ⊢ es [::] Ts; P,E ⊢ s1
√⟧
    ⟹ map Val vs = esi ∧ s2 = si by fact+
  from wt obtain C Ts where wte:P,E ⊢ e :: Class C and wtes:P,E ⊢ es [::] Ts

    by(cases Copt)auto
  show ?case
  proof(cases Copt)
    assume Copt = None
    with eval have P,E ⊢ ⟨e•M(es),s0⟩ ⇒ ⟨e2,s2'⟩ by simp
    thus ?thesis
    proof(rule eval-cases)
      fix ex' assume eval-throw:P,E ⊢ ⟨e,s0⟩ ⇒ ⟨throw ex',s2'⟩
      from IH1[OF eval-throw wte sconf] show THROW NullPointer = e2 ∧ s2
= s2'
        by simp
    next
      fix es' ex' s w ws
      assume eval-val:P,E ⊢ ⟨e,s0⟩ ⇒ ⟨Val w,s⟩
        and evals-throw:P,E ⊢ ⟨es,s⟩ [⇒] ⟨map Val ws@throw ex'#es',s2'⟩
      from IH1[OF eval-val wte sconf] have eq:s = s1 by simp
      with wf eval-val wte sconf have sconf':P,E ⊢ s1 √
        by(fastforce intro:eval-preserves-sconf)
      from IH2[OF evals-throw[simplified eq] wtes this]
    show THROW NullPointer = e2 ∧ s2 = s2' by(fastforce dest:map-Val-throw-False)
    next
      fix C' Xs Xs' Ds' S' U U' Us Us' a' body'' body''' h h' l l' pns'' pns'''
        s ws ws'
      assume eval-ref:P,E ⊢ ⟨e,s0⟩ ⇒ ⟨ref(a',Xs),s⟩
      from IH1[OF eval-ref wte sconf] show THROW NullPointer = e2 ∧ s2 =
s2'
        by simp
    next
      fix s ws
      assume eval-null:P,E ⊢ ⟨e,s0⟩ ⇒ ⟨null,s⟩
        and evals-vals:P,E ⊢ ⟨es,s⟩ [⇒] ⟨map Val ws,s2'⟩
        and e2:e2 = THROW NullPointer
      from IH1[OF eval-null wte sconf] have eq:s = s1 by simp
      with wf eval-null wte sconf have sconf':P,E ⊢ s1 √
        by(fastforce intro:eval-preserves-sconf)
```

from *IH2*[*OF evals-vals*[*simplified eq*] *wtes this*] *e2*
show *THROW NullPointer* = e_2 ∧ s_2 = s_2′ **by** *simp*
qed
next
fix C' **assume** *Copt* = *Some* C'
with *eval* **have** $P,E \vdash \langle e\bullet(C'::)M(es),s_0\rangle \Rightarrow \langle e_2,s_2'\rangle$ **by** *simp*
thus *?thesis*
proof(*rule eval-cases*)
fix *ex*′ **assume** *eval-throw*:$P,E \vdash \langle e,s_0\rangle \Rightarrow \langle throw\ ex',s_2'\rangle$
from *IH1*[*OF eval-throw wte sconf*] **show** *THROW NullPointer* = e_2 ∧ s_2 = s_2′
by *simp*
next
fix *es*′ *ex*′ *s w ws*
assume *eval-val*:$P,E \vdash \langle e,s_0\rangle \Rightarrow \langle Val\ w,s\rangle$
and *evals-throw*:$P,E \vdash \langle es,s\rangle\ [\Rightarrow]\ \langle map\ Val\ ws@throw\ ex'\#es',s_2'\rangle$
from *IH1*[*OF eval-val wte sconf*] **have** *eq*:$s = s_1$ **by** *simp*
with *wf eval-val wte sconf* **have** *sconf*′:$P,E \vdash s_1\ \surd$
by(*fastforce intro*:*eval-preserves-sconf*)
from *IH2*[*OF evals-throw*[*simplified eq*] *wtes this*]
show *THROW NullPointer* = e_2 ∧ s_2 = s_2′ **by**(*fastforce dest*:*map-Val-throw-False*)
next
fix *C*′ *Xs Xs*′ *Ds*′ *S*′ *U U*′ *Us Us*′ *a*′ *body*″ *body*‴ *h h*′ *l l*′ *pns*″ *pns*‴ *s ws ws*′
assume *eval-ref*:$P,E \vdash \langle e,s_0\rangle \Rightarrow \langle ref(a',Xs),s\rangle$
from *IH1*[*OF eval-ref wte sconf*] **show** *THROW NullPointer* = e_2 ∧ s_2 = s_2′
by *simp*
next
fix *s ws*
assume *eval-null*:$P,E \vdash \langle e,s_0\rangle \Rightarrow \langle null,s\rangle$
and *evals-vals*:$P,E \vdash \langle es,s\rangle\ [\Rightarrow]\ \langle map\ Val\ ws,s_2'\rangle$
and *e2*:e_2 = *THROW NullPointer*
from *IH1*[*OF eval-null wte sconf*] **have** *eq*:$s = s_1$ **by** *simp*
with *wf eval-null wte sconf* **have** *sconf*′:$P,E \vdash s_1\ \surd$
by(*fastforce intro*:*eval-preserves-sconf*)
from *IH2*[*OF evals-vals*[*simplified eq*] *wtes this*] *e2*
show *THROW NullPointer* = e_2 ∧ s_2 = s_2′ **by** *simp*
qed
qed
next
case (*Block E V T* e_0 h_0 l_0 e_1 h_1 l_1 e_2 s_2 *T*′)
have *eval*:$P,E \vdash \langle\{V:T;\ e_0\},(h_0,\ l_0)\rangle \Rightarrow \langle e_2,s_2\rangle$
and *wt*:$P,E \vdash \{V:T;\ e_0\} :: T'$ **and** *sconf*:$P,E \vdash (h_0,\ l_0)\ \surd$
and *IH*:$\bigwedge e_2\ s_2\ T'.\ [\![P,E(V \mapsto T) \vdash \langle e_0,(h_0,\ l_0(V := None))\rangle \Rightarrow \langle e_2,s_2\rangle;$
$P,E(V \mapsto T) \vdash e_0 :: T';\ P,E(V \mapsto T) \vdash (h_0,\ l_0(V := None))\ \surd]\!]$
$\Longrightarrow e_1 = e_2 \wedge (h_1,\ l_1) = s_2$ **by** *fact*+
from *wt* **have** *type*:*is-type P T* **and** *wte*:$P,E(V \mapsto T) \vdash e_0 :: T'$ **by** *auto*
from *sconf type* **have** *sconf*′:$P,E(V \mapsto T) \vdash (h_0,\ l_0(V := None))\ \surd$

```
    by(auto simp:sconf-def lconf-def envconf-def)
  from eval obtain h l where
    eval':P,E(V ↦ T) ⊢ ⟨e0,(h0,l0(V:=None))⟩ ⇒ ⟨e2,(h,l)⟩
    and s2:s2 = (h,l(V:=l0 V)) by (auto elim:eval-cases)
  from IH[OF eval' wte sconf'] s2 show ?case by simp
next
  case (Seq E e0 s0 v s1 e1 e2 s2 e2' s2' T)
  have eval:P,E ⊢ ⟨e0;; e1,s0⟩ ⇒ ⟨e2',s2'⟩
    and wt:P,E ⊢ e0;; e1 :: T and sconf:P,E ⊢ s0 √
    and IH1:⋀ei si T. ⟦P,E ⊢ ⟨e0,s0⟩ ⇒ ⟨ei,si⟩; P,E ⊢ e0 :: T; P,E ⊢ s0 √⟧
                  ⟹ Val v = ei ∧ s1 = si
    and IH2:⋀ei si T. ⟦P,E ⊢ ⟨e1,s1⟩ ⇒ ⟨ei,si⟩; P,E ⊢ e1 :: T; P,E ⊢ s1 √⟧
                  ⟹ e2 = ei ∧ s2 = si by fact+
  from wt obtain T' where wte0:P,E ⊢ e0 :: T' and wte1:P,E ⊢ e1 :: T by
auto
  from eval show ?case
  proof(rule eval-cases)
    fix s w
    assume eval-val:P,E ⊢ ⟨e0,s0⟩ ⇒ ⟨Val w,s⟩
      and eval':P,E ⊢ ⟨e1,s⟩ ⇒ ⟨e2',s2'⟩
    from IH1[OF eval-val wte0 sconf] have eq:s = s1 by simp
    with wf eval-val wte0 sconf have P,E ⊢ s1 √
      by(fastforce intro:eval-preserves-sconf)
    from IH2[OF eval'[simplified eq] wte1 this] show e2 = e2' ∧ s2 = s2' .
  next
    fix ex assume eval-throw:P,E ⊢ ⟨e0,s0⟩ ⇒ ⟨throw ex,s2'⟩
    from IH1[OF eval-throw wte0 sconf] show e2 = e2' ∧ s2 = s2' by simp
  qed
next
  case (SeqThrow E e0 s0 e s1 e1 e2 s2 T)
  have eval:P,E ⊢ ⟨e0;; e1,s0⟩ ⇒ ⟨e2,s2⟩
    and wt:P,E ⊢ e0;; e1 :: T and sconf:P,E ⊢ s0 √
    and IH:⋀ei si T. ⟦P,E ⊢ ⟨e0,s0⟩ ⇒ ⟨ei,si⟩; P,E ⊢ e0 :: T; P,E ⊢ s0 √⟧
                  ⟹ throw e = ei ∧ s1 = si by fact+
  from wt obtain T' where wte0:P,E ⊢ e0 :: T' by auto
  from eval show ?case
  proof(rule eval-cases)
    fix s w
    assume eval-val:P,E ⊢ ⟨e0,s0⟩ ⇒ ⟨Val w,s⟩
    from IH[OF eval-val wte0 sconf] show throw e = e2 ∧ s1 = s2 by simp
  next
    fix ex
    assume eval-throw:P,E ⊢ ⟨e0,s0⟩ ⇒ ⟨throw ex,s2⟩ and e2:e2 = throw ex
    from IH[OF eval-throw wte0 sconf] e2 show throw e = e2 ∧ s1 = s2 by simp
  qed
next
  case (CondT E e s0 s1 e1 e' s2 e2 e2' s2' T)
  have eval:P,E ⊢ ⟨if (e) e1 else e2,s0⟩ ⇒ ⟨e2',s2'⟩
    and wt:P,E ⊢ if (e) e1 else e2 :: T and sconf:P,E ⊢ s0 √
```

and *IH1*:$\bigwedge$*ei si T*. $[\![P,E \vdash \langle e,s_0\rangle \Rightarrow \langle ei,si\rangle;\ P,E \vdash e :: T;\ P,E \vdash s_0\ \surd]\!]$
$\Longrightarrow$ *true* $=$ *ei* $\wedge$ $s_1 = si$
and *IH2*:$\bigwedge$*ei si T*. $[\![P,E \vdash \langle e_1,s_1\rangle \Rightarrow \langle ei,si\rangle;\ P,E \vdash e_1 :: T;\ P,E \vdash s_1\ \surd]\!]$
$\Longrightarrow$ $e' = ei \wedge s_2 = si$ **by** *fact+*
from *wt* **have** *wte*:$P,E \vdash e ::$ *Boolean* **and** *wte1*:$P,E \vdash e_1 :: T$ **by** *auto*
from *eval* **show** *?case*
proof(*rule eval-cases*)
fix *s*
assume *eval-true*:$P,E \vdash \langle e,s_0\rangle \Rightarrow \langle true,s\rangle$ **and** *eval'*:$P,E \vdash \langle e_1,s\rangle \Rightarrow \langle {e_2}',{s_2}'\rangle$
from *IH1*[*OF eval-true wte sconf*] **have** *eq*:$s = s_1$ **by** *simp*
with *wf eval-true wte sconf* **have** $P,E \vdash s_1\ \surd$
by(*fastforce intro*:*eval-preserves-sconf*)
from *IH2*[*OF eval'*[*simplified eq*] *wte1 this*] **show** $e' = {e_2}' \wedge s_2 = {s_2}'$ **.**
next
fix *s* **assume** *eval-false*:$P,E \vdash \langle e,s_0\rangle \Rightarrow \langle false,s\rangle$
from *IH1*[*OF eval-false wte sconf*] **show** $e' = {e_2}' \wedge s_2 = {s_2}'$ **by** *simp*
next
fix *ex* **assume** *eval-throw*:$P,E \vdash \langle e,s_0\rangle \Rightarrow \langle throw\ ex,{s_2}'\rangle$
from *IH1*[*OF eval-throw wte sconf*] **show** $e' = {e_2}' \wedge s_2 = {s_2}'$ **by** *simp*
qed
next
case (*CondF E e* s_0 s_1 e_2 e' s_2 e_1 ${e_2}'$ ${s_2}'$ *T*)
have *eval*:$P,E \vdash \langle$*if* (*e*) e_1 *else* $e_2,s_0\rangle \Rightarrow \langle {e_2}',{s_2}'\rangle$
and *wt*:$P,E \vdash$ *if* (*e*) e_1 *else* $e_2 ::$ *T* **and** *sconf*:$P,E \vdash s_0\ \surd$
and *IH1*:$\bigwedge$*ei si T*. $[\![P,E \vdash \langle e,s_0\rangle \Rightarrow \langle ei,si\rangle;\ P,E \vdash e :: T;\ P,E \vdash s_0\ \surd]\!]$
$\Longrightarrow$ *false* $=$ *ei* $\wedge$ $s_1 = si$
and *IH2*:$\bigwedge$*ei si T*. $[\![P,E \vdash \langle e_2,s_1\rangle \Rightarrow \langle ei,si\rangle;\ P,E \vdash e_2 :: T;\ P,E \vdash s_1\ \surd]\!]$
$\Longrightarrow$ $e' = ei \wedge s_2 = si$ **by** *fact+*
from *wt* **have** *wte*:$P,E \vdash e ::$ *Boolean* **and** *wte2*:$P,E \vdash e_2 :: T$ **by** *auto*
from *eval* **show** *?case*
proof(*rule eval-cases*)
fix *s*
assume *eval-true*:$P,E \vdash \langle e,s_0\rangle \Rightarrow \langle true,s\rangle$
from *IH1*[*OF eval-true wte sconf*] **show** $e' = {e_2}' \wedge s_2 = {s_2}'$ **by** *simp*
next
fix *s*
assume *eval-false*:$P,E \vdash \langle e,s_0\rangle \Rightarrow \langle false,s\rangle$
and *eval'*:$P,E \vdash \langle e_2,s\rangle \Rightarrow \langle {e_2}',{s_2}'\rangle$
from *IH1*[*OF eval-false wte sconf*] **have** *eq*:$s = s_1$ **by** *simp*
with *wf eval-false wte sconf* **have** $P,E \vdash s_1\ \surd$
by(*fastforce intro*:*eval-preserves-sconf*)
from *IH2*[*OF eval'*[*simplified eq*] *wte2 this*] **show** $e' = {e_2}' \wedge s_2 = {s_2}'$ **.**
next
fix *ex* **assume** *eval-throw*:$P,E \vdash \langle e,s_0\rangle \Rightarrow \langle throw\ ex,{s_2}'\rangle$
from *IH1*[*OF eval-throw wte sconf*] **show** $e' = {e_2}' \wedge s_2 = {s_2}'$ **by** *simp*
qed
next
case (*CondThrow E e* s_0 e' s_1 e_1 e_2 ${e_2}'$ s_2 *T*)
have *eval*:$P,E \vdash \langle$*if* (*e*) e_1 *else* $e_2,s_0\rangle \Rightarrow \langle {e_2}',s_2\rangle$

and *wt*:$P,E \vdash$ *if* (e) e_1 *else* $e_2 :: T$ **and** *sconf*:$P,E \vdash s_0\ \surd$
and *IH*:$\bigwedge ei\ si\ T.\ [\![P,E \vdash \langle e,s_0\rangle \Rightarrow \langle ei,si\rangle;\ P,E \vdash e :: T;\ P,E \vdash s_0\ \surd]\!]$
$\Longrightarrow$ *throw* $e' = ei \wedge s_1 = si$ **by** *fact+*
from *wt* **have** *wte*:$P,E \vdash e ::$ *Boolean* **by** *auto*
from *eval* **show** *?case*
proof(*rule eval-cases*)
fix s
assume *eval-true*:$P,E \vdash \langle e,s_0\rangle \Rightarrow \langle true,s\rangle$
from *IH*[*OF eval-true wte sconf*] **show** *throw* $e' = e_2' \wedge s_1 = s_2$ **by** *simp*
next
fix s **assume** *eval-false*:$P,E \vdash \langle e,s_0\rangle \Rightarrow \langle false,s\rangle$
from *IH*[*OF eval-false wte sconf*] **show** *throw* $e' = e_2' \wedge s_1 = s_2$ **by** *simp*
next
fix *ex*
assume *eval-throw*:$P,E \vdash \langle e,s_0\rangle \Rightarrow \langle throw\ ex,s_2\rangle$ **and** *e2'*:$e_2' =$ *throw ex*
from *IH*[*OF eval-throw wte sconf*] *e2'* **show** *throw* $e' = e_2' \wedge s_1 = s_2$ **by** *simp*
qed
next
case (*WhileF E e* $s_0\ s_1$ *c* $e_2\ s_2\ T$)
have *eval*:$P,E \vdash \langle while\ (e)\ c,s_0\rangle \Rightarrow \langle e_2,s_2\rangle$
and *wt*:$P,E \vdash$ *while* (e) $c :: T$ **and** *sconf*:$P,E \vdash s_0\ \surd$
and *IH*:$\bigwedge e_2\ s_2\ T.\ [\![P,E \vdash \langle e,s_0\rangle \Rightarrow \langle e_2,s_2\rangle;\ P,E \vdash e :: T;\ P,E \vdash s_0\ \surd]\!]$
$\Longrightarrow$ *false* $= e_2 \wedge s_1 = s_2$ **by** *fact+*
from *wt* **have** *wte*:$P,E \vdash e ::$ *Boolean* **by** *auto*
from *eval* **show** *?case*
proof(*rule eval-cases*)
assume *eval-false*:$P,E \vdash \langle e,s_0\rangle \Rightarrow \langle false,s_2\rangle$ **and** *e2*:$e_2 =$ *unit*
from *IH*[*OF eval-false wte sconf*] *e2* **show** *unit* $= e_2 \wedge s_1 = s_2$ **by** *simp*
next
fix $s\ s'\ w$
assume *eval-true*:$P,E \vdash \langle e,s_0\rangle \Rightarrow \langle true,s\rangle$
from *IH*[*OF eval-true wte sconf*] **show** *unit* $= e_2 \wedge s_1 = s_2$ **by** *simp*
next
fix *ex* **assume** *eval-throw*:$P,E \vdash \langle e,s_0\rangle \Rightarrow \langle throw\ ex,s_2\rangle$
from *IH*[*OF eval-throw wte sconf*] **show** *unit* $= e_2 \wedge s_1 = s_2$ **by** *simp*
next
fix *ex s*
assume *eval-true*:$P,E \vdash \langle e,s_0\rangle \Rightarrow \langle true,s\rangle$
from *IH*[*OF eval-true wte sconf*] **show** *unit* $= e_2 \wedge s_1 = s_2$ **by** *simp*
qed
next
case (*WhileT E e* $s_0\ s_1$ *c* $v_1\ s_2\ e_3\ s_3\ e_2\ s_2'\ T$)
have *eval*:$P,E \vdash \langle while\ (e)\ c,s_0\rangle \Rightarrow \langle e_2,s_2'\rangle$
and *wt*:$P,E \vdash$ *while* (e) $c :: T$ **and** *sconf*:$P,E \vdash s_0\ \surd$
and *IH1*:$\bigwedge ei\ si\ T.\ [\![P,E \vdash \langle e,s_0\rangle \Rightarrow \langle ei,si\rangle;\ P,E \vdash e :: T;\ P,E \vdash s_0\ \surd]\!]$
$\Longrightarrow$ *true* $= ei \wedge s_1 = si$
and *IH2*:$\bigwedge ei\ si\ T.\ [\![P,E \vdash \langle c,s_1\rangle \Rightarrow \langle ei,si\rangle;\ P,E \vdash c :: T;\ P,E \vdash s_1\ \surd]\!]$
$\Longrightarrow$ *Val* $v_1 = ei \wedge s_2 = si$
and *IH3*:$\bigwedge ei\ si\ T.\ [\![P,E \vdash \langle while\ (e)\ c,s_2\rangle \Rightarrow \langle ei,si\rangle;\ P,E \vdash while\ (e)\ c :: T;$

$P,E \vdash s_2\ \surd\rrbracket$
$\Longrightarrow e_3 = ei \wedge s_3 = si$ **by** *fact+*
from *wt* **obtain** T' **where** *wte*:$P,E \vdash e :: Boolean$ **and** *wtc*:$P,E \vdash c :: T'$ **by** *auto*
from *eval* **show** *?case*
proof(*rule eval-cases*)
assume *eval-false*:$P,E \vdash \langle e,s_0\rangle \Rightarrow \langle false,s_2{}'\rangle$
from *IH1*[*OF eval-false wte sconf*] **show** $e_3 = e_2 \wedge s_3 = s_2{}'$ **by** *simp*
next
fix $s\ s'\ w$
assume *eval-true*:$P,E \vdash \langle e,s_0\rangle \Rightarrow \langle true,s\rangle$
and *eval-val*:$P,E \vdash \langle c,s\rangle \Rightarrow \langle Val\ w,s'\rangle$
and *eval-while*:$P,E \vdash \langle while\ (e)\ c,s'\rangle \Rightarrow \langle e_2,s_2{}'\rangle$
from *IH1*[*OF eval-true wte sconf*] **have** *eq*:$s = s_1$ **by** *simp*
with *wf eval-true wte sconf* **have** *sconf'*:$P,E \vdash s_1\ \surd$
by(*fastforce intro*:*eval-preserves-sconf*)
from *IH2*[*OF eval-val*[*simplified eq*] *wtc this*] **have** *eq'*:$s' = s_2$ **by** *simp*
with *wf eval-val wtc sconf' eq* **have** $P,E \vdash s_2\ \surd$
by(*fastforce intro*:*eval-preserves-sconf*)
from *IH3*[*OF eval-while*[*simplified eq'*] *wt this*] **show** $e_3 = e_2 \wedge s_3 = s_2{}'$ **.**
next
fix *ex* **assume** *eval-throw*:$P,E \vdash \langle e,s_0\rangle \Rightarrow \langle throw\ ex,s_2{}'\rangle$
from *IH1*[*OF eval-throw wte sconf*] **show** $e_3 = e_2 \wedge s_3 = s_2{}'$ **by** *simp*
next
fix *ex s*
assume *eval-true*:$P,E \vdash \langle e,s_0\rangle \Rightarrow \langle true,s\rangle$
and *eval-throw*:$P,E \vdash \langle c,s\rangle \Rightarrow \langle throw\ ex,s_2{}'\rangle$
from *IH1*[*OF eval-true wte sconf*] **have** *eq*:$s = s_1$ **by** *simp*
with *wf eval-true wte sconf* **have** *sconf'*:$P,E \vdash s_1\ \surd$
by(*fastforce intro*:*eval-preserves-sconf*)
from *IH2*[*OF eval-throw*[*simplified eq*] *wtc this*] **show** $e_3 = e_2 \wedge s_3 = s_2{}'$ **by** *simp*
qed
next
case (*WhileCondThrow E e* $s_0\ e'\ s_1\ c\ e_2\ s_2\ T$)
have *eval*:$P,E \vdash \langle while\ (e)\ c,s_0\rangle \Rightarrow \langle e_2,s_2\rangle$
and *wt*:$P,E \vdash while\ (e)\ c :: T$ **and** *sconf*:$P,E \vdash s_0\ \surd$
and *IH*:$\bigwedge ei\ si\ T.\ \llbracket P,E \vdash \langle e,s_0\rangle \Rightarrow \langle ei,si\rangle;\ P,E \vdash e :: T;\ P,E \vdash s_0\ \surd\rrbracket$
$\Longrightarrow throw\ e' = ei \wedge s_1 = si$ **by** *fact+*
from *wt* **have** *wte*:$P,E \vdash e :: Boolean$ **by** *auto*
from *eval* **show** *?case*
proof(*rule eval-cases*)
assume *eval-false*:$P,E \vdash \langle e,s_0\rangle \Rightarrow \langle false,s_2\rangle$
from *IH*[*OF eval-false wte sconf*] **show** $throw\ e' = e_2 \wedge s_1 = s_2$ **by** *simp*
next
fix $s\ s'\ w$
assume *eval-true*:$P,E \vdash \langle e,s_0\rangle \Rightarrow \langle true,s\rangle$
from *IH*[*OF eval-true wte sconf*] **show** $throw\ e' = e_2 \wedge s_1 = s_2$ **by** *simp*
next

fix *ex*
assume *eval-throw*:$P,E \vdash \langle e,s_0\rangle \Rightarrow \langle throw\ ex,s_2\rangle$ **and** *e2*:$e_2 = throw\ ex$
from *IH*[*OF eval-throw wte sconf*] *e2* **show** $throw\ e' = e_2 \wedge s_1 = s_2$ **by** *simp*
next
fix *ex s*
assume *eval-true*:$P,E \vdash \langle e,s_0\rangle \Rightarrow \langle true,s\rangle$
from *IH*[*OF eval-true wte sconf*] **show** $throw\ e' = e_2 \wedge s_1 = s_2$ **by** *simp*
qed
next
case (*WhileBodyThrow E e* $s_0\ s_1$ *c* $e'\ s_2\ e_2\ {s_2}'\ T$)
have *eval*:$P,E \vdash \langle while\ (e)\ c,s_0\rangle \Rightarrow \langle e_2,{s_2}'\rangle$
and *wt*:$P,E \vdash while\ (e)\ c :: T$ **and** *sconf*:$P,E \vdash s_0\ \surd$
and *IH1*:$\bigwedge ei\ si\ T.\ [\![P,E \vdash \langle e,s_0\rangle \Rightarrow \langle ei,si\rangle;\ P,E \vdash e :: T;\ P,E \vdash s_0\ \surd]\!]$
$\Longrightarrow true = ei \wedge s_1 = si$
and *IH2*:$\bigwedge ei\ si\ T.\ [\![P,E \vdash \langle c,s_1\rangle \Rightarrow \langle ei,si\rangle;\ P,E \vdash c :: T;\ P,E \vdash s_1\ \surd]\!]$
$\Longrightarrow throw\ e' = ei \wedge s_2 = si$ **by** *fact+*
from *wt* **obtain** T' **where** *wte*:$P,E \vdash e :: Boolean$ **and** *wtc*:$P,E \vdash c :: T'$ **by** *auto*
from *eval* **show** *?case*
proof(*rule eval-cases*)
assume *eval-false*:$P,E \vdash \langle e,s_0\rangle \Rightarrow \langle false,{s_2}'\rangle$
from *IH1*[*OF eval-false wte sconf*] **show** $throw\ e' = e_2 \wedge s_2 = {s_2}'$ **by** *simp*
next
fix $s\ s'\ w$
assume *eval-true*:$P,E \vdash \langle e,s_0\rangle \Rightarrow \langle true,s\rangle$
and *eval-val*:$P,E \vdash \langle c,s\rangle \Rightarrow \langle Val\ w,s'\rangle$
from *IH1*[*OF eval-true wte sconf*] **have** *eq*:$s = s_1$ **by** *simp*
with *wf eval-true wte sconf* **have** *sconf'*:$P,E \vdash s_1\ \surd$
by(*fastforce intro*:*eval-preserves-sconf*)
from *IH2*[*OF eval-val*[*simplified eq*] *wtc this*] **show** $throw\ e' = e_2 \wedge s_2 = {s_2}'$
by *simp*
next
fix *ex* **assume** *eval-throw*:$P,E \vdash \langle e,s_0\rangle \Rightarrow \langle throw\ ex,{s_2}'\rangle$
from *IH1*[*OF eval-throw wte sconf*] **show** $throw\ e' = e_2 \wedge s_2 = {s_2}'$ **by** *simp*
next
fix *ex s*
assume *eval-true*:$P,E \vdash \langle e,s_0\rangle \Rightarrow \langle true,s\rangle$
and *eval-throw*:$P,E \vdash \langle c,s\rangle \Rightarrow \langle throw\ ex,{s_2}'\rangle$ **and** *e2*:$e_2 = throw\ ex$
from *IH1*[*OF eval-true wte sconf*] **have** *eq*:$s = s_1$ **by** *simp*
with *wf eval-true wte sconf* **have** *sconf'*:$P,E \vdash s_1\ \surd$
by(*fastforce intro*:*eval-preserves-sconf*)
from *IH2*[*OF eval-throw*[*simplified eq*] *wtc this*] *e2* **show** $throw\ e' = e_2 \wedge s_2 = {s_2}'$
by *simp*
qed
next
case (*Throw E e* $s_0\ r\ s_1\ e_2\ s_2\ T$)
have *eval*:$P,E \vdash \langle throw\ e,s_0\rangle \Rightarrow \langle e_2,s_2\rangle$
and *wt*:$P,E \vdash throw\ e :: T$ **and** *sconf*:$P,E \vdash s_0\ \surd$

and *IH*:$\bigwedge$*ei si T*. $[\![P,E \vdash \langle e,s_0\rangle \Rightarrow \langle ei,si\rangle;\ P,E \vdash e :: T;\ P,E \vdash s_0\ \surd]\!]$
$\Longrightarrow$ *ref r* = *ei* $\wedge$ $s_1 = si$ **by** *fact+*
from *wt* **obtain** *C* **where** *wte*:$P,E \vdash e :: Class\ C$ **by** *auto*
from *eval* **show** *?case*
proof(*rule eval-cases*)
fix r'
assume *eval-ref*:$P,E \vdash \langle e,s_0\rangle \Rightarrow \langle ref\ r',s_2\rangle$ **and** *e2*:$e_2 = Throw\ r'$
from *IH*[*OF eval-ref wte sconf*] *e2* **show** *Throw r* = $e_2 \wedge s_1 = s_2$ **by** *simp*
next
assume *eval-null*:$P,E \vdash \langle e,s_0\rangle \Rightarrow \langle null,s_2\rangle$
from *IH*[*OF eval-null wte sconf*] **show** *Throw r* = $e_2 \wedge s_1 = s_2$ **by** *simp*
next
fix *ex* **assume** *eval-throw*:$P,E \vdash \langle e,s_0\rangle \Rightarrow \langle throw\ ex,s_2\rangle$
from *IH*[*OF eval-throw wte sconf*] **show** *Throw r* = $e_2 \wedge s_1 = s_2$ **by** *simp*
qed
next
case (*ThrowNull E e* s_0 s_1 e_2 s_2 *T*)
have *eval*:$P,E \vdash \langle throw\ e,s_0\rangle \Rightarrow \langle e_2,s_2\rangle$
and *wt*:$P,E \vdash throw\ e :: T$ **and** *sconf*:$P,E \vdash s_0\ \surd$
and *IH*:$\bigwedge$*ei si T*. $[\![P,E \vdash \langle e,s_0\rangle \Rightarrow \langle ei,si\rangle;\ P,E \vdash e :: T;\ P,E \vdash s_0\ \surd]\!]$
$\Longrightarrow$ *null* = *ei* $\wedge$ $s_1 = si$ **by** *fact+*
from *wt* **obtain** *C* **where** *wte*:$P,E \vdash e :: Class\ C$ **by** *auto*
from *eval* **show** *?case*
proof(*rule eval-cases*)
fix r' **assume** *eval-ref*:$P,E \vdash \langle e,s_0\rangle \Rightarrow \langle ref\ r',s_2\rangle$
from *IH*[*OF eval-ref wte sconf*] **show** *THROW NullPointer* = $e_2 \wedge s_1 = s_2$
by *simp*
next
assume *eval-null*:$P,E \vdash \langle e,s_0\rangle \Rightarrow \langle null,s_2\rangle$ **and** *e2*:$e_2 = THROW\ NullPointer$
from *IH*[*OF eval-null wte sconf*] *e2* **show** *THROW NullPointer* = $e_2 \wedge s_1 = s_2$
by *simp*
next
fix *ex* **assume** *eval-throw*:$P,E \vdash \langle e,s_0\rangle \Rightarrow \langle throw\ ex,s_2\rangle$
from *IH*[*OF eval-throw wte sconf*] **show** *THROW NullPointer* = $e_2 \wedge s_1 = s_2$ **by** *simp*
qed
next
case (*ThrowThrow E e* s_0 e' s_1 e_2 s_2 *T*)
have *eval*:$P,E \vdash \langle throw\ e,s_0\rangle \Rightarrow \langle e_2,s_2\rangle$
and *wt*:$P,E \vdash throw\ e :: T$ **and** *sconf*:$P,E \vdash s_0\ \surd$
and *IH*:$\bigwedge$*ei si T*. $[\![P,E \vdash \langle e,s_0\rangle \Rightarrow \langle ei,si\rangle;\ P,E \vdash e :: T;\ P,E \vdash s_0\ \surd]\!]$
$\Longrightarrow$ *throw* e' = *ei* $\wedge$ $s_1 = si$ **by** *fact+*
from *wt* **obtain** *C* **where** *wte*:$P,E \vdash e :: Class\ C$ **by** *auto*
from *eval* **show** *?case*
proof(*rule eval-cases*)
fix r' **assume** *eval-ref*:$P,E \vdash \langle e,s_0\rangle \Rightarrow \langle ref\ r',s_2\rangle$
from *IH*[*OF eval-ref wte sconf*] **show** *throw* e' = $e_2 \wedge s_1 = s_2$ **by** *simp*
next

assume *eval-null*:$P,E \vdash \langle e,s_0\rangle \Rightarrow \langle null,s_2\rangle$
from *IH*[*OF eval-null wte sconf*] **show** *throw* $e' = e_2 \wedge s_1 = s_2$ **by** *simp*
next
fix *ex*
assume *eval-throw*:$P,E \vdash \langle e,s_0\rangle \Rightarrow \langle throw\ ex,s_2\rangle$ **and** *e2*:$e_2 = throw\ ex$
from *IH*[*OF eval-throw wte sconf*] *e2* **show** *throw* $e' = e_2 \wedge s_1 = s_2$ **by** *simp*
qed
next
case *Nil* **thus** *?case* **by** (*auto elim*:*evals-cases*)
next
case (*Cons E e* s_0 *v* s_1 *es es'* s_2 *es*$_2$ s_2' *Ts*)
have *evals*:$P,E \vdash \langle e\#es,s_0\rangle$ [⇒] $\langle es_2,s_2'\rangle$
and *wt*:$P,E \vdash e\#es$ [::] *Ts* **and** *sconf*:$P,E \vdash s_0\ \surd$
and *IH1*:$\bigwedge ei\ si\ T.\ [\![P,E \vdash \langle e,s_0\rangle \Rightarrow \langle ei,si\rangle;\ P,E \vdash e :: T;\ P,E \vdash s_0\ \surd]\!]$
$\Longrightarrow$ *Val v* $= ei \wedge s_1 = si$
and *IH2*:$\bigwedge esi\ si\ Ts.\ [\![P,E \vdash \langle es,s_1\rangle$ [⇒] $\langle esi,si\rangle;\ P,E \vdash es$ [::] *Ts*; $P,E \vdash s_1\ \surd]\!]$
$\Longrightarrow es' = esi \wedge s_2 = si$ **by** *fact*+
from *wt* **obtain** T' Ts' **where** *Ts*:$Ts = T'\#Ts'$ **by**(*cases Ts*) *auto*
with *wt* **have** *wte*:$P,E \vdash e :: T'$ **and** *wtes*:$P,E \vdash es$ [::] Ts' **by** *auto*
from *evals* **show** *?case*
proof(*rule evals-cases*)
fix *es''* *s w*
assume *eval-val*:$P,E \vdash \langle e,s_0\rangle \Rightarrow \langle Val\ w,s\rangle$
and *evals-vals*:$P,E \vdash \langle es,s\rangle$ [⇒] $\langle es'',s_2'\rangle$ **and** *es2*:$es_2 = Val\ w\#es''$
from *IH1*[*OF eval-val wte sconf*] **have** *s*:$s = s_1$ **and** *v*:$v = w$ **by** *simp-all*
with *wf eval-val wte sconf* **have** $P,E \vdash s_1\ \surd$
by(*fastforce intro*:*eval-preserves-sconf*)
from *IH2*[*OF evals-vals*[*simplified s*] *wtes this*] **have** $es' = es'' \wedge s_2 = s_2'$ **.**
with *es2 v* **show** *Val v* # $es' = es_2 \wedge s_2 = s_2'$ **by** *simp*
next
fix *ex* **assume** *eval-throw*:$P,E \vdash \langle e,s_0\rangle \Rightarrow \langle throw\ ex,s_2'\rangle$
from *IH1*[*OF eval-throw wte sconf*] **show** *Val v* # $es' = es_2 \wedge s_2 = s_2'$ **by** *simp*
qed
next
case (*ConsThrow E e* s_0 e' s_1 *es es*$_2$ s_2 *Ts*)
have *evals*:$P,E \vdash \langle e\#es,s_0\rangle$ [⇒] $\langle es_2,s_2\rangle$
and *wt*:$P,E \vdash e\#es$ [::] *Ts* **and** *sconf*:$P,E \vdash s_0\ \surd$
and *IH*:$\bigwedge ei\ si\ T.\ [\![P,E \vdash \langle e,s_0\rangle \Rightarrow \langle ei,si\rangle;\ P,E \vdash e :: T;\ P,E \vdash s_0\ \surd]\!]$
$\Longrightarrow$ *throw* $e' = ei \wedge s_1 = si$ **by** *fact*+
from *wt* **obtain** T' Ts' **where** *Ts*:$Ts = T'\#Ts'$ **by**(*cases Ts*) *auto*
with *wt* **have** *wte*:$P,E \vdash e :: T'$ **by** *auto*
from *evals* **show** *?case*
proof(*rule evals-cases*)
fix *es''* *s w*
assume *eval-val*:$P,E \vdash \langle e,s_0\rangle \Rightarrow \langle Val\ w,s\rangle$
from *IH*[*OF eval-val wte sconf*] **show** *throw* $e'\#es = es_2 \wedge s_1 = s_2$ **by** *simp*
next

fix *ex*
assume *eval-throw*:$P,E \vdash \langle e,s_0\rangle \Rightarrow \langle throw\ ex,s_2\rangle$ **and** *es2*:$es_2 = throw\ ex\#es$
from *IH*[*OF eval-throw wte sconf*] *es2* **show** *throw e′#es* $= es_2 \wedge s_1 = s_2$ **by** *simp*
qed
qed

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* ⟹ $P,E \vdash$ *new C* ⇝ *new C*
| *AnnoCast*: $P,E \vdash e \rightsquigarrow e' \Longrightarrow P,E \vdash$ *Cast C e* ⇝ *Cast C e′*
| *AnnoStatCast*: $P,E \vdash e \rightsquigarrow e' \Longrightarrow P,E \vdash$ *StatCast C e* ⇝ *StatCast C e′*
| *AnnoVal*: $P,E \vdash$ *Val v* ⇝ *Val v*
| *AnnoVarVar*: *E V* = ⌊*T*⌋ ⟹ $P,E \vdash$ *Var V* ⇝ *Var V*
| *AnnoVarField*: ⟦ *E V* = *None*; *E this* = ⌊*Class C*⌋; *P* ⊢ *C has least V*:*T via Cs* ⟧
⟹ $P,E \vdash$ *Var V* ⇝ *Var this*•*V*{*Cs*}
| *AnnoBinOp*:
⟦ $P,E \vdash e1 \rightsquigarrow e1'$; $P,E \vdash e2 \rightsquigarrow e2'$ ⟧
⟹ $P,E \vdash e1$ ≪*bop*≫ *e2* ⇝ *e1′* ≪*bop*≫ *e2′*
| *AnnoLAss*:
$P,E \vdash e \rightsquigarrow e' \Longrightarrow P,E \vdash V{:=}e \rightsquigarrow V{:=}e'$
| *AnnoFAcc*:
⟦ $P,E \vdash e \rightsquigarrow e'$; $P,E \vdash e'$ :: *Class C*; *P* ⊢ *C has least F*:*T via Cs* ⟧
⟹ $P,E \vdash e$•*F*{[]} ⇝ *e′*•*F*{*Cs*}
| *AnnoFAss*: ⟦ $P,E \vdash e1 \rightsquigarrow e1'$; $P,E \vdash e2 \rightsquigarrow e2'$;

$P,E \vdash e1' :: Class\ C;\ P \vdash C\ has\ least\ F{:}T\ via\ Cs\ ]\!]$
$\Longrightarrow P,E \vdash e1{\cdot}F\{[]\} := e2 \rightsquigarrow e1'{\cdot}F\{Cs\} := e2'$
| *AnnoCall*:
$[\![\ P,E \vdash e \rightsquigarrow e';\ P,E \vdash es\ [\rightsquigarrow]\ es'\ ]\!]$
$\Longrightarrow P,E \vdash Call\ e\ Copt\ M\ es \rightsquigarrow Call\ e'\ Copt\ M\ es'$
| *AnnoBlock*:
$P,E(V \mapsto T) \vdash e \rightsquigarrow e' \Longrightarrow P,E \vdash \{V{:}T;\ e\} \rightsquigarrow \{V{:}T;\ e'\}$
| *AnnoComp*: $[\![\ P,E \vdash e1 \rightsquigarrow e1';\ P,E \vdash e2 \rightsquigarrow e2'\ ]\!]$
$\Longrightarrow P,E \vdash e1;;e2 \rightsquigarrow e1';;e2'$
| *AnnoCond*: $[\![\ P,E \vdash e \rightsquigarrow e';\ P,E \vdash e1 \rightsquigarrow e1';\ P,E \vdash e2 \rightsquigarrow e2'\ ]\!]$
$\Longrightarrow P,E \vdash if\ (e)\ e1\ else\ e2 \rightsquigarrow if\ (e')\ e1'\ else\ e2'$
| *AnnoLoop*: $[\![\ P,E \vdash e \rightsquigarrow e';\ P,E \vdash c \rightsquigarrow c'\ ]\!]$
$\Longrightarrow P,E \vdash while\ (e)\ c \rightsquigarrow while\ (e')\ c'$
| *AnnoThrow*: $P,E \vdash e \rightsquigarrow e' \Longrightarrow P,E \vdash throw\ e \rightsquigarrow throw\ e'$

| *AnnoNil*: $P,E \vdash [] \ [\rightsquigarrow]\ []$
| *AnnoCons*: $[\![\ P,E \vdash e \rightsquigarrow e';\ P,E \vdash es\ [\rightsquigarrow]\ es'\ ]\!]$
$\Longrightarrow P,E \vdash e\#es\ [\rightsquigarrow]\ e'\#es'$

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*
apply (*induct xs*)
apply *simp*
apply (*rule app.intros*)
apply *simp*
apply (*iprover intro*: *app.intros*)
done

theorem *app-eq2*: *app xs ys zs* ⟹ *zs* = *xs* @ *ys*
by (*erule app.induct*) *simp-all*

theorem *app-eq*: *app xs ys zs* = (*zs* = *xs* @ *ys*)
apply (*rule iffI*)
apply (*erule app-eq2*)
apply (*erule app-eq1*)
done

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
.

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
by(*erule converse-rtranclpE*) *blast+*

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*)
by(*auto simp add*: *Set-project-def map-filter-def intro*: *rev-image-eqI split*: *if-split-asm*)

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
.

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*: $i \Rightarrow i \Rightarrow i \Rightarrow bool$, $i \Rightarrow o \Rightarrow o \Rightarrow bool$)
map-val2
.

theorem *map-val-conv*: $(xs = map\ Val\ ys) = map\text{-}val\ xs\ ys$
theorem *map-val2-conv*:
$(xs = map\ Val\ ys\ @\ throw\ e\ \#\ zs) = map\text{-}val2\ xs\ ys\ (throw\ e\ \#\ zs)$

29.2 Code generation

lemma *subclsRp-code* [*code-pred-intro*]:
$[\![\ class\ P\ C = \lfloor(Bs,\ rest)\rfloor;\ Predicate\text{-}Compile.contains\ (set\ Bs)\ (Repeats\ D)\ ]\!]$
$\Longrightarrow subclsRp\ P\ C\ D$
by(*auto intro*: *subclsRp.intros simp add*: *contains-def*)

code-pred
(*modes*: $i \Rightarrow i \Rightarrow i \Rightarrow bool$, $i \Rightarrow i \Rightarrow o \Rightarrow bool$)
subclsRp
by(*erule subclsRp.cases*)(*fastforce simp add*: *Predicate-Compile.contains-def*)

lemma *subclsR-code* [*code-pred-inline*]:
$P \vdash C \prec_R D \longleftrightarrow subclsRp\ P\ C\ D$
by(*simp add*: *subclsR-def*)

lemma *subclsSp-code* [*code-pred-intro*]:
$[\![\ class\ P\ C = \lfloor(Bs,\ rest)\rfloor;\ Predicate\text{-}Compile.contains\ (set\ Bs)\ (Shares\ D)\ ]\!] \Longrightarrow$
$subclsSp\ P\ C\ D$
by(*auto intro*: *subclsSp.intros simp add*: *Predicate-Compile.contains-def*)

code-pred
(*modes*: $i \Rightarrow i \Rightarrow i \Rightarrow bool$, $i \Rightarrow i \Rightarrow o \Rightarrow bool$)
subclsSp
by(*erule subclsSp.cases*)(*fastforce simp add*: *Predicate-Compile.contains-def*)

declare *SubobjsR-Base* [*code-pred-intro*]
lemma *SubobjsR-Rep-code* [*code-pred-intro*]:
$[\![subclsRp\ P\ C\ D;\ Subobjs_R\ P\ D\ Cs]\!] \Longrightarrow Subobjs_R\ P\ C\ (C\ \#\ Cs)$
by(*simp add*: *SubobjsR-Rep subclsR-def*)

code-pred
(*modes*: $i \Rightarrow i \Rightarrow i \Rightarrow bool$, $i \Rightarrow i \Rightarrow o \Rightarrow bool$)
$Subobjs_R$
by(*erule* $Subobjs_R$*.cases*)(*auto simp add*: *subclsR-code*)

lemma *subcls1p-code* [*code-pred-intro*]:
$[\![class\ P\ C = Some\ (Bs,rest);\ Predicate\text{-}Compile.contains\ (baseClasses\ Bs)\ D\ ]\!]$
$\Longrightarrow subcls1p\ P\ C\ D$
by(*auto intro*: *subcls1p.intros simp add*: *Predicate-Compile.contains-def*)

code-pred (*modes*: $i \Rightarrow i \Rightarrow i \Rightarrow bool$, $i \Rightarrow i \Rightarrow o \Rightarrow bool$)
subcls1p
by(*fastforce elim*!: *subcls1p.cases simp add*: *Predicate-Compile.contains-def*)

declare *Subobjs-Rep* [*code-pred-intro*]
lemma *Subobjs-Sh-code* [*code-pred-intro*]:
$[\![(subcls1p\ P)\hat{}**\ C\ C'; subclsSp\ P\ C'\ D; Subobjs_R\ P\ D\ Cs]\!]$
$\Longrightarrow Subobjs\ P\ C\ Cs$
by(*rule Subobjs-Sh*)(*simp-all add*: *rtrancl-def subcls1-def subclsS-def*)

code-pred
(*modes*: $i \Rightarrow i \Rightarrow i \Rightarrow bool$, $i \Rightarrow i \Rightarrow o \Rightarrow bool$)
Subobjs
by(*erule Subobjs.cases*)(*auto simp add*: *rtrancl-def subcls1-def subclsS-def*)

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* **.**

lemma *widen-subcls'*:
$[\![Subobjs\ P\ C\ Cs'; last\ Cs' = D; widen\text{-}unique\ P\ C\ D\ Cs']\!]$
$\Longrightarrow P \vdash Class\ C \leq Class\ D$
by(*rule widen-subcls*,*auto simp*:*path-unique-def widen-unique-def*)

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
by(*erule widen.cases*)(*auto simp add*: *path-unique-def widen-unique-def*)

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
.

lemma *leq-path-unfold*: $P,C \vdash Cs \sqsubseteq Ds \longleftrightarrow (leq\text{-}path1p\ P\ C)\hat{}**\ Cs\ Ds$
by(*simp add*: *leq-path1-def rtrancl-def*)

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

path-via
.

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

code-pred
(*modes*: *i* => *i* => *o* => *bool*, *i* => *i* => *i* => *bool*)
[*inductify*, *skip-proof*]
path-unique .

Redefine MethodDefs and FieldDecls

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

lemma [*code-pred-intro*]:
Subobjs P C Cs ⟹ *class P* (*last Cs*) = ⌊(*Bs*,*fs*,*ms*)⌋ ⟹ *map-of ms M* = ⌊*mthd*⌋ ⟹
MethodDefs' P C M Cs mthd
by (*simp add*: *MethodDefs-def MethodDefs'-def*)

code-pred
(*modes*: *i* ⇒ *i* ⇒ *i* ⇒ *o* ⇒ *o* ⇒ *bool*, *i* ⇒ *i* ⇒ *i* ⇒ *i* ⇒ *i* ⇒ *bool*)
MethodDefs'
by(*fastforce simp add*: *MethodDefs-def MethodDefs'-def*)

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
by (*simp add*: *FieldDecls-def FieldDecls'-def*)

code-pred
(*modes*: *i* ⇒ *i* ⇒ *i* ⇒ *o* ⇒ *o* ⇒ *bool*, *i* ⇒ *i* ⇒ *i* ⇒ *i* ⇒ *i* ⇒ *bool*)
FieldDecls'
by(*fastforce simp add*: *FieldDecls-def FieldDecls'-def*)

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* .

lemma [*code-pred-intro*]:
MethodDefs′ P C M Cs mthd ⟹ *MinimalMethodDefs-unique P C M Cs* ⟹
MinimalMethodDefs′ P C M Cs mthd
by (*fastforce simp add*:*MinimalMethodDefs-def MinimalMethodDefs′-def MethodDefs′-def MinimalMethodDefs-unique-def leq-path-unfold*)

code-pred
(*modes*: *i* ⇒ *i* ⇒ *i* ⇒ *o* ⇒ *o* ⇒ *bool*)
MinimalMethodDefs′
by(*fastforce simp add*:*MinimalMethodDefs-def MinimalMethodDefs′-def MethodDefs′-def MinimalMethodDefs-unique-def leq-path-unfold*)

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* .

lemma *LeastMethodDef-unfold*:
P ⊢ *C has least M* = *mthd via Cs* ⟷
MethodDefs′ P C M Cs mthd ∧ *LeastMethodDef-unique P C M Cs*
by(*fastforce simp add*: *LeastMethodDef-def MethodDefs′-def leq-path-unfold LeastMethodDef-unique-def*)

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*
by(*simp add*: *LeastMethodDef-unfold LeastMethodDef-unique-def*)

code-pred (*modes*: *i* => *i* => *i* => *o* => *o* => *bool*)
LeastMethodDef
by(*simp add*: *LeastMethodDef-unfold LeastMethodDef-unique-def*)

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
apply(*simp add*:*OverriderMethodDefs-def OverriderMethodDefs′-def MinimalMethodDefs′-def appendPath-def leq-path-unfold*)
apply(*rule-tac x*=*Cs′* **in** *exI*)
apply *clarsimp*
apply(*cases mthd′*)
apply *blast*
done

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
by(*auto simp add*:*OverriderMethodDefs-def OverriderMethodDefs′-def MinimalMethodDefs′-def appendPath-def leq-path-unfold simp del*: *split-paired-Ex*)

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′
apply(*clarsimp simp add*: *OverriderMethodDefs′-def MinimalMethodDefs′-def MethodDefs′-def OverriderMethodDefs-def appendPath-def leq-path-unfold*)
apply(*case-tac last xb* = *hd Cs′*)
apply(*simp*)

apply(*thin-tac PROP* -)
apply(*simp add*: *leq-path1-def*)
done

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* **.**

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*
by(*rule WTDynCast*)(*auto simp add*: *WTDynCast-ex-def*)

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* .

lemma *WTStaticCast-new*:
⟦*P*,*E* ⊢ *e* :: *Class D*; *is-class P C*; *WTStaticCast-sub P C D* ⟧
⟹ *P*,*E* ⊢ (|*C*|)*e* :: *Class C*
by (*rule WTStaticCast*)(*auto simp add*: *WTStaticCast-sub-def subcls1-def rtrancl-def*)

lemma *WTBinOp1*: ⟦ *P*,*E* ⊢ e_1 :: *T*; *P*,*E* ⊢ e_2 :: *T*⟧
⟹ *P*,*E* ⊢ e_1 «*Eq*» e_2 :: *Boolean*
apply (*rule WTBinOp*)
apply *assumption*+
apply *simp*
done

lemma *WTBinOp2*: ⟦ *P*,*E* ⊢ e_1 :: *Integer*; *P*,*E* ⊢ e_2 :: *Integer* ⟧
⟹ *P*,*E* ⊢ e_1 «*Add*» e_2 :: *Integer*
apply (*rule WTBinOp*)
apply *assumption*+
apply *simp*
done

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*′)
by(*auto simp add*: *LeastFieldDecl-def FieldDecls*′*-def leq-path-unfold*)

code-pred [*inductify*, *skip-proof*] *LeastFieldDecl* .

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 WTStaticCall
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 −
case *WT*
from *WT.prems* **show** *thesis*
proof(*cases* (*no-simp*) *rule*: *WT.cases*)
case *WTDynCast* **thus** *thesis*
by(*rule WT.WTDynCast-new*[*OF refl*, *unfolded WTDynCast-ex-def*, *simplified*])
next
case *WTStaticCast* **thus** *?thesis*
unfolding *subcls1-def rtrancl-def mem-Collect-eq prod.case*
by(*rule WT.WTStaticCast-new*[*OF refl*, *unfolded WTStaticCast-sub-def*])
next
case *WTBinOp* **thus** *?thesis*
by(*split bop.split-asm*)(*simp-all*, (*erule* (*4*) *WT.WTBinOp1*[*OF refl*] *WT.WTBinOp2*[*OF refl*])+)
qed(*assumption*|*erule* (*2*) *WT.that*[*OF refl*])+
next
case *WTs*
from *WTs.prems* **show** *thesis*
by(*cases* (*no-simp*) *rule*: *WTs.cases*)(*assumption*|*erule* (*2*) *WTs.that*[*OF refl*])+
qed

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'*)
by(*auto intro*: *casts-to.intros simp add*: *path-via-def appendPath-def*)

code-pred (*modes*: $i \Rightarrow i \Rightarrow i \Rightarrow o \Rightarrow bool$, $i \Rightarrow i \Rightarrow i \Rightarrow i \Rightarrow bool$)
casts-to
apply(*erule casts-to.cases*)
apply(*fastforce split*: *ty.splits*)
apply *simp*
apply(*fastforce simp add*: *appendPath-def path-via-def split*: *if-split-asm*)
done

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

Casts-to
.

lemma *card-eq-1-iff-ex1*: $x \in A \Longrightarrow card\ A = 1 \longleftrightarrow A = \{x\}$
apply(*rule iffI*)
apply(*rule equalityI*)
apply(*rule subsetI*)
apply(*subgoal-tac card* $\{x, xa\} \leq card\ A$)
apply(*auto intro*: *ccontr*)[*1*]
apply(*rule card-mono*)
apply *simp-all*
apply(*metis Suc-n-not-n card-infinite*)
done

lemma *FinalOverriderMethodDef-unfold* [*code-pred-def*]:
$P \vdash R$ *has overrider* $M = mthd$ *via* $Cs \longleftrightarrow$
OverriderMethodDefs' $P\ R\ M\ Cs\ mthd \wedge$
$(\forall Cs'\ mthd'.$ *OverriderMethodDefs'* $P\ R\ M\ Cs'\ mthd' \longrightarrow Cs = Cs' \wedge mthd = mthd')$
by(*auto simp add*: *FinalOverriderMethodDef-def OverriderMethodDefs'-def card-eq-1-iff-ex1 simp del*: *One-nat-def*)

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

code-pred
(*modes*: *i* => *i* => *i* => *i* => *o* => *o* => *bool*, *i* ⇒ *i* ⇒ *i* ⇒ *i* ⇒ *i* ⇒ *i* ⇒ *bool*)
[*inductify*]
SelectMethodDef
.

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* .

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*)))))
by *transfer*(*simp add*: *init-class-fieldmap-def*)

lift-definition *init-obj'* :: *prog* ⇒ *cname* ⇒ *subo'* ⇒ *bool* **is** *init-obj* .

lemma *init-obj'-intros* [*code-pred-intro*]:
Subobjs P C Cs $\Longrightarrow$ *init-obj' P C* (*Cs*, *init-class-fieldmap' P* (*last Cs*))
by(*transfer*)(*rule init-obj.intros*)

code-pred
(*modes*: *i* $\Rightarrow$ *i* $\Rightarrow$ *o* $\Rightarrow$ *bool as init-obj-pred*)
init-obj'
by *transfer*(*erule init-obj.cases*, *blast*)

lemma *init-obj-pred-conv*: *set-of-pred* (*init-obj-pred P C*) = *Collect* (*init-obj' P C*)
by(*auto elim*: *init-obj-predE intro*: *init-obj-predI*)

lift-definition *blank'* :: *prog* $\Rightarrow$ *cname* $\Rightarrow$ *obj'* **is** *blank* **.**

lemma *blank'-code* [*code*]:
blank' P C = (*C*, *set-of-pred* (*init-obj-pred P C*))
unfolding *init-obj-pred-conv* **by** *transfer*(*simp add*: *blank-def*)

type-synonym *heap'* = *addr* $\rightharpoonup$ *obj'*

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

lift-definition *new-Addr'* :: *heap'* $\Rightarrow$ *addr option* **is** *new-Addr* **.**

lift-definition *start-heap'* :: *prog* $\Rightarrow$ *heap'* **is** *start-heap* **.**

lemma *start-heap'-code* [*code*]:
start-heap' P = *Map.empty* (*addr-of-sys-xcpt NullPointer* $\mapsto$ *blank' P NullPointer*)
(*addr-of-sys-xcpt ClassCast* $\mapsto$ *blank' P ClassCast*)
(*addr-of-sys-xcpt OutOfMemory* $\mapsto$ *blank' P OutOfMemory*)
by *transfer*(*simp add*: *start-heap-def*)

type-synonym
state' = *heap'* $\times$ *locals*

lift-definition *hp'* :: *state'* $\Rightarrow$ *heap'* **is** *hp* **.**

lemma *hp'-code* [*code*]: *hp'* = *fst*
by *transfer simp*

lift-definition *lcl'* :: *state'* $\Rightarrow$ *locals* **is** *lcl* **.**

lemma *lcl-code* [*code*]: *lcl'* = *snd*

by *transfer simp*

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

lemma *New'*:
⟦ *new-Addr' h* = *Some a*; *h'* = *h*(*a*↦(*blank' P C*)) ⟧
⟹ *P*,*E* ⊢ ⟨*new C*,(*h*,*l*)⟩ ⇒' ⟨*ref* (*a*,[*C*]),(*h'*,*l*)⟩
by *transfer*(*unfold blank-def*, *rule New*)

lemma *NewFail'*:
new-Addr' h = *None* ⟹
P,*E* ⊢ ⟨*new C*, (*h*,*l*)⟩ ⇒' ⟨*THROW OutOfMemory*,(*h*,*l*)⟩
by *transfer*(*rule NewFail*)

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*$_1$⟩
by *transfer*(*rule StaticUpCast*)

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*$_1$⟩
apply *transfer*
apply (*rule StaticDownCast*)
apply (*simp add*: *app-eq*)
done

lemma *StaticCastNull'*:
P,*E* ⊢ ⟨*e*,*s*$_0$⟩ ⇒' ⟨*null*,*s*$_1$⟩ ⟹
P,*E* ⊢ ⟨(| *C* |)*e*,*s*$_0$⟩ ⇒' ⟨*null*,*s*$_1$⟩
by *transfer*(*rule StaticCastNull*)

lemma *StaticCastFail'-new*:
⟦ *P*,*E* ⊢ ⟨*e*,*s*$_0$⟩⇒' ⟨*ref* (*a*,*Cs*),*s*$_1$⟩; ¬ (*subcls1p P*) ^** (*last Cs*) *C*; *C* ∉ *set Cs*⟧
⟹ *P*,*E* ⊢ ⟨(| *C* |)*e*,*s*$_0$⟩ ⇒' ⟨*THROW ClassCast*,*s*$_1$⟩
apply *transfer*
by (*fastforce intro*:*StaticCastFail simp add*: *rtrancl-def subcls1-def*)

lemma *StaticCastThrow'*:
P,*E* ⊢ ⟨*e*,*s*$_0$⟩ ⇒' ⟨*throw e'*,*s*$_1$⟩ ⟹
P,*E* ⊢ ⟨(| *C* |)*e*,*s*$_0$⟩ ⇒' ⟨*throw e'*,*s*$_1$⟩

by *transfer*(*rule StaticCastThrow*)

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*$,s_0\rangle \Rightarrow' \langle ref(a,Ds),s_1\rangle$
by *transfer*(*rule StaticUpDynCast*)

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*$,s_0\rangle \Rightarrow' \langle ref(a,Cs@[C]),s_1\rangle$
apply *transfer*
apply (*rule StaticDownDynCast*)
apply (*simp add*: *app-eq*)
done

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* $(a,Cs'),(h,l)\rangle$
by *transfer*(*rule DynCast*)

lemma *DynCastNull'*:
$P,E \vdash \langle e,s_0\rangle \Rightarrow' \langle null,s_1\rangle \Longrightarrow$
$P,E \vdash \langle$*Cast C e*$,s_0\rangle \Rightarrow' \langle null,s_1\rangle$
by *transfer*(*rule DynCastNull*)

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*$,s_0\rangle \Rightarrow' \langle null,(h,l)\rangle$
by *transfer*(*rule DynCastFail*)

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'*$,s_1\rangle$
by *transfer*(*rule DynCastThrow*)

lemma *Val'*:
$P,E \vdash \langle$*Val v*$,s\rangle \Rightarrow' \langle$*Val v*$,s\rangle$
by *transfer*(*rule Val*)

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* $v_2,s_2\rangle$;
$binop(bop,v_1,v_2) = Some\ v$ ⟧
$\Longrightarrow P,E \vdash \langle e_1 \ll bop \gg e_2,s_0\rangle \Rightarrow' \langle$*Val* $v,s_2\rangle$
by *transfer*(*rule BinOp*)

lemma *BinOpThrow1′*:
$P,E \vdash \langle e_1,s_0\rangle \Rightarrow' \langle throw\ e,s_1\rangle \Longrightarrow$
$P,E \vdash \langle e_1 \ll bop \gg e_2,\ s_0\rangle \Rightarrow' \langle throw\ e,s_1\rangle$
by *transfer*(*rule BinOpThrow1*)

lemma *BinOpThrow2′*:
$[\![P,E \vdash \langle e_1,s_0\rangle \Rightarrow' \langle Val\ v_1,s_1\rangle;\ P,E \vdash \langle e_2,s_1\rangle \Rightarrow' \langle throw\ e,s_2\rangle\ ]\!]$
$\Longrightarrow P,E \vdash \langle e_1 \ll bop \gg e_2,s_0\rangle \Rightarrow' \langle throw\ e,s_2\rangle$
by *transfer*(*rule BinOpThrow2*)

lemma *Var′*:
$l\ V = Some\ v \Longrightarrow$
$P,E \vdash \langle Var\ V,(h,l)\rangle \Rightarrow' \langle Val\ v,(h,l)\rangle$
by *transfer*(*rule Var*)

lemma *LAss′*:
$[\![P,E \vdash \langle e,s_0\rangle \Rightarrow' \langle Val\ v,(h,l)\rangle;\ E\ V = Some\ T;$
$P \vdash T\ casts\ v\ to\ v';\ l' = l(V \mapsto v')\ ]\!]$
$\Longrightarrow P,E \vdash \langle V:=e,s_0\rangle \Rightarrow' \langle Val\ v',(h,l')\rangle$
by (*transfer*) (*erule* (*3*) *LAss*)

lemma *LAssThrow′*:
$P,E \vdash \langle e,s_0\rangle \Rightarrow' \langle throw\ e',s_1\rangle \Longrightarrow$
$P,E \vdash \langle V:=e,s_0\rangle \Rightarrow' \langle throw\ e',s_1\rangle$
by *transfer*(*rule LAssThrow*)

lemma *FAcc′-new*:
$[\![P,E \vdash \langle e,s_0\rangle \Rightarrow' \langle ref\ (a,Cs'),(h,l)\rangle;\ h\ a = Some(D,S);$
$Ds = Cs'@_pCs$; *Predicate-Compile.contains* (*Set-project S Ds*) *fs*; *Mapping.lookup* *fs F = Some v* $]\!]$
$\Longrightarrow P,E \vdash \langle e{\cdot}F\{Cs\},s_0\rangle \Rightarrow' \langle Val\ v,(h,l)\rangle$
unfolding *Set-project-def mem-Collect-eq Predicate-Compile.contains-def*
by *transfer*(*rule FAcc*)

lemma *FAccNull′*:
$P,E \vdash \langle e,s_0\rangle \Rightarrow' \langle null,s_1\rangle \Longrightarrow$
$P,E \vdash \langle e{\cdot}F\{Cs\},s_0\rangle \Rightarrow' \langle THROW\ NullPointer,s_1\rangle$
by *transfer*(*rule FAccNull*)

lemma *FAccThrow′*:
$P,E \vdash \langle e,s_0\rangle \Rightarrow' \langle throw\ e',s_1\rangle \Longrightarrow$
$P,E \vdash \langle e{\cdot}F\{Cs\},s_0\rangle \Rightarrow' \langle throw\ e',s_1\rangle$
by *transfer*(*rule FAccThrow*)

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*;

$S' = S - \{(Ds,fs)\} \cup \{(Ds,fs')\};\ h_2' = h_2(a \mapsto (D,S'))]\!]$
$\Longrightarrow P,E \vdash \langle e_1 \cdot F\{Cs\} := e_2, s_0\rangle \Rightarrow' \langle Val\ v', (h_2', l_2)\rangle$
unfolding *Predicate-Compile.contains-def Set-project-def mem-Collect-eq*
by *transfer*(*rule FAss*)

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

lemma *FAssThrow1'*:
$P,E \vdash \langle e_1, s_0\rangle \Rightarrow' \langle throw\ e', s_1\rangle \Longrightarrow$
$P,E \vdash \langle e_1 \cdot F\{Cs\} := e_2, s_0\rangle \Rightarrow' \langle throw\ e', s_1\rangle$
by *transfer*(*rule FAssThrow1*)

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

lemma *CallObjThrow'*:
$P,E \vdash \langle e, s_0\rangle \Rightarrow' \langle throw\ e', s_1\rangle \Longrightarrow$
$P,E \vdash \langle Call\ e\ Copt\ M\ es, s_0\rangle \Rightarrow' \langle throw\ e', s_1\rangle$
by *transfer*(*rule CallObjThrow*)

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-val2 evs vs $(throw\ ex\ \#\ es')\ ]\!]$
$\Longrightarrow P,E \vdash \langle Call\ e\ Copt\ M\ es, s0\rangle \Rightarrow' \langle throw\ ex, s2\rangle$
apply *transfer*
apply(*rule eval-evals.CallParamsThrow*, *assumption*+)
apply(*simp add*: *map-val2-conv*[*symmetric*])
done

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-val evs vs;
$h_2\ a = Some(C,S);\ P \vdash last\ Cs\ has\ least\ M = (Ts',T',pns',body')\ via\ Ds;$
$P \vdash (C, Cs@_p Ds)\ selects\ M = (Ts,T,pns,body)\ via\ Cs';\ length\ vs = length\ pns;$
$P \vdash Ts\ Casts\ vs\ to\ vs';\ l_2' = [this \mapsto Ref\ (a,Cs'),\ pns[\mapsto]vs'];$
$new\text{-}body = (case\ T'\ of\ Class\ D \Rightarrow (\!|D|\!)body\ \ |\ -\ \Rightarrow body);$
$P,E(this \mapsto Class(last\ Cs'),\ pns[\mapsto]Ts) \vdash \langle new\text{-}body, (h_2,l_2')\rangle \Rightarrow' \langle e', (h_3,l_3)\rangle\ ]\!]$
$\Longrightarrow P,E \vdash \langle e \cdot M(ps), s_0\rangle \Rightarrow' \langle e', (h_3,l_2)\rangle$
apply *transfer*
apply(*rule Call*)
apply *assumption*+
apply(*simp add*: *map-val-conv*[*symmetric*])
apply *assumption*+

done

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-val evs vs;
$P \vdash Path\ (last\ Cs)\ to\ C\ unique$; $P \vdash Path\ (last\ Cs)\ to\ C\ via\ Cs''$;
$P \vdash C\ has\ least\ M = (Ts,T,pns,body)\ via\ Cs'$; $Ds = (Cs@_pCs'')@_pCs'$;
$length\ vs = length\ pns$; $P \vdash Ts\ Casts\ vs\ to\ vs'$;
$l_2' = [this\mapsto Ref\ (a,Ds),\ pns[\mapsto]vs']$;
$P,E(this\mapsto Class(last\ Ds),\ pns[\mapsto]Ts) \vdash \langle body,(h_2,l_2')\rangle \Rightarrow' \langle e',(h_3,l_3)\rangle$ ⟧
⟹ $P,E \vdash \langle e\cdot(C::)M(ps),s_0\rangle \Rightarrow' \langle e',(h_3,l_2)\rangle$
apply *transfer*
apply(*rule StaticCall*)
apply(*assumption*)+
apply(*simp add*: *map-val-conv*[*symmetric*])
apply *assumption*+
done

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* ⟧
⟹ $P,E \vdash \langle Call\ e\ Copt\ M\ es,s_0\rangle \Rightarrow' \langle THROW\ NullPointer,s_2\rangle$
apply *transfer*
apply(*rule CallNull*, *assumption*+)
apply(*simp add*: *map-val-conv*[*symmetric*])
done

lemma *Block′*:
⟦$P,E(V \mapsto T) \vdash \langle e_0,(h_0,l_0(V:=None))\rangle \Rightarrow' \langle e_1,(h_1,l_1)\rangle$ ⟧ ⟹
$P,E \vdash \langle\{V:T;\ e_0\},(h_0,l_0)\rangle \Rightarrow' \langle e_1,(h_1,l_1(V:=l_0\ V))\rangle$
by *transfer*(*rule Block*)

lemma *Seq′*:
⟦ $P,E \vdash \langle e_0,s_0\rangle \Rightarrow' \langle Val\ v,s_1\rangle$; $P,E \vdash \langle e_1,s_1\rangle \Rightarrow' \langle e_2,s_2\rangle$ ⟧
⟹ $P,E \vdash \langle e_0;;e_1,s_0\rangle \Rightarrow' \langle e_2,s_2\rangle$
by *transfer*(*rule Seq*)

lemma *SeqThrow′*:
$P,E \vdash \langle e_0,s_0\rangle \Rightarrow' \langle throw\ e,s_1\rangle$ ⟹
$P,E \vdash \langle e_0;;e_1,s_0\rangle \Rightarrow' \langle throw\ e,s_1\rangle$
by *transfer*(*rule SeqThrow*)

lemma *CondT′*:
⟦ $P,E \vdash \langle e,s_0\rangle \Rightarrow' \langle true,s_1\rangle$; $P,E \vdash \langle e_1,s_1\rangle \Rightarrow' \langle e',s_2\rangle$ ⟧
⟹ $P,E \vdash \langle if\ (e)\ e_1\ else\ e_2,s_0\rangle \Rightarrow' \langle e',s_2\rangle$
by *transfer*(*rule CondT*)

lemma *CondF′*:
⟦ $P,E \vdash \langle e,s_0\rangle \Rightarrow' \langle false,s_1\rangle$; $P,E \vdash \langle e_2,s_1\rangle \Rightarrow' \langle e',s_2\rangle$ ⟧
⟹ $P,E \vdash \langle if\ (e)\ e_1\ else\ e_2,s_0\rangle \Rightarrow' \langle e',s_2\rangle$

by *transfer*(*rule CondF*)

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

lemma *WhileF'*:
$P,E \vdash \langle e,s_0\rangle \Rightarrow' \langle false,s_1\rangle \Longrightarrow$
$P,E \vdash \langle while\ (e)\ c,s_0\rangle \Rightarrow' \langle unit,s_1\rangle$
by *transfer*(*rule WhileF*)

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

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

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

lemma *Throw'*:
$P,E \vdash \langle e,s_0\rangle \Rightarrow' \langle ref\ r,s_1\rangle \Longrightarrow$
$P,E \vdash \langle throw\ e,s_0\rangle \Rightarrow' \langle Throw\ r,s_1\rangle$
by *transfer*(*rule eval-evals.Throw*)

lemma *ThrowNull'*:
$P,E \vdash \langle e,s_0\rangle \Rightarrow' \langle null,s_1\rangle \Longrightarrow$
$P,E \vdash \langle throw\ e,s_0\rangle \Rightarrow' \langle THROW\ NullPointer,s_1\rangle$
by *transfer*(*rule ThrowNull*)

lemma *ThrowThrow'*:
$P,E \vdash \langle e,s_0\rangle \Rightarrow' \langle throw\ e',s_1\rangle \Longrightarrow$
$P,E \vdash \langle throw\ e,s_0\rangle \Rightarrow' \langle throw\ e',s_1\rangle$
by *transfer*(*rule ThrowThrow*)

lemma *Nil'*:
$P,E \vdash \langle[],s\rangle\ [\Rightarrow']\ \langle[],s\rangle$
by *transfer*(*rule eval-evals.Nil*)

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

$\Longrightarrow P,E \vdash \langle e\#es,s_0\rangle\ [\Rightarrow']\ \langle Val\ v\ \#\ es',s_2\rangle$
by *transfer*(*rule eval-evals.Cons*)

lemma *ConsThrow'*:
$P,E \vdash \langle e,\ s_0\rangle \Rightarrow' \langle throw\ e',\ s_1\rangle \Longrightarrow$
$P,E \vdash \langle e\#es,\ s_0\rangle\ [\Rightarrow']\ \langle throw\ e'\ \#\ es,\ s_1\rangle$
by *transfer*(*rule ConsThrow*)

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 $\bigwedge E\ e_1\ s_0\ v_1\ s_1\ e_2\ v_2\ s_2\ bop\ va.\ x = E \Longrightarrow y = e_1 \ll bop \gg e_2 \Longrightarrow z = s_0 \Longrightarrow$
$u = Val\ va \Longrightarrow v = s_2 \Longrightarrow P,E \vdash \langle e_1,s_0\rangle \Rightarrow' \langle Val\ v_1,s_1\rangle \Longrightarrow$
$P,E \vdash \langle e_2,s_1\rangle \Rightarrow' \langle Val\ v_2,s_2\rangle \Longrightarrow binop\ (bop,\ v_1,\ v_2) = \lfloor va\rfloor \Longrightarrow thesis$
and $\bigwedge E\ e_1\ s_0\ e\ s_1\ bop\ e_2.\ x = E \Longrightarrow y = e_1 \ll bop \gg e_2 \Longrightarrow z = s_0 \Longrightarrow u = throw\ e \Longrightarrow v = s_1\ \Longrightarrow$
$P,E \vdash \langle e_1,s_0\rangle \Rightarrow' \langle throw\ e,s_1\rangle \Longrightarrow thesis$
and $\bigwedge E\ e_1\ s_0\ v_1\ s_1\ e_2\ e\ s_2\ bop.\ x = E \Longrightarrow y = e_1 \ll bop \gg e_2 \Longrightarrow z = s_0 \Longrightarrow u = throw\ e \Longrightarrow$
$v = s_2 \Longrightarrow P,E \vdash \langle e_1,s_0\rangle \Rightarrow' \langle Val\ v_1,s_1\rangle \Longrightarrow P,E \vdash \langle e_2,s_1\rangle \Rightarrow' \langle throw\ e,s_2\rangle \Longrightarrow thesis$
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 $\bigwedge E\ e\ s_0\ e'\ s_1\ V.\ x = E \Longrightarrow y = V{:=}e \Longrightarrow z = s_0 \Longrightarrow u = throw\ e' \Longrightarrow v = s_1 \Longrightarrow$

$P,E \vdash \langle e,s_0\rangle \Rightarrow' \langle \mathit{throw}\ e',s_1\rangle \Longrightarrow \mathit{thesis}$
and $\bigwedge E\ e\ s_0\ a\ Cs'\ h\ l\ D\ S\ Ds\ Cs\ fs\ F\ va.\ x = E \Longrightarrow y = e{\cdot}F\{Cs\} \Longrightarrow z = s_0 \Longrightarrow$
$u = \mathit{Val}\ va \Longrightarrow v = (h,\ l) \Longrightarrow P,E \vdash \langle e,s_0\rangle \Rightarrow' \langle \mathit{ref}\ (a,\ Cs'),(h,\ l)\rangle \Longrightarrow$
$h\ a = \lfloor (D,\ S)\rfloor \Longrightarrow Ds = Cs'\ @_p\ Cs \Longrightarrow (Ds,\ fs) \in S \Longrightarrow \mathit{Mapping.lookup}\ fs\ F = \lfloor va\rfloor \Longrightarrow \mathit{thesis}$
and $\bigwedge E\ e\ s_0\ s_1\ F\ Cs.\ x = E \Longrightarrow y = e{\cdot}F\{Cs\} \Longrightarrow z = s_0 \Longrightarrow$
$u = \mathit{Throw}\ (\mathit{addr\text{-}of\text{-}sys\text{-}xcpt}\ \mathit{NullPointer},\ [\mathit{NullPointer}]) \Longrightarrow$
$v = s_1 \Longrightarrow P,E \vdash \langle e,s_0\rangle \Rightarrow' \langle \mathit{null},s_1\rangle \Longrightarrow \mathit{thesis}$
and $\bigwedge E\ e\ s_0\ e'\ s_1\ F\ Cs.\ x = E \Longrightarrow y = e{\cdot}F\{Cs\} \Longrightarrow z = s_0 \Longrightarrow u = \mathit{throw}\ e' \Longrightarrow v = s_1 \Longrightarrow$
$P,E \vdash \langle e,s_0\rangle \Rightarrow' \langle \mathit{throw}\ e',s_1\rangle \Longrightarrow \mathit{thesis}$
and $\bigwedge E\ e_1\ s_0\ a\ Cs'\ s_1\ e_2\ va\ h_2\ l_2\ D\ S\ F\ T\ Cs\ v'\ Ds\ fs\ fs'\ S'\ h_2'.$
$x = E \Longrightarrow y = e_1{\cdot}F\{Cs\} := e_2 \Longrightarrow z = s_0 \Longrightarrow u = \mathit{Val}\ v' \Longrightarrow v = (h_2',\ l_2) \Longrightarrow$
$P,E \vdash \langle e_1,s_0\rangle \Rightarrow' \langle \mathit{ref}\ (a,\ Cs'),s_1\rangle \Longrightarrow P,E \vdash \langle e_2,s_1\rangle \Rightarrow' \langle \mathit{Val}\ va,(h_2,\ l_2)\rangle \Longrightarrow$
$h_2\ a = \lfloor (D,\ S)\rfloor \Longrightarrow P \vdash \mathit{last}\ Cs'\ \mathit{has\ least}\ F{:}T\ \mathit{via}\ Cs \Longrightarrow$
$P \vdash T\ \mathit{casts}\ va\ \mathit{to}\ v' \Longrightarrow Ds = Cs'\ @_p\ Cs \Longrightarrow (Ds,\ fs) \in S \Longrightarrow fs' = \mathit{Mapping.update}\ F\ v'\ fs \Longrightarrow$
$S' = S - \{(Ds,\ fs)\} \cup \{(Ds,\ fs')\} \Longrightarrow h_2' = h_2(a \mapsto (D,\ S')) \Longrightarrow \mathit{thesis}$
and $\bigwedge E\ e_1\ s_0\ s_1\ e_2\ va\ s_2\ F\ Cs.\ x = E \Longrightarrow y = e_1{\cdot}F\{Cs\} := e_2 \Longrightarrow z = s_0 \Longrightarrow$
$u = \mathit{Throw}\ (\mathit{addr\text{-}of\text{-}sys\text{-}xcpt}\ \mathit{NullPointer},\ [\mathit{NullPointer}]) \Longrightarrow$
$v = s_2 \Longrightarrow P,E \vdash \langle e_1,s_0\rangle \Rightarrow' \langle \mathit{null},s_1\rangle \Longrightarrow P,E \vdash \langle e_2,s_1\rangle \Rightarrow' \langle \mathit{Val}\ va,s_2\rangle \Longrightarrow \mathit{thesis}$
and $\bigwedge E\ e_1\ s_0\ e'\ s_1\ F\ Cs\ e_2.\ x = E \Longrightarrow y = e_1{\cdot}F\{Cs\} := e_2 \Longrightarrow$
$z = s_0 \Longrightarrow u = \mathit{throw}\ e' \Longrightarrow v = s_1 \Longrightarrow P,E \vdash \langle e_1,s_0\rangle \Rightarrow' \langle \mathit{throw}\ e',s_1\rangle \Longrightarrow \mathit{thesis}$
and $\bigwedge E\ e_1\ s_0\ va\ s_1\ e_2\ e'\ s_2\ F\ Cs.\ x = E \Longrightarrow y = e_1{\cdot}F\{Cs\} := e_2 \Longrightarrow z = s_0 \Longrightarrow$
$u = \mathit{throw}\ e' \Longrightarrow v = s_2 \Longrightarrow P,E \vdash \langle e_1,s_0\rangle \Rightarrow' \langle \mathit{Val}\ va,s_1\rangle \Longrightarrow P,E \vdash \langle e_2,s_1\rangle \Rightarrow' \langle \mathit{throw}\ e',s_2\rangle \Longrightarrow$
thesis
and $\bigwedge E\ e\ s_0\ e'\ s_1\ \mathit{Copt}\ M\ es.\ x = E \Longrightarrow y = \mathit{Call}\ e\ \mathit{Copt}\ M\ es \Longrightarrow$
$z = s_0 \Longrightarrow u = \mathit{throw}\ e' \Longrightarrow v = s_1 \Longrightarrow P,E \vdash \langle e,s_0\rangle \Rightarrow' \langle \mathit{throw}\ e',s_1\rangle \Longrightarrow \mathit{thesis}$
and $\bigwedge E\ e\ s_0\ va\ s_1\ es\ vs\ ex\ es'\ s_2\ \mathit{Copt}\ M.\ x = E \Longrightarrow y = \mathit{Call}\ e\ \mathit{Copt}\ M\ es \Longrightarrow$
$z = s_0 \Longrightarrow u = \mathit{throw}\ ex \Longrightarrow v = s_2 \Longrightarrow P,E \vdash \langle e,s_0\rangle \Rightarrow' \langle \mathit{Val}\ va,s_1\rangle \Longrightarrow$
$P,E \vdash \langle es,s_1\rangle\ [\Rightarrow']\ \langle \mathit{map}\ \mathit{Val}\ vs\ @\ \mathit{throw}\ ex\ \#\ es',s_2\rangle \Longrightarrow \mathit{thesis}$
and $\bigwedge E\ e\ s_0\ a\ Cs\ s_1\ ps\ vs\ h_2\ l_2\ C\ S\ M\ Ts'\ T'\ pns'\ body'\ Ds\ Ts\ T\ pns\ body\ Cs'\ vs'\ l_2'\ \mathit{new\text{-}body}\ e'$
$h_3\ l_3.\ x = E \Longrightarrow y = \mathit{Call}\ e\ \mathit{None}\ M\ ps \Longrightarrow z = s_0 \Longrightarrow u = e' \Longrightarrow v = (h_3,\ l_2) \Longrightarrow$
$P,E \vdash \langle e,s_0\rangle \Rightarrow' \langle \mathit{ref}\ (a,\ Cs),s_1\rangle \Longrightarrow P,E \vdash \langle ps,s_1\rangle\ [\Rightarrow']\ \langle \mathit{map}\ \mathit{Val}\ vs,(h_2,\ l_2)\rangle \Longrightarrow$
$h_2\ a = \lfloor (C,\ S)\rfloor \Longrightarrow P \vdash \mathit{last}\ Cs\ \mathit{has\ least}\ M = (Ts',\ T',\ pns',\ body')\ \mathit{via}\ Ds \Longrightarrow$
$P \vdash (C,Cs\ @_p\ Ds)\ \mathit{selects}\ M = (Ts,\ T,\ pns,\ body)\ \mathit{via}\ Cs' \Longrightarrow \mathit{length}\ vs =$

length pns $\Longrightarrow$
$P \vdash$ *Ts Casts vs to vs'* $\Longrightarrow$ $l_2' = [$*this* $\mapsto$ *Ref* $(a,$ *Cs'*$)$, *pns* $[\mapsto]$ *vs'*$] \Longrightarrow$
new-body $=$ (*case T' of Class D* $\Rightarrow$ $(\!|D|\!)$*body* $|$ - $\Rightarrow$ *body*) $\Longrightarrow$
P,*E*(*this* $\mapsto$ *Class* (*last Cs'*), *pns* $[\mapsto]$ *Ts*) $\vdash \langle$*new-body*,$(h_2, l_2')\rangle \Rightarrow' \langle e', (h_3, l_3)\rangle \Longrightarrow$
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-of-sys-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 $\bigwedge E\ e\ s_0\ e'\ s_1\ c.\ x = E \Longrightarrow y = \mathit{while}\ (e)\ c \Longrightarrow z = s_0 \Longrightarrow u = \mathit{throw}\ e' \Longrightarrow v = s_1 \Longrightarrow$
$P,E \vdash \langle e,s_0\rangle \Rightarrow' \langle \mathit{throw}\ e',s_1\rangle \Longrightarrow \mathit{thesis}$
and $\bigwedge E\ e\ s_0\ s_1\ c\ e'\ s_2.\ x = E \Longrightarrow y = \mathit{while}\ (e)\ c \Longrightarrow z = s_0 \Longrightarrow u = \mathit{throw}\ e' \Longrightarrow$
$v = s_2 \Longrightarrow P,E \vdash \langle e,s_0\rangle \Rightarrow' \langle \mathit{true},s_1\rangle \Longrightarrow P,E \vdash \langle c,s_1\rangle \Rightarrow' \langle \mathit{throw}\ e',s_2\rangle \Longrightarrow \mathit{thesis}$
and $\bigwedge E\ e\ s_0\ r\ s_1.\ x = E \Longrightarrow y = \mathit{throw}\ e \Longrightarrow$
$z = s_0 \Longrightarrow u = \mathit{Throw}\ r \Longrightarrow v = s_1 \Longrightarrow P,E \vdash \langle e,s_0\rangle \Rightarrow' \langle \mathit{ref}\ r,s_1\rangle \Longrightarrow \mathit{thesis}$
and $\bigwedge E\ e\ s_0\ s_1.\ x = E \Longrightarrow y = \mathit{throw}\ e \Longrightarrow z = s_0 \Longrightarrow$
$u = \mathit{Throw}\ (\mathit{addr\text{-}of\text{-}sys\text{-}xcpt}\ \mathit{NullPointer},\ [\mathit{NullPointer}]) \Longrightarrow$
$v = s_1 \Longrightarrow P,E \vdash \langle e,s_0\rangle \Rightarrow' \langle \mathit{null},s_1\rangle \Longrightarrow \mathit{thesis}$
and $\bigwedge E\ e\ s_0\ e'\ s_1.\ x = E \Longrightarrow y = \mathit{throw}\ e \Longrightarrow$
$z = s_0 \Longrightarrow u = \mathit{throw}\ e' \Longrightarrow v = s_1 \Longrightarrow P,E \vdash \langle e,s_0\rangle \Rightarrow' \langle \mathit{throw}\ e',s_1\rangle \Longrightarrow \mathit{thesis}$
shows *thesis*
using *assms*
by(*transfer*)(*erule eval.cases*, *unfold blank-def*, *assumption*+)

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 −
case *eval′*
from *eval′.prems* **show** *thesis*
proof(*cases* (*no-simp*) *rule*: *eval′-cases*)
case (*StaticDownCast E C e* s_0 *a Cs Cs′* s_1)
moreover
have *app a* [*Cs*] (*a* @ [*Cs*]) *app* (*a* @ [*Cs*]) *Cs′* (*a* @ [*Cs*] @ *Cs′*)
by(*simp-all add*: *app-eq*)
ultimately show *?thesis* **by**(*rule eval′.StaticDownCast′*[*OF refl*])
next
case *StaticCastFail* **thus** *?thesis*
unfolding *rtrancl-def subcls1-def mem-Collect-eq prod.case*
by(*rule eval′.StaticCastFail′*[*OF refl*])
next
case (*StaticDownDynCast E e* s_0 *a Cs C Cs′* s_1)
moreover have *app Cs* [*C*] (*Cs* @ [*C*]) *app* (*Cs* @ [*C*]) *Cs′* (*Cs* @ [*C*] @ *Cs′*)
by(*simp-all add*: *app-eq*)
ultimately show *thesis* **by**(*rule eval′.StaticDownDynCast′*[*OF refl*])
next
case *DynCast* **thus** *?thesis* **by**(*rule eval′.DynCast′*[*OF refl*])
next
case *DynCastFail* **thus** *?thesis* **by**(*rule eval′.DynCastFail′*[*OF refl*])
next
case *BinOpThrow2* **thus** *?thesis* **by**(*rule eval′.BinOpThrow2′*[*OF refl*])
next
case *FAcc* **thus** *?thesis*
by(*rule eval′.FAcc′*[*OF refl*, *unfolded Predicate-Compile.contains-def Set-project-def mem-Collect-eq*])
next
case *FAss* **thus** *?thesis*
by(*rule eval′.FAss′*[*OF refl*, *unfolded Predicate-Compile.contains-def Set-project-def mem-Collect-eq*])
next
case *FAssThrow2* **thus** *?thesis* **by**(*rule eval′.FAssThrow2′*[*OF refl*])
next
case (*CallParamsThrow E e* s_0 *v* s_1 *es vs ex es′* s_2 *Copt M*)

```
    moreover have map-val2 (map Val vs @ throw ex # es') vs (throw ex # es')
      by(simp add: map-val2-conv[symmetric])
    ultimately show ?thesis by(rule eval'.CallParamsThrow'[OF refl])
  next
    case (Call E e s0 a Cs s1 ps vs)
   moreover have map-val (map Val vs) vs by(simp add: map-val-conv[symmetric])
    ultimately show ?thesis by−(rule eval'.Call'[OF refl])
  next
    case (StaticCall E e s0 a Cs s1 ps vs)
   moreover have map-val (map Val vs) vs by(simp add: map-val-conv[symmetric])
    ultimately show ?thesis by−(rule eval'.StaticCall'[OF refl])
  next
    case (CallNull E e s0 s1 es vs)
   moreover have map-val (map Val vs) vs by(simp add: map-val-conv[symmetric])
    ultimately show ?thesis by−(rule eval'.CallNull'[OF refl])
  next
    case SeqThrow thus ?thesis by(rule eval'.SeqThrow'[OF refl])
  next
    case CondF thus ?thesis by(rule eval'.CondF'[OF refl])
  next
    case CondThrow thus ?thesis by(rule eval'.CondThrow'[OF refl])
  next
    case WhileCondThrow thus ?thesis by(rule eval'.WhileCondThrow'[OF refl])
  next
    case WhileBodyThrow thus ?thesis by(rule eval'.WhileBodyThrow'[OF refl])
  next
    case ThrowNull thus ?thesis by(rule eval'.ThrowNull'[OF refl])
  qed(assumption|erule (4) eval'.that[OF refl])+
next
  case evals'
  from evals'.prems evals'.that[OF refl]
  show thesis by transfer(erule evals.cases)
qed
```

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 · ''x'' {[''Left'',''Top'']} ≪Add≫ Val (Intg 5))])

definition
  classRight :: cdecl where
  classRight = (''Right'', [Shares ''Right2''],[],
   [(''f'', [Class ''Top'', Integer], Integer,[''V'',''W''],Var this · ''x'' {[''Right2'',''Top'']}
≪Add≫ Val (Intg 7)),(''g'',[],Class ''Left'',[],new ''Left'')])

definition
  classRight2 :: cdecl where
  classRight2 = (''Right2'', [Repeats ''Top''],[],
   [(''f'', [Class ''Top'', Integer], Integer,[''V'',''W''],Var this · ''x'' {[''Right2'',''Top'']}
≪Add≫ Val (Intg 9)),(''g'',[],Class ''Top'',[],new ''Top'')])
```

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*''↦*Class* ''*Right*''] ⊢
⟨''*V*'' := *new* ''*Bottom*'' ;; *Cast* ''*Left*'' (*Var* ''*V*''),(*Map.empty*,*Map.empty*)⟩
⇒′ ⟨*e'*, *s'*⟩}

values [*expected* {*Val*(*Ref*(*0*,[''*Right*''])}] — dynCastViaSh
{*fst* (*e'*, *s'*) | *e' s'*.
progOverrider,[''*V*''↦*Class* ''*Right2*''] ⊢
⟨''*V*'' := *new* ''*Right*'' ;; *Cast* ''*Right*'' (*Var* ''*V*''),(*Map.empty*,*Map.empty*)⟩
⇒′ ⟨*e'*, *s'*⟩}

values [*expected* {*Val* (*Intg 42*)}] — block
{*fst* (*e'*, *s'*) | *e' s'*.
progOverrider,[''*V*''↦*Integer*]
⊢ ⟨''*V*'' := *Val*(*Intg 42*) ;; {''*V*'':*Class* ''*Left*''; ''*V*'' := *new* ''*Bottom*''} ;; *Var* ''*V*'',
(*Map.empty*,*Map.empty*)⟩ ⇒′ ⟨*e'*, *s'*⟩}

values [*expected* {*Val* (*Intg 8*)}] — staticCall
{*fst* (*e'*, *s'*) | *e' s'*.
progOverrider,[''*V*''↦*Class* ''*Right*'',''*W*''↦*Class* ''*Bottom*'']
⊢ ⟨''*V*'' := *new* ''*Bottom*'' ;; ''*W*'' := *new* ''*Bottom*'' ;;
((*Cast* ''*Left*'' (*Var* ''*W*''))•''*x*''{[''*Left*'',''*Top*'']} := *Val*(*Intg 3*));;
(*Var* ''*W*''•(''*Left*''::)''*f*''([*Var* ''*V*'',*Val*(*Intg 2*)])),(*Map.empty*,*Map.empty*)⟩
⇒′ ⟨*e'*, *s'*⟩}

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*));;

$''W'' := new\ ''Left'' ;;\ Var\ ''V''\cdot''f''([Var\ ''W'', Val(Intg\ 42)]),$
$(Map.empty, Map.empty)\rangle \Rightarrow' \langle e',\ s'\rangle\}$

values $[expected\ \{Val(Ref(1, [''Left'', ''Top'']))\}]$ — callClass
$\{fst\ (e',\ s') \mid e'\ s'.$
$progOverrider, [''V'' \mapsto Class\ ''Right2'']$
$\vdash \langle ''V'' := new\ ''Right'' ;;\ Var\ ''V''\cdot''g''([]), (Map.empty, Map.empty)\rangle \Rightarrow' \langle e', s'\rangle\}$

values $[expected\ \{Val(Intg\ 42)\}]$ — fieldAss
$\{fst\ (e',\ s') \mid e'\ s'.$
$progOverrider, [''V'' \mapsto Class\ ''Right2'']$
$\vdash \langle ''V'' := new\ ''Right'' ;;$
$(Var\ ''V''\cdot''x''\{[''Right2'', ''Top'']\} := (Val(Intg\ 42))) ;;$
$(Var\ ''V''\cdot''x''\{[''Right2'', ''Top'']\}), (Map.empty, Map.empty)\rangle \Rightarrow' \langle e',\ s'\rangle\}$

typing rules

values $[expected\ \{Class\ ''Bottom''\}]$ — typeNew
$\{T.\ progOverrider, Map.empty \vdash new\ ''Bottom'' :: T\}$

values $[expected\ \{Class\ ''Left''\}]$ — typeDynCast
$\{T.\ progOverrider, Map.empty \vdash Cast\ ''Left''\ (new\ ''Bottom'') :: T\}$

values $[expected\ \{Class\ ''Left''\}]$ — typeStaticCast
$\{T.\ progOverrider, Map.empty \vdash (\!|''Left''|\!)\ (new\ ''Bottom'') :: T\}$

values $[expected\ \{Integer\}]$ — typeVal
$\{T.\ [], Map.empty \vdash Val(Intg\ 17) :: T\}$

values $[expected\ \{Integer\}]$ — typeVar
$\{T.\ [], [''V'' \mapsto Integer] \vdash Var\ ''V'' :: T\}$

values $[expected\ \{Boolean\}]$ — typeBinOp
$\{T.\ [], Map.empty \vdash (Val(Intg\ 5))\ \ll Eq\gg\ (Val(Intg\ 6)) :: T\}$

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*''↦*Class* ''*Right2*'']
⊢ ''*V*'' := *new* ''*Right*'' ;; *Var* ''*V*''•''*g*''([]) :: *T*}

values [*expected* {*Class* ''*Top*''}] — typeBlock
{*T*. *progOverrider*,*Map.empty* ⊢ {''*V*'':*Class* ''*Top*''; ''*V*'' := *new* ''*Left*''} :: *T*}

values [*expected* {*Integer*}] — typeCond
{*T*. [],*Map.empty* ⊢ *if* (*true*) *Val*(*Intg* 6) *else Val*(*Intg* 9) :: *T*}

values [*expected* {*Void*}] — typeWhile
{*T*. [],*Map.empty* ⊢ *while* (*false*) *Val*(*Intg* 17) :: *T*}

values [*expected* {*Void*}] — typeThrow
{*T*. *progOverrider*,*Map.empty* ⊢ *throw* (*new* ''*Bottom*'') :: *T*}

values [*expected* {*Integer*}] — typeBig
{*T*. *progOverrider*,[''*V*''↦*Class* ''*Right2*'',''*W*''↦*Class* ''*Left*'']
⊢ ''*V*'' := *new* ''*Right*'' ;; ''*W*'' := *new* ''*Left*'' ;;
(*Var* ''*V*''•''*f*''([*Var* ''*W*'', *Val*(*Intg* 7)])) ≪*Add*≫ (*Var* ''*W*''•''*f*''([*Var* ''*V*'',
Val(*Intg* 13)]))
:: *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.