JinjaDCI: a Java semantics with dynamic class initialization

Susannah Mansky

September 1, 2025

Abstract. This work is an extension of the Jinja semantics for Java and the JVM by Klein and Nipkow to include static fields and methods and dynamic class initialization. In Java, class initialization methods are run dynamically, called when classes are first used. Such calls are handled by the running of an initialization procedure, which interrupts execution and determines which initialization methods must be run before execution continues. This interrupting is modeled here in a couple of ways. In the Java semantics, evaluation is performed via expressions that are manipulated through evaluation until a final value is reached. In JinjaDCI, we have added two types of initialization expressions whose evaluations produce the steps of the initialization procedure. These expressions can occur during evaluation and store the calling expression away to continue being evaluated once the procedure is complete. In the JVM semantics, since programs are static sequences of instructions, the initialization procedure is run instead by the execution function. This function performs steps of the procedure rather than calling instructions when the initialization procedure has been called.

This extension includes the necessary updates to all major proofs from the original Jinja, including type safety and correctness of compilation from the Java semantics to the JVM semantics.

This work is partially described in [1].

Contents

Chapter 1

Jinja Source Language

1.1 Auxiliary Definitions

theory *Auxiliary* **imports** *Main* **begin**

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

notation *Some* (‹(⌊-⌋)›)

1.1.1 *distinct-fst*

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

lemma *distinct-fst-Nil* [*simp*]:
 distinct-fst []
⟨*proof*⟩
lemma *distinct-fst-Cons* [*simp*]:
 $distinct\text{-}fst\ ((k,x)\#kxs) = (distinct\text{-}fst\ kxs \wedge (\forall\, y.\ (k,y) \notin set\ kxs))$⟨*proof*⟩
lemma *distinct-fst-appendD*:
 $distinct\text{-}fst(kxs\ @\ kxs') \Longrightarrow distinct\text{-}fst\ kxs \wedge distinct\text{-}fst\ kxs'$⟨*proof*⟩
lemma *map-of-SomeI*:
 ⟦ *distinct-fst kxs*; $(k,x) \in set\ kxs$ ⟧ ⟹ *map-of kxs k* = *Some x*⟨*proof*⟩

1.1.2 Using *list-all2* for relations

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

Convenience lemmas

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

⟨*proof*⟩
lemma *rel-list-all2-Cons1*:
list-all2 (*fun-of S*) (*x*#*xs*) *ys* =
(∃ *z zs*. *ys* = *z*#*zs* ∧ (*x*,*z*) ∈ *S* ∧ *list-all2* (*fun-of S*) *xs zs*)
⟨*proof*⟩
lemma *rel-list-all2-Cons2*:
list-all2 (*fun-of S*) *xs* (*y*#*ys*) =
(∃ *z zs*. *xs* = *z*#*zs* ∧ (*z*,*y*) ∈ *S* ∧ *list-all2* (*fun-of S*) *zs ys*)
⟨*proof*⟩
lemma *rel-list-all2-refl*:
(⋀*x*. (*x*,*x*) ∈ *S*) ⟹ *list-all2* (*fun-of S*) *xs xs*
⟨*proof*⟩
lemma *rel-list-all2-antisym*:
⟦ (⋀*x y*. ⟦(*x*,*y*) ∈ *S*; (*y*,*x*) ∈ *T*⟧ ⟹ *x* = *y*);
list-all2 (*fun-of S*) *xs ys*; *list-all2* (*fun-of T*) *ys xs* ⟧ ⟹ *xs* = *ys*
⟨*proof*⟩
lemma *rel-list-all2-trans*:
⟦ ⋀*a b c*. ⟦(*a*,*b*) ∈ *R*; (*b*,*c*) ∈ *S*⟧ ⟹ (*a*,*c*) ∈ *T*;
list-all2 (*fun-of R*) *as bs*; *list-all2* (*fun-of S*) *bs cs*⟧
⟹ *list-all2* (*fun-of T*) *as cs*
⟨*proof*⟩
lemma *rel-list-all2-update-cong*:
⟦ *i*<*size xs*; *list-all2* (*fun-of S*) *xs ys*; (*x*,*y*) ∈ *S* ⟧
⟹ *list-all2* (*fun-of S*) (*xs*[*i*:=*x*]) (*ys*[*i*:=*y*])
⟨*proof*⟩
lemma *rel-list-all2-nthD*:
⟦ *list-all2* (*fun-of S*) *xs ys*; *p* < *size xs* ⟧ ⟹ (*xs*!*p*,*ys*!*p*) ∈ *S*
⟨*proof*⟩
lemma *rel-list-all2I*:
⟦ *length a* = *length b*; ⋀*n*. *n* < *length a* ⟹ (*a*!*n*,*b*!*n*) ∈ *S* ⟧ ⟹ *list-all2* (*fun-of S*) *a b*
⟨*proof*⟩

1.1.3 Auxiliary properties of *map-of* function

lemma *map-of-set-pcs-notin*: *C* ∉ (λ*t*. *snd* (*fst t*)) ' *set FDTs* ⟹ *map-of FDTs* (*F*, *C*) = *None*⟨*proof*⟩
lemma *map-of-insertmap-SomeD'*:
map-of fs F = *Some y* ⟹ *map-of* (*map* (λ(*F*, *y*). (*F*, *D*, *y*)) *fs*) *F* = *Some*(*D*,*y*)⟨*proof*⟩
lemma *map-of-reinsert-neq-None*:
Ca ≠ *D* ⟹ *map-of* (*map* (λ(*F*, *y*). ((*F*, *Ca*), *y*)) *fs*) (*F*, *D*) = *None*⟨*proof*⟩
lemma *map-of-remap-insertmap*:
map-of (*map* ((λ((*F*, *D*), *b*, *T*). (*F*, *D*, *b*, *T*)) ∘ (λ(*F*, *y*). ((*F*, *D*), *y*))) *fs*)
= *map-of* (*map* (λ(*F*, *y*). (*F*, *D*, *y*)) *fs*)⟨*proof*⟩

lemma *map-of-reinsert-SomeD*:
map-of (*map* (λ(*F*, *y*). ((*F*, *D*), *y*)) *fs*) (*F*, *D*) = *Some T* ⟹ *map-of fs F* = *Some T*⟨*proof*⟩
lemma *map-of-filtered-SomeD*:
map-of fs (*F*,*D*) = *Some* (*a*, *T*) ⟹ *Q* ((*F*,*D*),*a*,*T*) ⟹
map-of (*map* (λ((*F*,*D*), *b*, *T*). ((*F*,*D*), *P T*)) (*filter Q fs*))
(*F*,*D*) = *Some* (*P T*)⟨*proof*⟩

lemma *map-of-remove-filtered-SomeD*:
map-of fs (*F*,*C*) = *Some* (*a*, *T*) ⟹ *Q* ((*F*,*C*),*a*,*T*) ⟹
map-of (*map* (λ((*F*,*D*), *b*, *T*). (*F*, *P T*)) [((*F*, *D*), *b*, *T*)←*fs* . *Q* ((*F*, *D*), *b*, *T*) ∧ *D* = *C*])
F = *Some* (*P T*)⟨*proof*⟩

lemma *map-of-Some-None-split*:
assumes $t = map\ (\lambda(F,\ y).\ ((F,\ C),\ y))\ fs\ @\ t'\ map\text{-}of\ t'\ (F,\ C) = None\ map\text{-}of\ t\ (F,\ C) = Some\ y$
shows $map\text{-}of\ (map\ (\lambda((F,\ D),\ b,\ T).\ (F,\ D,\ b,\ T))\ t)\ F = Some\ (C,\ y)\langle proof\rangle$
end

1.2 Jinja 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 *Object* :: *cname*
where
  $Object \equiv ''Object''$

definition *this* :: *vname*
where
  $this \equiv ''this''$

definition *clinit* :: *string* **where** $clinit = ''<clinit>''$
definition *init* :: *string* **where** $init = ''<init>''$

definition *start-m* :: *string* **where** $start\text{-}m = ''<start>''$
definition *Start* :: *string* **where** $Start = ''<Start>''$

lemma *start-m-neq-clinit* [*simp*]: $start\text{-}m \neq clinit\ \langle proof\rangle$
lemma *Object-neq-Start* [*simp*]: $Object \neq Start\ \langle proof\rangle$
lemma *Start-neq-Object* [*simp*]: $Start \neq Object\ \langle proof\rangle$

datatype *staticb* = *Static* | *NonStatic*

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

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

lemma [*iff*]: $is\text{-}refT\ NT\langle proof\rangle$
lemma [*iff*]: $is\text{-}refT(Class\ C)\langle proof\rangle$
lemma *refTE*:
  $[\![is\text{-}refT\ T;\ T = NT \Longrightarrow P;\ \bigwedge C.\ T = Class\ C \Longrightarrow P\ ]\!] \Longrightarrow P\langle proof\rangle$
lemma *not-refTE*:
  $[\![\ \neg is\text{-}refT\ T;\ T = Void \vee T = Boolean \vee T = Integer \Longrightarrow P\ ]\!] \Longrightarrow P\langle proof\rangle$
end

1.3 Class Declarations and Programs

theory *Decl* **imports** *Type* **begin**

type-synonym
fdecl = *vname* × *staticb* × *ty* — field declaration
type-synonym
'm mdecl = *mname* × *staticb* × *ty list* × *ty* × *'m* — method = name, static flag, arg. types, return type, body
type-synonym
'm class = *cname* × *fdecl list* × *'m mdecl list* — class = superclass, fields, methods
type-synonym
'm cdecl = *cname* × *'m class* — class declaration
type-synonym
'm prog = *'m cdecl list* — program

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

lemma *class-cons*: ⟦ *C* ≠ *fst x* ⟧ ⟹ *class* (*x* # *P*) *C* = *class P C*
⟨*proof*⟩

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

lemma *finite-is-class*: *finite* {*C*. *is-class P C*}⟨*proof*⟩
definition *is-type* :: *'m prog* ⇒ *ty* ⇒ *bool*
where
is-type P T ≡
(*case T of Void* ⇒ *True* | *Boolean* ⇒ *True* | *Integer* ⇒ *True* | *NT* ⇒ *True*
| *Class C* ⇒ *is-class P C*)

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

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

lemma *class-exists-equiv*:
(∃ *x*. *fst x* = *cn* ∧ *x* ∈ *set P*) = (*class P cn* ≠ *None*)
⟨*proof*⟩

lemma *class-exists-equiv2*:
(∃ *x*. *fst x* = *cn* ∧ *x* ∈ *set* (*P1* @ *P2*)) = (*class P1 cn* ≠ *None* ∨ *class P2 cn* ≠ *None*)
⟨*proof*⟩

end

1.4 Relations between Jinja Types

theory *TypeRel* **imports**
HOL−Library.Transitive-Closure-Table
Decl
begin

1.4.1 The subclass relations

inductive-set
subcls1 :: *'m prog* ⇒ (*cname* × *cname*) *set*
and *subcls1'* :: *'m prog* ⇒ [*cname*, *cname*] ⇒ *bool* (‹- ⊢ - ≺¹ -› [*71*,*71*,*71*] *70*)
for *P* :: *'m prog*
where
P ⊢ *C* ≺¹ *D* ≡ (*C*,*D*) ∈ *subcls1 P*
| *subcls1I*: ⟦*class P C* = *Some* (*D*,*rest*); *C* ≠ *Object*⟧ ⟹ *P* ⊢ *C* ≺¹ *D*

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

lemma *subcls1D*: *P* ⊢ *C* ≺¹ *D* ⟹ *C* ≠ *Object* ∧ (∃*fs ms*. *class P C* = *Some* (*D*,*fs*,*ms*))⟨*proof*⟩
lemma [*iff*]: ¬ *P* ⊢ *Object* ≺¹ *C*⟨*proof*⟩
lemma [*iff*]: (*P* ⊢ *Object* ≼* *C*) = (*C* = *Object*)⟨*proof*⟩
lemma *subcls1-def2*:
subcls1 P =
(*SIGMA C*:{*C*. *is-class P C*}. {*D*. *C*≠*Object* ∧ *fst* (*the* (*class P C*))=*D*})⟨*proof*⟩
lemma *finite-subcls1*: *finite* (*subcls1 P*)⟨*proof*⟩
primrec *supercls-lst* :: *'m prog* ⇒ *cname list* ⇒ *bool* **where**
supercls-lst P (*C*#*Cs*) = ((∀ *C'* ∈ *set Cs*. *P* ⊢ *C'* ≼* *C*) ∧ *supercls-lst P Cs*) |
supercls-lst P [] = *True*

lemma *supercls-lst-app*:
⟦ *supercls-lst P* (*C*#*Cs*); *P* ⊢ *C* ≼* *C'* ⟧ ⟹ *supercls-lst P* (*C'*#*C*#*Cs*)
⟨*proof*⟩

1.4.2 The subtype relations

inductive
widen :: *'m prog* ⇒ *ty* ⇒ *ty* ⇒ *bool* (‹- ⊢ - ≤ -› [*71*,*71*,*71*] *70*)
for *P* :: *'m prog*
where
widen-refl[*iff*]: *P* ⊢ *T* ≤ *T*
| *widen-subcls*: *P* ⊢ *C* ≼* *D* ⟹ *P* ⊢ *Class C* ≤ *Class D*
| *widen-null*[*iff*]: *P* ⊢ *NT* ≤ *Class C*

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

lemma [*iff*]: (*P* ⊢ *T* ≤ *Void*) = (*T* = *Void*)⟨*proof*⟩
lemma [*iff*]: (*P* ⊢ *T* ≤ *Boolean*) = (*T* = *Boolean*)⟨*proof*⟩
lemma [*iff*]: (*P* ⊢ *T* ≤ *Integer*) = (*T* = *Integer*)⟨*proof*⟩
lemma [*iff*]: (*P* ⊢ *Void* ≤ *T*) = (*T* = *Void*)⟨*proof*⟩

lemma [*iff*]: $(P \vdash Boolean \leq T) = (T = Boolean)\langle proof\rangle$
lemma [*iff*]: $(P \vdash Integer \leq T) = (T = Integer)\langle proof\rangle$

lemma *Class-widen*: $P \vdash Class\ C \leq T \implies \exists D.\ T = Class\ D\langle proof\rangle$
lemma [*iff*]: $(P \vdash T \leq NT) = (T = NT)\langle proof\rangle$
lemma *Class-widen-Class* [*iff*]: $(P \vdash Class\ C \leq Class\ D) = (P \vdash C \preceq^* D)\langle proof\rangle$
lemma *widen-Class*: $(P \vdash T \leq Class\ C) = (T = NT \vee (\exists D.\ T = Class\ D \wedge P \vdash D \preceq^* C))\langle proof\rangle$

lemma *widen-trans*[*trans*]: $[\![P \vdash S \leq U;\ P \vdash U \leq T]\!] \implies P \vdash S \leq T\langle proof\rangle$
lemma *widens-trans* [*trans*]: $[\![P \vdash Ss\ [\leq]\ Ts;\ P \vdash Ts\ [\leq]\ Us]\!] \implies P \vdash Ss\ [\leq]\ Us\langle proof\rangle$

1.4.3 Method lookup

inductive
  Methods :: $['m\ prog,\ cname,\ mname \rightharpoonup (staticb \times ty\ list \times ty \times 'm) \times cname] \Rightarrow bool$
    $(\langle - \vdash - \ sees'\text{-}methods\ - \rangle\ [51,51,51]\ 50)$
  for $P :: 'm\ prog$
where
  sees-methods-Object:
 $[\![\ class\ P\ Object = Some(D,fs,ms);\ Mm = map\text{-}option\ (\lambda m.\ (m,Object)) \circ map\text{-}of\ ms\ ]\!]$
  $\implies P \vdash Object\ sees\text{-}methods\ Mm$
| *sees-methods-rec*:
 $[\![\ class\ P\ C = Some(D,fs,ms);\ C \neq Object;\ P \vdash D\ sees\text{-}methods\ Mm;$
    $Mm' = Mm\ ++\ (map\text{-}option\ (\lambda m.\ (m,C)) \circ map\text{-}of\ ms)\ ]\!]$
  $\implies P \vdash C\ sees\text{-}methods\ Mm'$

lemma *sees-methods-fun*:
assumes *1*: $P \vdash C\ sees\text{-}methods\ Mm$
shows $\bigwedge Mm'.\ P \vdash C\ sees\text{-}methods\ Mm' \implies Mm' = Mm$
$\langle proof\rangle$
lemma *visible-methods-exist*:
  $P \vdash C\ sees\text{-}methods\ Mm \implies Mm\ M = Some(m,D) \implies$
   $(\exists D'\ fs\ ms.\ class\ P\ D = Some(D',fs,ms) \wedge map\text{-}of\ ms\ M = Some\ m)$
$\langle proof\rangle$
lemma *sees-methods-decl-above*:
assumes *Csees*: $P \vdash C\ sees\text{-}methods\ Mm$
shows $Mm\ M = Some(m,D) \implies P \vdash C \preceq^* D$
$\langle proof\rangle$
lemma *sees-methods-idemp*:
assumes *Cmethods*: $P \vdash C\ sees\text{-}methods\ Mm$
shows $\bigwedge m\ D.\ Mm\ M = Some(m,D) \implies$
        $\exists Mm'.\ (P \vdash D\ sees\text{-}methods\ Mm') \wedge Mm'\ M = Some(m,D)\langle proof\rangle$

lemma *sees-methods-decl-mono*:
assumes *sub*: $P \vdash C' \preceq^* C$
shows $P \vdash C\ sees\text{-}methods\ Mm \implies$
    $\exists Mm'\ Mm_2.\ P \vdash C'\ sees\text{-}methods\ Mm' \wedge Mm' = Mm\ ++\ Mm_2 \wedge$
        $(\forall M\ m\ D.\ Mm_2\ M = Some(m,D) \longrightarrow P \vdash D \preceq^* C)\langle proof\rangle$
lemma *sees-methods-is-class-Object*:
 $P \vdash D\ sees\text{-}methods\ Mm \implies is\text{-}class\ P\ Object$
$\langle proof\rangle$

lemma *sees-methods-sub-Obj*: $P \vdash C\ sees\text{-}methods\ Mm \implies P \vdash C \preceq^* Object$

⟨*proof*⟩

definition *Method* :: *'m prog* ⇒ *cname* ⇒ *mname* ⇒ *staticb* ⇒ *ty list* ⇒ *ty* ⇒ *'m* ⇒ *cname* ⇒ *bool*
(‹- ⊢ - *sees* -, - : - →- = - *in* -› [*51*,*51*,*51*,*51*,*51*,*51*,*51*,*51*] *50*)
where
P ⊢ *C sees M*, *b*: *Ts*→*T* = *m in D* ≡
∃ *Mm*. *P* ⊢ *C sees-methods Mm* ∧ *Mm M* = *Some*((*b*,*Ts*,*T*,*m*),*D*)

definition *has-method* :: *'m prog* ⇒ *cname* ⇒ *mname* ⇒ *staticb* ⇒ *bool*
(‹- ⊢ - *has* -, -› [*51*,*0*,*0*,*51*] *50*)
where
P ⊢ *C has M*, *b* ≡ ∃ *Ts T m D*. *P* ⊢ *C sees M*,*b*:*Ts*→*T* = *m in D*

lemma *sees-method-fun*:
⟦*P* ⊢ *C sees M*,*b*:*TS*→*T* = *m in D*; *P* ⊢ *C sees M*,*b'*:*TS'*→*T'* = *m'* *in D'* ⟧
⟹ *b* = *b'* ∧ *TS'* = *TS* ∧ *T'* = *T* ∧ *m'* = *m* ∧ *D'* = *D*
⟨*proof*⟩
lemma *sees-method-decl-above*:
P ⊢ *C sees M*,*b*:*Ts*→*T* = *m in D* ⟹ *P* ⊢ *C* ≼* *D*
⟨*proof*⟩
lemma *visible-method-exists*:
P ⊢ *C sees M*,*b*:*Ts*→*T* = *m in D* ⟹
∃ *D' fs ms*. *class P D* = *Some*(*D'*,*fs*,*ms*) ∧ *map-of ms M* = *Some*(*b*,*Ts*,*T*,*m*)⟨*proof*⟩

lemma *sees-method-idemp*:
P ⊢ *C sees M*,*b*:*Ts*→*T*=*m in D* ⟹ *P* ⊢ *D sees M*,*b*:*Ts*→*T*=*m in D*
⟨*proof*⟩
lemma *sees-method-decl-mono*:
assumes *sub*: *P* ⊢ *C'* ≼* *C* **and**
C-sees: *P* ⊢ *C sees M*,*b*:*Ts*→*T*=*m in D* **and**
C'-sees: *P* ⊢ *C' sees M*,*b'*:*Ts'*→*T'*=*m'* *in D'*
shows *P* ⊢ *D'* ≼* *D*
⟨*proof*⟩
lemma *sees-methods-is-class*: *P* ⊢ *C sees-methods Mm* ⟹ *is-class P C*⟨*proof*⟩
lemma *sees-method-is-class*:
⟦ *P* ⊢ *C sees M*,*b*:*Ts*→*T*=*m in D* ⟧ ⟹ *is-class P C*⟨*proof*⟩
lemma *sees-method-is-class'*:
⟦ *P* ⊢ *C sees M*,*b*:*Ts*→*T*=*m in D* ⟧ ⟹ *is-class P D*⟨*proof*⟩
lemma *sees-method-sub-Obj*: *P* ⊢ *C sees M*,*b*: *Ts*→*T* = *m in D* ⟹ *P* ⊢ *C* ≼* *Object*
⟨*proof*⟩

1.4.4 Field lookup

inductive
Fields :: [*'m prog*, *cname*, ((*vname* × *cname*) × *staticb* × *ty*) *list*] ⇒ *bool*
(‹- ⊢ - *has'-fields* -› [*51*,*51*,*51*] *50*)
for *P* :: *'m prog*
where
has-fields-rec:
⟦ *class P C* = *Some*(*D*,*fs*,*ms*); *C* ≠ *Object*; *P* ⊢ *D has-fields FDTs*;
FDTs' = *map* (λ(*F*,*b*,*T*). ((*F*,*C*),*b*,*T*)) *fs* @ *FDTs* ⟧
⟹ *P* ⊢ *C has-fields FDTs'*
| *has-fields-Object*:

⟦ *class P Object = Some(D,fs,ms)*; *FDTs = map* (λ(*F*,*b*,*T*). ((*F*,*Object*),*b*,*T*)) *fs* ⟧
⟹ *P* ⊢ *Object has-fields FDTs*

lemma *has-fields-is-class*:
P ⊢ *C has-fields FDTs* ⟹ *is-class P C*⟨*proof*⟩
lemma *has-fields-fun*:
assumes *1*: *P* ⊢ *C has-fields FDTs*
shows ⋀*FDTs′*. *P* ⊢ *C has-fields FDTs′* ⟹ *FDTs′* = *FDTs*
⟨*proof*⟩
lemma *all-fields-in-has-fields*:
assumes *sub*: *P* ⊢ *C has-fields FDTs*
shows ⟦ *P* ⊢ *C* ≼* *D*; *class P D = Some*(*D′*,*fs*,*ms*); (*F*,*b*,*T*) ∈ *set fs* ⟧
⟹ ((*F*,*D*),*b*,*T*) ∈ *set FDTs*⟨*proof*⟩
lemma *has-fields-decl-above*:
assumes *fields*: *P* ⊢ *C has-fields FDTs*
shows ((*F*,*D*),*b*,*T*) ∈ *set FDTs* ⟹ *P* ⊢ *C* ≼* *D*⟨*proof*⟩

lemma *subcls-notin-has-fields*:
assumes *fields*: *P* ⊢ *C has-fields FDTs*
shows ((*F*,*D*),*b*,*T*) ∈ *set FDTs* ⟹ (*D*,*C*) ∉ (*subcls1 P*)⁺⟨*proof*⟩
lemma *subcls-notin-has-fields2*:
assumes *fields*: *P* ⊢ *C has-fields FDTs*
shows ⟦ *C* ≠ *Object*; *P* ⊢ *C* ≺¹ *D* ⟧ ⟹ (*D*,*C*) ∉ (*subcls1 P*)*
⟨*proof*⟩

lemma *has-fields-mono-lem*:
assumes *sub*: *P* ⊢ *D* ≼* *C*
shows *P* ⊢ *C has-fields FDTs*
⟹ ∃ *pre*. *P* ⊢ *D has-fields pre*@*FDTs* ∧ *dom*(*map-of pre*) ∩ *dom*(*map-of FDTs*) = {}⟨*proof*⟩
lemma *has-fields-declaring-classes*:
shows *P* ⊢ *C has-fields FDTs*
⟹ ∃ *pre FDTs′*. *FDTs* = *pre*@*FDTs′*
∧ (*C* ≠ *Object* ⟶ (∃ *D fs ms*. *class P C* = ⌊(*D*,*fs*,*ms*)⌋ ∧ *P* ⊢ *D has-fields FDTs′*))
∧ *set*(*map* (λ*t*. *snd*(*fst t*)) *pre*) ⊆ {*C*}
∧ *set*(*map* (λ*t*. *snd*(*fst t*)) *FDTs′*) ⊆ {*C′*. *C′* ≠ *C* ∧ *P* ⊢ *C* ≼* *C′*}
⟨*proof*⟩

lemma *has-fields-mono-lem2*:
assumes *hf*: *P* ⊢ *C has-fields FDTs*
and *cls*: *class P C = Some*(*D*,*fs*,*ms*) **and** *map-of*: *map-of FDTs* (*F*,*C*) = ⌊(*b*,*T*)⌋
shows ∃ *FDTs′*. *FDTs* = (*map* (λ(*F*,*b*,*T*). ((*F*,*C*),*b*,*T*)) *fs*) @ *FDTs′* ∧ *map-of FDTs′* (*F*,*C*) = *None*
⟨*proof*⟩

lemma *has-fields-is-class-Object*:
P ⊢ *D has-fields FDTs* ⟹ *is-class P Object*
⟨*proof*⟩

lemma *Object-fields*:
⟦ *P* ⊢ *Object has-fields FDTs*; *C* ≠ *Object* ⟧ ⟹ *map-of FDTs* (*F*,*C*) = *None*
⟨*proof*⟩

definition *has-field* :: *'m prog* ⇒ *cname* ⇒ *vname* ⇒ *staticb* ⇒ *ty* ⇒ *cname* ⇒ *bool*

(‹- ⊢ - *has* -,-:- *in* -› [51,51,51,51,51,51] 50)
where
$P \vdash C$ *has* $F,b{:}T$ *in* $D \equiv$
$\exists FDTs.\ P \vdash C$ *has-fields* $FDTs \wedge$ *map-of* $FDTs\ (F,D) = Some\ (b,T)$

lemma *has-field-mono*:
assumes *has*: $P \vdash C$ *has* $F,b{:}T$ *in* D **and** *sub*: $P \vdash C' \preceq^* C$
shows $P \vdash C'$ *has* $F,b{:}T$ *in* $D\langle proof\rangle$
lemma *has-field-fun*:
$[\![P \vdash C$ *has* $F,b{:}T$ *in* $D;\ P \vdash C$ *has* $F,b'{:}T'$ *in* $D]\!] \Longrightarrow b = b' \wedge T' = T\langle proof\rangle$

lemma *has-field-idemp*:
assumes *has*: $P \vdash C$ *has* $F,b{:}T$ *in* D
shows $P \vdash D$ *has* $F,b{:}T$ *in* $D\langle proof\rangle$
lemma *visible-fields-exist*:
assumes *fields*: $P \vdash C$ *has-fields* $FDTs$ **and**
FDTs: *map-of* $FDTs\ (F,D) = Some\ (b,\ T)$
shows $\exists D'\ fs\ ms.$ *class* $P\ D = Some(D',fs,ms) \wedge$ *map-of* $fs\ F = Some(b,T)$
$\langle proof\rangle$

lemma *map-of-remap-SomeD*:
map-of (*map* ($\lambda((k,k'),x).\ (k,(k',x))$) t) $k = Some\ (k',x) \Longrightarrow$ *map-of* $t\ (k,\ k') = Some\ x\langle proof\rangle$
lemma *map-of-remap-SomeD2*:
map-of (*map* ($\lambda((k,k'),x,x').\ (k,(k',x,x'))$) t) $k = Some\ (k',x,x') \Longrightarrow$ *map-of* $t\ (k,\ k') = Some\ (x,\ x')\langle proof\rangle$
lemma *has-field-decl-above*:
$P \vdash C$ *has* $F,b{:}T$ *in* $D \Longrightarrow P \vdash C \preceq^* D\langle proof\rangle$
definition *sees-field* :: *'m prog* ⇒ *cname* ⇒ *vname* ⇒ *staticb* ⇒ *ty* ⇒ *cname* ⇒ *bool*
(‹- ⊢ - *sees* -,-:- *in* -› [51,51,51,51,51,51] 50)
where
$P \vdash C$ *sees* $F,b{:}T$ *in* $D \equiv$
$\exists FDTs.\ P \vdash C$ *has-fields* $FDTs \wedge$
map-of (*map* ($\lambda((F,D),b,T).\ (F,(D,b,T))$) $FDTs$) $F = Some(D,b,T)$

lemma *has-visible-field*:
$P \vdash C$ *sees* $F,b{:}T$ *in* $D \Longrightarrow P \vdash C$ *has* $F,b{:}T$ *in* $D\langle proof\rangle$
lemma *sees-field-fun*:
$[\![P \vdash C$ *sees* $F,b{:}T$ *in* $D;\ P \vdash C$ *sees* $F,b'{:}T'$ *in* $D']\!] \Longrightarrow b = b' \wedge T' = T \wedge D' = D\langle proof\rangle$
lemma *sees-field-decl-above*:
$P \vdash C$ *sees* $F,b{:}T$ *in* $D \Longrightarrow P \vdash C \preceq^* D\langle proof\rangle$

lemma *sees-field-idemp*:
assumes *sees*: $P \vdash C$ *sees* $F,b{:}T$ *in* D
shows $P \vdash D$ *sees* $F,b{:}T$ *in* $D\langle proof\rangle$
lemma *has-field-sees-aux*:
assumes *hf*: $P \vdash C$ *has-fields* $FDTs$ **and** *map*: *map-of* $FDTs\ (F,\ C) = \lfloor(b,\ T)\rfloor$
shows *map-of* (*map* ($\lambda((F,\ D),\ b,\ T).\ (F,\ D,\ b,\ T)$) $FDTs$) $F = \lfloor(C,\ b,\ T)\rfloor$
$\langle proof\rangle$

lemma *has-field-sees*: $P \vdash C$ *has* $F,b{:}T$ *in* $C \Longrightarrow P \vdash C$ *sees* $F,b{:}T$ *in* C
$\langle proof\rangle$

lemma *has-field-is-class*:

$P \vdash C$ *has* $F,b{:}T$ *in* $D \Longrightarrow$ *is-class* $P\ C\langle proof\rangle$
lemma *has-field-is-class'*:
$P \vdash C$ *has* $F,b{:}T$ *in* $D \Longrightarrow$ *is-class* $P\ D\langle proof\rangle$

1.4.5 Functional lookup

definition *method* :: *'m prog* ⇒ *cname* ⇒ *mname* ⇒ *cname* × *staticb* × *ty list* × *ty* × *'m*
where
method $P\ C\ M \equiv$ *THE* $(D,b,Ts,T,m).\ P \vdash C$ *sees* $M,b{:}Ts \to T = m$ *in* D

definition *field* :: *'m prog* ⇒ *cname* ⇒ *vname* ⇒ *cname* × *staticb* × *ty*
where
field $P\ C\ F \equiv$ *THE* $(D,b,T).\ P \vdash C$ *sees* $F,b{:}T$ *in* D

definition *fields* :: *'m prog* ⇒ *cname* ⇒ ((*vname* × *cname*) × *staticb* × *ty*) *list*
where
fields $P\ C \equiv$ *THE FDTs*. $P \vdash C$ *has-fields FDTs*

lemma *fields-def2* [*simp*]: $P \vdash C$ *has-fields FDTs* $\Longrightarrow$ *fields* $P\ C = FDTs\langle proof\rangle$
lemma *field-def2* [*simp*]: $P \vdash C$ *sees* $F,b{:}T$ *in* $D \Longrightarrow$ *field* $P\ C\ F = (D,b,T)\langle proof\rangle$
lemma *method-def2* [*simp*]: $P \vdash C$ *sees* $M,b{:}\ Ts \to T = m$ *in* $D \Longrightarrow$ *method* $P\ C\ M = (D,b,Ts,T,m)\langle proof\rangle$

The following are the fields for initializing an object (non-static fields) and a class (just that class's static fields), respectively.

definition *ifields* :: *'m prog* ⇒ *cname* ⇒ ((*vname* × *cname*) × *staticb* × *ty*) *list*
where
ifields $P\ C \equiv$ *filter* $(\lambda((F,D),b,T).\ b = NonStatic)$ (*fields* $P\ C$)

definition *isfields* :: *'m prog* ⇒ *cname* ⇒ ((*vname* × *cname*) × *staticb* × *ty*) *list*
where
isfields $P\ C \equiv$ *filter* $(\lambda((F,D),b,T).\ b = Static \wedge D = C)$ (*fields* $P\ C$)

lemma *ifields-def2*[*simp*]: ⟦ $P \vdash C$ *has-fields FDTs* ⟧ ⟹ *ifields* $P\ C$ = *filter* $(\lambda((F,D),b,T).\ b = NonStatic)$ *FDTs*
$\langle proof\rangle$

lemma *isfields-def2*[*simp*]: ⟦ $P \vdash C$ *has-fields FDTs* ⟧ ⟹ *isfields* $P\ C$ = *filter* $(\lambda((F,D),b,T).\ b = Static \wedge D = C)$ *FDTs*
$\langle proof\rangle$

lemma *ifields-def3*: ⟦ $P \vdash C$ *sees* $F,b{:}T$ *in* D; $b = NonStatic$ ⟧ ⟹ $(((F,D),b,T) \in$ *set* (*ifields* $P\ C))\langle proof\rangle$
lemma *isfields-def3*: ⟦ $P \vdash C$ *sees* $F,b{:}T$ *in* D; $b = Static$; $D = C$ ⟧ ⟹ $(((F,D),b,T) \in$ *set* (*isfields* $P\ C))\langle proof\rangle$

definition *seeing-class* :: *'m prog* ⇒ *cname* ⇒ *mname* ⇒ *cname option* **where**
seeing-class $P\ C\ M =$
(*if* $\exists\ Ts\ T\ m\ D.\ P \vdash C$ *sees* $M,Static{:}Ts \to T = m$ *in* D
then Some (*fst*(*method* $P\ C\ M$))
else None)

lemma *seeing-class-def2*[*simp*]:
$P \vdash C$ *sees* $M,Static{:}Ts \to T = m$ *in* $D \Longrightarrow$ *seeing-class* $P\ C\ M$ = *Some* D
$\langle proof\rangle$

1.5 Jinja Values

theory *Value* **imports** *TypeRel* **begin**

type-synonym *addr* = *nat*

datatype *val*
= *Unit* — dummy result value of void expressions
| *Null* — null reference
| *Bool bool* — Boolean value
| *Intg int* — integer value
| *Addr addr* — addresses of objects in the heap

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

primrec *the-Addr* :: *val* ⇒ *addr* **where**
the-Addr (*Addr a*) = *a*

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*

end

1.6 Objects and the Heap

theory *Objects* **imports** *TypeRel Value* **begin**

1.6.1 Objects

type-synonym
fields = *vname* × *cname* ⇀ *val* — field name, defining class, value
type-synonym
obj = *cname* × *fields* — class instance with class name and fields
type-synonym
sfields = *vname* ⇀ *val* — field name to value

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

— initializes a given list of fields
definition *init-fields* :: ((*vname* × *cname*) × *staticb* × *ty*) *list* ⇒ *fields*
where
init-fields FDTs ≡ (*map-of* ∘ *map* (λ((*F*,*D*),*b*,*T*). ((*F*,*D*),*default-val T*))) *FDTs*

definition *init-sfields* :: ((*vname* × *cname*) × *staticb* × *ty*) *list* ⇒ *sfields*
where
init-sfields FDTs ≡ (*map-of* ∘ *map* (λ((*F*,*D*),*b*,*T*). (*F*,*default-val T*))) *FDTs*

— a new, blank object with default values for instance fields:
definition *blank* :: *'m prog* ⇒ *cname* ⇒ *obj*
where
blank P C ≡ (*C*,*init-fields* (*ifields P C*))

— a new, blank object with default values for static fields:
definition *sblank* :: *'m prog* ⇒ *cname* ⇒ *sfields*
where
sblank P C ≡ *init-sfields* (*isfields P C*)

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

translations
(*type*) *fields* <= (*type*) *char list* × *char list* ⇒ *val option*
(*type*) *obj* <= (*type*) *char list* × *fields*
(*type*) *sfields* <= (*type*) *char list* ⇒ *val option*

1.6.2 Heap

type-synonym *heap* = *addr* ⇀ *obj*

translations
(*type*) *heap* <= (*type*) *nat* ⇒ *obj option*

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*(*LEAST a*. *h a* = *None*) *else None*

definition *cast-ok* :: *'m prog* ⇒ *cname* ⇒ *heap* ⇒ *val* ⇒ *bool*
where
cast-ok P C h v ≡ *v* = *Null* ∨ *P* ⊢ *cname-of h* (*the-Addr v*) ≼* *C*

definition *hext* :: *heap* ⇒ *heap* ⇒ *bool* (‹- ⊴ -› [*51*,*51*] *50*)
where
h ⊴ *h'* ≡ ∀ *a C fs*. *h a* = *Some*(*C*,*fs*) ⟶ (∃ *fs'*. *h' a* = *Some*(*C*,*fs'*))

primrec *typeof-h* :: *heap* ⇒ *val* ⇒ *ty option* (‹*typeof*_›)
where
typeof$_h$ *Unit* = *Some Void*
| *typeof*$_h$ *Null* = *Some NT*
| *typeof*$_h$ (*Bool b*) = *Some Boolean*
| *typeof*$_h$ (*Intg i*) = *Some Integer*
| *typeof*$_h$ (*Addr a*) = (*case h a of None* ⇒ *None* | *Some*(*C*,*fs*) ⇒ *Some*(*Class C*))

lemma *new-Addr-SomeD*:
new-Addr h = *Some a* ⟹ *h a* = *None*
⟨*proof*⟩
lemma [*simp*]: (*typeof*$_h$ *v* = *Some Boolean*) = (∃ *b*. *v* = *Bool b*)
⟨*proof*⟩

lemma [*simp*]: $(typeof_h\ v = Some\ Integer) = (\exists i.\ v = Intg\ i)\langle proof\rangle$
lemma [*simp*]: $(typeof_h\ v = Some\ NT) = (v = Null)$
$\langle proof\rangle$
lemma [*simp*]: $(typeof_h\ v = Some(Class\ C)) = (\exists a\ fs.\ v = Addr\ a \wedge h\ a = Some(C,fs))$
$\langle proof\rangle$
lemma [*simp*]: $h\ a = Some(C,fs) \Longrightarrow typeof_{(h(a\mapsto(C,fs')))}\ v = typeof_h\ v$
$\langle proof\rangle$

For literal values the first parameter of *typeof* can be set to {} because they do not contain addresses:

abbreviation
typeof :: *val* ⇒ *ty option* **where**
typeof v == *typeof-h Map.empty v*

lemma *typeof-lit-typeof*:
$typeof\ v = Some\ T \Longrightarrow typeof_h\ v = Some\ T$
$\langle proof\rangle$
lemma *typeof-lit-is-type*:
$typeof\ v = Some\ T \Longrightarrow is\text{-}type\ P\ T$
$\langle proof\rangle$

1.6.3 Heap extension $\trianglelefteq$

lemma *hextI*: $\forall a\ C\ fs.\ h\ a = Some(C,fs) \longrightarrow (\exists fs'.\ h'\ a = Some(C,fs')) \Longrightarrow h \trianglelefteq h'\langle proof\rangle$
lemma *hext-objD*: $[\![\ h \trianglelefteq h';\ h\ a = Some(C,fs)\ ]\!] \Longrightarrow \exists fs'.\ h'\ a = Some(C,fs')\langle proof\rangle$
lemma *hext-refl* [*iff*]: $h \trianglelefteq h\langle proof\rangle$
lemma *hext-new* [*simp*]: $h\ a = None \Longrightarrow h \trianglelefteq h(a\mapsto x)\langle proof\rangle$
lemma *hext-trans*: $[\![\ h \trianglelefteq h';\ h' \trianglelefteq h''\ ]\!] \Longrightarrow h \trianglelefteq h''\langle proof\rangle$
lemma *hext-upd-obj*: $h\ a = Some\ (C,fs) \Longrightarrow h \trianglelefteq h(a\mapsto(C,fs'))\langle proof\rangle$
lemma *hext-typeof-mono*: $[\![\ h \trianglelefteq h';\ typeof_h\ v = Some\ T\ ]\!] \Longrightarrow typeof_{h'}\ v = Some\ T\langle proof\rangle$

1.6.4 Static field information function

datatype *init-state* = *Done* | *Processing* | *Prepared* | *Error*
— *Done* = initialized
— *Processing* = currently being initialized
— *Prepared* = uninitialized and not currently being initialized
— *Error* = previous initialization attempt resulted in erroneous state

inductive *iprog* :: *init-state* ⇒ *init-state* ⇒ *bool* (‹- $\leq_i$ -› [*51*,*51*] *50*)
where
[*simp*]: *Prepared* $\leq_i$ *i*
| [*simp*]: *Processing* $\leq_i$ *Done*
| [*simp*]: *Processing* $\leq_i$ *Error*
| [*simp*]: *i* $\leq_i$ *i*

lemma *iprog-Done*[*simp*]: $(Done \leq_i i) = (i = Done)$
$\langle proof\rangle$

lemma *iprog-Error*[*simp*]: $(Error \leq_i i) = (i = Error)$
$\langle proof\rangle$

lemma *iprog-Processing*[*simp*]: $(Processing \leq_i i) = (i = Done \vee i = Error \vee i = Processing)$
$\langle proof\rangle$

lemma *iprog-trans*: $[\![\ i \le_i i';\ i' \le_i i''\ ]\!] \Longrightarrow i \le_i i'' \langle proof \rangle$

1.6.5 Static Heap

The static heap (sheap) is used for storing information about static field values and initialization status for classes.

type-synonym
sheap = *cname* $\rightharpoonup$ *sfields* $\times$ *init-state*

translations
(*type*) *sheap* <= (*type*) *char list* $\Rightarrow$ (*sfields* $\times$ *init-state*) *option*

definition *shext* :: *sheap* $\Rightarrow$ *sheap* $\Rightarrow$ *bool* ($\langle$- $\preceq_s$ -$\rangle$ [*51*,*51*] *50*)
where
$sh \preceq_s sh' \equiv \forall C\ sfs\ i.\ sh\ C = Some(sfs,i) \longrightarrow (\exists sfs'\ i'.\ sh'\ C = Some(sfs',i') \wedge i \le_i i')$

lemma *shextI*: $\forall C\ sfs\ i.\ sh\ C = Some(sfs,i) \longrightarrow (\exists sfs'\ i'.\ sh'\ C = Some(sfs',i') \wedge i \le_i i') \Longrightarrow sh \preceq_s sh' \langle proof \rangle$
lemma *shext-objD*: $[\![\ sh \preceq_s sh';\ sh\ C = Some(sfs,i)\ ]\!] \Longrightarrow \exists sfs'\ i'.\ sh'\ C = Some(sfs',\ i') \wedge i \le_i i' \langle proof \rangle$
lemma *shext-refl* [*iff*]: $sh \preceq_s sh \langle proof \rangle$
lemma *shext-new* [*simp*]: $sh\ C = None \Longrightarrow sh \preceq_s sh(C \mapsto x) \langle proof \rangle$
lemma *shext-trans*: $[\![\ sh \preceq_s sh';\ sh' \preceq_s sh''\ ]\!] \Longrightarrow sh \preceq_s sh'' \langle proof \rangle$
lemma *shext-upd-obj*: $[\![\ sh\ C = Some\ (sfs,i);\ i \le_i i'\ ]\!] \Longrightarrow sh \preceq_s sh(C \mapsto (sfs',i')) \langle proof \rangle$
end

1.7 Exceptions

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

definition *ErrorCl* :: *string* **where** *ErrorCl* = *''Error''*
definition *ThrowCl* :: *string* **where** *ThrowCl* = *''Throwable''*

definition *NullPointer* :: *cname*
where
NullPointer $\equiv$ *''NullPointer''*

definition *ClassCast* :: *cname*
where
ClassCast $\equiv$ *''ClassCast''*

definition *OutOfMemory* :: *cname*
where
OutOfMemory $\equiv$ *''OutOfMemory''*

definition *NoClassDefFoundError* :: *cname*
where
NoClassDefFoundError $\equiv$ *''NoClassDefFoundError''*

definition *IncompatibleClassChangeError* :: *cname*
where

IncompatibleClassChangeError ≡ *''IncompatibleClassChangeError''*

definition *NoSuchFieldError* :: *cname*
where
NoSuchFieldError ≡ *''NoSuchFieldError''*

definition *NoSuchMethodError* :: *cname*
where
NoSuchMethodError ≡ *''NoSuchMethodError''*

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

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*
if s = *NoClassDefFoundError then 3 else*
if s = *IncompatibleClassChangeError then 4 else*
if s = *NoSuchFieldError then 5 else*
if s = *NoSuchMethodError then 6 else undefined*

lemmas *sys-xcpts-defs* = *NullPointer-def ClassCast-def OutOfMemory-def NoClassDefFoundError-def*
IncompatibleClassChangeError-def NoSuchFieldError-def NoSuchMethodError-def

lemma *Start-nsys-xcpts*: *Start* ∉ *sys-xcpts*
⟨*proof*⟩

lemma *Start-nsys-xcpts1* [*simp*]: *Start* ≠ *NullPointer Start* ≠ *ClassCast*
Start ≠ *OutOfMemory Start* ≠ *NoClassDefFoundError*
Start ≠ *IncompatibleClassChangeError Start* ≠ *NoSuchFieldError*
Start ≠ *NoSuchMethodError*
⟨*proof*⟩

lemma *Start-nsys-xcpts2* [*simp*]: *NullPointer* ≠ *Start ClassCast* ≠ *Start*
OutOfMemory ≠ *Start NoClassDefFoundError* ≠ *Start*
IncompatibleClassChangeError ≠ *Start NoSuchFieldError* ≠ *Start*
NoSuchMethodError ≠ *Start*
⟨*proof*⟩

definition *start-heap* :: *'c prog* ⇒ *heap*
where
start-heap G ≡ *Map.empty* (*addr-of-sys-xcpt NullPointer* ↦ *blank G NullPointer*,
addr-of-sys-xcpt ClassCast ↦ *blank G ClassCast*,
addr-of-sys-xcpt OutOfMemory ↦ *blank G OutOfMemory*,
addr-of-sys-xcpt NoClassDefFoundError ↦ *blank G NoClassDefFoundError*,
addr-of-sys-xcpt IncompatibleClassChangeError ↦ *blank G IncompatibleClass-*
ChangeError,
addr-of-sys-xcpt NoSuchFieldError ↦ *blank G NoSuchFieldError*,

addr-of-sys-xcpt NoSuchMethodError $\mapsto$ *blank G NoSuchMethodError*)

definition *preallocated* :: *heap* $\Rightarrow$ *bool*
where
preallocated h $\equiv \forall$ *C* $\in$ *sys-xcpts*. $\exists$*fs*. *h*(*addr-of-sys-xcpt C*) = *Some* (*C*,*fs*)

1.7.1 System exceptions

lemma *sys-xcpts-incl* [*simp*]: *NullPointer* $\in$ *sys-xcpts* $\wedge$ *OutOfMemory* $\in$ *sys-xcpts*
$\wedge$ *ClassCast* $\in$ *sys-xcpts* $\wedge$ *NoClassDefFoundError* $\in$ *sys-xcpts*
$\wedge$ *IncompatibleClassChangeError* $\in$ *sys-xcpts* $\wedge$ *NoSuchFieldError* $\in$ *sys-xcpts*
$\wedge$ *NoSuchMethodError* $\in$ *sys-xcpts*$\langle$*proof*$\rangle$
lemma *sys-xcpts-cases* [*consumes 1*, *cases set*]:
$[\![$ *C* $\in$ *sys-xcpts*; *P NullPointer*; *P OutOfMemory*; *P ClassCast*; *P NoClassDefFoundError*;
P IncompatibleClassChangeError; *P NoSuchFieldError*;
P NoSuchMethodError $]\!] \Longrightarrow$ *P C*$\langle$*proof*$\rangle$

1.7.2 Starting heap

lemma *start-heap-sys-xcpts*:
assumes *C* $\in$ *sys-xcpts*
shows *start-heap P* (*addr-of-sys-xcpt C*) = *Some*(*blank P C*)
$\langle$*proof*$\rangle$

lemma *start-heap-classes*:
start-heap P a = *Some*(*C*,*fs*) $\Longrightarrow$ *C* $\in$ *sys-xcpts*
$\langle$*proof*$\rangle$

lemma *start-heap-nStart*: *start-heap P a* = *Some obj* $\Longrightarrow$ *fst*(*obj*) $\neq$ *Start*
$\langle$*proof*$\rangle$

1.7.3 *preallocated*

lemma *preallocated-dom* [*simp*]:
$[\![$ *preallocated h*; *C* $\in$ *sys-xcpts* $]\!] \Longrightarrow$ *addr-of-sys-xcpt C* $\in$ *dom h*$\langle$*proof*$\rangle$

lemma *preallocatedD*:
$[\![$ *preallocated h*; *C* $\in$ *sys-xcpts* $]\!] \Longrightarrow \exists$*fs*. *h*(*addr-of-sys-xcpt C*) = *Some* (*C*, *fs*)$\langle$*proof*$\rangle$

lemma *preallocatedE* [*elim?*]:
$[\![$ *preallocated h*; *C* $\in$ *sys-xcpts*; $\bigwedge$*fs*. *h*(*addr-of-sys-xcpt C*) = *Some*(*C*,*fs*) $\Longrightarrow$ *P h C*$]\!]$
$\Longrightarrow$ *P h C*$\langle$*proof*$\rangle$

lemma *cname-of-xcp* [*simp*]:
$[\![$ *preallocated h*; *C* $\in$ *sys-xcpts* $]\!] \Longrightarrow$ *cname-of h* (*addr-of-sys-xcpt C*) = *C*$\langle$*proof*$\rangle$

lemma *typeof-ClassCast* [*simp*]:
preallocated h $\Longrightarrow$ *typeof*$_h$ (*Addr*(*addr-of-sys-xcpt ClassCast*)) = *Some*(*Class ClassCast*)$\langle$*proof*$\rangle$

lemma *typeof-OutOfMemory* [*simp*]:
preallocated h $\Longrightarrow$ *typeof*$_h$ (*Addr*(*addr-of-sys-xcpt OutOfMemory*)) = *Some*(*Class OutOfMemory*)$\langle$*proof*$\rangle$

lemma *typeof-NullPointer* [*simp*]:
preallocated h $\Longrightarrow$ *typeof*$_h$ (*Addr*(*addr-of-sys-xcpt NullPointer*)) = *Some*(*Class NullPointer*)$\langle$*proof*$\rangle$
lemma *typeof-NoClassDefFoundError* [*simp*]:

preallocated h ⟹ *typeof*$_h$ (*Addr*(*addr-of-sys-xcpt NoClassDefFoundError*)) = *Some*(*Class NoClassDefFoundError*)⟨*proof*⟩
lemma *typeof-IncompatibleClassChangeError* [*simp*]:
preallocated h ⟹ *typeof*$_h$ (*Addr*(*addr-of-sys-xcpt IncompatibleClassChangeError*)) = *Some*(*Class IncompatibleClassChangeError*)⟨*proof*⟩
lemma *typeof-NoSuchFieldError* [*simp*]:
preallocated h ⟹ *typeof*$_h$ (*Addr*(*addr-of-sys-xcpt NoSuchFieldError*)) = *Some*(*Class NoSuchFieldError*)⟨*proof*⟩
lemma *typeof-NoSuchMethodError* [*simp*]:
preallocated h ⟹ *typeof*$_h$ (*Addr*(*addr-of-sys-xcpt NoSuchMethodError*)) = *Some*(*Class NoSuchMethodError*)⟨*proof*⟩
lemma *preallocated-hext*:
⟦ *preallocated h*; *h* ⊴ *h'* ⟧ ⟹ *preallocated h'*⟨*proof*⟩
lemma *preallocated-start*:
preallocated (*start-heap P*)
⟨*proof*⟩

end

1.8 Expressions

theory *Expr*
imports *../Common/Exceptions*
begin

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

datatype *'a exp*
= *new cname* — class instance creation
| *Cast cname* (*'a exp*) — type cast
| *Val val* — value
| *BinOp* (*'a exp*) *bop* (*'a exp*) (‹- «-» -› [*80*,*0*,*81*] *80*) — binary operation
| *Var 'a* — local variable (incl. parameter)
| *LAss 'a* (*'a exp*) (‹-:=-› [*90*,*90*]*90*) — local assignment
| *FAcc* (*'a exp*) *vname cname* (‹-•-{-}› [*10*,*90*,*99*]*90*) — field access
| *SFAcc cname vname cname* (‹-•$_s$-{-}› [*10*,*90*,*99*]*90*) — static field access
| *FAss* (*'a exp*) *vname cname* (*'a exp*) (‹-•-{-} := -› [*10*,*90*,*99*,*90*]*90*) — field assignment
| *SFAss cname vname cname* (*'a exp*) (‹-•$_s$-{-} := -› [*10*,*90*,*99*,*90*]*90*) — static field assignment
| *Call* (*'a exp*) *mname* (*'a exp list*) (‹-•-'(-')› [*90*,*99*,*0*] *90*) — method call
| *SCall cname mname* (*'a exp list*) (‹-•$_s$-'(-')› [*90*,*99*,*0*] *90*) — static method call
| *Block 'a ty* (*'a exp*) (‹'{-:-; -}›)
| *Seq* (*'a exp*) (*'a exp*) (‹-;;/ -› [*61*,*60*]*60*)
| *Cond* (*'a exp*) (*'a exp*) (*'a exp*) (‹*if* '(-') -/ *else* -› [*80*,*79*,*79*]*70*)
| *While* (*'a exp*) (*'a exp*) (‹*while* '(-') -› [*80*,*79*]*70*)
| *throw* (*'a exp*)
| *TryCatch* (*'a exp*) *cname 'a* (*'a exp*) (‹*try* -/ *catch*'(- -') -› [*0*,*99*,*80*,*79*] *70*)
| *INIT cname cname list bool* (*'a exp*) (‹*INIT* - '(-,-') ← -› [*60*,*60*,*60*,*60*] *60*) — internal initialization command: class, list of superclasses to initialize, preparation flag; command on hold
| *RI cname* (*'a exp*) *cname list* (*'a exp*) (‹*RI* '(-,-') ; - ← -› [*60*,*60*,*60*,*60*] *60*) — running of the initialization procedure for class with expression, classes still to initialize command on hold

type-synonym
expr = *vname exp* — Jinja expression

type-synonym
J-mb = *vname list* × *expr* — Jinja method body: parameter names and expression
type-synonym
J-prog = *J-mb prog* — Jinja program

type-synonym
init-stack = *expr list* × *bool* — Stack of expressions waiting on initialization in small step; indicator boolean True if current expression has been init checked

The semantics of binary operators:

fun *binop* :: *bop* × *val* × *val* ⇒ *val option* **where**
binop(*Eq*,v_1,v_2) = *Some*(*Bool* (v_1 = v_2))
| *binop*(*Add*,*Intg* i_1,*Intg* i_2) = *Some*(*Intg*(i_1+i_2))
| *binop*(*bop*,v_1,v_2) = *None*

lemma [*simp*]:
(*binop*(*Add*,v_1,v_2) = *Some* *v*) = (∃ i_1 i_2. v_1 = *Intg* i_1 ∧ v_2 = *Intg* i_2 ∧ *v* = *Intg*(i_1+i_2))⟨*proof*⟩

lemma *map-Val-throw-eq*:
map Val vs @ *throw ex* # *es* = *map Val vs'* @ *throw ex'* # *es'* ⟹ *ex* = *ex'*⟨*proof*⟩
lemma *map-Val-nthrow-neq*:
map Val vs = *map Val vs'* @ *throw ex'* # *es'* ⟹ *False*⟨*proof*⟩
lemma *map-Val-eq*:
map Val vs = *map Val vs'* ⟹ *vs* = *vs'*⟨*proof*⟩

lemma *init-rhs-neq* [*simp*]: *e* ≠ *INIT C* (*Cs*,*b*) ← *e*
⟨*proof*⟩

lemma *init-rhs-neq'* [*simp*]: *INIT C* (*Cs*,*b*) ← *e* ≠ *e*
⟨*proof*⟩

lemma *ri-rhs-neq* [*simp*]: *e* ≠ *RI*(*C*,*e'*);*Cs* ← *e*
⟨*proof*⟩

lemma *ri-rhs-neq'* [*simp*]: *RI*(*C*,*e'*);*Cs* ← *e* ≠ *e*
⟨*proof*⟩

1.8.1 Syntactic sugar

abbreviation (*input*)
InitBlock:: *'a* ⇒ *ty* ⇒ *'a exp* ⇒ *'a exp* ⇒ *'a exp* (‹(*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 *addr a* == *Val*(*Addr a*)
abbreviation *true* == *Val*(*Bool True*)
abbreviation *false* == *Val*(*Bool False*)

abbreviation
Throw :: *addr* ⇒ *'a exp* **where**
Throw a == *throw*(*Val*(*Addr a*))

abbreviation

THROW :: *cname* ⇒ *'a exp* **where**
THROW xc == *Throw*(*addr-of-sys-xcpt xc*)

1.8.2 Free Variables

primrec *fv* :: *expr* ⇒ *vname set* **and** *fvs* :: *expr list* ⇒ *vname set* **where**
fv(*new C*) = {}
| *fv*(*Cast C e*) = *fv e*
| *fv*(*Val v*) = {}
| *fv*(e_1 «*bop*» e_2) = *fv* e_1 ∪ *fv* e_2
| *fv*(*Var V*) = {*V*}
| *fv*(*LAss V e*) = {*V*} ∪ *fv e*
| *fv*(*e*·*F*{*D*}) = *fv e*
| *fv*($C·_sF${*D*}) = {}
| *fv*(e_1·*F*{*D*}:=e_2) = *fv* e_1 ∪ *fv* e_2
| *fv*($C·_sF${*D*}:=e_2) = *fv* e_2
| *fv*(*e*·*M*(*es*)) = *fv e* ∪ *fvs es*
| *fv*($C·_sM$(*es*)) = *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*
| *fv*(*try* e_1 *catch*(*C V*) e_2) = *fv* e_1 ∪ (*fv* e_2 − {*V*})
| *fv*(*INIT C* (*Cs*,*b*) ← *e*) = *fv e*
| *fv*(*RI* (*C*,*e*);*Cs* ← *e'*) = *fv e* ∪ *fv e'*
| *fvs*([]) = {}
| *fvs*(*e*#*es*) = *fv e* ∪ *fvs es*

lemma [*simp*]: *fvs*(es_1 @ es_2) = *fvs* es_1 ∪ *fvs* es_2⟨*proof*⟩
lemma [*simp*]: *fvs*(*map Val vs*) = {}⟨*proof*⟩

1.8.3 Accessing expression constructor arguments

fun *val-of* :: *'a exp* ⇒ *val option* **where**
val-of (*Val v*) = *Some v* |
val-of - = *None*

lemma *val-of-spec*: *val-of e* = *Some v* ⟹ *e* = *Val v*
⟨*proof*⟩

fun *lass-val-of* :: *'a exp* ⇒ (*'a* × *val*) *option* **where**
lass-val-of (*V*:=*Val v*) = *Some* (*V*, *v*) |
lass-val-of - = *None*

lemma *lass-val-of-spec*:
assumes *lass-val-of e* = ⌊*a*⌋
shows *e* = (*fst a*:=*Val* (*snd a*))
⟨*proof*⟩

fun *map-vals-of* :: *'a exp list* ⇒ *val list option* **where**
map-vals-of (*e*#*es*) = (*case val-of e of Some v* ⇒ (*case map-vals-of es of Some vs* ⇒ *Some* (*v*#*vs*)
| - ⇒ *None*)
| - ⇒ *None*) |

map-vals-of [] = *Some* []

lemma *map-vals-of-spec*: *map-vals-of es* = *Some vs* ⟹ *es* = *map Val vs*
⟨*proof*⟩

lemma *map-vals-of-Vals*[*simp*]: *map-vals-of* (*map Val vs*) = ⌊*vs*⌋ ⟨*proof*⟩

lemma *map-vals-of-throw*[*simp*]:
map-vals-of (*map Val vs* @ *throw e* # *es'*) = *None*
⟨*proof*⟩

fun *bool-of* :: 'a *exp* ⇒ *bool option* **where**
bool-of true = *Some True* |
bool-of false = *Some False* |
bool-of - = *None*

lemma *bool-of-specT*:
assumes *bool-of e* = *Some True* **shows** *e* = *true*
⟨*proof*⟩

lemma *bool-of-specF*:
assumes *bool-of e* = *Some False* **shows** *e* = *false*
⟨*proof*⟩

fun *throw-of* :: 'a *exp* ⇒ 'a *exp option* **where**
throw-of (*throw e'*) = *Some e'* |
throw-of - = *None*

lemma *throw-of-spec*: *throw-of e* = *Some e'* ⟹ *e* = *throw e'*
⟨*proof*⟩

fun *init-exp-of* :: 'a *exp* ⇒ 'a *exp option* **where**
init-exp-of (*INIT C* (*Cs*,*b*) ← *e*) = *Some e* |
init-exp-of (*RI*(*C*,*e'*);*Cs* ← *e*) = *Some e* |
init-exp-of - = *None*

lemma *init-exp-of-neq* [*simp*]: *init-exp-of e* = ⌊*e'*⌋ ⟹ *e'* ≠ *e* ⟨*proof*⟩
lemma *init-exp-of-neq'*[*simp*]: *init-exp-of e* = ⌊*e'*⌋ ⟹ *e* ≠ *e'* ⟨*proof*⟩

1.8.4 Class initialization

This section defines a few functions that return information about an expression's current initialization status.

primrec *sub-RI* :: 'a *exp* ⇒ *bool* **and** *sub-RIs* :: 'a *exp list* ⇒ *bool* **where**
  sub-RI(*new C*) = *False*
| *sub-RI*(*Cast C e*) = *sub-RI e*
| *sub-RI*(*Val v*) = *False*
| *sub-RI*(*e*$_1$ «*bop*» *e*$_2$) = (*sub-RI e*$_1$ ∨ *sub-RI e*$_2$)
| *sub-RI*(*Var V*) = *False*
| *sub-RI*(*LAss V e*) = *sub-RI e*
| *sub-RI*(*e*•*F*{*D*}) = *sub-RI e*
| *sub-RI*(*C*•$_s$*F*{*D*}) = *False*

| *sub-RI*(e_1·F{D}:=e_2) = (*sub-RI* e_1 ∨ *sub-RI* e_2)
| *sub-RI*(C·$_s F${D}:=e_2) = *sub-RI* e_2
| *sub-RI*(e·M(es)) = (*sub-RI* e ∨ *sub-RIs* es)
| *sub-RI*(C·$_s M$(es)) = (M = *clinit* ∨ *sub-RIs* es)
| *sub-RI*({ V:T; e}) = *sub-RI* e
| *sub-RI*(e_1;;e_2) = (*sub-RI* e_1 ∨ *sub-RI* e_2)
| *sub-RI*(*if* (b) e_1 *else* e_2) = (*sub-RI* b ∨ *sub-RI* e_1 ∨ *sub-RI* e_2)
| *sub-RI*(*while* (b) e) = (*sub-RI* b ∨ *sub-RI* e)
| *sub-RI*(*throw* e) = *sub-RI* e
| *sub-RI*(*try* e_1 *catch*(C V) e_2) = (*sub-RI* e_1 ∨ *sub-RI* e_2)
| *sub-RI*(*INIT* C (Cs,b) ← e) = *True*
| *sub-RI*(*RI* (C,e);Cs ← e') = *True*
| *sub-RIs*([]) = *False*
| *sub-RIs*(e#es) = (*sub-RI* e ∨ *sub-RIs* es)

lemmas *sub-RI-sub-RIs-induct* = *sub-RI.induct sub-RIs.induct*

lemma *nsub-RIs-def*[*simp*]:
¬*sub-RIs* es ⟹ ∀ e ∈ *set* es. ¬*sub-RI* e
⟨*proof*⟩

lemma *sub-RI-base*:
e = *INIT* C (Cs, b) ← e' ∨ e = *RI*(C,e_0);Cs ← e' ⟹ *sub-RI* e
⟨*proof*⟩

lemma *nsub-RI-Vals*[*simp*]: ¬*sub-RIs* (*map Val vs*)
⟨*proof*⟩

lemma *lass-val-of-nsub-RI*: *lass-val-of* e = ⌊a⌋ ⟹ ¬*sub-RI* e
⟨*proof*⟩
primrec *not-init* :: *cname* ⇒ *'a exp* ⇒ *bool* **and** *not-inits* :: *cname* ⇒ *'a exp list* ⇒ *bool* **where**
not-init C' (*new* C) = *True*
| *not-init* C' (*Cast* C e) = *not-init* C' e
| *not-init* C' (*Val* v) = *True*
| *not-init* C' (e_1 «*bop*» e_2) = (*not-init* C' e_1 ∧ *not-init* C' e_2)
| *not-init* C' (*Var* V) = *True*
| *not-init* C' (*LAss* V e) = *not-init* C' e
| *not-init* C' (e·F{D}) = *not-init* C' e
| *not-init* C' (C·$_s F${D}) = *True*
| *not-init* C' (e_1·F{D}:=e_2) = (*not-init* C' e_1 ∧ *not-init* C' e_2)
| *not-init* C' (C·$_s F${D}:=e_2) = *not-init* C' e_2
| *not-init* C' (e·M(es)) = (*not-init* C' e ∧ *not-inits* C' es)
| *not-init* C' (C·$_s M$(es)) = *not-inits* C' es
| *not-init* C' ({ V:T; e}) = *not-init* C' e
| *not-init* C' (e_1;;e_2) = (*not-init* C' e_1 ∧ *not-init* C' e_2)
| *not-init* C' (*if* (b) e_1 *else* e_2) = (*not-init* C' b ∧ *not-init* C' e_1 ∧ *not-init* C' e_2)
| *not-init* C' (*while* (b) e) = (*not-init* C' b ∧ *not-init* C' e)
| *not-init* C' (*throw* e) = *not-init* C' e
| *not-init* C' (*try* e_1 *catch*(C V) e_2) = (*not-init* C' e_1 ∧ *not-init* C' e_2)
| *not-init* C' (*INIT* C (Cs,b) ← e) = (((b ⟶ Cs = *Nil* ∨ C' ≠ *hd* Cs) ∧ C' ∉ *set*(*tl* Cs) ∧ *not-init* C' e)
| *not-init* C' (*RI* (C,e);Cs ← e') = (C' ∉ *set* (C#Cs) ∧ *not-init* C' e ∧ *not-init* C' e')
| *not-inits* C' ([]) = *True*

| *not-inits C′ (e#es) = (not-init C′ e ∧ not-inits C′ es)*

lemma *not-inits-def′*[*simp*]:
not-inits C es ⟹ ∀ e ∈ set es. not-init C e
⟨*proof*⟩

lemma *nsub-RIs-not-inits-aux*: *∀ e ∈ set es. ¬sub-RI e ⟶ not-init C e*
⟹ ¬sub-RIs es ⟹ not-inits C es
⟨*proof*⟩

lemma *nsub-RI-not-init*: *¬sub-RI e ⟹ not-init C e*
⟨*proof*⟩

lemma *nsub-RIs-not-inits*: *¬sub-RIs es ⟹ not-inits C es*
⟨*proof*⟩

1.8.5 Subexpressions

primrec *subexp* :: *′a exp ⇒ ′a exp set* **and** *subexps* :: *′a exp list ⇒ ′a exp set* **where**
subexp(new C) = {}
| *subexp(Cast C e) = {e} ∪ subexp e*
| *subexp(Val v) = {}*
| *subexp(e₁ «bop» e₂) = {e₁, e₂} ∪ subexp e₁ ∪ subexp e₂*
| *subexp(Var V) = {}*
| *subexp(LAss V e) = {e} ∪ subexp e*
| *subexp(e·F{D}) = {e} ∪ subexp e*
| *subexp(C·ₛF{D}) = {}*
| *subexp(e₁·F{D}:=e₂) = {e₁, e₂} ∪ subexp e₁ ∪ subexp e₂*
| *subexp(C·ₛF{D}:=e₂) = {e₂} ∪subexp e₂*
| *subexp(e·M(es)) = {e} ∪ set es ∪ subexp e ∪ subexps es*
| *subexp(C·ₛM(es)) = set es ∪ subexps es*
| *subexp({V:T; e}) = {e} ∪ subexp e*
| *subexp(e₁;;e₂) = {e₁, e₂} ∪ subexp e₁ ∪ subexp e₂*
| *subexp(if (b) e₁ else e₂) = {b, e₁, e₂} ∪ subexp b ∪ subexp e₁ ∪ subexp e₂*
| *subexp(while (b) e) = {b, e} ∪ subexp b ∪ subexp e*
| *subexp(throw e) = {e} ∪ subexp e*
| *subexp(try e₁ catch(C V) e₂) = {e₁, e₂} ∪ subexp e₁ ∪ subexp e₂*
| *subexp(INIT C (Cs,b) ← e) = {e} ∪ subexp e*
| *subexp(RI (C,e);Cs ← e′) = {e, e′} ∪ subexp e ∪ subexp e′*
| *subexps([]) = {}*
| *subexps(e#es) = {e} ∪ subexp e ∪ subexps es*

lemmas *subexp-subexps-induct = subexp.induct subexps.induct*

abbreviation *subexp-of* :: *′a exp ⇒ ′a exp ⇒ bool* **where**
subexp-of e e′ ≡ e ∈ subexp e′

lemma *subexp-size-le*:
(e′ ∈ subexp e ⟶ size e′ < size e) ∧ (e′ ∈ subexps es ⟶ size e′ < size-list size es)
⟨*proof*⟩

lemma *subexps-def2*: *subexps es = set es ∪ (⋃ e ∈ set es. subexp e)* ⟨*proof*⟩
lemma shows *subexp-induct*[*consumes 1*]:

$(\bigwedge e.\ subexp\ e = \{\} \Longrightarrow R\ e) \Longrightarrow (\bigwedge e.\ (\bigwedge e'.\ e' \in subexp\ e \Longrightarrow R\ e') \Longrightarrow R\ e)$
$\Longrightarrow (\bigwedge es.\ (\bigwedge e'.\ e' \in subexps\ es \Longrightarrow R\ e') \Longrightarrow Rs\ es) \Longrightarrow (\forall e'.\ e' \in subexp\ e \longrightarrow R\ e') \wedge R\ e$
and *subexps-induct*[*consumes 1*]:
$(\bigwedge es.\ subexps\ es = \{\} \Longrightarrow Rs\ es) \Longrightarrow (\bigwedge e.\ (\bigwedge e'.\ e' \in subexp\ e \Longrightarrow R\ e') \Longrightarrow R\ e)$
$\Longrightarrow (\bigwedge es.\ (\bigwedge e'.\ e' \in subexps\ es \Longrightarrow R\ e') \Longrightarrow Rs\ es) \Longrightarrow (\forall e'.\ e' \in subexps\ es \longrightarrow R\ e') \wedge Rs\ es$
⟨*proof*⟩

1.8.6 Final expressions

definition *final* :: *'a exp* ⇒ *bool*
where
final e ≡ (∃ *v*. *e* = *Val v*) ∨ (∃ *a*. *e* = *Throw a*)

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

lemma [*simp*]: *final*(*Val v*)⟨*proof*⟩
lemma [*simp*]: *final*(*throw e*) = (∃ *a*. *e* = *addr a*)⟨*proof*⟩
lemma *finalE*: ⟦ *final e*; ⋀*v*. *e* = *Val v* ⟹ *R*; ⋀*a*. *e* = *Throw a* ⟹ *R* ⟧ ⟹ *R*⟨*proof*⟩
lemma *final-fv*[*iff*]: *final e* ⟹ *fv e* = {}
⟨*proof*⟩

lemma *finalsE*:
⟦ *finals es*; ⋀*vs*. *es* = *map Val vs* ⟹ *R*; ⋀*vs a es'*. *es* = *map Val vs* @ *Throw a* # *es'* ⟹ *R* ⟧ ⟹
R⟨*proof*⟩
lemma [*iff*]: *finals* []⟨*proof*⟩
lemma [*iff*]: *finals* (*Val v* # *es*) = *finals es*⟨*proof*⟩
lemma *finals-app-map*[*iff*]: *finals* (*map Val vs* @ *es*) = *finals es*⟨*proof*⟩
lemma [*iff*]: *finals* (*map Val vs*)⟨*proof*⟩
lemma [*iff*]: *finals* (*throw e* # *es*) = (∃ *a*. *e* = *addr a*)⟨*proof*⟩
lemma *not-finals-ConsI*: ¬ *final e* ⟹ ¬ *finals*(*e*#*es*)⟨*proof*⟩
lemma *not-finals-ConsI2*: *e* = *Val v* ⟹ ¬ *finals es* ⟹ ¬ *finals*(*e*#*es*)⟨*proof*⟩

end

1.9 Well-typedness of Jinja expressions

theory *WellType*
imports *../Common/Objects Expr*
begin

type-synonym
env = *vname* ⇀ *ty*

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

WTNew:

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

| *WTCast*:
$[\![\ P,E \vdash e :: Class\ D;\ \ is\text{-}class\ P\ C;\ \ P \vdash C \preceq^* D \vee P \vdash D \preceq^* C\ ]\!]$
$\Longrightarrow P,E \vdash Cast\ C\ e :: Class\ C$

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

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

| *WTBinOpEq*:
$[\![\ P,E \vdash e_1 :: T_1;\ \ P,E \vdash e_2 :: T_2;\ P \vdash T_1 \leq T_2 \vee P \vdash T_2 \leq T_1\ ]\!]$
$\Longrightarrow P,E \vdash e_1\ «Eq»\ e_2 :: Boolean$

| *WTBinOpAdd*:
$[\![\ P,E \vdash e_1 :: Integer;\ \ P,E \vdash e_2 :: Integer\ ]\!]$
$\Longrightarrow P,E \vdash e_1\ «Add»\ e_2 :: Integer$

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

| *WTFAcc*:
$[\![\ P,E \vdash e :: Class\ C;\ \ P \vdash C\ sees\ F,NonStatic{:}T\ in\ D\ ]\!]$
$\Longrightarrow P,E \vdash e{\cdot}F\{D\} :: T$

| *WTSFAcc*:
$[\![\ P \vdash C\ sees\ F,Static{:}T\ in\ D\ ]\!]$
$\Longrightarrow P,E \vdash C{\cdot}_sF\{D\} :: T$

| *WTFAss*:
$[\![\ P,E \vdash e_1 :: Class\ C;\ \ P \vdash C\ sees\ F,NonStatic{:}T\ in\ D;\ \ P,E \vdash e_2 :: T';\ \ P \vdash T' \leq T\ ]\!]$
$\Longrightarrow P,E \vdash e_1{\cdot}F\{D\}{:=}e_2 :: Void$

| *WTSFAss*:
$[\![\ \ P \vdash C\ sees\ F,Static{:}T\ in\ D;\ \ P,E \vdash e_2 :: T';\ \ P \vdash T' \leq T\ ]\!]$
$\Longrightarrow P,E \vdash C{\cdot}_sF\{D\}{:=}e_2 :: Void$

| *WTCall*:
$[\![\ P,E \vdash e :: Class\ C;\ \ P \vdash C\ sees\ M,NonStatic{:}Ts \to T = (pns,body)\ in\ D;$
$\ \ P,E \vdash es\ [::]\ Ts';\ \ P \vdash Ts'\ [\leq]\ Ts\ ]\!]$
$\Longrightarrow P,E \vdash e{\cdot}M(es) :: T$

| *WTSCall*:
$[\![\ P \vdash C\ sees\ M,Static{:}Ts \to T = (pns,body)\ in\ D;$
$\ \ P,E \vdash es\ [::]\ Ts';\ \ P \vdash Ts'\ [\leq]\ Ts;\ M \neq clinit\ ]\!]$
$\Longrightarrow P,E \vdash C{\cdot}_sM(es) :: T$

| *WTBlock*:

⟦ *is-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*:
⟦ $P,E \vdash e :: Boolean$; $P,E \vdash e_1{::}T_1$; $P,E \vdash e_2{::}T_2$;
$P \vdash T_1 \leq T_2 \vee P \vdash T_2 \leq T_1$; $P \vdash T_1 \leq T_2 \longrightarrow T = T_2$; $P \vdash T_2 \leq T_1 \longrightarrow T = T_1$ ⟧
$\Longrightarrow$ $P,E \vdash$ *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* $e :: Void$

| *WTTry*:
⟦ $P,E \vdash e_1 :: T$; $P,E(V \mapsto Class\ C) \vdash e_2 :: T$; *is-class P C* ⟧
$\Longrightarrow$ $P,E \vdash$ *try* e_1 *catch*$(C\ V)$ $e_2 :: T$

— well-typed expression lists

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

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

lemma *init-nwt* [*simp*]:$\neg P,E \vdash INIT\ C\ (Cs,b) \leftarrow e :: T$
⟨*proof*⟩
lemma *ri-nwt* [*simp*]:$\neg P,E \vdash RI(C,e);Cs \leftarrow e' :: T$
⟨*proof*⟩

lemma [*iff*]: $(P,E \vdash [] [::] Ts) = (Ts = [])$⟨*proof*⟩
lemma [*iff*]: $(P,E \vdash e\#es [::] T\#Ts) = (P,E \vdash e :: T \wedge P,E \vdash es [::] Ts)$⟨*proof*⟩
lemma [*iff*]: $(P,E \vdash (e\#es) [::] Ts) =$
$(\exists U\ Us.\ Ts = U\#Us \wedge P,E \vdash e :: U \wedge P,E \vdash es [::] Us)$⟨*proof*⟩
lemma [*iff*]: $\bigwedge Ts.\ (P,E \vdash es_1 @ es_2 [::] Ts) =$
$(\exists Ts_1\ Ts_2.\ Ts = Ts_1 @ Ts_2 \wedge P,E \vdash es_1 [::] Ts_1 \wedge P,E \vdash es_2[::]Ts_2)$⟨*proof*⟩
lemma [*iff*]: $P,E \vdash Val\ v :: T = (typeof\ v = Some\ T)$⟨*proof*⟩
lemma [*iff*]: $P,E \vdash Var\ V :: T = (E\ V = Some\ T)$⟨*proof*⟩
lemma [*iff*]: $P,E \vdash e_1;;e_2 :: T_2 = (\exists T_1.\ P,E \vdash e_1{::}T_1 \wedge P,E \vdash e_2{::}T_2)$⟨*proof*⟩
lemma [*iff*]: $(P,E \vdash \{V{:}T;\ e\} :: T') = (is\text{-}type\ P\ T \wedge P,E(V{\mapsto}T) \vdash e :: T')$⟨*proof*⟩

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

lemma *WT-fv*: $P,E \vdash e :: T \Longrightarrow fv\ e \subseteq dom\ E$

and $P,E \vdash es\ [::]\ Ts \Longrightarrow fvs\ es \subseteq dom\ E\langle proof\rangle$
lemma *WT-nsub-RI*: $P,E \vdash e :: T \Longrightarrow \neg sub\text{-}RI\ e$
and *WTs-nsub-RIs*: $P,E \vdash es\ [::]\ Ts \Longrightarrow \neg sub\text{-}RIs\ es\langle proof\rangle$

1.10 Runtime Well-typedness

theory *WellTypeRT*
imports *WellType*
begin

inductive
WTrt :: *J-prog* ⇒ *heap* ⇒ *sheap* ⇒ *env* ⇒ *expr* ⇒ *ty* ⇒ *bool*
and *WTrts* :: *J-prog* ⇒ *heap* ⇒ *sheap* ⇒ *env* ⇒ *expr list* ⇒ *ty list* ⇒ *bool*
and *WTrt2* :: [*J-prog*,*env*,*heap*,*sheap*,*expr*,*ty*] ⇒ *bool*
(‹-,-,-,- ⊢ - : -› [*51*,*51*,*51*,*51*]*50*)
and *WTrts2* :: [*J-prog*,*env*,*heap*,*sheap*,*expr list*, *ty list*] ⇒ *bool*
(‹-,-,-,- ⊢ - [:] -› [*51*,*51*,*51*,*51*]*50*)
for *P* :: *J-prog* **and** *h* :: *heap* **and** *sh* :: *sheap*
where

$P,E,h,sh \vdash e : T \equiv WTrt\ P\ h\ sh\ E\ e\ T$
| $P,E,h,sh \vdash es[:]Ts \equiv WTrts\ P\ h\ sh\ E\ es\ Ts$

| *WTrtNew*:
$is\text{-}class\ P\ C \Longrightarrow$
$P,E,h,sh \vdash new\ C : Class\ C$

| *WTrtCast*:
$[\![\ P,E,h,sh \vdash e : T;\ is\text{-}refT\ T;\ is\text{-}class\ P\ C\ ]\!]$
$\Longrightarrow P,E,h,sh \vdash Cast\ C\ e : Class\ C$

| *WTrtVal*:
$typeof_h\ v = Some\ T \Longrightarrow$
$P,E,h,sh \vdash Val\ v : T$

| *WTrtVar*:
$E\ V = Some\ T \Longrightarrow$
$P,E,h,sh \vdash Var\ V : T$

| *WTrtBinOpEq*:
$[\![\ P,E,h,sh \vdash e_1 : T_1;\ \ P,E,h,sh \vdash e_2 : T_2\ ]\!]$
$\Longrightarrow P,E,h,sh \vdash e_1\ «Eq»\ e_2 : Boolean$

| *WTrtBinOpAdd*:
$[\![\ P,E,h,sh \vdash e_1 : Integer;\ \ P,E,h,sh \vdash e_2 : Integer\ ]\!]$
$\Longrightarrow P,E,h,sh \vdash e_1\ «Add»\ e_2 : Integer$

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

| *WTrtFAcc*:
$[\![\ P,E,h,sh \vdash e : Class\ C;\ P \vdash C\ has\ F,NonStatic{:}T\ in\ D\ ]\!] \Longrightarrow$

$P,E,h,sh \vdash e{\cdot}F\{D\} : T$

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

| *WTrtSFAcc*:
$[\![P \vdash C\ has\ F,Static{:}T\ in\ D]\!] \Longrightarrow$
$P,E,h,sh \vdash C{\cdot}_s F\{D\} : T$

| *WTrtFAss*:
$[\![P,E,h,sh \vdash e_1 : Class\ C;\ \ P \vdash C\ has\ F,NonStatic{:}T\ in\ D;\ P,E,h,sh \vdash e_2 : T_2;\ \ P \vdash T_2 \leq T]\!]$
$\Longrightarrow P,E,h,sh \vdash e_1{\cdot}F\{D\}{:=}e_2 : Void$

| *WTrtFAssNT*:
$[\![P,E,h,sh \vdash e_1{:}NT;\ P,E,h,sh \vdash e_2 : T_2]\!]$
$\Longrightarrow P,E,h,sh \vdash e_1{\cdot}F\{D\}{:=}e_2 : Void$

| *WTrtSFAss*:
$[\![P,E,h,sh \vdash e_2 : T_2;\ P \vdash C\ has\ F,Static{:}T\ in\ D;\ P \vdash T_2 \leq T]\!]$
$\Longrightarrow P,E,h,sh \vdash C{\cdot}_s F\{D\}{:=}e_2 : Void$

| *WTrtCall*:
$[\![P,E,h,sh \vdash e : Class\ C;\ P \vdash C\ sees\ M,NonStatic{:}Ts \to T = (pns,body)\ in\ D;$
$\quad P,E,h,sh \vdash es\ [:]\ Ts';\ P \vdash Ts'\ [\leq]\ Ts]\!]$
$\Longrightarrow P,E,h,sh \vdash e{\cdot}M(es) : T$

| *WTrtCallNT*:
$[\![P,E,h,sh \vdash e : NT;\ P,E,h,sh \vdash es\ [:]\ Ts]\!]$
$\Longrightarrow P,E,h,sh \vdash e{\cdot}M(es) : T$

| *WTrtSCall*:
$[\![P \vdash C\ sees\ M,Static{:}Ts \to T = (pns,body)\ in\ D;$
$\quad P,E,h,sh \vdash es\ [:]\ Ts';\ P \vdash Ts'\ [\leq]\ Ts;$
$\quad M = clinit \longrightarrow sh\ D = \lfloor(sfs,Processing)\rfloor \wedge es = map\ Val\ vs]\!]$
$\Longrightarrow P,E,h,sh \vdash C{\cdot}_s M(es) : T$

| *WTrtBlock*:
$P,E(V{\mapsto}T),h,sh \vdash e : T' \Longrightarrow$
$P,E,h,sh \vdash \{V{:}T;\ e\} : T'$

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

| *WTrtCond*:
$[\![P,E,h,sh \vdash e : Boolean;\ \ P,E,h,sh \vdash e_1{:}T_1;\ \ P,E,h,sh \vdash e_2{:}T_2;$
$\quad P \vdash T_1 \leq T_2 \vee P \vdash T_2 \leq T_1;\ P \vdash T_1 \leq T_2 \longrightarrow T = T_2;\ P \vdash T_2 \leq T_1 \longrightarrow T = T_1]\!]$
$\Longrightarrow P,E,h,sh \vdash if\ (e)\ e_1\ else\ e_2 : T$

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

| *WTrtThrow*:
$[\![P,E,h,sh \vdash e : T_r;\ is\text{-}refT\ T_r]\!] \Longrightarrow$
$P,E,h,sh \vdash throw\ e : T$

| *WTrtTry*:
$[\![P,E,h,sh \vdash e_1 : T_1;\ P,E(V \mapsto Class\ C),h,sh \vdash e_2 : T_2;\ P \vdash T_1 \leq T_2]\!]$
$\Longrightarrow P,E,h,sh \vdash try\ e_1\ catch(C\ V)\ e_2 : T_2$

| *WTrtInit*:
$[\![P,E,h,sh \vdash e : T;\ \forall C' \in set\ (C\#Cs).\ is\text{-}class\ P\ C';\ \neg sub\text{-}RI\ e;$
$\forall C' \in set\ (tl\ Cs).\ \exists sfs.\ sh\ C' = \lfloor(sfs,Processing)\rfloor;$
$b \longrightarrow (\forall C' \in set\ Cs.\ \exists sfs.\ sh\ C' = \lfloor(sfs,Processing)\rfloor);$
$distinct\ Cs;\ supercls\text{-}lst\ P\ Cs]\!]$
$\Longrightarrow P,E,h,sh \vdash INIT\ C\ (Cs,\ b) \leftarrow e : T$

| *WTrtRI*:
$[\![P,E,h,sh \vdash e : T;\ P,E,h,sh \vdash e' : T';\ \forall C' \in set\ (C\#Cs).\ is\text{-}class\ P\ C';\ \neg sub\text{-}RI\ e';$
$\forall C' \in set\ (C\#Cs).\ not\text{-}init\ C'\ e;$
$\forall C' \in set\ Cs.\ \exists sfs.\ sh\ C' = \lfloor(sfs,Processing)\rfloor;$
$\exists sfs.\ sh\ C = \lfloor(sfs,\ Processing)\rfloor \vee (sh\ C = \lfloor(sfs,\ Error)\rfloor \wedge e = THROW\ NoClassDefFoundError);$
$distinct\ (C\#Cs);\ supercls\text{-}lst\ P\ (C\#Cs)]\!]$
$\Longrightarrow P,E,h,sh \vdash RI(C,\ e);Cs \leftarrow e' : T'$

— well-typed expression lists

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

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

1.10.1 Easy consequences

lemma [*iff*]: $(P,E,h,sh \vdash [] [:] Ts) = (Ts = [])\langle proof\rangle$
lemma [*iff*]: $(P,E,h,sh \vdash e\#es\ [:]\ T\#Ts) = (P,E,h,sh \vdash e : T \wedge P,E,h,sh \vdash es\ [:]\ Ts)\langle proof\rangle$
lemma [*iff*]: $(P,E,h,sh \vdash (e\#es)\ [:]\ Ts) =$
$(\exists U\ Us.\ Ts = U\#Us \wedge P,E,h,sh \vdash e : U \wedge P,E,h,sh \vdash es\ [:]\ Us)\langle proof\rangle$
lemma [*simp*]: $\forall Ts.\ (P,E,h,sh \vdash es_1\ @\ es_2\ [:]\ Ts) =$
$(\exists Ts_1\ Ts_2.\ Ts = Ts_1\ @\ Ts_2 \wedge P,E,h,sh \vdash es_1\ [:]\ Ts_1\ \&\ P,E,h,sh \vdash es_2[:]Ts_2)\langle proof\rangle$
lemma [*iff*]: $P,E,h,sh \vdash Val\ v : T = (typeof_h\ v = Some\ T)\langle proof\rangle$
lemma [*iff*]: $P,E,h,sh \vdash Var\ v : T = (E\ v = Some\ T)\langle proof\rangle$
lemma [*iff*]: $P,E,h,sh \vdash e_1;;e_2 : T_2 = (\exists T_1.\ P,E,h,sh \vdash e_1:T_1 \wedge P,E,h,sh \vdash e_2:T_2)\langle proof\rangle$
lemma [*iff*]: $P,E,h,sh \vdash \{V:T;\ e\} : T' = (P,E(V\mapsto T),h,sh \vdash e : T')\langle proof\rangle$

1.10.2 Some interesting lemmas

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

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

lemma *WTrt-env-mono*:

$P,E,h,sh \vdash e : T \Longrightarrow (\bigwedge E'.\ E \subseteq_m E' \Longrightarrow P,E',h,sh \vdash e : T)$ **and**
$P,E,h,sh \vdash es\ [:]\ Ts \Longrightarrow (\bigwedge E'.\ E \subseteq_m E' \Longrightarrow P,E',h,sh \vdash es\ [:]\ Ts) \langle proof \rangle$

lemma *WTrt-hext-mono*: $P,E,h,sh \vdash e : T \Longrightarrow h \trianglelefteq h' \Longrightarrow P,E,h',sh \vdash e : T$
and *WTrts-hext-mono*: $P,E,h,sh \vdash es\ [:]\ Ts \Longrightarrow h \trianglelefteq h' \Longrightarrow P,E,h',sh \vdash es\ [:]\ Ts \langle proof \rangle$
lemma *WTrt-shext-mono*: $P,E,h,sh \vdash e : T \Longrightarrow sh \trianglelefteq_s sh' \Longrightarrow \neg sub\text{-}RI\ e \Longrightarrow P,E,h,sh' \vdash e : T$
and *WTrts-shext-mono*: $P,E,h,sh \vdash es\ [:]\ Ts \Longrightarrow sh \trianglelefteq_s sh' \Longrightarrow \neg sub\text{-}RIs\ es \Longrightarrow P,E,h,sh' \vdash es\ [:]\ Ts \langle proof \rangle$
lemma *WTrt-hext-shext-mono*: $P,E,h,sh \vdash e : T$
$\Longrightarrow h \trianglelefteq h' \Longrightarrow sh \trianglelefteq_s sh' \Longrightarrow \neg sub\text{-}RI\ e \Longrightarrow P,E,h',sh' \vdash e : T$
$\langle proof \rangle$

lemma *WTrts-hext-shext-mono*: $P,E,h,sh \vdash es\ [:]\ Ts$
$\Longrightarrow h \trianglelefteq h' \Longrightarrow sh \trianglelefteq_s sh' \Longrightarrow \neg sub\text{-}RIs\ es \Longrightarrow P,E,h',sh' \vdash es\ [:]\ Ts$
$\langle proof \rangle$

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

1.11 Program State

theory *State* **imports** *../Common/Exceptions* **begin**

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

definition *hp* :: *state* ⇒ *heap*
where
hp ≡ *fst*
definition *lcl* :: *state* ⇒ *locals*
where
lcl ≡ *fst* ∘ *snd*
definition *shp* :: *state* ⇒ *sheap*
where
shp ≡ *snd* ∘ *snd*

end

1.12 System Classes

theory *SystemClasses*
imports *Decl Exceptions*
begin

This theory provides definitions for the *Object* class, and the system exceptions.

definition *ObjectC* :: *'m cdecl*
where
ObjectC ≡ (*Object*, (*undefined*,[],[]))

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

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

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

definition *NoClassDefFoundC* :: *'m cdecl*
where
NoClassDefFoundC ≡ (*NoClassDefFoundError*, (*Object*,[],[]))

definition *IncompatibleClassChangeC* :: *'m cdecl*
where
IncompatibleClassChangeC ≡ (*IncompatibleClassChangeError*, (*Object*,[],[]))

definition *NoSuchFieldC* :: *'m cdecl*
where
NoSuchFieldC ≡ (*NoSuchFieldError*, (*Object*,[],[]))

definition *NoSuchMethodC* :: *'m cdecl*
where
NoSuchMethodC ≡ (*NoSuchMethodError*, (*Object*,[],[]))

definition *SystemClasses* :: *'m cdecl list*
where
SystemClasses ≡ [*ObjectC*, *NullPointerC*, *ClassCastC*, *OutOfMemoryC*, *NoClassDefFoundC*, *IncompatibleClassChangeC*, *NoSuchFieldC*, *NoSuchMethodC*]

end

1.13 Generic Well-formedness of programs

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

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

Because Jinja does not have method overloading, its policy for method overriding is the classical one: *covariant in the result type but contravariant in the argument types.* This means the result type of the overriding method becomes more specific, the argument types become more general.

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

definition *wf-fdecl* :: *'m prog* ⇒ *fdecl* ⇒ *bool*
where
wf-fdecl P ≡ λ(*F*,*b*,*T*). *is-type P T*

definition *wf-mdecl* :: *'m wf-mdecl-test* ⇒ *'m wf-mdecl-test*

where
wf-mdecl wf-md P C $\equiv \lambda(M,b,Ts,T,m)$.
$(\forall\, T{\in}set\ Ts.\ is\text{-}type\ P\ T) \wedge is\text{-}type\ P\ T \wedge wf\text{-}md\ P\ C\ (M,b,Ts,T,m)$

definition *wf-clinit* :: *'m mdecl list* ⇒ *bool* **where**
wf-clinit ms = $(\exists\, m.\ (clinit,Static,[],Void,m){\in}set\ ms)$

definition *wf-cdecl* :: *'m wf-mdecl-test* ⇒ *'m prog* ⇒ *'m cdecl* ⇒ *bool*
where
wf-cdecl wf-md P ≡ $\lambda(C,(D,fs,ms))$.
$(\forall f{\in}set\ fs.\ wf\text{-}fdecl\ P\ f) \wedge\ distinct\text{-}fst\ fs \wedge$
$(\forall\, m{\in}set\ ms.\ wf\text{-}mdecl\ wf\text{-}md\ P\ C\ m) \wedge\ distinct\text{-}fst\ ms \wedge$
$(C \neq Object \longrightarrow$
$is\text{-}class\ P\ D \wedge \neg\ P \vdash D \preceq^* C \wedge$
$(\forall\,(M,b,Ts,T,m){\in}set\ ms.$
$\forall\, D'\ b'\ Ts'\ T'\ m'.\ P \vdash D\ sees\ M,b':Ts' \to T' = m'\ in\ D' \longrightarrow$
$b = b' \wedge P \vdash Ts'\ [\leq]\ Ts \wedge P \vdash T \leq T'))\ \wedge$
wf-clinit ms

definition *wf-syscls* :: *'m prog* ⇒ *bool*
where
wf-syscls P ≡ $\{Object\} \cup sys\text{-}xcpts \subseteq set(map\ fst\ P)$

definition *wf-prog* :: *'m wf-mdecl-test* ⇒ *'m prog* ⇒ *bool*
where
wf-prog wf-md P ≡ *wf-syscls P* ∧ $(\forall\, c \in set\ P.\ wf\text{-}cdecl\ wf\text{-}md\ P\ c) \wedge distinct\text{-}fst\ P$

1.13.1 Well-formedness lemmas

lemma *class-wf*:
$[\![class\ P\ C = Some\ c;\ wf\text{-}prog\ wf\text{-}md\ P]\!] \Longrightarrow wf\text{-}cdecl\ wf\text{-}md\ P\ (C,c)\langle proof\rangle$

lemma *class-Object* [*simp*]:
wf-prog wf-md P ⟹ $\exists\, C\ fs\ ms.\ class\ P\ Object = Some\ (C,fs,ms)\langle proof\rangle$

lemma *is-class-Object* [*simp*]:
wf-prog wf-md P ⟹ *is-class P Object*⟨*proof*⟩
lemma *is-class-supclass*:
assumes *wf*: *wf-prog wf-md P* **and** *sub*: $P \vdash C \preceq^* D$
shows *is-class P C* ⟹ *is-class P D*⟨*proof*⟩
lemma *is-class-xcpt*:
$[\![\ C \in sys\text{-}xcpts;\ wf\text{-}prog\ wf\text{-}md\ P\ ]\!] \Longrightarrow is\text{-}class\ P\ C\langle proof\rangle$

lemma *subcls1-wfD*:
assumes *sub1*: $P \vdash C \prec^1 D$ **and** *wf*: *wf-prog wf-md P*
shows $D \neq C \wedge (D,C) \notin (subcls1\ P)^+\langle proof\rangle$

lemma *wf-cdecl-supD*:
$[\![wf\text{-}cdecl\ wf\text{-}md\ P\ (C,D,r);\ C \neq Object]\!] \Longrightarrow is\text{-}class\ P\ D\langle proof\rangle$

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

lemma *subcls-irrefl*:

⟦ *wf-prog wf-md P*; (*C*,*D*) ∈ (*subcls1 P*)$^+$ ⟧ ⟹ *C* ≠ *D*⟨*proof*⟩

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

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

lemma *single-valued-subcls1*:
wf-prog wf-md G ⟹ *single-valued* (*subcls1 G*)⟨*proof*⟩

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

lemma *subcls1-induct-aux*:
assumes *is-class P C* **and** *wf*: *wf-prog wf-md P* **and** *QObj*: *Q Object*
shows
⟦ ⋀*C D fs ms*.
⟦ *C* ≠ *Object*; *is-class P C*; *class P C* = *Some* (*D*,*fs*,*ms*) ∧
wf-cdecl wf-md P (*C*,*D*,*fs*,*ms*) ∧ *P* ⊢ *C* ≺^{1} *D* ∧ *is-class P D* ∧ *Q D*⟧ ⟹ *Q C* ⟧
⟹ *Q C*⟨*proof*⟩

lemma *subcls1-induct* [*consumes 2*, *case-names Object Subcls*]:
⟦ *wf-prog wf-md P*; *is-class P C*; *Q Object*;
⋀*C D*. ⟦*C* ≠ *Object*; *P* ⊢ *C* ≺^{1} *D*; *is-class P D*; *Q D*⟧ ⟹ *Q C* ⟧
⟹ *Q C*⟨*proof*⟩

lemma *subcls-C-Object*:
assumes *class*: *is-class P C* **and** *wf*: *wf-prog wf-md P*
shows *P* ⊢ *C* ≼* *Object*⟨*proof*⟩

lemma *is-type-pTs*:
assumes *wf-prog wf-md P* **and** (*C*,*S*,*fs*,*ms*) ∈ *set P* **and** (*M*,*b*,*Ts*,*T*,*m*) ∈ *set ms*
shows *set Ts* ⊆ *types P*⟨*proof*⟩
lemma *wf-supercls-distinct-app*:
assumes *wf*:*wf-prog wf-md P*
and *nObj*: *C* ≠ *Object* **and** *cls*: *class P C* = ⌊(*D*, *fs*, *ms*)⌋
and *super*: *supercls-lst P* (*C*#*Cs*) **and** *dist*: *distinct* (*C*#*Cs*)
shows *distinct* (*D*#*C*#*Cs*)
⟨*proof*⟩

1.13.2 Well-formedness and method lookup

lemma *sees-wf-mdecl*:
assumes *wf*: *wf-prog wf-md P* **and** *sees*: *P* ⊢ *C sees M*,*b*:*Ts*→*T* = *m in D*
shows *wf-mdecl wf-md P D* (*M*,*b*,*Ts*,*T*,*m*)⟨*proof*⟩
lemma *sees-method-mono* [*rule-format* (*no-asm*)]:
assumes *sub*: *P* ⊢ *C*′ ≼* *C* **and** *wf*: *wf-prog wf-md P*
shows ∀ *D b Ts T m*. *P* ⊢ *C sees M*,*b*:*Ts*→*T* = *m in D* ⟶
(∃ *D*′ *Ts*′ *T*′ *m*′. *P* ⊢ *C*′ *sees M*,*b*:*Ts*′→*T*′ = *m*′ *in D*′ ∧ *P* ⊢ *Ts* [≤] *Ts*′ ∧ *P* ⊢ *T*′ ≤ *T*)⟨*proof*⟩

lemma *sees-method-mono2*:
⟦ *P* ⊢ *C*′ ≼* *C*; *wf-prog wf-md P*;
P ⊢ *C sees M*,*b*:*Ts*→*T* = *m in D*; *P* ⊢ *C*′ *sees M*,*b*′:*Ts*′→*T*′ = *m*′ *in D*′ ⟧

$\Longrightarrow b = b' \wedge P \vdash Ts\ [\leq]\ Ts' \wedge P \vdash T' \leq T \langle proof \rangle$
lemma *mdecls-visible*:
assumes *wf*: *wf-prog wf-md P* **and** *class*: *is-class P C*
shows $\bigwedge D\ fs\ ms.\ class\ P\ C = Some(D,fs,ms)$
$\Longrightarrow \exists Mm.\ P \vdash C\ sees\text{-}methods\ Mm \wedge (\forall (M,b,Ts,T,m) \in set\ ms.\ Mm\ M = Some((b,Ts,T,m),C)) \langle proof \rangle$
lemma *mdecl-visible*:
assumes *wf*: *wf-prog wf-md P* **and** *C*: $(C,S,fs,ms) \in set\ P$ **and** *m*: $(M,b,Ts,T,m) \in set\ ms$
shows $P \vdash C\ sees\ M,b:Ts \rightarrow T = m\ in\ C \langle proof \rangle$

lemma *Call-lemma*:
assumes *sees*: $P \vdash C\ sees\ M,b:Ts \rightarrow T = m\ in\ D$ **and** *sub*: $P \vdash C' \preceq^* C$ **and** *wf*: *wf-prog wf-md P*
shows $\exists D'\ Ts'\ T'\ m'.$
$P \vdash C'\ sees\ M,b:Ts' \rightarrow T' = m'\ in\ D' \wedge P \vdash Ts\ [\leq]\ Ts' \wedge P \vdash T' \leq T \wedge P \vdash C' \preceq^* D'$
$\wedge\ is\text{-}type\ P\ T' \wedge (\forall T \in set\ Ts'.\ is\text{-}type\ P\ T) \wedge wf\text{-}md\ P\ D'\ (M,b,Ts',T',m') \langle proof \rangle$

lemma *wf-prog-lift*:
assumes *wf*: *wf-prog* $(\lambda P\ C\ bd.\ A\ P\ C\ bd)\ P$
and *rule*:
$\bigwedge$*wf-md C M b Ts C T m bd.*
wf-prog wf-md P $\Longrightarrow$
$P \vdash C\ sees\ M,b:Ts \rightarrow T = m\ in\ C \Longrightarrow$
$set\ Ts \subseteq types\ P \Longrightarrow$
$bd = (M,b,Ts,T,m) \Longrightarrow$
A P C bd $\Longrightarrow$
B P C bd
shows *wf-prog* $(\lambda P\ C\ bd.\ B\ P\ C\ bd)\ P \langle proof \rangle$
lemma *wf-sees-clinit*:
assumes *wf*:*wf-prog wf-md P* **and** *ex*: *class P C = Some a*
shows $\exists m.\ P \vdash C\ sees\ clinit,Static:[] \rightarrow Void = m\ in\ C$
$\langle proof \rangle$

lemma *wf-sees-clinit1*:
assumes *wf*:*wf-prog wf-md P* **and** *ex*: *class P C = Some a*
and $P \vdash C\ sees\ clinit,b:Ts \rightarrow T = m\ in\ D$
shows $b = Static \wedge Ts = [] \wedge T = Void \wedge D = C$
$\langle proof \rangle$

lemma *wf-NonStatic-nclinit*:
assumes *wf*: *wf-prog wf-md P* **and** *meth*: $P \vdash C\ sees\ M,NonStatic:Ts \rightarrow T = (mxs,mxl,ins,xt)\ in\ D$
shows $M \neq clinit$
$\langle proof \rangle$

1.13.3 Well-formedness and field lookup

lemma *wf-Fields-Ex*:
assumes *wf*: *wf-prog wf-md P* **and** *is-class P C*
shows $\exists FDTs.\ P \vdash C\ has\text{-}fields\ FDTs \langle proof \rangle$

lemma *has-fields-types*:
$[\![P \vdash C\ has\text{-}fields\ FDTs;\ (FD,b,T) \in set\ FDTs;\ wf\text{-}prog\ wf\text{-}md\ P]\!] \Longrightarrow is\text{-}type\ P\ T \langle proof \rangle$
lemma *sees-field-is-type*:
$[\![P \vdash C\ sees\ F,b:T\ in\ D;\ wf\text{-}prog\ wf\text{-}md\ P]\!] \Longrightarrow is\text{-}type\ P\ T \langle proof \rangle$

lemma *wf-syscls*:
set SystemClasses $\subseteq$ *set P* $\Longrightarrow$ *wf-syscls P*⟨*proof*⟩

1.13.4 Well-formedness and subclassing

lemma *wf-subcls-nCls*:
assumes *wf*: *wf-prog wf-md P* **and** *ns*: $\neg$ *is-class P C*
shows ⟦ *P* $\vdash$ *D* $\preceq^*$ *D'*; *D* $\neq$ *C* ⟧ $\Longrightarrow$ *D'* $\neq$ *C*
⟨*proof*⟩

lemma *wf-subcls-nCls'*:
assumes *wf*: *wf-prog wf-md P* **and** *ns*: $\neg$*is-class P* C_0
shows $\bigwedge$*cd D'*. *cd* $\in$ *set P* $\Longrightarrow$ $\neg$*P* $\vdash$ *fst cd* $\preceq^*$ C_0
⟨*proof*⟩

lemma *wf-nclass-nsub*:
⟦ *wf-prog wf-md P*; *is-class P C*; $\neg$*is-class P C'* ⟧ $\Longrightarrow$ $\neg$*P* $\vdash$ *C* $\preceq^*$ *C'*
⟨*proof*⟩

lemma *wf-sys-xcpt-nsub-Start*:
assumes *wf*: *wf-prog wf-md P* **and** *ns*: $\neg$*is-class P Start* **and** *sx*: *C* $\in$ *sys-xcpts*
shows $\neg$*P* $\vdash$ *C* $\preceq^*$ *Start*
⟨*proof*⟩

end

1.14 Weak well-formedness of Jinja programs

theory *WWellForm* **imports** *../Common/ WellForm Expr* **begin**

definition *wwf-J-mdecl* :: *J-prog* $\Rightarrow$ *cname* $\Rightarrow$ *J-mb mdecl* $\Rightarrow$ *bool*
where
wwf-J-mdecl P C $\equiv$ λ(*M*,*b*,*Ts*,*T*,(*pns*,*body*)).
length Ts = *length pns* $\wedge$ *distinct pns* $\wedge$ $\neg$*sub-RI body* $\wedge$
(*case b of*
NonStatic $\Rightarrow$ *this* $\notin$ *set pns* $\wedge$ *fv body* $\subseteq$ {*this*} $\cup$ *set pns*
| *Static* $\Rightarrow$ *fv body* $\subseteq$ *set pns*)

lemma *wwf-J-mdecl-NonStatic*[*simp*]:
wwf-J-mdecl P C (*M*,*NonStatic*,*Ts*,*T*,*pns*,*body*) =
(*length Ts* = *length pns* $\wedge$ *distinct pns* $\wedge$ $\neg$*sub-RI body* $\wedge$ *this* $\notin$ *set pns* $\wedge$ *fv body* $\subseteq$ {*this*} $\cup$ *set pns*)⟨*proof*⟩
lemma *wwf-J-mdecl-Static*[*simp*]:
wwf-J-mdecl P C (*M*,*Static*,*Ts*,*T*,*pns*,*body*) =
(*length Ts* = *length pns* $\wedge$ *distinct pns* $\wedge$ $\neg$*sub-RI body* $\wedge$ *fv body* $\subseteq$ *set pns*)⟨*proof*⟩
abbreviation
wwf-J-prog :: *J-prog* $\Rightarrow$ *bool* **where**
wwf-J-prog $\equiv$ *wf-prog wwf-J-mdecl*

lemma *sees-wwf-nsub-RI*:
⟦ *wwf-J-prog P*; *P* $\vdash$ *C sees M*,*b* : *Ts*$\rightarrow$*T* = (*pns*, *body*) *in D* ⟧ $\Longrightarrow$ $\neg$*sub-RI body*⟨*proof*⟩
end

1.15 Big Step Semantics

theory *BigStep* **imports** *Expr State WWellForm* **begin**

inductive
  eval :: *J-prog* ⇒ *expr* ⇒ *state* ⇒ *expr* ⇒ *state* ⇒ *bool*
        (‹- ⊢ ((1⟨-,/-⟩) ⇒/ (1⟨-,/-⟩))› [51,0,0,0,0] 81)
  and *evals* :: *J-prog* ⇒ *expr list* ⇒ *state* ⇒ *expr list* ⇒ *state* ⇒ *bool*
         (‹- ⊢ ((1⟨-,/-⟩) [⇒]/ (1⟨-,/-⟩))› [51,0,0,0,0] 81)
  for *P* :: *J-prog*
where

  New:
  ⟦ *sh C* = *Some* (*sfs*, *Done*); *new-Addr h* = *Some a*;
    P ⊢ *C has-fields FDTs*; *h'* = *h*(*a*↦*blank P C*) ⟧
  ⟹ *P* ⊢ ⟨*new C*,(*h*,*l*,*sh*)⟩ ⇒ ⟨*addr a*,(*h'*,*l*,*sh*)⟩

| *NewFail*:
  ⟦ *sh C* = *Some* (*sfs*, *Done*); *new-Addr h* = *None*; *is-class P C* ⟧ ⟹
  P ⊢ ⟨*new C*, (*h*,*l*,*sh*)⟩ ⇒ ⟨*THROW OutOfMemory*,(*h*,*l*,*sh*)⟩

| *NewInit*:
  ⟦ ∄*sfs*. *sh C* = *Some* (*sfs*, *Done*); *P* ⊢ ⟨*INIT C* ([*C*],*False*) ← *unit*,(*h*,*l*,*sh*)⟩ ⇒ ⟨*Val v'*,(*h'*,*l'*,*sh'*)⟩;
    new-Addr h' = *Some a*; *P* ⊢ *C has-fields FDTs*; *h''* = *h'*(*a*↦*blank P C*) ⟧
  ⟹ *P* ⊢ ⟨*new C*,(*h*,*l*,*sh*)⟩ ⇒ ⟨*addr a*,(*h''*,*l'*,*sh'*)⟩

| *NewInitOOM*:
  ⟦ ∄*sfs*. *sh C* = *Some* (*sfs*, *Done*); *P* ⊢ ⟨*INIT C* ([*C*],*False*) ← *unit*,(*h*,*l*,*sh*)⟩ ⇒ ⟨*Val v'*,(*h'*,*l'*,*sh'*)⟩;
    new-Addr h' = *None*; *is-class P C* ⟧
  ⟹ *P* ⊢ ⟨*new C*,(*h*,*l*,*sh*)⟩ ⇒ ⟨*THROW OutOfMemory*,(*h'*,*l'*,*sh'*)⟩

| *NewInitThrow*:
  ⟦ ∄*sfs*. *sh C* = *Some* (*sfs*, *Done*); *P* ⊢ ⟨*INIT C* ([*C*],*False*) ← *unit*,(*h*,*l*,*sh*)⟩ ⇒ ⟨*throw a*,*s'*⟩;
    is-class P C ⟧
  ⟹ *P* ⊢ ⟨*new C*,(*h*,*l*,*sh*)⟩ ⇒ ⟨*throw a*,*s'*⟩

| *Cast*:
  ⟦ *P* ⊢ ⟨*e*,*s*$_0$⟩ ⇒ ⟨*addr a*,(*h*,*l*,*sh*)⟩; *h a* = *Some*(*D*,*fs*); *P* ⊢ *D* ⪯* *C* ⟧
  ⟹ *P* ⊢ ⟨*Cast C e*,*s*$_0$⟩ ⇒ ⟨*addr a*,(*h*,*l*,*sh*)⟩

| *CastNull*:
  P ⊢ ⟨*e*,*s*$_0$⟩ ⇒ ⟨*null*,*s*$_1$⟩ ⟹
  P ⊢ ⟨*Cast C e*,*s*$_0$⟩ ⇒ ⟨*null*,*s*$_1$⟩

| *CastFail*:
  ⟦ *P* ⊢ ⟨*e*,*s*$_0$⟩⇒ ⟨*addr a*,(*h*,*l*,*sh*)⟩; *h a* = *Some*(*D*,*fs*); ¬ *P* ⊢ *D* ⪯* *C* ⟧
  ⟹ *P* ⊢ ⟨*Cast C e*,*s*$_0$⟩ ⇒ ⟨*THROW ClassCast*,(*h*,*l*,*sh*)⟩

| *CastThrow*:
  P ⊢ ⟨*e*,*s*$_0$⟩ ⇒ ⟨*throw e'*,*s*$_1$⟩ ⟹
  P ⊢ ⟨*Cast C e*,*s*$_0$⟩ ⇒ ⟨*throw e'*,*s*$_1$⟩

| *Val*:
  P ⊢ ⟨*Val v*,*s*⟩ ⇒ ⟨*Val v*,*s*⟩

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

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

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

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

| *LAss*:
$[\![P \vdash \langle e,s_0\rangle \Rightarrow \langle Val\ v,(h,l,sh)\rangle;\ l' = l(V \mapsto v)]\!]$
$\Longrightarrow P \vdash \langle V{:=}e,s_0\rangle \Rightarrow \langle unit,(h,l',sh)\rangle$

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

| *FAcc*:
$[\![P \vdash \langle e,s_0\rangle \Rightarrow \langle addr\ a,(h,l,sh)\rangle;\ h\ a = Some(C,fs);$
$P \vdash C\ has\ F,NonStatic{:}t\ in\ D;$
$fs(F,D) = Some\ v]\!]$
$\Longrightarrow P \vdash \langle e{\cdot}F\{D\},s_0\rangle \Rightarrow \langle Val\ v,(h,l,sh)\rangle$

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

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

| *FAccNone*:
$[\![P \vdash \langle e,s_0\rangle \Rightarrow \langle addr\ a,(h,l,sh)\rangle;\ h\ a = Some(C,fs);$
$\neg(\exists\ b\ t.\ P \vdash C\ has\ F,b{:}t\ in\ D)]\!]$
$\Longrightarrow P \vdash \langle e{\cdot}F\{D\},s_0\rangle \Rightarrow \langle THROW\ NoSuchFieldError,(h,l,sh)\rangle$

| *FAccStatic*:
$[\![P \vdash \langle e,s_0\rangle \Rightarrow \langle addr\ a,(h,l,sh)\rangle;\ h\ a = Some(C,fs);$
$P \vdash C\ has\ F,Static{:}t\ in\ D]\!]$
$\Longrightarrow P \vdash \langle e{\cdot}F\{D\},s_0\rangle \Rightarrow \langle THROW\ IncompatibleClassChangeError,(h,l,sh)\rangle$

| *SFAcc*:
$[\![P \vdash C\ has\ F,Static{:}t\ in\ D;$
$sh\ D = Some\ (sfs,Done);$
$sfs\ F = Some\ v]\!]$
$\Longrightarrow P \vdash \langle C{\cdot}_sF\{D\},(h,l,sh)\rangle \Rightarrow \langle Val\ v,(h,l,sh)\rangle$

| *SFAccInit*:
⟦ $P \vdash C$ *has* F,*Static*:t *in* D;
$\nexists$ *sfs*. *sh* D = *Some* (*sfs*,*Done*); $P \vdash \langle INIT\ D\ ([D],False) \leftarrow unit,(h,l,sh)\rangle \Rightarrow \langle Val\ v',(h',l',sh')\rangle$;
$sh'\ D$ = *Some* (*sfs*,i);
sfs F = *Some* v ⟧
$\Longrightarrow P \vdash \langle C{\cdot}_s F\{D\},(h,l,sh)\rangle \Rightarrow \langle Val\ v,(h',l',sh')\rangle$

| *SFAccInitThrow*:
⟦ $P \vdash C$ *has* F,*Static*:t *in* D;
$\nexists$ *sfs*. *sh* D = *Some* (*sfs*,*Done*); $P \vdash \langle INIT\ D\ ([D],False) \leftarrow unit,(h,l,sh)\rangle \Rightarrow \langle throw\ a,s'\rangle$ ⟧
$\Longrightarrow P \vdash \langle C{\cdot}_s F\{D\},(h,l,sh)\rangle \Rightarrow \langle throw\ a,s'\rangle$

| *SFAccNone*:
⟦ $\neg(\exists\ b\ t.\ P \vdash C$ *has* F,b:t *in* $D)$ ⟧
$\Longrightarrow P \vdash \langle C{\cdot}_s F\{D\},s\rangle \Rightarrow \langle THROW\ NoSuchFieldError,s\rangle$

| *SFAccNonStatic*:
⟦ $P \vdash C$ *has* F,*NonStatic*:t *in* D ⟧
$\Longrightarrow P \vdash \langle C{\cdot}_s F\{D\},s\rangle \Rightarrow \langle THROW\ IncompatibleClassChangeError,s\rangle$

| *FAss*:
⟦ $P \vdash \langle e_1,s_0\rangle \Rightarrow \langle addr\ a,s_1\rangle$; $P \vdash \langle e_2,s_1\rangle \Rightarrow \langle Val\ v,(h_2,l_2,sh_2)\rangle$;
$h_2\ a$ = *Some*(C,*fs*); $P \vdash C$ *has* F,*NonStatic*:t *in* D;
fs' = $fs((F,D)\mapsto v)$; $h_2' = h_2(a\mapsto(C,fs'))$ ⟧
$\Longrightarrow P \vdash \langle e_1{\cdot}F\{D\}:=e_2,s_0\rangle \Rightarrow \langle unit,(h_2',l_2,sh_2)\rangle$

| *FAssNull*:
⟦ $P \vdash \langle e_1,s_0\rangle \Rightarrow \langle null,s_1\rangle$; $P \vdash \langle e_2,s_1\rangle \Rightarrow \langle Val\ v,s_2\rangle$ ⟧ $\Longrightarrow$
$P \vdash \langle e_1{\cdot}F\{D\}:=e_2,s_0\rangle \Rightarrow \langle THROW\ NullPointer,s_2\rangle$

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

| *FAssThrow2*:
⟦ $P \vdash \langle e_1,s_0\rangle \Rightarrow \langle Val\ v,s_1\rangle$; $P \vdash \langle e_2,s_1\rangle \Rightarrow \langle throw\ e',s_2\rangle$ ⟧
$\Longrightarrow P \vdash \langle e_1{\cdot}F\{D\}:=e_2,s_0\rangle \Rightarrow \langle throw\ e',s_2\rangle$

| *FAssNone*:
⟦ $P \vdash \langle e_1,s_0\rangle \Rightarrow \langle addr\ a,s_1\rangle$; $P \vdash \langle e_2,s_1\rangle \Rightarrow \langle Val\ v,(h_2,l_2,sh_2)\rangle$;
$h_2\ a$ = *Some*(C,*fs*); $\neg(\exists\ b\ t.\ P \vdash C$ *has* F,b:t *in* $D)$ ⟧
$\Longrightarrow P \vdash \langle e_1{\cdot}F\{D\}:=e_2,s_0\rangle \Rightarrow \langle THROW\ NoSuchFieldError,(h_2,l_2,sh_2)\rangle$

| *FAssStatic*:
⟦ $P \vdash \langle e_1,s_0\rangle \Rightarrow \langle addr\ a,s_1\rangle$; $P \vdash \langle e_2,s_1\rangle \Rightarrow \langle Val\ v,(h_2,l_2,sh_2)\rangle$;
$h_2\ a$ = *Some*(C,*fs*); $P \vdash C$ *has* F,*Static*:t *in* D ⟧
$\Longrightarrow P \vdash \langle e_1{\cdot}F\{D\}:=e_2,s_0\rangle \Rightarrow \langle THROW\ IncompatibleClassChangeError,(h_2,l_2,sh_2)\rangle$

| *SFAss*:
⟦ $P \vdash \langle e_2,s_0\rangle \Rightarrow \langle Val\ v,(h_1,l_1,sh_1)\rangle$;
$P \vdash C$ *has* F,*Static*:t *in* D;
$sh_1\ D$ = *Some*(*sfs*,*Done*); sfs' = $sfs(F\mapsto v)$; $sh_1' = sh_1(D\mapsto(sfs',Done))$ ⟧
$\Longrightarrow P \vdash \langle C{\cdot}_s F\{D\}:=e_2,s_0\rangle \Rightarrow \langle unit,(h_1,l_1,sh_1')\rangle$

| *SFAssInit*:
$[\![\ P \vdash \langle e_2,s_0\rangle \Rightarrow \langle Val\ v,(h_1,l_1,sh_1)\rangle;$
$P \vdash C\ has\ F,Static{:}t\ in\ D;$
$\nexists sfs.\ sh_1\ D = Some(sfs,Done);\ P \vdash \langle INIT\ D\ ([D],False) \leftarrow unit,(h_1,l_1,sh_1)\rangle \Rightarrow \langle Val\ v',(h',l',sh')\rangle;$
$sh'\ D = Some(sfs,i);$
$sfs' = sfs(F{\mapsto}v);\ sh'' = sh'(D{\mapsto}(sfs',i))\ ]\!]$
$\Longrightarrow P \vdash \langle C{\cdot}_sF\{D\}{:=}e_2,s_0\rangle \Rightarrow \langle unit,(h',l',sh'')\rangle$

| *SFAssInitThrow*:
$[\![\ P \vdash \langle e_2,s_0\rangle \Rightarrow \langle Val\ v,(h_1,l_1,sh_1)\rangle;$
$P \vdash C\ has\ F,Static{:}t\ in\ D;$
$\nexists sfs.\ sh_1\ D = Some(sfs,Done);\ P \vdash \langle INIT\ D\ ([D],False) \leftarrow unit,(h_1,l_1,sh_1)\rangle \Rightarrow \langle throw\ a,s'\rangle\ ]\!]$
$\Longrightarrow P \vdash \langle C{\cdot}_sF\{D\}{:=}e_2,s_0\rangle \Rightarrow \langle throw\ a,s'\rangle$

| *SFAssThrow*:
$P \vdash \langle e_2,s_0\rangle \Rightarrow \langle throw\ e',s_2\rangle$
$\Longrightarrow P \vdash \langle C{\cdot}_sF\{D\}{:=}e_2,s_0\rangle \Rightarrow \langle throw\ e',s_2\rangle$

| *SFAssNone*:
$[\![\ P \vdash \langle e_2,s_0\rangle \Rightarrow \langle Val\ v,(h_2,l_2,sh_2)\rangle;$
$\neg(\exists\ b\ t.\ P \vdash C\ has\ F,b{:}t\ in\ D)\ ]\!]$
$\Longrightarrow P \vdash \langle C{\cdot}_sF\{D\}{:=}e_2,s_0\rangle \Rightarrow \langle THROW\ NoSuchFieldError,(h_2,l_2,sh_2)\rangle$

| *SFAssNonStatic*:
$[\![\ P \vdash \langle e_2,s_0\rangle \Rightarrow \langle Val\ v,(h_2,l_2,sh_2)\rangle;$
$P \vdash C\ has\ F,NonStatic{:}t\ in\ D\ ]\!]$
$\Longrightarrow P \vdash \langle C{\cdot}_sF\{D\}{:=}e_2,s_0\rangle \Rightarrow \langle THROW\ IncompatibleClassChangeError,(h_2,l_2,sh_2)\rangle$

| *CallObjThrow*:
$P \vdash \langle e,s_0\rangle \Rightarrow \langle throw\ e',s_1\rangle \Longrightarrow$
$P \vdash \langle e{\cdot}M(ps),s_0\rangle \Rightarrow \langle throw\ e',s_1\rangle$

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

| *CallNull*:
$[\![\ P \vdash \langle e,s_0\rangle \Rightarrow \langle null,s_1\rangle;\ \ P \vdash \langle ps,s_1\rangle\ [\Rightarrow]\ \langle map\ Val\ vs,s_2\rangle\ ]\!]$
$\Longrightarrow P \vdash \langle e{\cdot}M(ps),s_0\rangle \Rightarrow \langle THROW\ NullPointer,s_2\rangle$

| *CallNone*:
$[\![\ P \vdash \langle e,s_0\rangle \Rightarrow \langle addr\ a,s_1\rangle;\ \ P \vdash \langle ps,s_1\rangle\ [\Rightarrow]\ \langle map\ Val\ vs,(h_2,l_2,sh_2)\rangle;$
$h_2\ a = Some(C,fs);\ \neg(\exists\ b\ Ts\ T\ m\ D.\ P \vdash C\ sees\ M,b{:}Ts{\to}T = m\ in\ D)\ ]\!]$
$\Longrightarrow P \vdash \langle e{\cdot}M(ps),s_0\rangle \Rightarrow \langle THROW\ NoSuchMethodError,(h_2,l_2,sh_2)\rangle$

| *CallStatic*:
$[\![\ P \vdash \langle e,s_0\rangle \Rightarrow \langle addr\ a,s_1\rangle;\ \ P \vdash \langle ps,s_1\rangle\ [\Rightarrow]\ \langle map\ Val\ vs,(h_2,l_2,sh_2)\rangle;$
$h_2\ a = Some(C,fs);\ P \vdash C\ sees\ M,Static{:}Ts{\to}T = m\ in\ D\ ]\!]$
$\Longrightarrow P \vdash \langle e{\cdot}M(ps),s_0\rangle \Rightarrow \langle THROW\ IncompatibleClassChangeError,(h_2,l_2,sh_2)\rangle$

| *Call*:
$[\![\ P \vdash \langle e,s_0\rangle \Rightarrow \langle addr\ a,s_1\rangle;\ \ P \vdash \langle ps,s_1\rangle\ [\Rightarrow]\ \langle map\ Val\ vs,(h_2,l_2,sh_2)\rangle;$
$h_2\ a = Some(C,fs);\ P \vdash C\ sees\ M,NonStatic{:}Ts{\to}T = (pns,body)\ in\ D;$

$\quad$ *length vs = length pns*; $l_2' = [this \mapsto Addr\ a,\ pns[\mapsto]vs]$;
$\quad P \vdash \langle body,(h_2,l_2',sh_2)\rangle \Rightarrow \langle e',(h_3,l_3,sh_3)\rangle$ ⟧
$\Longrightarrow P \vdash \langle e{\cdot}M(ps),s_0\rangle \Rightarrow \langle e',(h_3,l_2,sh_3)\rangle$

| *SCallParamsThrow*:
⟦ $P \vdash \langle es,s_0\rangle$ [⇒] $\langle map\ Val\ vs\ @\ throw\ ex\ \#\ es',s_2\rangle$ ⟧
$\Longrightarrow P \vdash \langle C{\cdot}_s M(es),s_0\rangle \Rightarrow \langle throw\ ex,s_2\rangle$

| *SCallNone*:
⟦ $P \vdash \langle ps,s_0\rangle$ [⇒] $\langle map\ Val\ vs,s_2\rangle$;
$\quad \neg(\exists\, b\ Ts\ T\ m\ D.\ P \vdash C\ sees\ M,b{:}Ts\to T = m\ in\ D)$ ⟧
$\Longrightarrow P \vdash \langle C{\cdot}_s M(ps),s_0\rangle \Rightarrow \langle THROW\ NoSuchMethodError,s_2\rangle$

| *SCallNonStatic*:
⟦ $P \vdash \langle ps,s_0\rangle$ [⇒] $\langle map\ Val\ vs,s_2\rangle$;
$\quad P \vdash C\ sees\ M,NonStatic{:}Ts\to T = m\ in\ D$ ⟧
$\Longrightarrow P \vdash \langle C{\cdot}_s M(ps),s_0\rangle \Rightarrow \langle THROW\ IncompatibleClassChangeError,s_2\rangle$

| *SCallInitThrow*:
⟦ $P \vdash \langle ps,s_0\rangle$ [⇒] $\langle map\ Val\ vs,(h_1,l_1,sh_1)\rangle$;
$\quad P \vdash C\ sees\ M,Static{:}Ts\to T = (pns,body)\ in\ D$;
$\quad \nexists sfs.\ sh_1\ D = Some(sfs,Done);\ M \neq clinit$;
$\quad P \vdash \langle INIT\ D\ ([D],False) \leftarrow unit,(h_1,l_1,sh_1)\rangle \Rightarrow \langle throw\ a,s'\rangle$ ⟧
$\Longrightarrow P \vdash \langle C{\cdot}_s M(ps),s_0\rangle \Rightarrow \langle throw\ a,s'\rangle$

| *SCallInit*:
⟦ $P \vdash \langle ps,s_0\rangle$ [⇒] $\langle map\ Val\ vs,(h_1,l_1,sh_1)\rangle$;
$\quad P \vdash C\ sees\ M,Static{:}Ts\to T = (pns,body)\ in\ D$;
$\quad \nexists sfs.\ sh_1\ D = Some(sfs,Done);\ M \neq clinit$;
$\quad P \vdash \langle INIT\ D\ ([D],False) \leftarrow unit,(h_1,l_1,sh_1)\rangle \Rightarrow \langle Val\ v',(h_2,l_2,sh_2)\rangle$;
$\quad$ *length vs = length pns*; $l_2' = [pns[\mapsto]vs]$;
$\quad P \vdash \langle body,(h_2,l_2',sh_2)\rangle \Rightarrow \langle e',(h_3,l_3,sh_3)\rangle$ ⟧
$\Longrightarrow P \vdash \langle C{\cdot}_s M(ps),s_0\rangle \Rightarrow \langle e',(h_3,l_2,sh_3)\rangle$

| *SCall*:
⟦ $P \vdash \langle ps,s_0\rangle$ [⇒] $\langle map\ Val\ vs,(h_2,l_2,sh_2)\rangle$;
$\quad P \vdash C\ sees\ M,Static{:}Ts\to T = (pns,body)\ in\ D$;
$\quad sh_2\ D = Some(sfs,Done) \vee (M = clinit \wedge sh_2\ D = Some(sfs,Processing))$;
$\quad$ *length vs = length pns*; $l_2' = [pns[\mapsto]vs]$;
$\quad P \vdash \langle body,(h_2,l_2',sh_2)\rangle \Rightarrow \langle e',(h_3,l_3,sh_3)\rangle$ ⟧
$\Longrightarrow P \vdash \langle C{\cdot}_s M(ps),s_0\rangle \Rightarrow \langle e',(h_3,l_2,sh_3)\rangle$

| *Block*:
$P \vdash \langle e_0,(h_0,l_0(V{:=}None),sh_0)\rangle \Rightarrow \langle e_1,(h_1,l_1,sh_1)\rangle \Longrightarrow$
$P \vdash \langle\{V{:}T;\ e_0\},(h_0,l_0,sh_0)\rangle \Rightarrow \langle e_1,(h_1,l_1(V{:=}l_0\ V),sh_1)\rangle$

| *Seq*:
⟦ $P \vdash \langle e_0,s_0\rangle \Rightarrow \langle Val\ v,s_1\rangle$; $P \vdash \langle e_1,s_1\rangle \Rightarrow \langle e_2,s_2\rangle$ ⟧
$\Longrightarrow P \vdash \langle e_0;;e_1,s_0\rangle \Rightarrow \langle e_2,s_2\rangle$

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

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

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

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

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

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

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

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

| *Throw*:
$P \vdash \langle e,s_0\rangle \Rightarrow \langle addr\ a,s_1\rangle \Longrightarrow$
$P \vdash \langle throw\ e,s_0\rangle \Rightarrow \langle Throw\ a,s_1\rangle$

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

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

| *Try*:
$P \vdash \langle e_1,s_0\rangle \Rightarrow \langle Val\ v_1,s_1\rangle \Longrightarrow$
$P \vdash \langle try\ e_1\ catch(C\ V)\ e_2,s_0\rangle \Rightarrow \langle Val\ v_1,s_1\rangle$

| *TryCatch*:
$[\![P \vdash \langle e_1,s_0\rangle \Rightarrow \langle Throw\ a,(h_1,l_1,sh_1)\rangle;\ \ h_1\ a = Some(D,fs);\ \ P \vdash D \preceq^*\ C;$
$\quad P \vdash \langle e_2,(h_1,l_1(V\mapsto Addr\ a),sh_1)\rangle \Rightarrow \langle e_2',(h_2,l_2,sh_2)\rangle]\!]$
$\Longrightarrow P \vdash \langle try\ e_1\ catch(C\ V)\ e_2,s_0\rangle \Rightarrow \langle e_2',(h_2,l_2(V:=l_1\ V),sh_2)\rangle$

| *TryThrow*:
$[\![P \vdash \langle e_1,s_0\rangle \Rightarrow \langle Throw\ a,(h_1,l_1,sh_1)\rangle;\ \ h_1\ a = Some(D,fs);\ \ \neg\ P \vdash D \preceq^*\ C]\!]$
$\Longrightarrow P \vdash \langle try\ e_1\ catch(C\ V)\ e_2,s_0\rangle \Rightarrow \langle Throw\ a,(h_1,l_1,sh_1)\rangle$

| *Nil*:

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

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

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

— init rules

| *InitFinal*:
$P \vdash \langle e,s\rangle \Rightarrow \langle e',s'\rangle \Longrightarrow P \vdash \langle INIT\ C\ (Nil,b) \leftarrow e,s\rangle \Rightarrow \langle e',s'\rangle$

| *InitNone*:
$[\![\ sh\ C = None;\ P \vdash \langle INIT\ C'\ (C\#Cs,False) \leftarrow e,(h,l,sh(C \mapsto (sblank\ P\ C,\ Prepared)))\rangle \Rightarrow \langle e',s'\rangle\ ]\!]$
$\Longrightarrow P \vdash \langle INIT\ C'\ (C\#Cs,False) \leftarrow e,(h,l,sh)\rangle \Rightarrow \langle e',s'\rangle$

| *InitDone*:
$[\![\ sh\ C = Some(sfs,Done);\ P \vdash \langle INIT\ C'\ (Cs,True) \leftarrow e,(h,l,sh)\rangle \Rightarrow \langle e',s'\rangle\ ]\!]$
$\Longrightarrow P \vdash \langle INIT\ C'\ (C\#Cs,False) \leftarrow e,(h,l,sh)\rangle \Rightarrow \langle e',s'\rangle$

| *InitProcessing*:
$[\![\ sh\ C = Some(sfs,Processing);\ P \vdash \langle INIT\ C'\ (Cs,True) \leftarrow e,(h,l,sh)\rangle \Rightarrow \langle e',s'\rangle\ ]\!]$
$\Longrightarrow P \vdash \langle INIT\ C'\ (C\#Cs,False) \leftarrow e,(h,l,sh)\rangle \Rightarrow \langle e',s'\rangle$

— note that *RI* will mark all classes in the list Cs with the Error flag
| *InitError*:
$[\![\ sh\ C = Some(sfs,Error);$
$\quad P \vdash \langle RI\ (C,\ THROW\ NoClassDefFoundError);Cs \leftarrow e,(h,l,sh)\rangle \Rightarrow \langle e',s'\rangle\ ]\!]$
$\Longrightarrow P \vdash \langle INIT\ C'\ (C\#Cs,False) \leftarrow e,(h,l,sh)\rangle \Rightarrow \langle e',s'\rangle$

| *InitObject*:
$[\![\ sh\ C = Some(sfs,Prepared);$
$\quad C = Object;$
$\quad sh' = sh(C \mapsto (sfs,Processing));$
$\quad P \vdash \langle INIT\ C'\ (C\#Cs,True) \leftarrow e,(h,l,sh')\rangle \Rightarrow \langle e',s'\rangle\ ]\!]$
$\Longrightarrow P \vdash \langle INIT\ C'\ (C\#Cs,False) \leftarrow e,(h,l,sh)\rangle \Rightarrow \langle e',s'\rangle$

| *InitNonObject*:
$[\![\ sh\ C = Some(sfs,Prepared);$
$\quad C \neq Object;$
$\quad class\ P\ C = Some\ (D,r);$
$\quad sh' = sh(C \mapsto (sfs,Processing));$
$\quad P \vdash \langle INIT\ C'\ (D\#C\#Cs,False) \leftarrow e,(h,l,sh')\rangle \Rightarrow \langle e',s'\rangle\ ]\!]$
$\Longrightarrow P \vdash \langle INIT\ C'\ (C\#Cs,False) \leftarrow e,(h,l,sh)\rangle \Rightarrow \langle e',s'\rangle$

| *InitRInit*:
$P \vdash \langle RI\ (C,C\cdot_s clinit([]));Cs \leftarrow e,(h,l,sh)\rangle \Rightarrow \langle e',s'\rangle$
$\Longrightarrow P \vdash \langle INIT\ C'\ (C\#Cs,True) \leftarrow e,(h,l,sh)\rangle \Rightarrow \langle e',s'\rangle$

| *RInit*:
$[\![\ P \vdash \langle e',s\rangle \Rightarrow \langle Val\ v,\ (h',l',sh')\rangle;$

$sh'\ C = Some(sfs,\ i);\ sh'' = sh'(C \mapsto (sfs,\ Done));$
$C' = last(C\#Cs);$
$P \vdash \langle INIT\ C'\ (Cs,True) \leftarrow e,\ (h',l',sh'')\rangle \Rightarrow \langle e_1,s_1\rangle\ ]\!]$
$\Longrightarrow P \vdash \langle RI\ (C,e');Cs \leftarrow e,s\rangle \Rightarrow \langle e_1,s_1\rangle$

| *RInitInitFail*:
$[\![\ P \vdash \langle e',s\rangle \Rightarrow \langle throw\ a,\ (h',l',sh')\rangle;$
$sh'\ C = Some(sfs,\ i);\ sh'' = sh'(C \mapsto (sfs,\ Error));$
$P \vdash \langle RI\ (D,throw\ a);Cs \leftarrow e,\ (h',l',sh'')\rangle \Rightarrow \langle e_1,s_1\rangle\ ]\!]$
$\Longrightarrow P \vdash \langle RI\ (C,e');D\#Cs \leftarrow e,s\rangle \Rightarrow \langle e_1,s_1\rangle$

| *RInitFailFinal*:
$[\![\ P \vdash \langle e',s\rangle \Rightarrow \langle throw\ a,\ (h',l',sh')\rangle;$
$sh'\ C = Some(sfs,\ i);\ sh'' = sh'(C \mapsto (sfs,\ Error))\ ]\!]$
$\Longrightarrow P \vdash \langle RI\ (C,e');Nil \leftarrow e,s\rangle \Rightarrow \langle throw\ a,\ (h',l',sh'')\rangle$

1.15.1 Final expressions

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

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

lemma *eval-finalId*: $final\ e \Longrightarrow P \vdash \langle e,s\rangle \Rightarrow \langle e,s\rangle\langle proof\rangle$
lemma *eval-final-same*: $[\![\ P \vdash \langle e,s\rangle \Rightarrow \langle e',s'\rangle;\ final\ e\ ]\!] \Longrightarrow e = e' \wedge s = s'\langle proof\rangle$
lemma *eval-finalsId*:
assumes *finals*: *finals es* **shows** $P \vdash \langle es,s\rangle\ [\Rightarrow]\ \langle es,s\rangle\langle proof\rangle$
lemma *evals-finals-same*:
assumes *finals*: *finals es*
shows $P \vdash \langle es,s\rangle\ [\Rightarrow]\ \langle es',s'\rangle \Longrightarrow es = es' \wedge s = s'$
$\langle proof\rangle$

1.15.2 Property preservation

lemma *evals-length*: $P \vdash \langle es,s\rangle\ [\Rightarrow]\ \langle es',s'\rangle \Longrightarrow length\ es = length\ es'$
$\langle proof\rangle$

corollary *evals-empty*: $P \vdash \langle es,s\rangle\ [\Rightarrow]\ \langle es',s'\rangle \Longrightarrow (es = []) = (es' = [])$
$\langle proof\rangle$

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

lemma *eval-lcl-incr*: $P \vdash \langle e,(h_0,l_0,sh_0)\rangle \Rightarrow \langle e',(h_1,l_1,sh_1)\rangle \Longrightarrow dom\ l_0 \subseteq dom\ l_1$
and *evals-lcl-incr*: $P \vdash \langle es,(h_0,l_0,sh_0)\rangle\ [\Rightarrow]\ \langle es',(h_1,l_1,sh_1)\rangle \Longrightarrow dom\ l_0 \subseteq dom\ l_1\langle proof\rangle$
lemma
shows *init-ri-same-loc*: $P \vdash \langle e,(h,l,sh)\rangle \Rightarrow \langle e',(h',l',sh')\rangle$
$\Longrightarrow (\bigwedge C\ Cs\ b\ M\ a.\ e = INIT\ C\ (Cs,b) \leftarrow unit \vee e = C\cdot_s M([]) \vee e = RI\ (C,Throw\ a)\ ;\ Cs \leftarrow unit$
$\vee\ e = RI\ (C,C\cdot_s clinit([]))\ ;\ Cs \leftarrow unit$
$\Longrightarrow l = l')$
and $P \vdash \langle es,(h,l,sh)\rangle\ [\Rightarrow]\ \langle es',(h',l',sh')\rangle \Longrightarrow True$

⟨*proof*⟩

lemma *init-same-loc*: $P \vdash \langle INIT\ C\ (Cs,b) \leftarrow unit,(h,l,sh)\rangle \Rightarrow \langle e',(h',l',sh')\rangle \Longrightarrow l = l'$
⟨*proof*⟩

lemma assumes *wf*: *wwf-J-prog P*
shows *eval-proc-pres'*: $P \vdash \langle e,(h,l,sh)\rangle \Rightarrow \langle e',(h',l',sh')\rangle$
$\Longrightarrow$ *not-init C e* $\Longrightarrow \exists$ *sfs. sh C* $= \lfloor$(*sfs*, *Processing*)$\rfloor \Longrightarrow \exists$ *sfs'. sh' C* $= \lfloor$(*sfs'*, *Processing*)$\rfloor$
and *evals-proc-pres'*: $P \vdash \langle es,(h,l,sh)\rangle\ [\Rightarrow]\ \langle es',(h',l',sh')\rangle$
$\Longrightarrow$ *not-inits C es* $\Longrightarrow \exists$ *sfs. sh C* $= \lfloor$(*sfs*, *Processing*)$\rfloor \Longrightarrow \exists$ *sfs'. sh' C* $= \lfloor$(*sfs'*, *Processing*)$\rfloor$⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*pr*

1.16 Definite assignment

theory *DefAss* **imports** *BigStep* **begin**

1.16.1 Hypersets

type-synonym *'a hyperset* = *'a set option*

definition *hyperUn* :: *'a hyperset* ⇒ *'a hyperset* ⇒ *'a hyperset* (**infixl** ‹⊔› *65*)
where
$A \sqcup B \equiv$ *case A of None* ⇒ *None*
| $\lfloor A\rfloor$ ⇒ (*case B of None* ⇒ *None* | $\lfloor B\rfloor$ ⇒ $\lfloor A \cup B\rfloor$)

definition *hyperInt* :: *'a hyperset* ⇒ *'a hyperset* ⇒ *'a hyperset* (**infixl** ‹⊓› *70*)
where
$A \sqcap B \equiv$ *case A of None* ⇒ *B*
| $\lfloor A\rfloor$ ⇒ (*case B of None* ⇒ $\lfloor A\rfloor$ | $\lfloor B\rfloor$ ⇒ $\lfloor A \cap B\rfloor$)

definition *hyperDiff1* :: *'a hyperset* ⇒ *'a* ⇒ *'a hyperset* (**infixl** ‹⊖› *65*)
where
$A \ominus a \equiv$ *case A of None* ⇒ *None* | $\lfloor A\rfloor$ ⇒ $\lfloor A - \{a\}\rfloor$

definition *hyper-isin* :: *'a* ⇒ *'a hyperset* ⇒ *bool* (**infix** ‹∈∈› *50*)
where
$a \in\in A \equiv$ *case A of None* ⇒ *True* | $\lfloor A\rfloor$ ⇒ $a \in A$

definition *hyper-subset* :: *'a hyperset* ⇒ *'a hyperset* ⇒ *bool* (**infix** ‹⊑› *50*)
where
$A \sqsubseteq B \equiv$ *case B of None* ⇒ *True*
| $\lfloor B\rfloor$ ⇒ (*case A of None* ⇒ *False* | $\lfloor A\rfloor$ ⇒ $A \subseteq B$)

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

lemma [*simp*]: $\lfloor\{\}\rfloor \sqcup A = A \ \wedge\ A \sqcup \lfloor\{\}\rfloor = A$⟨*proof*⟩
lemma [*simp*]: $\lfloor A\rfloor \sqcup \lfloor B\rfloor = \lfloor A \cup B\rfloor \wedge \lfloor A\rfloor \ominus a = \lfloor A - \{a\}\rfloor$⟨*proof*⟩
lemma [*simp*]: *None* $\sqcup A =$ *None* $\wedge$ $A \sqcup$ *None* $=$ *None*⟨*proof*⟩
lemma [*simp*]: $a \in\in$ *None* $\wedge$ *None* $\ominus a =$ *None*⟨*proof*⟩
lemma *hyper-isin-union*: $x \in\in \lfloor A\rfloor \Longrightarrow x \in\in \lfloor A\rfloor \sqcup B$
⟨*proof*⟩

lemma *hyperUn-assoc*: $(A \sqcup B) \sqcup C = A \sqcup (B \sqcup C)\langle proof\rangle$
lemma *hyper-insert-comm*: $A \sqcup \lfloor\{a\}\rfloor = \lfloor\{a\}\rfloor \sqcup A \wedge A \sqcup (\lfloor\{a\}\rfloor \sqcup B) = \lfloor\{a\}\rfloor \sqcup (A \sqcup B)\langle proof\rangle$
lemma *hyper-comm*: $A \sqcup B = B \sqcup A \wedge A \sqcup B \sqcup C = B \sqcup A \sqcup C\langle proof\rangle$

1.16.2 Definite assignment

primrec
$\mathcal{A}$ :: *'a exp* ⇒ *'a hyperset*
and $\mathcal{A}s$ :: *'a exp list* ⇒ *'a hyperset*
where
$\mathcal{A}\ (new\ C) = \lfloor\{\}\rfloor$
| $\mathcal{A}\ (Cast\ C\ e) = \mathcal{A}\ e$
| $\mathcal{A}\ (Val\ v) = \lfloor\{\}\rfloor$
| $\mathcal{A}\ (e_1\ «bop»\ e_2) = \mathcal{A}\ e_1 \sqcup \mathcal{A}\ e_2$
| $\mathcal{A}\ (Var\ V) = \lfloor\{\}\rfloor$
| $\mathcal{A}\ (LAss\ V\ e) = \lfloor\{V\}\rfloor \sqcup \mathcal{A}\ e$
| $\mathcal{A}\ (e{\cdot}F\{D\}) = \mathcal{A}\ e$
| $\mathcal{A}\ (C{\cdot}_sF\{D\}) = \lfloor\{\}\rfloor$
| $\mathcal{A}\ (e_1{\cdot}F\{D\}{:=}e_2) = \mathcal{A}\ e_1 \sqcup \mathcal{A}\ e_2$
| $\mathcal{A}\ (C{\cdot}_sF\{D\}{:=}e_2) = \mathcal{A}\ e_2$
| $\mathcal{A}\ (e{\cdot}M(es)) = \mathcal{A}\ e \sqcup \mathcal{A}s\ es$
| $\mathcal{A}\ (C{\cdot}_sM(es)) = \mathcal{A}s\ es$
| $\mathcal{A}\ (\{V{:}T;\ e\}) = \mathcal{A}\ e \ominus V$
| $\mathcal{A}\ (e_1;;e_2) = \mathcal{A}\ e_1 \sqcup \mathcal{A}\ e_2$
| $\mathcal{A}\ (if\ (e)\ e_1\ else\ e_2) = \mathcal{A}\ e \sqcup (\mathcal{A}\ e_1 \sqcap \mathcal{A}\ e_2)$
| $\mathcal{A}\ (while\ (b)\ e) = \mathcal{A}\ b$
| $\mathcal{A}\ (throw\ e) = None$
| $\mathcal{A}\ (try\ e_1\ catch(C\ V)\ e_2) = \mathcal{A}\ e_1 \sqcap (\mathcal{A}\ e_2 \ominus V)$
| $\mathcal{A}\ (INIT\ C\ (Cs,b) \leftarrow e) = \lfloor\{\}\rfloor$
| $\mathcal{A}\ (RI\ (C,e);Cs \leftarrow e') = \mathcal{A}\ e$

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

primrec
$\mathcal{D}$ :: *'a exp* ⇒ *'a hyperset* ⇒ *bool*
and $\mathcal{D}s$ :: *'a exp list* ⇒ *'a hyperset* ⇒ *bool*
where
$\mathcal{D}\ (new\ C)\ A = True$
| $\mathcal{D}\ (Cast\ C\ e)\ A = \mathcal{D}\ e\ A$
| $\mathcal{D}\ (Val\ v)\ A = True$
| $\mathcal{D}\ (e_1\ «bop»\ e_2)\ A = (\mathcal{D}\ e_1\ A \wedge \mathcal{D}\ e_2\ (A \sqcup \mathcal{A}\ e_1))$
| $\mathcal{D}\ (Var\ V)\ A = (V \in\in A)$
| $\mathcal{D}\ (LAss\ V\ e)\ A = \mathcal{D}\ e\ A$
| $\mathcal{D}\ (e{\cdot}F\{D\})\ A = \mathcal{D}\ e\ A$
| $\mathcal{D}\ (C{\cdot}_sF\{D\})\ A = True$
| $\mathcal{D}\ (e_1{\cdot}F\{D\}{:=}e_2)\ A = (\mathcal{D}\ e_1\ A \wedge \mathcal{D}\ e_2\ (A \sqcup \mathcal{A}\ e_1))$
| $\mathcal{D}\ (C{\cdot}_sF\{D\}{:=}e_2)\ A = \mathcal{D}\ e_2\ A$
| $\mathcal{D}\ (e{\cdot}M(es))\ A = (\mathcal{D}\ e\ A \wedge \mathcal{D}s\ es\ (A \sqcup \mathcal{A}\ e))$
| $\mathcal{D}\ (C{\cdot}_sM(es))\ A = \mathcal{D}s\ es\ A$
| $\mathcal{D}\ (\{V{:}T;\ e\})\ A = \mathcal{D}\ e\ (A \ominus V)$
| $\mathcal{D}\ (e_1;;e_2)\ A = (\mathcal{D}\ e_1\ A \wedge \mathcal{D}\ e_2\ (A \sqcup \mathcal{A}\ e_1))$
| $\mathcal{D}\ (if\ (e)\ e_1\ else\ e_2)\ A =$
$(\mathcal{D}\ e\ A \wedge \mathcal{D}\ e_1\ (A \sqcup \mathcal{A}\ e) \wedge \mathcal{D}\ e_2\ (A \sqcup \mathcal{A}\ e))$

$| \mathcal{D}\ (while\ (e)\ c)\ A = (\mathcal{D}\ e\ A \wedge \mathcal{D}\ c\ (A \sqcup \mathcal{A}\ e))$
$| \mathcal{D}\ (throw\ e)\ A = \mathcal{D}\ e\ A$
$| \mathcal{D}\ (try\ e_1\ catch(C\ V)\ e_2)\ A = (\mathcal{D}\ e_1\ A \wedge \mathcal{D}\ e_2\ (A \sqcup \lfloor\{V\}\rfloor))$
$| \mathcal{D}\ (INIT\ C\ (Cs,b) \leftarrow e)\ A = \mathcal{D}\ e\ A$
$| \mathcal{D}\ (RI\ (C,e);Cs \leftarrow e')\ A = (\mathcal{D}\ e\ A \wedge \mathcal{D}\ e'\ A)$

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

lemma *As-map-Val[simp]*: $\mathcal{A}s\ (map\ Val\ vs) = \lfloor\{\}\rfloor\langle proof\rangle$
lemma *D-append[iff]*: $\bigwedge A.\ \mathcal{D}s\ (es\ @\ es')\ A = (\mathcal{D}s\ es\ A \wedge \mathcal{D}s\ es'\ (A \sqcup \mathcal{A}s\ es))\langle proof\rangle$

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

lemma *sqUn-lem*: $A \sqsubseteq A' \Longrightarrow A \sqcup B \sqsubseteq A' \sqcup B\langle proof\rangle$
lemma *diff-lem*: $A \sqsubseteq A' \Longrightarrow A \ominus b \sqsubseteq A' \ominus b\langle proof\rangle$

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

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

lemma *Ds-Vals*: $\mathcal{D}s\ (map\ Val\ vs)\ A\ \langle proof\rangle$

end

1.17 Conformance Relations for Type Soundness Proofs

theory *Conform*
imports *Exceptions*
begin

definition *conf* :: *'m prog* ⇒ *heap* ⇒ *val* ⇒ *ty* ⇒ *bool* (‹-,- ⊢ - :≤ -› [51,51,51,51] 50)
where
$P,h \vdash v :\leq T \equiv$
$\exists T'.\ typeof_h\ v = Some\ T' \wedge P \vdash T' \leq T$

definition *oconf* :: *'m prog* ⇒ *heap* ⇒ *obj* ⇒ *bool* (‹-,- ⊢ - √› [51,51,51] 50)
where
$P,h \vdash obj\ \surd \equiv$
$let\ (C,fs) = obj\ in\ \forall F\ D\ T.\ P \vdash C\ has\ F,NonStatic{:}T\ in\ D \longrightarrow$
$(\exists v.\ fs(F,D) = Some\ v \wedge P,h \vdash v :\leq T)$

definition *soconf* :: *'m prog* ⇒ *heap* ⇒ *cname* ⇒ *sfields* ⇒ *bool* (‹-,-,- ⊢$_s$ - √› [51,51,51,51] 50)
where
$P,h,C \vdash_s sfs\ \surd \equiv$
$\forall F\ T.\ P \vdash C\ has\ F,Static{:}T\ in\ C \longrightarrow$
$(\exists v.\ sfs\ F = Some\ v \wedge P,h \vdash v :\leq T)$

definition *hconf* :: *'m prog* ⇒ *heap* ⇒ *bool* (‹- ⊢ - √› [51,51] 50)
where

$P \vdash h \surd \equiv$
$(\forall a\ obj.\ h\ a = Some\ obj \longrightarrow P,h \vdash obj \surd) \wedge preallocated\ h$

definition *shconf* :: *'m prog* ⇒ *heap* ⇒ *sheap* ⇒ *bool* (‹-,- ⊢$_s$ - √› [*51*,*51*,*51*] *50*)
where
$P,h \vdash_s sh \surd \equiv$
$(\forall C\ sfs\ i.\ sh\ C = Some(sfs,i) \longrightarrow P,h,C \vdash_s sfs \surd)$

definition *lconf* :: *'m prog* ⇒ *heap* ⇒ (*vname* ⇀ *val*) ⇒ (*vname* ⇀ *ty*) ⇒ *bool* (‹-,- ⊢ - *'*(:≤*'*) -› [*51*,*51*,*51*,*51*] *50*)
where
$P,h \vdash l\ (:\leq)\ E \equiv$
$\forall V\ v.\ l\ V = Some\ v \longrightarrow (\exists T.\ E\ V = Some\ T \wedge P,h \vdash v :\leq T)$

abbreviation
confs :: *'m prog* ⇒ *heap* ⇒ *val list* ⇒ *ty list* ⇒ *bool*
(‹-,- ⊢ - [:≤] -› [*51*,*51*,*51*,*51*] *50*) **where**
$P,h \vdash vs\ [:\leq]\ Ts \equiv list\text{-}all2\ (conf\ P\ h)\ vs\ Ts$

1.17.1 Value conformance :≤

lemma *conf-Null* [*simp*]: $P,h \vdash Null :\leq T = P \vdash NT \leq T$⟨*proof*⟩
lemma *typeof-conf*[*simp*]: $typeof_h\ v = Some\ T \Longrightarrow P,h \vdash v :\leq T$⟨*proof*⟩
lemma *typeof-lit-conf*[*simp*]: $typeof\ v = Some\ T \Longrightarrow P,h \vdash v :\leq T$⟨*proof*⟩
lemma *defval-conf*[*simp*]: $P,h \vdash default\text{-}val\ T :\leq T$⟨*proof*⟩
lemma *conf-upd-obj*: $h\ a = Some(C,fs) \Longrightarrow (P,h(a\mapsto(C,fs')) \vdash x :\leq T) = (P,h \vdash x :\leq T)$⟨*proof*⟩
lemma *conf-widen*: $P,h \vdash v :\leq T \Longrightarrow P \vdash T \leq T' \Longrightarrow P,h \vdash v :\leq T'$⟨*proof*⟩
lemma *conf-hext*: $h \trianglelefteq h' \Longrightarrow P,h \vdash v :\leq T \Longrightarrow P,h' \vdash v :\leq T$⟨*proof*⟩
lemma *conf-ClassD*: $P,h \vdash v :\leq Class\ C \Longrightarrow$
$v = Null \vee (\exists a\ obj\ T.\ v = Addr\ a \wedge h\ a = Some\ obj \wedge obj\text{-}ty\ obj = T \wedge P \vdash T \leq Class\ C)$⟨*proof*⟩
lemma *conf-NT* [*iff*]: $P,h \vdash v :\leq NT = (v = Null)$⟨*proof*⟩
lemma *non-npD*: ⟦ $v \neq Null$; $P,h \vdash v :\leq Class\ C$ ⟧
$\Longrightarrow \exists a\ C'\ fs.\ v = Addr\ a \wedge h\ a = Some(C',fs) \wedge P \vdash C' \preceq^* C$⟨*proof*⟩

1.17.2 Value list conformance [:≤]

lemma *confs-widens* [*trans*]: ⟦$P,h \vdash vs\ [:\leq]\ Ts$; $P \vdash Ts\ [\leq]\ Ts'$⟧ ⟹ $P,h \vdash vs\ [:\leq]\ Ts'$⟨*proof*⟩
lemma *confs-rev*: $P,h \vdash rev\ s\ [:\leq]\ t = (P,h \vdash s\ [:\leq]\ rev\ t)$⟨*proof*⟩
lemma *confs-conv-map*:
$\bigwedge Ts'.\ P,h \vdash vs\ [:\leq]\ Ts' = (\exists Ts.\ map\ typeof_h\ vs = map\ Some\ Ts \wedge P \vdash Ts\ [\leq]\ Ts')$⟨*proof*⟩
lemma *confs-hext*: $P,h \vdash vs\ [:\leq]\ Ts \Longrightarrow h \trianglelefteq h' \Longrightarrow P,h' \vdash vs\ [:\leq]\ Ts$⟨*proof*⟩
lemma *confs-Cons2*: $P,h \vdash xs\ [:\leq]\ y\#ys = (\exists z\ zs.\ xs = z\#zs \wedge P,h \vdash z :\leq y \wedge P,h \vdash zs\ [:\leq]\ ys)$⟨*proof*⟩

1.17.3 Object conformance

lemma *oconf-hext*: $P,h \vdash obj \surd \Longrightarrow h \trianglelefteq h' \Longrightarrow P,h' \vdash obj \surd$⟨*proof*⟩
lemma *oconf-blank*:
$P \vdash C\ has\text{-}fields\ FDTs \Longrightarrow P,h \vdash blank\ P\ C \surd$⟨*proof*⟩
lemma *oconf-fupd* [*intro?*]:
⟦ $P \vdash C\ has\ F,NonStatic{:}T\ in\ D$; $P,h \vdash v :\leq T$; $P,h \vdash (C,fs) \surd$ ⟧
$\Longrightarrow P,h \vdash (C,\ fs((F,D)\mapsto v)) \surd$⟨*proof*⟩

1.17.4 Static object conformance

lemma *soconf-hext*: $P,h,C \vdash_s obj\ \surd \Longrightarrow h \trianglelefteq h' \Longrightarrow P,h',C \vdash_s obj\ \surd \langle proof \rangle$
lemma *soconf-sblank*:
$P \vdash C\ has\text{-}fields\ FDTs \Longrightarrow P,h,C \vdash_s sblank\ P\ C\ \surd \langle proof \rangle$
lemma *soconf-fupd* [*intro?*]:
$[\![P \vdash C\ has\ F,Static{:}T\ in\ C;\ P,h \vdash v :\leq T;\ P,h,C \vdash_s sfs\ \surd]\!]$
$\Longrightarrow P,h,C \vdash_s sfs(F{\mapsto}v)\ \surd \langle proof \rangle$

1.17.5 Heap conformance

lemma *hconfD*: $[\![P \vdash h\ \surd;\ h\ a = Some\ obj]\!] \Longrightarrow P,h \vdash obj\ \surd \langle proof \rangle$
lemma *hconf-new*: $[\![P \vdash h\ \surd;\ h\ a = None;\ P,h \vdash obj\ \surd]\!] \Longrightarrow P \vdash h(a{\mapsto}obj)\ \surd \langle proof \rangle$
lemma *hconf-upd-obj*: $[\![P \vdash h\surd;\ h\ a = Some(C,fs);\ P,h \vdash (C,fs')\surd]\!] \Longrightarrow P \vdash h(a{\mapsto}(C,fs'))\surd \langle proof \rangle$

1.17.6 Class statics conformance

lemma *shconfD*: $[\![P,h \vdash_s sh\ \surd;\ sh\ C = Some(sfs,i)]\!] \Longrightarrow P,h,C \vdash_s sfs\ \surd \langle proof \rangle$
lemma *shconf-upd-obj*: $[\![P,h \vdash_s sh\ \surd;\ P,h,C \vdash_s sfs'\ \surd]\!]$
$\Longrightarrow P,h \vdash_s sh(C{\mapsto}(sfs',i'))\surd \langle proof \rangle$
lemma *shconf-hnew*: $[\![P,h \vdash_s sh\ \surd;\ h\ a = None]\!] \Longrightarrow P,h(a{\mapsto}obj) \vdash_s sh\ \surd \langle proof \rangle$
lemma *shconf-hupd-obj*: $[\![P,h \vdash_s sh\ \surd;\ h\ a = Some(C,fs)]\!] \Longrightarrow P,h(a{\mapsto}(C,fs')) \vdash_s sh\ \surd \langle proof \rangle$

1.17.7 Local variable conformance

lemma *lconf-hext*: $[\![P,h \vdash l\ (:\leq)\ E;\ h \trianglelefteq h']\!] \Longrightarrow P,h' \vdash l\ (:\leq)\ E \langle proof \rangle$
lemma *lconf-upd*:
$[\![P,h \vdash l\ (:\leq)\ E;\ P,h \vdash v :\leq T;\ E\ V = Some\ T]\!] \Longrightarrow P,h \vdash l(V{\mapsto}v)\ (:\leq)\ E \langle proof \rangle$
lemma *lconf-empty*[*iff*]: $P,h \vdash Map.empty\ (:\leq)\ E \langle proof \rangle$
lemma *lconf-upd2*: $[\![P,h \vdash l\ (:\leq)\ E;\ P,h \vdash v :\leq T]\!] \Longrightarrow P,h \vdash l(V{\mapsto}v)\ (:\leq)\ E(V{\mapsto}T) \langle proof \rangle$

end

1.18 Small Step Semantics

theory *SmallStep*
imports *Expr State WWellForm*
begin

fun *blocks* :: *vname list* $*$ *ty list* $*$ *val list* $*$ *expr* $\Rightarrow$ *expr*
where
$blocks(V\#Vs,\ T\#Ts,\ v\#vs,\ e) = \{V{:}T := Val\ v;\ blocks(Vs,Ts,vs,e)\}$
$|blocks([],[],[],e) = e$

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

lemma [*simp*]:
$[\![size\ vs = size\ Vs;\ size\ Ts = size\ Vs]\!] \Longrightarrow fv(blocks(Vs,Ts,vs,e)) = fv\ e - set\ Vs \langle proof \rangle$

lemma *sub-RI-blocks-body*[*iff*]: $length\ vs = length\ pns \Longrightarrow length\ Ts = length\ pns$
$\Longrightarrow sub\text{-}RI\ (blocks\ (pns,\ Ts,\ vs,\ body)) \longleftrightarrow sub\text{-}RI\ body$
$\langle proof \rangle$

definition *assigned* :: *'a* ⇒ *'a exp* ⇒ *bool*
where
assigned V e ≡ ∃ *v e'*. *e* = (*V* := *Val v*;; *e'*)

— expression is okay to go the right side of *INIT* or *RI* ← or to have indicator Boolean be True (in latter case, given that class is also verified initialized)
fun *icheck* :: *'m prog* ⇒ *cname* ⇒ *'a exp* ⇒ *bool* **where**
icheck P C' (*new C*) = (*C'* = *C*) |
icheck P D' (*C*•*sF*{*D*}) = ((*D'* = *D*) ∧ (∃ *T*. *P* ⊢ *C has F*,*Static*:*T in D*)) |
icheck P D' (*C*•*sF*{*D*}:=(*Val v*)) = ((*D'* = *D*) ∧ (∃ *T*. *P* ⊢ *C has F*,*Static*:*T in D*)) |
icheck P D (*C*•*sM*(*es*)) = ((∃ *vs*. *es* = *map Val vs*) ∧ (∃ *Ts T m*. *P* ⊢ *C sees M*,*Static*:*Ts*→*T* = *m in D*)) |
icheck P - - = False

lemma *nicheck-SFAss-nonVal*: *val-of* e_2 = *None* ⟹ ¬*icheck P C'* (*C*•*sF*{*D*} := (e_2::*'a exp*))
⟨*proof*⟩

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

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

| *RedNew*:
⟦ *new-Addr h* = *Some a*; *P* ⊢ *C has-fields FDTs*; *h'* = *h*(*a*↦*blank P C*) ⟧
⟹ *P* ⊢ ⟨*new C*, (*h*,*l*,*sh*), *True*⟩ → ⟨*addr a*, (*h'*,*l*,*sh*), *False*⟩

| *RedNewFail*:
⟦ *new-Addr h* = *None*; *is-class P C* ⟧ ⟹
P ⊢ ⟨*new C*, (*h*,*l*,*sh*), *True*⟩ → ⟨*THROW OutOfMemory*, (*h*,*l*,*sh*), *False*⟩

| *NewInitDoneRed*:
sh C = *Some* (*sfs*, *Done*) ⟹
P ⊢ ⟨*new C*, (*h*,*l*,*sh*), *False*⟩ → ⟨*new C*, (*h*,*l*,*sh*), *True*⟩

| *NewInitRed*:
⟦ ∄ *sfs*. *sh C* = *Some* (*sfs*, *Done*); *is-class P C* ⟧
⟹ *P* ⊢ ⟨*new C*,(*h*,*l*,*sh*),*False*⟩ → ⟨*INIT C* ([*C*],*False*) ← *new C*,(*h*,*l*,*sh*),*False*⟩

| *CastRed*:
P ⊢ ⟨*e*,*s*,*b*⟩ → ⟨*e'*,*s'*,*b'*⟩ ⟹
P ⊢ ⟨*Cast C e*, *s*, *b*⟩ → ⟨*Cast C e'*, *s'*, *b'*⟩

| *RedCastNull*:
P ⊢ ⟨*Cast C null*, *s*, *b*⟩ → ⟨*null*,*s*,*b*⟩

| *RedCast*:

⟦ $h\ a = Some(D,fs)$; $P \vdash D \preceq^* C$ ⟧
⟹ $P \vdash \langle Cast\ C\ (addr\ a),\ (h,l,sh),\ b\rangle \to \langle addr\ a,\ (h,l,sh),\ b\rangle$

| *RedCastFail*:
⟦ $h\ a = Some(D,fs)$; $\neg\ P \vdash D \preceq^* C$ ⟧
⟹ $P \vdash \langle Cast\ C\ (addr\ a),\ (h,l,sh),\ b\rangle \to \langle THROW\ ClassCast,\ (h,l,sh),\ b\rangle$

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

| *BinOpRed2*:
$P \vdash \langle e,s,b\rangle \to \langle e',s',b'\rangle$ ⟹
$P \vdash \langle(Val\ v_1)\ «bop»\ e,\ s,\ b\rangle \to \langle(Val\ v_1)\ «bop»\ e',\ s',\ b'\rangle$

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

| *RedVar*:
$l\ V = Some\ v$ ⟹
$P \vdash \langle Var\ V,(h,l,sh),b\rangle \to \langle Val\ v,(h,l,sh),b\rangle$

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

| *RedLAss*:
$P \vdash \langle V{:=}(Val\ v),\ (h,l,sh),\ b\rangle \to \langle unit,\ (h,l(V{\mapsto}v),sh),\ b\rangle$

| *FAccRed*:
$P \vdash \langle e,s,b\rangle \to \langle e',s',b'\rangle$ ⟹
$P \vdash \langle e{\cdot}F\{D\},\ s,\ b\rangle \to \langle e'{\cdot}F\{D\},\ s',\ b'\rangle$

| *RedFAcc*:
⟦ $h\ a = Some(C,fs)$; $fs(F,D) = Some\ v$;
$P \vdash C\ has\ F,NonStatic{:}t\ in\ D$ ⟧
⟹ $P \vdash \langle(addr\ a){\cdot}F\{D\},\ (h,l,sh),\ b\rangle \to \langle Val\ v,(h,l,sh),b\rangle$

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

| *RedFAccNone*:
⟦ $h\ a = Some(C,fs)$; $\neg(\exists\ b\ t.\ P \vdash C\ has\ F,b{:}t\ in\ D)$ ⟧
⟹ $P \vdash \langle(addr\ a){\cdot}F\{D\},(h,l,sh),b\rangle \to \langle THROW\ NoSuchFieldError,(h,l,sh),b\rangle$

| *RedFAccStatic*:
⟦ $h\ a = Some(C,fs)$; $P \vdash C\ has\ F,Static{:}t\ in\ D$ ⟧
⟹ $P \vdash \langle(addr\ a){\cdot}F\{D\},(h,l,sh),b\rangle \to \langle THROW\ IncompatibleClassChangeError,(h,l,sh),b\rangle$

| *RedSFAcc*:
⟦ $P \vdash C\ has\ F,Static{:}t\ in\ D$;
$sh\ D = Some\ (sfs,i)$;
$sfs\ F = Some\ v$ ⟧

$\Longrightarrow P \vdash \langle C\cdot_s F\{D\},(h,l,sh),True\rangle \to \langle Val\ v,(h,l,sh),False\rangle$

| *SFAccInitDoneRed*:
$[\![P \vdash C\ has\ F,Static{:}t\ in\ D;$
$sh\ D = Some\ (sfs,Done)\ ]\!]$
$\Longrightarrow P \vdash \langle C\cdot_s F\{D\},(h,l,sh),False\rangle \to \langle C\cdot_s F\{D\},(h,l,sh),True\rangle$

| *SFAccInitRed*:
$[\![P \vdash C\ has\ F,Static{:}t\ in\ D;$
$\nexists sfs.\ sh\ D = Some\ (sfs,Done)\ ]\!]$
$\Longrightarrow P \vdash \langle C\cdot_s F\{D\},(h,l,sh),False\rangle \to \langle INIT\ D\ ([D],False) \leftarrow C\cdot_s F\{D\},(h,l,sh),False\rangle$

| *RedSFAccNone*:
$\neg(\exists\, b\ t.\ P \vdash C\ has\ F,b{:}t\ in\ D)$
$\Longrightarrow P \vdash \langle C\cdot_s F\{D\},(h,l,sh),b\rangle \to \langle THROW\ NoSuchFieldError,(h,l,sh),False\rangle$

| *RedSFAccNonStatic*:
$P \vdash C\ has\ F,NonStatic{:}t\ in\ D$
$\Longrightarrow P \vdash \langle C\cdot_s F\{D\},(h,l,sh),b\rangle \to \langle THROW\ IncompatibleClassChangeError,(h,l,sh),False\rangle$

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

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

| *RedFAss*:
$[\![P \vdash C\ has\ F,NonStatic{:}t\ in\ D;\ h\ a = Some(C,fs)\ ]\!] \Longrightarrow$
$P \vdash \langle (addr\ a){\cdot}F\{D\}{:=}(Val\ v),\ (h,l,sh),\ b\rangle \to \langle unit,\ (h(a \mapsto (C,fs((F,D) \mapsto v))),l,sh),\ b\rangle$

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

| *RedFAssNone*:
$[\![h\ a = Some(C,fs);\ \neg(\exists\, b\ t.\ P \vdash C\ has\ F,b{:}t\ in\ D)\ ]\!]$
$\Longrightarrow P \vdash \langle (addr\ a){\cdot}F\{D\}{:=}(Val\ v),(h,l,sh),b\rangle \to \langle THROW\ NoSuchFieldError,(h,l,sh),b\rangle$

| *RedFAssStatic*:
$[\![h\ a = Some(C,fs);\ P \vdash C\ has\ F,Static{:}t\ in\ D\ ]\!]$
$\Longrightarrow P \vdash \langle (addr\ a){\cdot}F\{D\}{:=}(Val\ v),(h,l,sh),b\rangle \to \langle THROW\ IncompatibleClassChangeError,(h,l,sh),b\rangle$

| *SFAssRed*:
$P \vdash \langle e,s,b\rangle \to \langle e',s',b'\rangle \Longrightarrow$
$P \vdash \langle C\cdot_s F\{D\}{:=}e,\ s,\ b\rangle \to \langle C\cdot_s F\{D\}{:=}e',\ s',\ b'\rangle$

| *RedSFAss*:
$[\![P \vdash C\ has\ F,Static{:}t\ in\ D;$
$sh\ D = Some(sfs,i);$
$sfs' = sfs(F{\mapsto}v);\ sh' = sh(D{\mapsto}(sfs',i))\ ]\!]$
$\Longrightarrow P \vdash \langle C\cdot_s F\{D\}{:=}(Val\ v),(h,l,sh),True\rangle \to \langle unit,(h,l,sh'),False\rangle$

| *SFAssInitDoneRed*:

⟦ $P \vdash C$ *has* F,*Static*:t *in* D;
  $sh\ D = Some(sfs,Done)$ ⟧
⟹ $P \vdash \langle C{\cdot}_s F\{D\}:=(Val\ v),(h,l,sh),False\rangle \to \langle C{\cdot}_s F\{D\}:=(Val\ v),(h,l,sh),True\rangle$

| *SFAssInitRed*:
⟦ $P \vdash C$ *has* F,*Static*:t *in* D;
  $\nexists sfs.\ sh\ D = Some(sfs,Done)$ ⟧
⟹ $P \vdash \langle C{\cdot}_s F\{D\}:=(Val\ v),(h,l,sh),False\rangle \to \langle INIT\ D\ ([D],False) \leftarrow C{\cdot}_s F\{D\}:=(Val\ v),(h,l,sh),False\rangle$

| *RedSFAssNone*:
$\neg(\exists\, b\ t.\ P \vdash C$ *has* F,b:t *in* $D)$
⟹ $P \vdash \langle C{\cdot}_s F\{D\}:=(Val\ v),s,b\rangle \to \langle THROW\ NoSuchFieldError,s,False\rangle$

| *RedSFAssNonStatic*:
$P \vdash C$ *has* F,*NonStatic*:t *in* D
⟹ $P \vdash \langle C{\cdot}_s F\{D\}:=(Val\ v),s,b\rangle \to \langle THROW\ IncompatibleClassChangeError,s,False\rangle$

| *CallObj*:
$P \vdash \langle e,s,b\rangle \to \langle e',s',b'\rangle \Longrightarrow$
$P \vdash \langle e{\cdot}M(es),s,b\rangle \to \langle e'{\cdot}M(es),s',b'\rangle$

| *CallParams*:
$P \vdash \langle es,s,b\rangle\ [\to]\ \langle es',s',b'\rangle \Longrightarrow$
$P \vdash \langle (Val\ v){\cdot}M(es),s,b\rangle \to \langle (Val\ v){\cdot}M(es'),s',b'\rangle$

| *RedCall*:
⟦ $h\ a = Some(C,fs)$; $P \vdash C$ *sees* M,*NonStatic*:$Ts \to T = (pns,body)$ *in* D; *size* vs = *size* pns; *size* Ts = *size* pns ⟧
⟹ $P \vdash \langle (addr\ a){\cdot}M(map\ Val\ vs),\ (h,l,sh),\ b\rangle \to \langle blocks(this\#pns,\ Class\ D\#Ts,\ Addr\ a\#vs,\ body),\ (h,l,sh),\ b\rangle$

| *RedCallNull*:
$P \vdash \langle null{\cdot}M(map\ Val\ vs),s,b\rangle \to \langle THROW\ NullPointer,s,b\rangle$

| *RedCallNone*:
⟦ $h\ a = Some(C,fs)$; $\neg(\exists\, b\ Ts\ T\ m\ D.\ P \vdash C$ *sees* M,b:$Ts \to T = m$ *in* $D)$ ⟧
⟹ $P \vdash \langle (addr\ a){\cdot}M(map\ Val\ vs),(h,l,sh),b\rangle \to \langle THROW\ NoSuchMethodError,(h,l,sh),b\rangle$

| *RedCallStatic*:
⟦ $h\ a = Some(C,fs)$; $P \vdash C$ *sees* M,*Static*:$Ts \to T = m$ *in* D ⟧
⟹ $P \vdash \langle (addr\ a){\cdot}M(map\ Val\ vs),(h,l,sh),b\rangle \to \langle THROW\ IncompatibleClassChangeError,(h,l,sh),b\rangle$

| *SCallParams*:
$P \vdash \langle es,s,b\rangle\ [\to]\ \langle es',s',b'\rangle \Longrightarrow$
$P \vdash \langle C{\cdot}_s M(es),s,b\rangle \to \langle C{\cdot}_s M(es'),s',b'\rangle$

| *RedSCall*:
⟦ $P \vdash C$ *sees* M,*Static*:$Ts \to T = (pns,body)$ *in* D;
  length vs = *length* pns; *size* Ts = *size* pns ⟧
⟹ $P \vdash \langle C{\cdot}_s M(map\ Val\ vs),s,True\rangle \to \langle blocks(pns,\ Ts,\ vs,\ body),\ s,\ False\rangle$

| *SCallInitDoneRed*:
⟦ $P \vdash C$ *sees* M,*Static*:$Ts \to T = (pns,body)$ *in* D;
  $sh\ D = Some(sfs,Done) \vee (M = clinit \wedge sh\ D = Some(sfs,Processing))$ ⟧

$\Longrightarrow P \vdash \langle C\bullet_s M(map\ Val\ vs),(h,l,sh),\ False\rangle \to \langle C\bullet_s M(map\ Val\ vs),(h,l,sh),\ True\rangle$

| *SCallInitRed*:
$[\![P \vdash C\ sees\ M,Static:Ts\to T = (pns,body)\ in\ D;$
$\quad \nexists sfs.\ sh\ D = Some(sfs,Done);\ M \neq clinit]\!]$
$\Longrightarrow P \vdash \langle C\bullet_s M(map\ Val\ vs),(h,l,sh),\ False\rangle \to \langle INIT\ D\ ([D],False) \leftarrow C\bullet_s M(map\ Val\ vs),(h,l,sh),False\rangle$

| *RedSCallNone*:
$[\![\neg(\exists b\ Ts\ T\ m\ D.\ P \vdash C\ sees\ M,b:Ts\to T = m\ in\ D)]\!]$
$\Longrightarrow P \vdash \langle C\bullet_s M(map\ Val\ vs),s,b\rangle \to \langle THROW\ NoSuchMethodError,s,False\rangle$

| *RedSCallNonStatic*:
$[\![P \vdash C\ sees\ M,NonStatic:Ts\to T = m\ in\ D]\!]$
$\Longrightarrow P \vdash \langle C\bullet_s M(map\ Val\ vs),s,b\rangle \to \langle THROW\ IncompatibleClassChangeError,s,False\rangle$

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

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

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

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

| *RedInitBlock*:
$P \vdash \langle\{V:T := Val\ v;\ Val\ u\},\ s,\ b\rangle \to \langle Val\ u,\ s,\ b\rangle$

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

| *RedSeq*:
$P \vdash \langle(Val\ v);;e_2,\ s,\ b\rangle \to \langle e_2,\ s,\ b\rangle$

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

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

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

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

| *ThrowRed*:

$P \vdash \langle e,s,b\rangle \to \langle e',s',b'\rangle \Longrightarrow$
$P \vdash \langle throw\ e,\ s,\ b\rangle \to \langle throw\ e',\ s',\ b'\rangle$

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

| *TryRed*:
$P \vdash \langle e,s,b\rangle \to \langle e',s',b'\rangle \Longrightarrow$
$P \vdash \langle try\ e\ catch(C\ V)\ e_2,\ s,\ b\rangle \to \langle try\ e'\ catch(C\ V)\ e_2,\ s',\ b'\rangle$

| *RedTry*:
$P \vdash \langle try\ (Val\ v)\ catch(C\ V)\ e_2,\ s,\ b\rangle \to \langle Val\ v,\ s,\ b\rangle$

| *RedTryCatch*:
$[\![\ hp\ s\ a = Some(D,fs);\ P \vdash D \preceq^*\ C\ ]\!]$
$\Longrightarrow P \vdash \langle try\ (Throw\ a)\ catch(C\ V)\ e_2,\ s,\ b\rangle \to \langle\{V{:}Class\ C := addr\ a;\ e_2\},\ s,\ b\rangle$

| *RedTryFail*:
$[\![\ hp\ s\ a = Some(D,fs);\ \neg\ P \vdash D \preceq^*\ C\ ]\!]$
$\Longrightarrow P \vdash \langle try\ (Throw\ a)\ catch(C\ V)\ e_2,\ s,\ b\rangle \to \langle Throw\ a,\ s,\ b\rangle$

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

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

— Initialization procedure

| *RedInit*:
$\neg sub\text{-}RI\ e \Longrightarrow P \vdash \langle INIT\ C\ (Nil,b) \leftarrow e,s,b'\rangle \to \langle e,s,icheck\ P\ C\ e\rangle$

| *InitNoneRed*:
$sh\ C = None$
$\Longrightarrow P \vdash \langle INIT\ C'\ (C\#Cs,False) \leftarrow e,(h,l,sh),b\rangle \to \langle INIT\ C'\ (C\#Cs,False) \leftarrow e,(h,l,sh(C \mapsto (sblank\ P\ C,\ Prepared))),b\rangle$

| *RedInitDone*:
$sh\ C = Some(sfs,Done)$
$\Longrightarrow P \vdash \langle INIT\ C'\ (C\#Cs,False) \leftarrow e,(h,l,sh),b\rangle \to \langle INIT\ C'\ (Cs,True) \leftarrow e,(h,l,sh),b\rangle$

| *RedInitProcessing*:
$sh\ C = Some(sfs,Processing)$
$\Longrightarrow P \vdash \langle INIT\ C'\ (C\#Cs,False) \leftarrow e,(h,l,sh),b\rangle \to \langle INIT\ C'\ (Cs,True) \leftarrow e,(h,l,sh),b\rangle$

| *RedInitError*:
$sh\ C = Some(sfs,Error)$
$\Longrightarrow P \vdash \langle INIT\ C'\ (C\#Cs,False) \leftarrow e,(h,l,sh),b\rangle \to \langle RI\ (C,THROW\ NoClassDefFoundError);Cs \leftarrow e,(h,l,sh),b\rangle$

| *InitObjectRed*:
$[\![\ sh\ C = Some(sfs,Prepared);$

$C = Object$;
$sh' = sh(C \mapsto (sfs,Processing))$ ⟧
⟹ $P \vdash \langle INIT\ C'\ (C\#Cs,False) \leftarrow e,(h,l,sh),b\rangle \to \langle INIT\ C'\ (C\#Cs,True) \leftarrow e,(h,l,sh'),b\rangle$

| *InitNonObjectSuperRed*:
⟦ $sh\ C = Some(sfs,Prepared)$;
$C \neq Object$;
$class\ P\ C = Some\ (D,r)$;
$sh' = sh(C \mapsto (sfs,Processing))$ ⟧
⟹ $P \vdash \langle INIT\ C'\ (C\#Cs,False) \leftarrow e,(h,l,sh),b\rangle \to \langle INIT\ C'\ (D\#C\#Cs,False) \leftarrow e,(h,l,sh'),b\rangle$

| *RedInitRInit*:
$P \vdash \langle INIT\ C'\ (C\#Cs,True) \leftarrow e,(h,l,sh),b\rangle \to \langle RI\ (C,C\cdot_s clinit([]));Cs \leftarrow e,(h,l,sh),b\rangle$

| *RInitRed*:
$P \vdash \langle e,s,b\rangle \to \langle e',s',b'\rangle \Longrightarrow$
$P \vdash \langle RI\ (C,e);Cs \leftarrow e_0,\ s,\ b\rangle \to \langle RI\ (C,e');Cs \leftarrow e_0,\ s',\ b'\rangle$

| *RedRInit*:
⟦ $sh\ C = Some\ (sfs,\ i)$;
$sh' = sh(C \mapsto (sfs,Done))$;
$C' = last(C\#Cs)$ ⟧ ⟹
$P \vdash \langle RI\ (C,\ Val\ v);Cs \leftarrow e,\ (h,l,sh),\ b\rangle \to \langle INIT\ C'\ (Cs,True) \leftarrow e,\ (h,l,sh'),\ b\rangle$

— Exception propagation

| *CastThrow*: $P \vdash \langle Cast\ C\ (throw\ e),\ s,\ b\rangle \to \langle throw\ e,\ s,\ b\rangle$
| *BinOpThrow1*: $P \vdash \langle (throw\ e)\ «bop»\ e_2,\ s,\ b\rangle \to \langle throw\ e,\ s,\ b\rangle$
| *BinOpThrow2*: $P \vdash \langle (Val\ v_1)\ «bop»\ (throw\ e),\ s,\ b\rangle \to \langle throw\ e,\ s,\ b\rangle$
| *LAssThrow*: $P \vdash \langle V:=(throw\ e),\ s,\ b\rangle \to \langle throw\ e,\ s,\ b\rangle$
| *FAccThrow*: $P \vdash \langle (throw\ e)\cdot F\{D\},\ s,\ b\rangle \to \langle throw\ e,\ s,\ b\rangle$
| *FAssThrow1*: $P \vdash \langle (throw\ e)\cdot F\{D\}:=e_2,\ s,\ b\rangle \to \langle throw\ e,\ s,\ b\rangle$
| *FAssThrow2*: $P \vdash \langle Val\ v\cdot F\{D\}:=(throw\ e),\ s,\ b\rangle \to \langle throw\ e,\ s,\ b\rangle$
| *SFAssThrow*: $P \vdash \langle C\cdot_s F\{D\}:=(throw\ e),\ s,\ b\rangle \to \langle throw\ e,\ s,\ b\rangle$
| *CallThrowObj*: $P \vdash \langle (throw\ e)\cdot M(es),\ s,\ b\rangle \to \langle throw\ e,\ s,\ b\rangle$
| *CallThrowParams*: ⟦ $es = map\ Val\ vs\ @\ throw\ e\ \#\ es'$ ⟧ ⟹ $P \vdash \langle (Val\ v)\cdot M(es),\ s,\ b\rangle \to \langle throw\ e,\ s,\ b\rangle$
| *SCallThrowParams*: ⟦ $es = map\ Val\ vs\ @\ throw\ e\ \#\ es'$ ⟧ ⟹ $P \vdash \langle C\cdot_s M(es),\ s,\ b\rangle \to \langle throw\ e,\ s,\ b\rangle$
| *BlockThrow*: $P \vdash \langle \{V:T;\ Throw\ a\},\ s,\ b\rangle \to \langle Throw\ a,\ s,\ b\rangle$
| *InitBlockThrow*: $P \vdash \langle \{V:T := Val\ v;\ Throw\ a\},\ s,\ b\rangle \to \langle Throw\ a,\ s,\ b\rangle$
| *SeqThrow*: $P \vdash \langle (throw\ e);;e_2,\ s,\ b\rangle \to \langle throw\ e,\ s,\ b\rangle$
| *CondThrow*: $P \vdash \langle if\ (throw\ e)\ e_1\ else\ e_2,\ s,\ b\rangle \to \langle throw\ e,\ s,\ b\rangle$
| *ThrowThrow*: $P \vdash \langle throw(throw\ e),\ s,\ b\rangle \to \langle throw\ e,\ s,\ b\rangle$
| *RInitInitThrow*: ⟦ $sh\ C = Some(sfs,i);\ sh' = sh(C \mapsto (sfs,Error))$ ⟧ ⟹
$P \vdash \langle RI\ (C,Throw\ a);D\#Cs \leftarrow e,(h,l,sh),b\rangle \to \langle RI\ (D,Throw\ a);Cs \leftarrow e,(h,l,sh'),b\rangle$
| *RInitThrow*: ⟦ $sh\ C = Some(sfs,\ i);\ sh' = sh(C \mapsto (sfs,Error))$ ⟧ ⟹
$P \vdash \langle RI\ (C,Throw\ a);Nil \leftarrow e,(h,l,sh),b\rangle \to \langle Throw\ a,(h,l,sh'),b\rangle$

1.18.1 The reflexive transitive closure

abbreviation
Step :: *J-prog* ⇒ *expr* ⇒ *state* ⇒ *bool* ⇒ *expr* ⇒ *state* ⇒ *bool* ⇒ *bool*
(‹- ⊢ ((1⟨-,/-,/-⟩) →*/ (1⟨-,/-,/-⟩))› [51,0,0,0,0,0,0] 81)

where $P \vdash \langle e,s,b\rangle \to* \langle e',s',b'\rangle \equiv ((e,s,b),\ e',s',b') \in (red\ P)^*$

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

lemmas *converse-rtrancl-induct3* =
converse-rtrancl-induct [*of* (*ax*, *ay*, *az*) (*bx*, *by*, *bz*), *split-format* (*complete*),
consumes 1, *case-names refl step*]

lemma *converse-rtrancl-induct-red*[*consumes 1*]:
assumes $P \vdash \langle e,(h,l,sh),b\rangle \to* \langle e',(h',l',sh'),b'\rangle$
and $\bigwedge e\ h\ l\ sh\ b.\ R\ e\ h\ l\ sh\ b\ e\ h\ l\ sh\ b$
and $\bigwedge e_0\ h_0\ l_0\ sh_0\ b_0\ e_1\ h_1\ l_1\ sh_1\ b_1\ e'\ h'\ l'\ sh'\ b'.$
$[\![\ P \vdash \langle e_0,(h_0,l_0,sh_0),b_0\rangle \to \langle e_1,(h_1,l_1,sh_1),b_1\rangle;\ R\ e_1\ h_1\ l_1\ sh_1\ b_1\ e'\ h'\ l'\ sh'\ b'\ ]\!]$
$\Longrightarrow R\ e_0\ h_0\ l_0\ sh_0\ b_0\ e'\ h'\ l'\ sh'\ b'$
shows $R\ e\ h\ l\ sh\ b\ e'\ h'\ l'\ sh'\ b'\langle proof\rangle$

1.18.2 Some easy lemmas

lemma [*iff*]: $\neg\ P \vdash \langle[],s,b\rangle\ [\to]\ \langle es',s',b'\rangle\langle proof\rangle$
lemma [*iff*]: $\neg\ P \vdash \langle Val\ v,s,b\rangle \to \langle e',s',b'\rangle\langle proof\rangle$
lemma *val-no-step*: *val-of e* = ⌊*v*⌋ ⟹ $\neg\ P \vdash \langle e,s,b\rangle \to \langle e',s',b'\rangle\langle proof\rangle$
lemma [*iff*]: $\neg\ P \vdash \langle Throw\ a,s,b\rangle \to \langle e',s',b'\rangle\langle proof\rangle$

lemma *map-Vals-no-step* [*iff*]: $\neg\ P \vdash \langle map\ Val\ vs,s,b\rangle\ [\to]\ \langle es',s',b'\rangle\langle proof\rangle$
lemma *vals-no-step*: *map-vals-of es* = ⌊*vs*⌋ ⟹ $\neg\ P \vdash \langle es,s,b\rangle\ [\to]\ \langle es',s',b'\rangle\langle proof\rangle$
lemma *vals-throw-no-step* [*iff*]: $\neg\ P \vdash \langle map\ Val\ vs\ @\ Throw\ a\ \#\ es,s,b\rangle\ [\to]\ \langle es',s',b'\rangle\langle proof\rangle$
lemma *lass-val-of-red*:
$[\![$ *lass-val-of e* = ⌊*a*⌋; $P \vdash \langle e,(h,\ l,\ sh),b\rangle \to \langle e',(h',\ l',\ sh'),b'\rangle\ ]\!]$
⟹ $e' = unit \wedge h' = h \wedge l' = l(fst\ a \mapsto snd\ a) \wedge sh' = sh \wedge b = b'\langle proof\rangle$

lemma *final-no-step* [*iff*]: *final e* ⟹ $\neg\ P \vdash \langle e,s,b\rangle \to \langle e',s',b'\rangle\langle proof\rangle$
lemma *finals-no-step* [*iff*]: *finals es* ⟹ $\neg\ P \vdash \langle es,s,b\rangle\ [\to]\ \langle es',s',b'\rangle\langle proof\rangle$
lemma *reds-final-same*:
$P \vdash \langle e,s,b\rangle \to* \langle e',s',b'\rangle \Longrightarrow final\ e \Longrightarrow e = e' \wedge s = s' \wedge b = b'$
$\langle proof\rangle$

lemma *reds-throw*:
$P \vdash \langle e,s,b\rangle \to* \langle e',s',b'\rangle \Longrightarrow (\bigwedge e_t.\ throw\text{-}of\ e = \lfloor e_t\rfloor \Longrightarrow \exists\, e_t'.\ throw\text{-}of\ e' = \lfloor e_t'\rfloor)$
$\langle proof\rangle$

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

lemma *red-lcl-incr*: $P \vdash \langle e,(h_0,l_0,sh_0),b\rangle \to \langle e',(h_1,l_1,sh_1),b'\rangle \Longrightarrow dom\ l_0 \subseteq dom\ l_1$
and *reds-lcl-incr*: $P \vdash \langle es,(h_0,l_0,sh_0),b\rangle\ [\to]\ \langle es',(h_1,l_1,sh_1),b'\rangle \Longrightarrow dom\ l_0 \subseteq dom\ l_1\langle proof\rangle$
lemma *red-lcl-add*: $P \vdash \langle e,(h,l,sh),b\rangle \to \langle e',(h',l',sh'),b'\rangle \Longrightarrow (\bigwedge l_0.\ P \vdash \langle e,(h,l_0{+}{+}l,sh),b\rangle \to \langle e',(h',l_0{+}{+}l',sh'),b'\rangle)$
and *reds-lcl-add*: $P \vdash \langle es,(h,l,sh),b\rangle\ [\to]\ \langle es',(h',l',sh'),b'\rangle \Longrightarrow (\bigwedge l_0.\ P \vdash \langle es,(h,l_0{+}{+}l,sh),b\rangle\ [\to]$
$\langle es',(h',l_0{+}{+}l',sh'),b'\rangle)\langle proof\rangle$

lemma *Red-lcl-add*:

assumes $P \vdash \langle e,(h,l,sh),\ b\rangle \to* \langle e',(h',l',sh'),\ b'\rangle$ **shows** $P \vdash \langle e,(h,l_0@@l,sh),b\rangle \to* \langle e',(h',l_0@@l',sh'),b'\rangle\langle proof\rangle$
lemma assumes *wf*: *wwf-J-prog P*
shows *red-proc-pres*: $P \vdash \langle e,(h,l,sh),b\rangle \to \langle e',(h',l',sh'),b'\rangle$
$\Longrightarrow$ *not-init C e* $\Longrightarrow$ *sh C* $= \lfloor(sfs,\ Processing)\rfloor \Longrightarrow$ *not-init C e'* $\wedge$ $(\exists\, sfs'.\ sh'\ C = \lfloor(sfs',\ Processing)\rfloor)$
and *reds-proc-pres*: $P \vdash \langle es,(h,l,sh),b\rangle\ [\to]\ \langle es',(h',l',sh'),b'\rangle$
$\Longrightarrow$ *not-inits C es* $\Longrightarrow$ *sh C* $= \lfloor(sfs,\ Processing)\rfloor \Longrightarrow$ *not-inits C es'* $\wedge$ $(\exists\, sfs'.\ sh'\ C = \lfloor(sfs',\ Processing)\rfloor)\langle proof\rangle$

1.19 Expression conformance properties

theory *EConform*
imports *SmallStep BigStep*
begin

lemma *cons-to-append*: *list* $\neq []$ $\longrightarrow$ $(\exists\, ls.\ a\ \#\ list = ls\ @\ [last\ list])$
$\langle proof\rangle$

1.19.1 Initialization conformance

fun *init-class* :: *'m prog* $\Rightarrow$ *'a exp* $\Rightarrow$ *cname option* **where**
init-class P (new C) = *Some C* |
init-class P $(C{\cdot}_sF\{D\})$ = *Some D* |
init-class P $(C{\cdot}_sF\{D\}{:=}e_2)$ = *Some D* |
init-class P $(C{\cdot}_sM(es))$ = *seeing-class P C M* |
init-class - - = None

lemma *icheck-init-class*: *icheck P C e* $\Longrightarrow$ *init-class P e* $= \lfloor C\rfloor$
$\langle proof\rangle$
fun *ss-exp* :: *'a exp* $\Rightarrow$ *'a exp* **and** *ss-exps* :: *'a exp list* $\Rightarrow$ *'a exp option* **where**
ss-exp (new C) = *new C*
| *ss-exp (Cast C e)* = (*case val-of e of Some v* $\Rightarrow$ *Cast C e* | *-* $\Rightarrow$ *ss-exp e*)
| *ss-exp (Val v)* = *Val v*
| *ss-exp* $(e_1\ «bop»\ e_2)$ = (*case val-of* e_1 *of Some v* $\Rightarrow$ (*case val-of* e_2 *of Some v* $\Rightarrow$ $e_1\ «bop»\ e_2$ | *-* $\Rightarrow$ *ss-exp* e_2)
| *-* $\Rightarrow$ *ss-exp* e_1)
| *ss-exp (Var V)* = *Var V*
| *ss-exp (LAss V e)* = (*case val-of e of Some v* $\Rightarrow$ *LAss V e* | *-* $\Rightarrow$ *ss-exp e*)
| *ss-exp* $(e{\cdot}F\{D\})$ = (*case val-of e of Some v* $\Rightarrow$ $e{\cdot}F\{D\}$ | *-* $\Rightarrow$ *ss-exp e*)
| *ss-exp* $(C{\cdot}_sF\{D\})$ = $C{\cdot}_sF\{D\}$
| *ss-exp* $(e_1{\cdot}F\{D\}{:=}e_2)$ = (*case val-of* e_1 *of Some v* $\Rightarrow$ (*case val-of* e_2 *of Some v* $\Rightarrow$ $e_1{\cdot}F\{D\}{:=}e_2$ | *-* $\Rightarrow$ *ss-exp* e_2)
| *-* $\Rightarrow$ *ss-exp* e_1)
| *ss-exp* $(C{\cdot}_sF\{D\}{:=}e_2)$ = (*case val-of* e_2 *of Some v* $\Rightarrow$ $C{\cdot}_sF\{D\}{:=}e_2$ | *-* $\Rightarrow$ *ss-exp* e_2)
| *ss-exp* $(e{\cdot}M(es))$ = (*case val-of e of Some v* $\Rightarrow$ (*case map-vals-of es of Some t* $\Rightarrow$ $e{\cdot}M(es)$ | *-* $\Rightarrow$ *the*(*ss-exps es*))
| *-* $\Rightarrow$ *ss-exp e*)
| *ss-exp* $(C{\cdot}_sM(es))$ = (*case map-vals-of es of Some t* $\Rightarrow$ $C{\cdot}_sM(es)$ | *-* $\Rightarrow$ *the*(*ss-exps es*))
| *ss-exp* $(\{V{:}T;\ e\})$ = *ss-exp e*
| *ss-exp* $(e_1;;e_2)$ = (*case val-of* e_1 *of Some v* $\Rightarrow$ *ss-exp* e_2
| *None* $\Rightarrow$ (*case lass-val-of* e_1 *of Some p* $\Rightarrow$ *ss-exp* e_2
| *None* $\Rightarrow$ *ss-exp* e_1))
| *ss-exp* $(if\ (b)\ e_1\ else\ e_2)$ = (*case bool-of b of Some True* $\Rightarrow$ $if\ (b)\ e_1\ else\ e_2$
| *Some False* $\Rightarrow$ $if\ (b)\ e_1\ else\ e_2$

$| \ \text{-} \Rightarrow$ *ss-exp b*)
| *ss-exp* (*while* (*b*) *e*) = *while* (*b*) *e*
| *ss-exp* (*throw e*) = (*case val-of e of Some v* ⇒ *throw e* | - ⇒ *ss-exp e*)
| *ss-exp* (*try* e_1 *catch*(*C V*) e_2) = (*case val-of* e_1 *of Some v* ⇒ *try* e_1 *catch*(*C V*) e_2
| - ⇒ *ss-exp* e_1)
| *ss-exp* (*INIT C* (*Cs*,*b*) ← *e*) = *INIT C* (*Cs*,*b*) ← *e*
| *ss-exp* (*RI* (*C*,*e*);*Cs* ← *e'*) = (*case val-of e of Some v* ⇒ *RI* (*C*,*e*);*Cs* ← *e* | - ⇒ *ss-exp e*)
| *ss-exps*([]) = *None*
| *ss-exps*(*e#es*) = (*case val-of e of Some v* ⇒ *ss-exps es* | - ⇒ *Some* (*ss-exp e*))

lemma *icheck-ss-exp*:
assumes *icheck P C e* **shows** *ss-exp e* = *e*
⟨*proof*⟩

lemma *ss-exps-Vals-None*[*simp*]:
ss-exps (*map Val vs*) = *None*
⟨*proof*⟩

lemma *ss-exps-Vals-NoneI*:
ss-exps es = *None* ⟹ ∃ *vs*. *es* = *map Val vs*
⟨*proof*⟩

lemma *ss-exps-throw-nVal*:
⟦ *val-of e* = *None*; *ss-exps* (*map Val vs* @ *throw e* # *es'*) = ⌊*e'*⌋ ⟧
⟹ *e'* = *ss-exp e*
⟨*proof*⟩

lemma *ss-exps-throw-Val*:
⟦ *val-of e* = ⌊*a*⌋; *ss-exps* (*map Val vs* @ *throw e* # *es'*) = ⌊*e'*⌋ ⟧
⟹ *e'* = *throw e*
⟨*proof*⟩

abbreviation *curr-init* :: *'m prog* ⇒ *'a exp* ⇒ *cname option* **where**
curr-init P e ≡ *init-class P* (*ss-exp e*)
abbreviation *curr-inits* :: *'m prog* ⇒ *'a exp list* ⇒ *cname option* **where**
curr-inits P es ≡ *case ss-exps es of Some e* ⇒ *init-class P e* | - ⇒ *None*

lemma *icheck-curr-init'*: ⋀*e'*. *ss-exp e* = *e'* ⟹ *icheck P C e'* ⟹ *curr-init P e* = ⌊*C*⌋
and *icheck-curr-inits'*: ⋀*e*. *ss-exps es* = ⌊*e*⌋ ⟹ *icheck P C e* ⟹ *curr-inits P es* = ⌊*C*⌋
⟨*proof*⟩

lemma *icheck-curr-init*: *icheck P C e'* ⟹ *ss-exp e* = *e'* ⟹ *curr-init P e* = ⌊*C*⌋
⟨*proof*⟩

lemma *icheck-curr-inits*: *icheck P C e* ⟹ *ss-exps es* = ⌊*e*⌋ ⟹ *curr-inits P es* = ⌊*C*⌋
⟨*proof*⟩

definition *initPD* :: *sheap* ⇒ *cname* ⇒ *bool* **where**
initPD sh C ≡ ∃ *sfs i*. *sh C* = *Some* (*sfs*, *i*) ∧ (*i* = *Done* ∨ *i* = *Processing*)

— checks that *INIT* and *RI* conform and are only in the main computation
fun *iconf* :: *sheap* ⇒ *'a exp* ⇒ *bool* **and** *iconfs* :: *sheap* ⇒ *'a exp list* ⇒ *bool* **where**
iconf sh (*new C*) = *True*

| *iconf sh* (*Cast C e*) = *iconf sh e*
| *iconf sh* (*Val v*) = *True*
| *iconf sh* (*e*$_1$ «*bop*» *e*$_2$) = (*case val-of e*$_1$ *of Some v* ⇒ *iconf sh e*$_2$ | - ⇒ *iconf sh e*$_1$ ∧ ¬*sub-RI e*$_2$)
| *iconf sh* (*Var V*) = *True*
| *iconf sh* (*LAss V e*) = *iconf sh e*
| *iconf sh* (*e*•*F*{*D*}) = *iconf sh e*
| *iconf sh* (*C*•$_s$*F*{*D*}) = *True*
| *iconf sh* (*e*$_1$•*F*{*D*}:=*e*$_2$) = (*case val-of e*$_1$ *of Some v* ⇒ *iconf sh e*$_2$ | - ⇒ *iconf sh e*$_1$ ∧ ¬*sub-RI e*$_2$)
| *iconf sh* (*C*•$_s$*F*{*D*}:=*e*$_2$) = *iconf sh e*$_2$
| *iconf sh* (*e*•*M*(*es*)) = (*case val-of e of Some v* ⇒ *iconfs sh es* | - ⇒ *iconf sh e* ∧ ¬*sub-RIs es*)
| *iconf sh* (*C*•$_s$*M*(*es*)) = *iconfs sh es*
| *iconf sh* ({*V*:*T*; *e*}) = *iconf sh e*
| *iconf sh* (*e*$_1$;;*e*$_2$) = (*case val-of e*$_1$ *of Some v* ⇒ *iconf sh e*$_2$
| *None* ⇒ (*case lass-val-of e*$_1$ *of Some p* ⇒ *iconf sh e*$_2$
| *None* ⇒ *iconf sh e*$_1$ ∧ ¬*sub-RI e*$_2$))
| *iconf sh* (*if* (*b*) *e*$_1$ *else e*$_2$) = (*iconf sh b* ∧ ¬*sub-RI e*$_1$ ∧ ¬*sub-RI e*$_2$)
| *iconf sh* (*while* (*b*) *e*) = (¬*sub-RI b* ∧ ¬*sub-RI e*)
| *iconf sh* (*throw e*) = *iconf sh e*
| *iconf sh* (*try e*$_1$ *catch*(*C V*) *e*$_2$) = (*iconf sh e*$_1$ ∧ ¬*sub-RI e*$_2$)
| *iconf sh* (*INIT C* (*Cs*,*b*) ← *e*) = ((*case Cs of Nil* ⇒ *initPD sh C* | *C*′#*Cs*′ ⇒ *last Cs* = *C*) ∧ ¬*sub-RI e*)
| *iconf sh* (*RI* (*C*,*e*);*Cs* ← *e*′) = (*iconf sh e* ∧ ¬*sub-RI e*′)
| *iconfs sh* ([]) = *True*
| *iconfs sh* (*e*#*es*) = (*case val-of e of Some v* ⇒ *iconfs sh es* | - ⇒ *iconf sh e* ∧ ¬*sub-RIs es*)

lemma *iconfs-map-throw*: *iconfs sh* (*map Val vs* @ *throw e* # *es*′) ⟹ *iconf sh e*
⟨*proof*⟩

lemma *nsub-RI-iconf-aux*:
(¬*sub-RI* (*e*::′*a exp*) ⟶ (∀ *e*′. *e*′ ∈ *subexp e* ⟶ ¬*sub-RI e*′ ⟶ *iconf sh e*′) ⟶ *iconf sh e*)
∧ (¬*sub-RIs* (*es*::′*a exp list*) ⟶ (∀ *e*′. *e*′ ∈ *subexps es* ⟶ ¬*sub-RI e*′ ⟶ *iconf sh e*′) ⟶ *iconfs sh es*)
⟨*proof*⟩

lemma *nsub-RI-iconf-aux*′:
(⋀*e*′. *subexp-of e*′ *e* ⟹ ¬*sub-RI e*′ ⟶ *iconf sh e*′) ⟹ (¬*sub-RI e* ⟹ *iconf sh e*)
⟨*proof*⟩

lemma *nsub-RI-iconf*: ¬*sub-RI e* ⟹ *iconf sh e*
and *nsub-RIs-iconfs*: ¬*sub-RIs es* ⟹ *iconfs sh es*
⟨*proof*⟩

lemma *lass-val-of-iconf*: *lass-val-of e* = ⌊*a*⌋ ⟹ *iconf sh e*
⟨*proof*⟩

lemma *icheck-iconf*:
assumes *icheck P C e* **shows** *iconf sh e*
⟨*proof*⟩

1.19.2 Indicator boolean conformance

definition *bconf* :: ′*m prog* ⇒ *sheap* ⇒ ′*a exp* ⇒ *bool* ⇒ *bool* (‹-,- ⊢$_b$ ′(-,-′) √› [*51*,*51*,*0*,*0*] *50*)
where
P,*sh* ⊢$_b$ (*e*,*b*) √ ≡ *b* ⟶ (∃ *C*. *icheck P C* (*ss-exp e*) ∧ *initPD sh C*)

definition *bconfs* :: *'m prog* ⇒ *sheap* ⇒ *'a exp list* ⇒ *bool* ⇒ *bool* (‹-,- ⊢$_b$ '(-,-') √› [51,51,0,0] 50)
where
P,*sh* ⊢$_b$ (*es*,*b*) √ ≡ *b* ⟶ (∃ *C*. (*icheck P C* (*the*(*ss-exps es*))
∧ (*curr-inits P es* = *Some C*) ∧ *initPD sh C*))

— bconf helper lemmas

lemma *bconf-nonVal*[*simp*]:
P,*sh* ⊢$_b$ (*e*,*True*) √ ⟹ *val-of e* = *None*
⟨*proof*⟩

lemma *bconfs-nonVals*[*simp*]:
P,*sh* ⊢$_b$ (*es*,*True*) √ ⟹ *map-vals-of es* = *None*
⟨*proof*⟩

lemma *bconf-Cast*[*iff*]:
P,*sh* ⊢$_b$ (*Cast C e*,*b*) √ ⟷ *P*,*sh* ⊢$_b$ (*e*,*b*) √
⟨*proof*⟩

lemma *bconf-BinOp*[*iff*]:
P,*sh* ⊢$_b$ (*e1* «*bop*» *e2*,*b*) √
⟷ (*case val-of e1 of Some v* ⇒ *P*,*sh* ⊢$_b$ (*e2*,*b*) √ | - ⇒ *P*,*sh* ⊢$_b$ (*e1*,*b*) √)
⟨*proof*⟩

lemma *bconf-LAss*[*iff*]:
P,*sh* ⊢$_b$ (*LAss V e*,*b*) √ ⟷ *P*,*sh* ⊢$_b$ (*e*,*b*) √
⟨*proof*⟩

lemma *bconf-FAcc*[*iff*]:
P,*sh* ⊢$_b$ (*e*·*F*{*D*},*b*) √ ⟷ *P*,*sh* ⊢$_b$ (*e*,*b*) √
⟨*proof*⟩

lemma *bconf-FAss*[*iff*]:
P,*sh* ⊢$_b$ (*FAss e1 F D e2*,*b*) √
⟷ (*case val-of e1 of Some v* ⇒ *P*,*sh* ⊢$_b$ (*e2*,*b*) √ | - ⇒ *P*,*sh* ⊢$_b$ (*e1*,*b*) √)
⟨*proof*⟩

lemma *bconf-SFAss*[*iff*]:
val-of e2 = *None* ⟹ *P*,*sh* ⊢$_b$ (*SFAss C F D e2*,*b*) √ ⟷ *P*,*sh* ⊢$_b$ (*e2*,*b*) √
⟨*proof*⟩

lemma *bconfs-Vals*[*iff*]:
P,*sh* ⊢$_b$ (*map Val vs*, *b*) √ ⟷ ¬ *b*
⟨*proof*⟩

lemma *bconf-Call*[*iff*]:
P,*sh* ⊢$_b$ (*e*·*M*(*es*),*b*) √
⟷ (*case val-of e of Some v* ⇒ *P*,*sh* ⊢$_b$ (*es*,*b*) √ | - ⇒ *P*,*sh* ⊢$_b$ (*e*,*b*) √)
⟨*proof*⟩

lemma *bconf-SCall*[*iff*]:
assumes *mvn*: *map-vals-of es* = *None*

shows $P,sh \vdash_b (C\bullet_s M(es),b)\ \surd \longleftrightarrow P,sh \vdash_b (es,b)\ \surd$
⟨*proof*⟩

lemma *bconf-Cons*[*iff*]:
$P,sh \vdash_b (e\#es,b)\ \surd$
$\longleftrightarrow$ (*case val-of e of Some v* ⇒ $P,sh \vdash_b (es,b)\ \surd$ | - ⇒ $P,sh \vdash_b (e,b)\ \surd$)
⟨*proof*⟩

lemma *bconf-InitBlock*[*iff*]:
$P,sh \vdash_b (\{V{:}T;\ V{:=}Val\ v;;\ e_2\},b)\ \surd \longleftrightarrow P,sh \vdash_b (e_2,b)\ \surd$
⟨*proof*⟩

lemma *bconf-Block*[*iff*]:
$P,sh \vdash_b (\{V{:}T;\ e\},b)\ \surd \longleftrightarrow P,sh \vdash_b (e,b)\ \surd$
⟨*proof*⟩

lemma *bconf-Seq*[*iff*]:
$P,sh \vdash_b (e1;;e2,b)\ \surd$
$\longleftrightarrow$ (*case val-of e1 of Some v* ⇒ $P,sh \vdash_b (e2,b)\ \surd$
| - ⇒ (*case lass-val-of e1 of Some p* ⇒ $P,sh \vdash_b (e2,b)\ \surd$
| *None* ⇒ $P,sh \vdash_b (e1,b)\ \surd$))
⟨*proof*⟩

lemma *bconf-Cond*[*iff*]:
$P,sh \vdash_b (if\ (b)\ e_1\ else\ e_2,b')\ \surd \longleftrightarrow P,sh \vdash_b (b,b')\ \surd$
⟨*proof*⟩

lemma *bconf-While*[*iff*]:
$P,sh \vdash_b (while\ (b)\ e,b')\ \surd \longleftrightarrow \neg b'$
⟨*proof*⟩

lemma *bconf-Throw*[*iff*]:
$P,sh \vdash_b (throw\ e,b)\ \surd \longleftrightarrow P,sh \vdash_b (e,b)\ \surd$
⟨*proof*⟩

lemma *bconf-Try*[*iff*]:
$P,sh \vdash_b (try\ e_1\ catch(C\ V)\ e_2,b)\ \surd \longleftrightarrow P,sh \vdash_b (e_1,b)\ \surd$
⟨*proof*⟩

lemma *bconf-INIT*[*iff*]:
$P,sh \vdash_b (INIT\ C\ (Cs,b') \leftarrow e,b)\ \surd \longleftrightarrow \neg b$
⟨*proof*⟩

lemma *bconf-RI*[*iff*]:
$P,sh \vdash_b (RI(C,e);Cs \leftarrow e',b)\ \surd \longleftrightarrow P,sh \vdash_b (e,b)\ \surd$
⟨*proof*⟩

lemma *bconfs-map-throw*[*iff*]:
$P,sh \vdash_b (map\ Val\ vs\ @\ throw\ e\ \#\ es',b)\ \surd \longleftrightarrow P,sh \vdash_b (e,b)\ \surd$
⟨*proof*⟩

end

1.20 Progress of Small Step Semantics

theory *Progress*
imports *WellTypeRT DefAss ../Common/Conform EConform*
begin

lemma *final-addrE*:
⟦ *P,E,h,sh* ⊢ *e* : *Class C*; *final e*;
⋀*a*. *e* = *addr a* ⟹ *R*;
⋀*a*. *e* = *Throw a* ⟹ *R* ⟧ ⟹ *R*⟨*proof*⟩

lemma *finalRefE*:
⟦ *P,E,h,sh* ⊢ *e* : *T*; *is-refT T*; *final e*;
e = *null* ⟹ *R*;
⋀*a C*. ⟦ *e* = *addr a*; *T* = *Class C* ⟧ ⟹ *R*;
⋀*a*. *e* = *Throw a* ⟹ *R* ⟧ ⟹ *R*⟨*proof*⟩

Derivation of new induction scheme for well typing:

inductive
WTrt' :: [*J-prog*,*heap*,*sheap*,*env*,*expr*,*ty*] ⇒ *bool*
and *WTrts'* :: [*J-prog*,*heap*,*sheap*,*env*,*expr list*, *ty list*] ⇒ *bool*
and *WTrt2'* :: [*J-prog*,*env*,*heap*,*sheap*,*expr*,*ty*] ⇒ *bool*
(‹-,-,-,- ⊢ - :'' › [*51*,*51*,*51*,*51*]*50*)
and *WTrts2'* :: [*J-prog*,*env*,*heap*,*sheap*,*expr list*, *ty list*] ⇒ *bool*
(‹-,-,-,- ⊢ - [:''] › [*51*,*51*,*51*,*51*]*50*)
for *P* :: *J-prog* **and** *h* :: *heap* **and** *sh* :: *sheap*
where
P,E,h,sh ⊢ *e* :' *T* ≡ *WTrt' P h sh E e T*
| *P,E,h,sh* ⊢ *es* [:'] *Ts* ≡ *WTrts' P h sh E es Ts*

| *is-class P C* ⟹ *P,E,h,sh* ⊢ *new C* :' *Class C*
| ⟦ *P,E,h,sh* ⊢ *e* :' *T*; *is-refT T*; *is-class P C* ⟧
⟹ *P,E,h,sh* ⊢ *Cast C e* :' *Class C*
| *typeof*$_h$ *v* = *Some T* ⟹ *P,E,h,sh* ⊢ *Val v* :' *T*
| *E v* = *Some T* ⟹ *P,E,h,sh* ⊢ *Var v* :' *T*
| ⟦ *P,E,h,sh* ⊢ *e*$_1$:' *T*$_1$; *P,E,h,sh* ⊢ *e*$_2$:' *T*$_2$ ⟧
⟹ *P,E,h,sh* ⊢ *e*$_1$ «*Eq*» *e*$_2$:' *Boolean*
| ⟦ *P,E,h,sh* ⊢ *e*$_1$:' *Integer*; *P,E,h,sh* ⊢ *e*$_2$:' *Integer* ⟧
⟹ *P,E,h,sh* ⊢ *e*$_1$ «*Add*» *e*$_2$:' *Integer*
| ⟦ *P,E,h,sh* ⊢ *Var V* :' *T*; *P,E,h,sh* ⊢ *e* :' *T'*; *P* ⊢ *T'* ≤ *T* ⟧
⟹ *P,E,h,sh* ⊢ *V*:=*e* :' *Void*
| ⟦ *P,E,h,sh* ⊢ *e* :' *Class C*; *P* ⊢ *C has F*,*NonStatic*:*T in D* ⟧ ⟹ *P,E,h,sh* ⊢ *e*·*F*{*D*} :' *T*
| *P,E,h,sh* ⊢ *e* :' *NT* ⟹ *P,E,h,sh* ⊢ *e*·*F*{*D*} :' *T*
| ⟦ *P* ⊢ *C has F*,*Static*:*T in D* ⟧ ⟹ *P,E,h,sh* ⊢ *C*·$_s$*F*{*D*} :' *T*
| ⟦ *P,E,h,sh* ⊢ *e*$_1$:' *Class C*; *P* ⊢ *C has F*,*NonStatic*:*T in D*;
P,E,h,sh ⊢ *e*$_2$:' *T*$_2$; *P* ⊢ *T*$_2$ ≤ *T* ⟧
⟹ *P,E,h,sh* ⊢ *e*$_1$·*F*{*D*}:=*e*$_2$:' *Void*
| ⟦ *P,E,h,sh* ⊢ *e*$_1$:'*NT*; *P,E,h,sh* ⊢ *e*$_2$:' *T*$_2$ ⟧ ⟹ *P,E,h,sh* ⊢ *e*$_1$·*F*{*D*}:=*e*$_2$:' *Void*
| ⟦ *P* ⊢ *C has F*,*Static*:*T in D*;
P,E,h,sh ⊢ *e*$_2$:' *T*$_2$; *P* ⊢ *T*$_2$ ≤ *T* ⟧
⟹ *P,E,h,sh* ⊢ *C*·$_s$*F*{*D*}:=*e*$_2$:' *Void*
| ⟦ *P,E,h,sh* ⊢ *e* :' *Class C*; *P* ⊢ *C sees M*,*NonStatic*:*Ts* → *T* = (*pns*,*body*) *in D*;
P,E,h,sh ⊢ *es* [:'] *Ts'*; *P* ⊢ *Ts'* [≤] *Ts* ⟧
⟹ *P,E,h,sh* ⊢ *e*·*M*(*es*) :' *T*

| ⟦ $P,E,h,sh \vdash e :' NT$; $P,E,h,sh \vdash es\ [:'\rceil\ Ts$ ⟧ $\Longrightarrow P,E,h,sh \vdash e{\cdot}M(es) :' T$
| ⟦ $P \vdash C$ *sees* M,*Static*:$Ts \to T = (pns,body)$ *in* D;
$P,E,h,sh \vdash es\ [:'\rceil\ Ts'$; $P \vdash Ts'\ [\leq]\ Ts$;
$M = clinit \longrightarrow sh\ D = \lfloor(sfs,Processing)\rfloor \wedge es = map\ Val\ vs$ ⟧
$\Longrightarrow P,E,h,sh \vdash C{\cdot}_s M(es) :' T$
| $P,E,h,sh \vdash []\ [:'\rceil\ []$
| ⟦ $P,E,h,sh \vdash e :' T$; $P,E,h,sh \vdash es\ [:'\rceil\ Ts$ ⟧ $\Longrightarrow P,E,h,sh \vdash e\#es\ [:'\rceil\ T\#Ts$
| ⟦ $typeof_h\ v = Some\ T_1$; $P \vdash T_1 \leq T$; $P,E(V{\mapsto}T),h,sh \vdash e_2 :' T_2$ ⟧
$\Longrightarrow P,E,h,sh \vdash \{V{:}T := Val\ v;\ e_2\} :' T_2$
| ⟦ $P,E(V{\mapsto}T),h,sh \vdash e :' T'$; $\neg$ *assigned* $V\ e$ ⟧ $\Longrightarrow P,E,h,sh \vdash \{V{:}T;\ e\} :' T'$
| ⟦ $P,E,h,sh \vdash e_1{:}'\ T_1$; $P,E,h,sh \vdash e_2{:}'T_2$ ⟧ $\Longrightarrow P,E,h,sh \vdash e_1;;e_2 :' T_2$
| ⟦ $P,E,h,sh \vdash e :' Boolean$; $P,E,h,sh \vdash e_1{:}'\ T_1$; $P,E,h,sh \vdash e_2{:}'\ T_2$;
$P \vdash T_1 \leq T_2 \vee P \vdash T_2 \leq T_1$;
$P \vdash T_1 \leq T_2 \longrightarrow T = T_2$; $P \vdash T_2 \leq T_1 \longrightarrow T = T_1$ ⟧
$\Longrightarrow P,E,h,sh \vdash if\ (e)\ e_1\ else\ e_2 :' T$
| ⟦ $P,E,h,sh \vdash e :' Boolean$; $P,E,h,sh \vdash c{:}'\ T$ ⟧
$\Longrightarrow P,E,h,sh \vdash while(e)\ c :' Void$
| ⟦ $P,E,h,sh \vdash e :' T_r$; *is-refT* T_r ⟧ $\Longrightarrow P,E,h,sh \vdash throw\ e :' T$
| ⟦ $P,E,h,sh \vdash e_1 :' T_1$; $P,E(V \mapsto Class\ C),h,sh \vdash e_2 :' T_2$; $P \vdash T_1 \leq T_2$ ⟧
$\Longrightarrow P,E,h,sh \vdash try\ e_1\ catch(C\ V)\ e_2 :' T_2$
| ⟦ $P,E,h,sh \vdash e :' T$; $\forall C' \in set\ (C\#Cs).$ *is-class* $P\ C'$; $\neg$*sub-RI* e;
$\forall C' \in set\ (tl\ Cs).\ \exists\ sfs.\ sh\ C' = \lfloor(sfs,Processing)\rfloor$;
$b \longrightarrow (\forall C' \in set\ Cs.\ \exists\ sfs.\ sh\ C' = \lfloor(sfs,Processing)\rfloor)$;
distinct Cs; *supercls-lst* $P\ Cs$ ⟧ $\Longrightarrow P,E,h,sh \vdash INIT\ C\ (Cs,\ b) \leftarrow e :' T$
| ⟦ $P,E,h,sh \vdash e :' T$; $P,E,h,sh \vdash e' :' T'$; $\forall C' \in set\ (C\#Cs).$ *is-class* $P\ C'$; $\neg$*sub-RI* e';
$\forall C' \in set\ (C\#Cs).$ *not-init* $C'\ e$;
$\forall C' \in set\ Cs.\ \exists\ sfs.\ sh\ C' = \lfloor(sfs,Processing)\rfloor$;
$\exists\ sfs.\ sh\ C = \lfloor(sfs,\ Processing)\rfloor \vee (sh\ C = \lfloor(sfs,\ Error)\rfloor \wedge e = THROW\ NoClassDefFoundError)$;
distinct $(C\#Cs)$; *supercls-lst* $P\ (C\#Cs)$ ⟧
$\Longrightarrow P,E,h,sh \vdash RI(C,\ e);Cs \leftarrow e' :' T'$

lemma [*iff*]: $P,E,h,sh \vdash e_1;;e_2 :' T_2 = (\exists\ T_1.\ P,E,h,sh \vdash e_1{:}'\ T_1 \wedge P,E,h,sh \vdash e_2{:}'\ T_2)\langle proof\rangle$
lemma [*iff*]: $P,E,h,sh \vdash Val\ v :' T = (typeof_h\ v = Some\ T)\langle proof\rangle$
lemma [*iff*]: $P,E,h,sh \vdash Var\ v :' T = (E\ v = Some\ T)\langle proof\rangle$

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

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

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

corollary *wts'-iff-wts*: $(P,E,h,sh \vdash es\ [:'\rceil\ Ts) = (P,E,h,sh \vdash es\ [:]\ Ts)\langle proof\rangle$
theorem assumes *wf*: *wwf-J-prog* P **and** *hconf*: $P \vdash h\ \surd$ **and** *shconf*: $P,h \vdash_s sh\ \surd$
shows *progress*: $P,E,h,sh \vdash e : T \Longrightarrow$
$(\bigwedge l.$ ⟦ $\mathcal{D}\ e\ \lfloor dom\ l\rfloor$; $P,sh \vdash_b (e,b)\ \surd$; $\neg$ *final* e ⟧ $\Longrightarrow \exists\ e'\ s'\ b'.\ P \vdash \langle e,(h,l,sh),b\rangle \to \langle e',s',b'\rangle)$
and $P,E,h,sh \vdash es\ [:]\ Ts \Longrightarrow$
$(\bigwedge l.$ ⟦ $\mathcal{D}s\ es\ \lfloor dom\ l\rfloor$; $P,sh \vdash_b (es,b)\ \surd$; $\neg$ *finals* es ⟧ $\Longrightarrow \exists\ es'\ s'\ b'.\ P \vdash \langle es,(h,l,sh),b\rangle\ [\to]\ \langle es',s',b'\rangle)\langle proof\rangle$
end

1.21 Well-formedness Constraints

theory *JWellForm*
imports *../Common/ WellForm WWellForm WellType DefAss*
begin

definition *wf-J-mdecl* :: *J-prog* ⇒ *cname* ⇒ *J-mb mdecl* ⇒ *bool*
where
wf-J-mdecl P C ≡ λ(*M*,*b*,*Ts*,*T*,(*pns*,*body*)).
length Ts = *length pns* ∧
distinct pns ∧
¬*sub-RI body* ∧
(*case b of*
NonStatic ⇒ *this* ∉ *set pns* ∧
(∃ *T'*. *P*,[*this*↦*Class C*,*pns*[↦]*Ts*] ⊢ *body* :: *T'* ∧ *P* ⊢ *T'* ≤ *T*) ∧
𝒟 *body* ⌊{*this*} ∪ *set pns*⌋
| *Static* ⇒ (∃ *T'*. *P*,[*pns*[↦]*Ts*] ⊢ *body* :: *T'* ∧ *P* ⊢ *T'* ≤ *T*) ∧
𝒟 *body* ⌊*set pns*⌋)

lemma *wf-J-mdecl-NonStatic*[*simp*]:
wf-J-mdecl P C (*M*,*NonStatic*,*Ts*,*T*,*pns*,*body*) ≡
(*length Ts* = *length pns* ∧
distinct pns ∧
¬*sub-RI body* ∧
this ∉ *set pns* ∧
(∃ *T'*. *P*,[*this*↦*Class C*,*pns*[↦]*Ts*] ⊢ *body* :: *T'* ∧ *P* ⊢ *T'* ≤ *T*) ∧
𝒟 *body* ⌊{*this*} ∪ *set pns*⌋)⟨*proof*⟩
lemma *wf-J-mdecl-Static*[*simp*]:
wf-J-mdecl P C (*M*,*Static*,*Ts*,*T*,*pns*,*body*) ≡
(*length Ts* = *length pns* ∧
distinct pns ∧
¬*sub-RI body* ∧
(∃ *T'*. *P*,[*pns*[↦]*Ts*] ⊢ *body* :: *T'* ∧ *P* ⊢ *T'* ≤ *T*) ∧
𝒟 *body* ⌊*set pns*⌋)⟨*proof*⟩

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

lemma *wf-J-prog-wf-J-mdecl*:
⟦ *wf-J-prog P*; (*C*, *D*, *fds*, *mths*) ∈ *set P*; *jmdcl* ∈ *set mths* ⟧
⟹ *wf-J-mdecl P C jmdcl*⟨*proof*⟩
lemma *wf-mdecl-wwf-mdecl*: *wf-J-mdecl P C Md* ⟹ *wwf-J-mdecl P C Md*⟨*proof*⟩
lemma *wf-prog-wwf-prog*: *wf-J-prog P* ⟹ *wwf-J-prog P*⟨*proof*⟩

end

1.22 Type Safety Proof

theory *TypeSafe*
imports *Progress BigStep SmallStep JWellForm*
begin

lemma *red-shext-incr*: $P \vdash \langle e,(h,l,sh),b\rangle \to \langle e',(h',l',sh'),b'\rangle$
$\Longrightarrow (\bigwedge E\ T.\ P,E,h,sh \vdash e : T \Longrightarrow sh \trianglelefteq_s sh')$
and *reds-shext-incr*: $P \vdash \langle es,(h,l,sh),b\rangle\ [\to]\ \langle es',(h',l',sh'),b'\rangle$
$\Longrightarrow (\bigwedge E\ Ts.\ P,E,h,sh \vdash es\ [:]\ Ts \Longrightarrow sh \trianglelefteq_s sh')\langle proof\rangle$
lemma *wf-types-clinit*:
assumes *wf*:*wf-prog wf-md P* **and** *ex*: *class P C* = *Some a* **and** *proc*: *sh C* = $\lfloor$(*sfs*, *Processing*)$\rfloor$
shows $P,E,h,sh \vdash C\cdot_s clinit([]) : Void$
$\langle proof\rangle$

1.22.1 Basic preservation lemmas

First some easy preservation lemmas.

theorem *red-preserves-hconf*:
$P \vdash \langle e,(h,l,sh),b\rangle \to \langle e',(h',l',sh'),b'\rangle \Longrightarrow (\bigwedge T\ E.\ [\![\ P,E,h,sh \vdash e : T;\ P \vdash h\ \surd\ ]\!] \Longrightarrow P \vdash h'\ \surd)$
and *reds-preserves-hconf*:
$P \vdash \langle es,(h,l,sh),b\rangle\ [\to]\ \langle es',(h',l',sh'),b'\rangle \Longrightarrow (\bigwedge Ts\ E.\ [\![\ P,E,h,sh \vdash es\ [:]\ Ts;\ P \vdash h\ \surd\ ]\!] \Longrightarrow P \vdash h'\ \surd)\langle proof\rangle$

theorem *red-preserves-lconf*:
$P \vdash \langle e,(h,l,sh),b\rangle \to \langle e',(h',l',sh'),b'\rangle \Longrightarrow$
$(\bigwedge T\ E.\ [\![\ P,E,h,sh \vdash e{:}T;\ P,h \vdash l\ (:\leq)\ E\ ]\!] \Longrightarrow P,h' \vdash l'\ (:\leq)\ E)$
and *reds-preserves-lconf*:
$P \vdash \langle es,(h,l,sh),b\rangle\ [\to]\ \langle es',(h',l',sh'),b'\rangle \Longrightarrow$
$(\bigwedge Ts\ E.\ [\![\ P,E,h,sh \vdash es[:]Ts;\ P,h \vdash l\ (:\leq)\ E\ ]\!] \Longrightarrow P,h' \vdash l'\ (:\leq)\ E)\langle proof\rangle$

theorem *red-preserves-shconf*:
$P \vdash \langle e,(h,l,sh),b\rangle \to \langle e',(h',l',sh'),b'\rangle \Longrightarrow (\bigwedge T\ E.\ [\![\ P,E,h,sh \vdash e : T;\ P,h \vdash_s sh\ \surd\ ]\!] \Longrightarrow P,h' \vdash_s sh'\ \surd)$
and *reds-preserves-shconf*:
$P \vdash \langle es,(h,l,sh),b\rangle\ [\to]\ \langle es',(h',l',sh'),b'\rangle \Longrightarrow (\bigwedge Ts\ E.\ [\![\ P,E,h,sh \vdash es\ [:]\ Ts;\ P,h \vdash_s sh\ \surd\ ]\!] \Longrightarrow P,h' \vdash_s sh'\ \surd)\langle proof\rangle$
theorem assumes *wf*: *wwf-J-prog P*
shows *red-preserves-iconf*:
$P \vdash \langle e,(h,l,sh),b\rangle \to \langle e',(h',l',sh'),b'\rangle \Longrightarrow iconf\ sh\ e \Longrightarrow iconf\ sh'\ e'$
and *reds-preserves-iconf*:
$P \vdash \langle es,(h,l,sh),b\rangle\ [\to]\ \langle es',(h',l',sh'),b'\rangle \Longrightarrow iconfs\ sh\ es \Longrightarrow iconfs\ sh'\ es'\langle proof\rangle$

lemma *Seq-bconf-preserve-aux*:
assumes $P \vdash \langle e,(h,\ l,\ sh),b\rangle \to \langle e',(h',\ l',\ sh'),b'\rangle$ **and** $P,sh \vdash_b (e;;\ e_2,b)\ \surd$
and $P,sh \vdash_b (e{::}expr,b)\ \surd \longrightarrow P,sh' \vdash_b (e'{::}expr,b')\ \surd$
shows $P,sh' \vdash_b (e';;e_2,b')\ \surd$
$\langle proof\rangle$

theorem *red-preserves-bconf*:
$P \vdash \langle e,(h,l,sh),b\rangle \to \langle e',(h',l',sh'),b'\rangle \Longrightarrow iconf\ sh\ e \Longrightarrow P,sh \vdash_b (e,b)\ \surd \Longrightarrow P,sh' \vdash_b (e',b')\ \surd$
and *reds-preserves-bconf*:
$P \vdash \langle es,(h,l,sh),b\rangle\ [\to]\ \langle es',(h',l',sh'),b'\rangle \Longrightarrow iconfs\ sh\ es \Longrightarrow P,sh \vdash_b (es,b)\ \surd \Longrightarrow P,sh' \vdash_b (es',b')\ \surd\langle proof\rangle$

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

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

lemma *red-lA-incr*: $P \vdash \langle e,(h,l,sh),b\rangle \to \langle e',(h',l',sh'),b'\rangle$

$\Longrightarrow \lfloor dom\ l \rfloor \sqcup \mathcal{A}\ e \sqsubseteq \lfloor dom\ l' \rfloor \sqcup \mathcal{A}\ e'$
and *reds-lA-incr*: $P \vdash \langle es,(h,l,sh),b\rangle\ [\rightarrow]\ \langle es',(h',l',sh'),b'\rangle$
$\Longrightarrow \lfloor dom\ l \rfloor \sqcup \mathcal{A}s\ es \sqsubseteq \lfloor dom\ l' \rfloor \sqcup \mathcal{A}s\ es' \langle proof \rangle$

Now preservation of definite assignment.

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

Combining conformance of heap, static heap, and local variables:

definition *sconf* :: *J-prog* $\Rightarrow$ *env* $\Rightarrow$ *state* $\Rightarrow$ *bool* $(\langle \text{-,- } \vdash \text{ - } \surd \rangle \quad [51,51,51]50)$
where
$P,E \vdash s\ \surd \ \equiv\ let\ (h,l,sh) = s\ in\ P \vdash h\ \surd \wedge P,h \vdash l\ (:\leq)\ E \wedge P,h \vdash_s sh\ \surd$

lemma *red-preserves-sconf*:
$[\![\ P \vdash \langle e,s,b\rangle \rightarrow \langle e',s',b'\rangle;\ P,E,hp\ s,shp\ s \vdash e : T;\ P,E \vdash s\ \surd\ ]\!] \Longrightarrow P,E \vdash s'\ \surd \langle proof \rangle$
lemma *reds-preserves-sconf*:
$[\![\ P \vdash \langle es,s,b\rangle\ [\rightarrow]\ \langle es',s',b'\rangle;\ P,E,hp\ s,shp\ s \vdash es\ [:]\ Ts;\ P,E \vdash s\ \surd\ ]\!] \Longrightarrow P,E \vdash s'\ \surd \langle proof \rangle$

1.22.2 Subject reduction

lemma *wt-blocks*:
$\bigwedge E.\ [\![\ length\ Vs = length\ Ts;\ length\ vs = length\ Ts\ ]\!] \Longrightarrow$
$(P,E,h,sh \vdash blocks(Vs,Ts,vs,e) : T) =$
$(P,E(Vs[\mapsto]Ts),h,sh \vdash e:T \wedge (\exists Ts'.\ map\ (typeof_h)\ vs = map\ Some\ Ts' \wedge P \vdash Ts'\ [\leq]\ Ts))\langle proof \rangle$
theorem assumes *wf*: *wf-J-prog P*
shows *subject-reduction2*: $P \vdash \langle e,(h,l,sh),b\rangle \rightarrow \langle e',(h',l',sh'),b'\rangle \Longrightarrow$
$(\bigwedge E\ T.\ [\![\ P,E \vdash (h,l,sh)\ \surd;\ iconf\ sh\ e;\ P,E,h,sh \vdash e:T\ ]\!]$
$\Longrightarrow \exists T'.\ P,E,h',sh' \vdash e':T' \wedge P \vdash T' \leq T)$
and *subjects-reduction2*: $P \vdash \langle es,(h,l,sh),b\rangle\ [\rightarrow]\ \langle es',(h',l',sh'),b'\rangle \Longrightarrow$
$(\bigwedge E\ Ts.\ [\![\ P,E \vdash (h,l,sh)\ \surd;\ iconfs\ sh\ es;\ P,E,h,sh \vdash es\ [:]\ Ts\ ]\!]$
$\Longrightarrow \exists Ts'.\ P,E,h',sh' \vdash es'\ [:]\ Ts' \wedge P \vdash Ts'\ [\leq]\ Ts)\langle proof \rangle$

corollary *subject-reduction*:
$[\![\ wf\text{-}J\text{-}prog\ P;\ P \vdash \langle e,s,b\rangle \rightarrow \langle e',s',b'\rangle;\ P,E \vdash s\ \surd;\ iconf\ (shp\ s)\ e;\ P,E,hp\ s,shp\ s \vdash e:T\ ]\!]$
$\Longrightarrow \exists T'.\ P,E,hp\ s',shp\ s' \vdash e':T' \wedge P \vdash T' \leq T \langle proof \rangle$
corollary *subjects-reduction*:
$[\![\ wf\text{-}J\text{-}prog\ P;\ P \vdash \langle es,s,b\rangle\ [\rightarrow]\ \langle es',s',b'\rangle;\ P,E \vdash s\ \surd;\ iconfs\ (shp\ s)\ es;\ P,E,hp\ s,shp\ s \vdash es[:]Ts\ ]\!]$
$\Longrightarrow \exists Ts'.\ P,E,hp\ s',shp\ s' \vdash es'[:]Ts' \wedge P \vdash Ts'\ [\leq]\ Ts \langle proof \rangle$

1.22.3 Lifting to $\rightarrow*$

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

lemma *Red-preserves-sconf*:
assumes *wf*: *wf-J-prog P* **and** *Red*: $P \vdash \langle e,s,b\rangle \rightarrow* \langle e',s',b'\rangle$
shows $\bigwedge T.\ [\![\ P,E,hp\ s,shp\ s \vdash e : T;\ iconf\ (shp\ s)\ e;\ P,E \vdash s\ \surd\ ]\!] \Longrightarrow P,E \vdash s'\ \surd \langle proof \rangle$
lemma *Red-preserves-iconf*:
assumes *wf*: *wwf-J-prog P* **and** *Red*: $P \vdash \langle e,s,b\rangle \rightarrow* \langle e',s',b'\rangle$
shows $iconf\ (shp\ s)\ e \Longrightarrow iconf\ (shp\ s')\ e' \langle proof \rangle$
lemma *Reds-preserves-iconf*:
assumes *wf*: *wwf-J-prog P* **and** *Red*: $P \vdash \langle es,s,b\rangle\ [\rightarrow]* \langle es',s',b'\rangle$
shows $iconfs\ (shp\ s)\ es \Longrightarrow iconfs\ (shp\ s')\ es' \langle proof \rangle$
lemma *Red-preserves-bconf*:

assumes *wf*: *wwf-J-prog P* **and** *Red*: $P \vdash \langle e,s,b\rangle \to* \langle e',s',b'\rangle$
shows *iconf* (*shp s*) *e* $\Longrightarrow$ $P,(shp\ s) \vdash_b (e,b)\ \surd \Longrightarrow P,(shp\ s') \vdash_b (e'::expr,b')\ \surd\langle proof\rangle$
lemma *Reds-preserves-bconf*:
assumes *wf*: *wwf-J-prog P* **and** *Red*: $P \vdash \langle es,s,b\rangle\ [\to]* \langle es',s',b'\rangle$
shows *iconfs* (*shp s*) *es* $\Longrightarrow$ $P,(shp\ s) \vdash_b (es,b)\ \surd \Longrightarrow P,(shp\ s') \vdash_b (es'::expr\ list,b')\ \surd\langle proof\rangle$
lemma *Red-preserves-defass*:
assumes *wf*: *wf-J-prog P* **and** *reds*: $P \vdash \langle e,s,b\rangle \to* \langle e',s',b'\rangle$
shows $\mathcal{D}\ e\ \lfloor dom(lcl\ s)\rfloor \Longrightarrow \mathcal{D}\ e'\ \lfloor dom(lcl\ s')\rfloor$
$\langle proof\rangle$

lemma *Red-preserves-type*:
assumes *wf*: *wf-J-prog P* **and** *Red*: $P \vdash \langle e,s,b\rangle \to* \langle e',s',b'\rangle$
shows $!!T.\ [\![\ P,E \vdash s\surd;\ iconf\ (shp\ s)\ e;\ P,E,hp\ s,shp\ s \vdash e:T\ ]\!]$
$\Longrightarrow \exists T'.\ P \vdash T' \le T \wedge P,E,hp\ s',shp\ s' \vdash e':T'\langle proof\rangle$

1.22.4 The final polish

The above preservation lemmas are now combined and packed nicely.

definition *wf-config* :: *J-prog* ⇒ *env* ⇒ *state* ⇒ *expr* ⇒ *ty* ⇒ *bool* $(\langle\text{-},\text{-},\text{-} \vdash \text{-} : \text{-}\ \surd\rangle\ [51,0,0,0,0]50)$
where
$P,E,s \vdash e:T\ \surd \equiv P,E \vdash s\ \surd \wedge iconf\ (shp\ s)\ e \wedge P,E,hp\ s,shp\ s \vdash e:T$

theorem *Subject-reduction*: **assumes** *wf*: *wf-J-prog P*
shows $P \vdash \langle e,s,b\rangle \to \langle e',s',b'\rangle \Longrightarrow P,E,s \vdash e : T\ \surd$
$\Longrightarrow \exists T'.\ P,E,s' \vdash e' : T'\ \surd \wedge P \vdash T' \le T\langle proof\rangle$

theorem *Subject-reductions*:
assumes *wf*: *wf-J-prog P* **and** *reds*: $P \vdash \langle e,s,b\rangle \to* \langle e',s',b'\rangle$
shows $\bigwedge T.\ P,E,s \vdash e:T\ \surd \Longrightarrow \exists T'.\ P,E,s' \vdash e':T'\ \surd \wedge P \vdash T' \le T\langle proof\rangle$

corollary *Progress*: **assumes** *wf*: *wf-J-prog P*
shows $[\![\ P,E,s\ \vdash e : T\ \surd;\ \mathcal{D}\ e\ \lfloor dom(lcl\ s)\rfloor;\ P,shp\ s \vdash_b (e,b)\ \surd;\ \neg\ final\ e\ ]\!]$
$\Longrightarrow \exists e'\ s'\ b'.\ P \vdash \langle e,s,b\rangle \to \langle e',s',b'\rangle\langle proof\rangle$
corollary *TypeSafety*:
fixes *s*::*state* **and** *e*::*expr*
assumes *wf*: *wf-J-prog P* **and** *sconf*: $P,E \vdash s\ \surd$ **and** *wt*: $P,E \vdash e::T$
and $\mathcal{D}$: $\mathcal{D}\ e\ \lfloor dom(lcl\ s)\rfloor$
and *iconf*: *iconf* (*shp s*) *e* **and** *bconf*: $P,(shp\ s) \vdash_b (e,b)\ \surd$
and *steps*: $P \vdash \langle e,s,b\rangle \to* \langle e',s',b'\rangle$
and *nstep*: $\neg(\exists e''\ s''\ b''.\ P \vdash \langle e',s',b'\rangle \to \langle e'',s'',b''\rangle)$
shows $(\exists v.\ e' = Val\ v \wedge P,hp\ s' \vdash v :\le T) \vee$
$(\exists a.\ e' = Throw\ a \wedge a \in dom(hp\ s'))\langle proof\rangle$

end

1.23 Equivalence of Big Step and Small Step Semantics

theory *Equivalence* **imports** *TypeSafe WWellForm* **begin**

1.23.1 Small steps simulate big step

Init

The reduction of initialization expressions doesn't touch or use their on-hold expressions (the subexpression to the right of ←) until the initialization operation completes. This function is used to prove this and related properties. It is then used for general reduction of initialization expressions separately from their on-hold expressions by using the on-hold expression *unit*, then putting the real on-hold expression back in at the end.

fun *init-switch* :: *'a exp* ⇒ *'a exp* ⇒ *'a exp* **where**
init-switch (*INIT C* (*Cs*,*b*) ← e_i) *e* = (*INIT C* (*Cs*,*b*) ← *e*) |
init-switch (*RI*(*C*,*e'*);*Cs* ← e_i) *e* = (*RI*(*C*,*e'*);*Cs* ← *e*) |
init-switch e' e = *e'*

fun *INIT-class* :: *'a exp* ⇒ *cname option* **where**
INIT-class (*INIT C* (*Cs*,*b*) ← *e*) = (*if C* = *last* (*C*#*Cs*) *then Some C else None*) |
INIT-class (*RI*(*C*,e_0);*Cs* ← *e*) = *Some* (*last* (*C*#*Cs*)) |
INIT-class - = *None*

lemma *init-red-init*:
⟦ *init-exp-of* e_0 = ⌊*e*⌋; *P* ⊢ ⟨e_0,s_0,b_0⟩ → ⟨e_1,s_1,b_1⟩ ⟧
⟹ (*init-exp-of* e_1 = ⌊*e*⌋ ∧ (*INIT-class* e_0 = ⌊*C*⌋ ⟶ *INIT-class* e_1 = ⌊*C*⌋))
∨ (e_1 = *e* ∧ b_1 = *icheck P* (*the*(*INIT-class* e_0)) *e*) ∨ (∃ *a*. e_1 = *throw a*)
⟨*proof*⟩

lemma *init-exp-switch*[*simp*]:
init-exp-of e_0 = ⌊*e*⌋ ⟹ *init-exp-of* (*init-switch* e_0 *e'*) = ⌊*e'*⌋
⟨*proof*⟩

lemma *init-red-sync*:
⟦ *P* ⊢ ⟨e_0,s_0,b_0⟩ → ⟨e_1,s_1,b_1⟩; *init-exp-of* e_0 = ⌊*e*⌋; e_1 ≠ *e* ⟧
⟹ (⋀*e'*. *P* ⊢ ⟨*init-switch* e_0 *e'*,s_0,b_0⟩ → ⟨*init-switch* e_1 *e'*,s_1,b_1⟩)
⟨*proof*⟩

lemma *init-red-sync-end*:
⟦ *P* ⊢ ⟨e_0,s_0,b_0⟩ → ⟨e_1,s_1,b_1⟩; *init-exp-of* e_0 = ⌊*e*⌋; e_1 = *e*; *throw-of e* = *None* ⟧
⟹ (⋀*e'*. ¬*sub-RI e'* ⟹ *P* ⊢ ⟨*init-switch* e_0 *e'*,s_0,b_0⟩ → ⟨*e'*,s_1,*icheck P* (*the*(*INIT-class* e_0)) *e'*⟩)
⟨*proof*⟩

lemma *init-reds-sync-unit'*:
⟦ *P* ⊢ ⟨e_0,s_0,b_0⟩ →* ⟨*Val v'*,s_1,b_1⟩; *init-exp-of* e_0 = ⌊*unit*⌋; *INIT-class* e_0 = ⌊*C*⌋ ⟧
⟹ (⋀*e'*. ¬*sub-RI e'* ⟹ *P* ⊢ ⟨*init-switch* e_0 *e'*,s_0,b_0⟩ →* ⟨*e'*,s_1,*icheck P* (*the*(*INIT-class* e_0)) *e'*⟩)
⟨*proof*⟩

lemma *init-reds-sync-unit-throw'*:
⟦ *P* ⊢ ⟨e_0,s_0,b_0⟩ →* ⟨*throw a*,s_1,b_1⟩; *init-exp-of* e_0 = ⌊*unit*⌋ ⟧
⟹ (⋀*e'*. *P* ⊢ ⟨*init-switch* e_0 *e'*,s_0,b_0⟩ →* ⟨*throw a*,s_1,b_1⟩)
⟨*proof*⟩

lemma *init-reds-sync-unit*:
assumes *P* ⊢ ⟨e_0,s_0,b_0⟩ →* ⟨*Val v'*,s_1,b_1⟩ **and** *init-exp-of* e_0 = ⌊*unit*⌋ **and** *INIT-class* e_0 = ⌊*C*⌋
and ¬*sub-RI e'*
shows *P* ⊢ ⟨*init-switch* e_0 *e'*,s_0,b_0⟩ →* ⟨*e'*,s_1,*icheck P* (*the*(*INIT-class* e_0)) *e'*⟩
⟨*proof*⟩

lemma *init-reds-sync-unit-throw*:
assumes $P \vdash \langle e_0,s_0,b_0\rangle \rightarrow* \langle throw\ a,s_1,b_1\rangle$ **and** *init-exp-of* $e_0 = \lfloor unit \rfloor$
shows $P \vdash \langle init\text{-}switch\ e_0\ e',s_0,b_0\rangle \rightarrow* \langle throw\ a,s_1,b_1\rangle$
⟨*proof*⟩

lemma *InitSeqReds*:
assumes $P \vdash \langle INIT\ C\ ([C],b) \leftarrow unit,s_0,b_0\rangle \rightarrow* \langle Val\ v',s_1,b_1\rangle$
and $P \vdash \langle e,s_1,icheck\ P\ C\ e\rangle \rightarrow* \langle e_2,s_2,b_2\rangle$ **and** $\neg sub\text{-}RI\ e$
shows $P \vdash \langle INIT\ C\ ([C],b) \leftarrow e,s_0,b_0\rangle \rightarrow* \langle e_2,s_2,b_2\rangle$
⟨*proof*⟩

lemma *InitSeqThrowReds*:
assumes $P \vdash \langle INIT\ C\ ([C],b) \leftarrow unit,s_0,b_0\rangle \rightarrow* \langle throw\ a,s_1,b_1\rangle$
shows $P \vdash \langle INIT\ C\ ([C],b) \leftarrow e,s_0,b_0\rangle \rightarrow* \langle throw\ a,s_1,b_1\rangle$
⟨*proof*⟩

lemma *InitNoneReds*:
$[\![\ sh\ C = None;$
$P \vdash \langle INIT\ C'\ (C\ \#\ Cs,False) \leftarrow e,(h,\ l,\ sh(C \mapsto (sblank\ P\ C,\ Prepared))),b\rangle \rightarrow* \langle e',s',b'\rangle\ ]\!]$
$\Longrightarrow P \vdash \langle INIT\ C'\ (C\#Cs,False) \leftarrow e,(h,l,sh),b\rangle \rightarrow* \langle e',s',b'\rangle\langle proof\rangle$
lemma *InitDoneReds*:
$[\![\ sh\ C = Some(sfs,Done);\ P \vdash \langle INIT\ C'\ (Cs,True) \leftarrow e,(h,l,sh),b\rangle \rightarrow* \langle e',s',b'\rangle\ ]\!]$
$\Longrightarrow P \vdash \langle INIT\ C'\ (C\#Cs,False) \leftarrow e,(h,l,sh),b\rangle \rightarrow* \langle e',s',b'\rangle\langle proof\rangle$
lemma *InitProcessingReds*:
$[\![\ sh\ C = Some(sfs,Processing);\ P \vdash \langle INIT\ C'\ (Cs,True) \leftarrow e,(h,l,sh),b\rangle \rightarrow* \langle e',s',b'\rangle\ ]\!]$
$\Longrightarrow P \vdash \langle INIT\ C'\ (C\#Cs,False) \leftarrow e,(h,l,sh),b\rangle \rightarrow* \langle e',s',b'\rangle\langle proof\rangle$
lemma *InitErrorReds*:
$[\![\ sh\ C = Some(sfs,Error);\ P \vdash \langle RI\ (C,THROW\ NoClassDefFoundError);Cs \leftarrow e,(h,l,sh),b\rangle \rightarrow*$
$\langle e',s',b'\rangle\ ]\!]$
$\Longrightarrow P \vdash \langle INIT\ C'\ (C\#Cs,False) \leftarrow e,(h,l,sh),b\rangle \rightarrow* \langle e',s',b'\rangle\langle proof\rangle$
lemma *InitObjectReds*:
$[\![\ sh\ C = Some(sfs,Prepared);\ C = Object;\ sh' = sh(C \mapsto (sfs,Processing));$
$P \vdash \langle INIT\ C'\ (C\#Cs,True) \leftarrow e,(h,l,sh'),b\rangle \rightarrow* \langle e',s',b'\rangle\ ]\!]$
$\Longrightarrow P \vdash \langle INIT\ C'\ (C\#Cs,False) \leftarrow e,(h,l,sh),b\rangle \rightarrow* \langle e',s',b'\rangle\langle proof\rangle$
lemma *InitNonObjectReds*:
$[\![\ sh\ C = Some(sfs,Prepared);\ C \neq Object;\ class\ P\ C = Some\ (D,r);$
$sh' = sh(C \mapsto (sfs,Processing));$
$P \vdash \langle INIT\ C'\ (D\#C\#Cs,False) \leftarrow e,(h,l,sh'),b\rangle \rightarrow* \langle e',s',b'\rangle\ ]\!]$
$\Longrightarrow P \vdash \langle INIT\ C'\ (C\#Cs,False) \leftarrow e,(h,l,sh),b\rangle \rightarrow* \langle e',s',b'\rangle\langle proof\rangle$
lemma *RedsInitRInit*:
$P \vdash \langle RI\ (C,C\cdot_s clinit([]));Cs \leftarrow e,(h,l,sh),b\rangle \rightarrow* \langle e',s',b'\rangle$
$\Longrightarrow P \vdash \langle INIT\ C'\ (C\#Cs,True) \leftarrow e,(h,l,sh),b\rangle \rightarrow* \langle e',s',b'\rangle\langle proof\rangle$
lemmas *rtrancl-induct3* =
rtrancl-induct[*of* (ax,ay,az) (bx,by,bz), *split-format* (*complete*), *consumes 1*, *case-names refl step*]

lemma *RInitReds*:
$P \vdash \langle e,s,b\rangle \rightarrow* \langle e',s',b'\rangle$
$\Longrightarrow P \vdash \langle RI\ (C,e);Cs \leftarrow e_0,\ s,\ b\rangle \rightarrow* \langle RI\ (C,e');Cs \leftarrow e_0,\ s',\ b'\rangle\langle proof\rangle$
lemma *RedsRInit*:
$[\![\ P \vdash \langle e_0,s_0,b_0\rangle \rightarrow* \langle Val\ v,(h_1,l_1,sh_1),b_1\rangle;$
$sh_1\ C = Some\ (sfs,\ i);\ sh_2 = sh_1(C \mapsto (sfs,Done));\ C' = last(C\#Cs);$
$P \vdash \langle INIT\ C'\ (Cs,True) \leftarrow e,(h_1,l_1,sh_2),b_1\rangle \rightarrow* \langle e',s',b'\rangle\ ]\!]$
$\Longrightarrow P \vdash \langle RI\ (C,\ e_0);Cs \leftarrow e,s_0,b_0\rangle \rightarrow* \langle e',s',b'\rangle\langle proof\rangle$

lemma *RInitInitThrowReds*:
⟦ $P \vdash \langle e,s,b\rangle \to* \langle Throw\ a,(h',l',sh'),b'\rangle$;
$sh'\ C = Some\ (sfs,\ i)$; $sh'' = sh'(C \mapsto (sfs,Error))$;
$P \vdash \langle RI\ (D,Throw\ a);Cs \leftarrow e_0,\ (h',l',sh''),b'\rangle \to* \langle e_1,s_1,b_1\rangle$ ⟧
⟹ $P \vdash \langle RI\ (C,\ e);D\#Cs \leftarrow e_0,s,b\rangle \to* \langle e_1,s_1,b_1\rangle\langle proof\rangle$
lemma *RInitThrowReds*:
⟦ $P \vdash \langle e,s,b\rangle \to* \langle Throw\ a,\ (h',l',sh'),b'\rangle$;
$sh'\ C = Some(sfs,\ i)$; $sh'' = sh'(C \mapsto (sfs,\ Error))$ ⟧
⟹ $P \vdash \langle RI\ (C,e);Nil \leftarrow e_0,s,b\rangle \to* \langle Throw\ a,\ (h',l',sh''),b'\rangle\langle proof\rangle$

New

lemma *NewInitDoneReds*:
⟦ $sh\ C = Some\ (sfs,\ Done)$; $new\text{-}Addr\ h = Some\ a$;
$P \vdash C\ has\text{-}fields\ FDTs$; $h' = h(a \mapsto blank\ P\ C)$ ⟧
⟹ $P \vdash \langle new\ C,(h,l,sh),False\rangle \to* \langle addr\ a,(h',l,sh),False\rangle\langle proof\rangle$
lemma *NewInitDoneReds2*:
⟦ $sh\ C = Some\ (sfs,\ Done)$; $new\text{-}Addr\ h = None$; $is\text{-}class\ P\ C$ ⟧
⟹ $P \vdash \langle new\ C,(h,l,sh),False\rangle \to* \langle THROW\ OutOfMemory,\ (h,l,sh),\ False\rangle\langle proof\rangle$
lemma *NewInitReds*:
assumes *nDone*: $\nexists sfs.\ shp\ s\ C = Some\ (sfs,\ Done)$
and *INIT-steps*: $P \vdash \langle INIT\ C\ ([C],False) \leftarrow unit,s,False\rangle \to* \langle Val\ v',(h',l',sh'),b'\rangle$
and *Addr*: $new\text{-}Addr\ h' = Some\ a$ **and** *has*: $P \vdash C\ has\text{-}fields\ FDTs$
and h'': $h'' = h'(a \mapsto blank\ P\ C)$ **and** *cls-C*: $is\text{-}class\ P\ C$
shows $P \vdash \langle new\ C,s,False\rangle \to* \langle addr\ a,(h'',l',sh'),False\rangle\langle proof\rangle$
lemma *NewInitOOMReds*:
assumes *nDone*: $\nexists sfs.\ shp\ s\ C = Some\ (sfs,\ Done)$
and *INIT-steps*: $P \vdash \langle INIT\ C\ ([C],False) \leftarrow unit,s,False\rangle \to* \langle Val\ v',(h',l',sh'),b'\rangle$
and *Addr*: $new\text{-}Addr\ h' = None$ **and** *cls-C*: $is\text{-}class\ P\ C$
shows $P \vdash \langle new\ C,s,False\rangle \to* \langle THROW\ OutOfMemory,(h',l',sh'),False\rangle\langle proof\rangle$
lemma *NewInitThrowReds*:
assumes *nDone*: $\nexists sfs.\ shp\ s\ C = Some\ (sfs,\ Done)$
and *cls-C*: $is\text{-}class\ P\ C$
and *INIT-steps*: $P \vdash \langle INIT\ C\ ([C],False) \leftarrow unit,s,False\rangle \to* \langle throw\ a,s',b'\rangle$
shows $P \vdash \langle new\ C,s,False\rangle \to* \langle throw\ a,s',b'\rangle\langle proof\rangle$

Cast

lemma *CastReds*:
$P \vdash \langle e,s,b\rangle \to* \langle e',s',b'\rangle \Longrightarrow P \vdash \langle Cast\ C\ e,s,b\rangle \to* \langle Cast\ C\ e',s',b'\rangle\langle proof\rangle$
lemma *CastRedsNull*:
$P \vdash \langle e,s,b\rangle \to* \langle null,s',b'\rangle \Longrightarrow P \vdash \langle Cast\ C\ e,s,b\rangle \to* \langle null,s',b'\rangle\langle proof\rangle$
lemma *CastRedsAddr*:
⟦ $P \vdash \langle e,s,b\rangle \to* \langle addr\ a,s',b'\rangle$; $hp\ s'\ a = Some(D,fs)$; $P \vdash D \preceq^* C$ ⟧ ⟹
$P \vdash \langle Cast\ C\ e,s,b\rangle \to* \langle addr\ a,s',b'\rangle\langle proof\rangle$
lemma *CastRedsFail*:
⟦ $P \vdash \langle e,s,b\rangle \to* \langle addr\ a,s',b'\rangle$; $hp\ s'\ a = Some(D,fs)$; $\neg\ P \vdash D \preceq^* C$ ⟧ ⟹
$P \vdash \langle Cast\ C\ e,s,b\rangle \to* \langle THROW\ ClassCast,s',b'\rangle\langle proof\rangle$
lemma *CastRedsThrow*:
⟦ $P \vdash \langle e,s,b\rangle \to* \langle throw\ a,s',b'\rangle$ ⟧ ⟹ $P \vdash \langle Cast\ C\ e,s,b\rangle \to* \langle throw\ a,s',b'\rangle\langle proof\rangle$

LAss

lemma *LAssReds*:

$P \vdash \langle e,s,b\rangle \to* \langle e',s',b'\rangle \Longrightarrow P \vdash \langle\ V{:=}e,s,b\rangle \to* \langle\ V{:=}e',s',b'\rangle\langle proof\rangle$
lemma *LAssRedsVal*:
$[\![\ P \vdash \langle e,s,b\rangle \to* \langle Val\ v,(h',l',sh'),b'\rangle\ ]\!] \Longrightarrow P \vdash \langle\ V{:=}e,s,b\rangle \to* \langle unit,(h',l'(V{\mapsto}v),sh'),b'\rangle\langle proof\rangle$
lemma *LAssRedsThrow*:
$[\![\ P \vdash \langle e,s,b\rangle \to* \langle throw\ a,s',b'\rangle\ ]\!] \Longrightarrow P \vdash \langle\ V{:=}e,s,b\rangle \to* \langle throw\ a,s',b'\rangle\langle proof\rangle$

BinOp

lemma *BinOp1Reds*:
$P \vdash \langle e,s,b\rangle \to* \langle e',s',b'\rangle \Longrightarrow P \vdash \langle\ e\ «bop»\ e_2,\ s,b\rangle \to* \langle e'\ «bop»\ e_2,\ s',b'\rangle\langle proof\rangle$
lemma *BinOp2Reds*:
$P \vdash \langle e,s,b\rangle \to* \langle e',s',b'\rangle \Longrightarrow P \vdash \langle(Val\ v)\ «bop»\ e,\ s,b\rangle \to* \langle(Val\ v)\ «bop»\ e',\ s',b'\rangle\langle proof\rangle$
lemma *BinOpRedsVal*:
assumes e_1*-steps*: $P \vdash \langle e_1,s_0,b_0\rangle \to* \langle Val\ v_1,s_1,b_1\rangle$
and e_2*-steps*: $P \vdash \langle e_2,s_1,b_1\rangle \to* \langle Val\ v_2,s_2,b_2\rangle$
and *op*: $binop(bop,v_1,v_2) = Some\ v$
shows $P \vdash \langle e_1\ «bop»\ e_2,\ s_0,b_0\rangle \to* \langle Val\ v,s_2,b_2\rangle\langle proof\rangle$
lemma *BinOpRedsThrow1*:
$P \vdash \langle e,s,b\rangle \to* \langle throw\ e',s',b'\rangle \Longrightarrow P \vdash \langle e\ «bop»\ e_2,\ s,b\rangle \to* \langle throw\ e',\ s',b'\rangle\langle proof\rangle$
lemma *BinOpRedsThrow2*:
assumes e_1*-steps*: $P \vdash \langle e_1,s_0,b_0\rangle \to* \langle Val\ v_1,s_1,b_1\rangle$
and e_2*-steps*: $P \vdash \langle e_2,s_1,b_1\rangle \to* \langle throw\ e,s_2,b_2\rangle$
shows $P \vdash \langle e_1\ «bop»\ e_2,\ s_0,b_0\rangle \to* \langle throw\ e,s_2,b_2\rangle\langle proof\rangle$

FAcc

lemma *FAccReds*:
$P \vdash \langle e,s,b\rangle \to* \langle e',s',b'\rangle \Longrightarrow P \vdash \langle e{\cdot}F\{D\},\ s,b\rangle \to* \langle e'{\cdot}F\{D\},\ s',b'\rangle\langle proof\rangle$
lemma *FAccRedsVal*:
$[\![\ P \vdash \langle e,s,b\rangle \to* \langle addr\ a,s',b'\rangle;\ hp\ s'\ a = Some(C,fs);\ fs(F,D) = Some\ v;$
$P \vdash C\ has\ F,NonStatic{:}t\ in\ D\ ]\!]$
$\Longrightarrow P \vdash \langle e{\cdot}F\{D\},s,b\rangle \to* \langle Val\ v,s',b'\rangle\langle proof\rangle$
lemma *FAccRedsNull*:
$P \vdash \langle e,s,b\rangle \to* \langle null,s',b'\rangle \Longrightarrow P \vdash \langle e{\cdot}F\{D\},s,b\rangle \to* \langle THROW\ NullPointer,s',b'\rangle\langle proof\rangle$
lemma *FAccRedsNone*:
$[\![\ P \vdash \langle e,s,b\rangle \to* \langle addr\ a,s',b'\rangle;$
$hp\ s'\ a = Some(C,fs);$
$\neg(\exists\ b\ t.\ P \vdash C\ has\ F,b{:}t\ in\ D)\ ]\!]$
$\Longrightarrow P \vdash \langle e{\cdot}F\{D\},s,b\rangle \to* \langle THROW\ NoSuchFieldError,s',b'\rangle\langle proof\rangle$
lemma *FAccRedsStatic*:
$[\![\ P \vdash \langle e,s,b\rangle \to* \langle addr\ a,s',b'\rangle;$
$hp\ s'\ a = Some(C,fs);$
$P \vdash C\ has\ F,Static{:}t\ in\ D\ ]\!]$
$\Longrightarrow P \vdash \langle e{\cdot}F\{D\},s,b\rangle \to* \langle THROW\ IncompatibleClassChangeError,s',b'\rangle\langle proof\rangle$
lemma *FAccRedsThrow*:
$P \vdash \langle e,s,b\rangle \to* \langle throw\ a,s',b'\rangle \Longrightarrow P \vdash \langle e{\cdot}F\{D\},s,b\rangle \to* \langle throw\ a,s',b'\rangle\langle proof\rangle$

SFAcc

lemma *SFAccReds*:
$[\![\ P \vdash C\ has\ F,Static{:}t\ in\ D;$
$shp\ s\ D = Some(sfs,Done);\ sfs\ F = Some\ v\ ]\!]$
$\Longrightarrow P \vdash \langle C{\cdot}_s F\{D\},s,True\rangle \to* \langle Val\ v,s,False\rangle\langle proof\rangle$
lemma *SFAccRedsNone*:
$\neg(\exists\ b\ t.\ P \vdash C\ has\ F,b{:}t\ in\ D)$

$\Longrightarrow P \vdash \langle C\cdot_s F\{D\},s,b\rangle \to* \langle THROW\ NoSuchFieldError,s,False\rangle\langle proof\rangle$
lemma *SFAccRedsNonStatic*:
$P \vdash C\ has\ F,NonStatic{:}t\ in\ D$
$\Longrightarrow P \vdash \langle C\cdot_s F\{D\},s,b\rangle \to* \langle THROW\ IncompatibleClassChangeError,s,False\rangle\langle proof\rangle$
lemma *SFAccInitDoneReds*:
assumes *cF*: $P \vdash C\ has\ F,Static{:}t\ in\ D$
and *shp*: $shp\ s\ D = Some\ (sfs,Done)$ **and** *sfs*: $sfs\ F = Some\ v$
shows $P \vdash \langle C\cdot_s F\{D\},\ s,\ b\rangle \to* \langle Val\ v,\ s,\ False\rangle\langle proof\rangle$
lemma *SFAccInitReds*:
assumes *cF*: $P \vdash C\ has\ F,Static{:}t\ in\ D$
and *nDone*: $\nexists sfs.\ shp\ s\ D = Some\ (sfs,Done)$
and *INIT-steps*: $P \vdash \langle INIT\ D\ ([D],False) \leftarrow unit,s,False\rangle \to* \langle Val\ v',s',b'\rangle$
and *shp'*: $shp\ s'\ D = Some\ (sfs,i)$ **and** *sfs*: $sfs\ F = Some\ v$
shows $P \vdash \langle C\cdot_s F\{D\},s,False\rangle \to* \langle Val\ v,s',False\rangle\langle proof\rangle$
lemma *SFAccInitThrowReds*:
$[\![\ P \vdash C\ has\ F,Static{:}t\ in\ D;$
$\nexists sfs.\ shp\ s\ D = Some\ (sfs,Done);$
$P \vdash \langle INIT\ D\ ([D],False) \leftarrow unit,s,False\rangle \to* \langle throw\ a,s',b'\rangle\ ]\!]$
$\Longrightarrow P \vdash \langle C\cdot_s F\{D\},s,False\rangle \to* \langle throw\ a,s',b'\rangle\langle proof\rangle$

FAss

lemma *FAssReds1*:
$P \vdash \langle e,s,b\rangle \to* \langle e',s',b'\rangle \Longrightarrow P \vdash \langle e\cdot F\{D\}{:=}e_2,\ s,b\rangle \to* \langle e'\cdot F\{D\}{:=}e_2,\ s',b'\rangle\langle proof\rangle$
lemma *FAssReds2*:
$P \vdash \langle e,s,b\rangle \to* \langle e',s',b'\rangle \Longrightarrow P \vdash \langle Val\ v\cdot F\{D\}{:=}e,\ s,b\rangle \to* \langle Val\ v\cdot F\{D\}{:=}e',\ s',b'\rangle\langle proof\rangle$
lemma *FAssRedsVal*:
assumes e_1*-steps*: $P \vdash \langle e_1,s_0,b_0\rangle \to* \langle addr\ a,s_1,b_1\rangle$
and e_2*-steps*: $P \vdash \langle e_2,s_1,b_1\rangle \to* \langle Val\ v,(h_2,l_2,sh_2),b_2\rangle$
and *cF*: $P \vdash C\ has\ F,NonStatic{:}t\ in\ D$ **and** *hC*: $Some(C,fs) = h_2\ a$
shows $P \vdash \langle e_1\cdot F\{D\}{:=}e_2,\ s_0,\ b_0\rangle \to* \langle unit,\ (h_2(a{\mapsto}(C,fs((F,D){\mapsto}v))),l_2,sh_2),b_2\rangle\langle proof\rangle$
lemma *FAssRedsNull*:
assumes e_1*-steps*: $P \vdash \langle e_1,s_0,b_0\rangle \to* \langle null,s_1,b_1\rangle$
and e_2*-steps*: $P \vdash \langle e_2,s_1,b_1\rangle \to* \langle Val\ v,s_2,b_2\rangle$
shows $P \vdash \langle e_1\cdot F\{D\}{:=}e_2,\ s_0,\ b_0\rangle \to* \langle THROW\ NullPointer,\ s_2,\ b_2\rangle\langle proof\rangle$
lemma *FAssRedsThrow1*:
$P \vdash \langle e,s,b\rangle \to* \langle throw\ e',s',b'\rangle \Longrightarrow P \vdash \langle e\cdot F\{D\}{:=}e_2,\ s,b\rangle \to* \langle throw\ e',\ s',b'\rangle\langle proof\rangle$
lemma *FAssRedsThrow2*:
assumes e_1*-steps*: $P \vdash \langle e_1,s_0,b_0\rangle \to* \langle Val\ v,s_1,b_1\rangle$
and e_2*-steps*: $P \vdash \langle e_2,s_1,b_1\rangle \to* \langle throw\ e,s_2,b_2\rangle$
shows $P \vdash \langle e_1\cdot F\{D\}{:=}e_2,s_0,b_0\rangle \to* \langle throw\ e,s_2,b_2\rangle\langle proof\rangle$
lemma *FAssRedsNone*:
assumes e_1*-steps*: $P \vdash \langle e_1,s_0,b_0\rangle \to* \langle addr\ a,s_1,b_1\rangle$
and e_2*-steps*: $P \vdash \langle e_2,s_1,b_1\rangle \to* \langle Val\ v,(h_2,l_2,sh_2),b_2\rangle$
and *hC*: $h_2\ a = Some(C,fs)$ **and** *ncF*: $\neg(\exists\ b\ t.\ P \vdash C\ has\ F,b{:}t\ in\ D)$
shows $P \vdash \langle e_1\cdot F\{D\}{:=}e_2,\ s_0,\ b_0\rangle \to* \langle THROW\ NoSuchFieldError,\ (h_2,l_2,sh_2),\ b_2\rangle\langle proof\rangle$
lemma *FAssRedsStatic*:
assumes e_1*-steps*: $P \vdash \langle e_1,s_0,b_0\rangle \to* \langle addr\ a,s_1,b_1\rangle$
and e_2*-steps*: $P \vdash \langle e_2,s_1,b_1\rangle \to* \langle Val\ v,(h_2,l_2,sh_2),b_2\rangle$
and *hC*: $h_2\ a = Some(C,fs)$ **and** *cF-Static*: $P \vdash C\ has\ F,Static{:}t\ in\ D$
shows $P \vdash \langle e_1\cdot F\{D\}{:=}e_2,\ s_0,\ b_0\rangle \to* \langle THROW\ IncompatibleClassChangeError,\ (h_2,l_2,sh_2),\ b_2\rangle\langle proof\rangle$

SFAss

lemma *SFAssReds*:
$P \vdash \langle e,s,b\rangle \to* \langle e',s',b'\rangle \Longrightarrow P \vdash \langle C\cdot_s F\{D\}:=e,s,b\rangle \to* \langle C\cdot_s F\{D\}:=e',s',b'\rangle\langle proof\rangle$
lemma *SFAssRedsVal*:
assumes e_2*-steps*: $P \vdash \langle e_2,s_0,b_0\rangle \to* \langle Val\ v,(h_2,l_2,sh_2),b_2\rangle$
and *cF*: $P \vdash C$ *has* F,*Static*:t *in* D
and *shD*: $sh_2\ D = \lfloor(sfs,Done)\rfloor$
shows $P \vdash \langle C\cdot_s F\{D\}:=e_2,\ s_0,\ b_0\rangle \to* \langle unit,\ (h_2,l_2,sh_2(D\mapsto(sfs(F\mapsto v),\ Done))),False\rangle\langle proof\rangle$
lemma *SFAssRedsThrow*:
$[\![\ P \vdash \langle e_2,s_0,b_0\rangle \to* \langle throw\ e,s_2,b_2\rangle\ ]\!]$
$\Longrightarrow P \vdash \langle C\cdot_s F\{D\}:=e_2,s_0,b_0\rangle \to* \langle throw\ e,s_2,b_2\rangle\langle proof\rangle$
lemma *SFAssRedsNone*:
$[\![\ P \vdash \langle e_2,s_0,b_0\rangle \to* \langle Val\ v,(h_2,l_2,sh_2),b_2\rangle;$
$\neg(\exists\ b\ t.\ P \vdash C$ *has* F,b:t *in* $D)\ ]\!] \Longrightarrow$
$P \vdash \langle C\cdot_s F\{D\}:=e_2,s_0,b_0\rangle \to* \langle THROW\ NoSuchFieldError,\ (h_2,l_2,sh_2),False\rangle\langle proof\rangle$
lemma *SFAssRedsNonStatic*:
$[\![\ P \vdash \langle e_2,s_0,b_0\rangle \to* \langle Val\ v,(h_2,l_2,sh_2),b_2\rangle;$
$P \vdash C$ *has* F,*NonStatic*:t *in* $D\ ]\!] \Longrightarrow$
$P \vdash \langle C\cdot_s F\{D\}:=e_2,s_0,b_0\rangle \to* \langle THROW\ IncompatibleClassChangeError,(h_2,l_2,sh_2),False\rangle\langle proof\rangle$
lemma *SFAssInitReds*:
assumes e_2*-steps*: $P \vdash \langle e_2,s_0,b_0\rangle \to* \langle Val\ v,(h_2,l_2,sh_2),False\rangle$
and *cF*: $P \vdash C$ *has* F,*Static*:t *in* D
and *nDone*: $\nexists\, sfs.\ sh_2\ D = Some\ (sfs,\ Done)$
and *INIT-steps*: $P \vdash \langle INIT\ D\ ([D],False) \leftarrow unit,(h_2,l_2,sh_2),False\rangle \to* \langle Val\ v',(h',l',sh'),b'\rangle$
and *sh'D*: $sh'\ D = Some(sfs,i)$
and *sfs'*: $sfs' = sfs(F\mapsto v)$ **and** *sh''*: $sh'' = sh'(D\mapsto(sfs',i))$
shows $P \vdash \langle C\cdot_s F\{D\}:=e_2,s_0,b_0\rangle \to* \langle unit,(h',l',sh''),False\rangle\langle proof\rangle$
lemma *SFAssInitThrowReds*:
assumes e_2*-steps*: $P \vdash \langle e_2,s_0,b_0\rangle \to* \langle Val\ v,(h_2,l_2,sh_2),False\rangle$
and *cF*: $P \vdash C$ *has* F,*Static*:t *in* D
and *nDone*: $\nexists\, sfs.\ sh_2\ D = Some\ (sfs,\ Done)$
and *INIT-steps*: $P \vdash \langle INIT\ D\ ([D],False) \leftarrow unit,(h_2,l_2,sh_2),False\rangle \to* \langle throw\ a,s',b'\rangle$
shows $P \vdash \langle C\cdot_s F\{D\}:=e_2,s_0,b_0\rangle \to* \langle throw\ a,s',b'\rangle\langle proof\rangle$

;;

lemma *SeqReds*:
$P \vdash \langle e,s,b\rangle \to* \langle e',s',b'\rangle \Longrightarrow P \vdash \langle e;;e_2,\ s,b\rangle \to* \langle e';;e_2,\ s',b'\rangle\langle proof\rangle$
lemma *SeqRedsThrow*:
$P \vdash \langle e,s,b\rangle \to* \langle throw\ e',s',b'\rangle \Longrightarrow P \vdash \langle e;;e_2,\ s,b\rangle \to* \langle throw\ e',\ s',b'\rangle\langle proof\rangle$
lemma *SeqReds2*:
assumes e_1*-steps*: $P \vdash \langle e_1,s_0,b_0\rangle \to* \langle Val\ v_1,s_1,b_1\rangle$
and e_2*-steps*: $P \vdash \langle e_2,s_1,b_1\rangle \to* \langle e_2',s_2,b_2\rangle$
shows $P \vdash \langle e_1;;e_2,\ s_0,b_0\rangle \to* \langle e_2',s_2,b_2\rangle\langle proof\rangle$

If

lemma *CondReds*:
$P \vdash \langle e,s,b\rangle \to* \langle e',s',b'\rangle \Longrightarrow P \vdash \langle if\ (e)\ e_1\ else\ e_2,s,b\rangle \to* \langle if\ (e')\ e_1\ else\ e_2,s',b'\rangle\langle proof\rangle$
lemma *CondRedsThrow*:
$P \vdash \langle e,s,b\rangle \to* \langle throw\ a,s',b'\rangle \Longrightarrow P \vdash \langle if\ (e)\ e_1\ else\ e_2,\ s,b\rangle \to* \langle throw\ a,s',b'\rangle\langle proof\rangle$
lemma *CondReds2T*:
assumes *e-steps*: $P \vdash \langle e,s_0,b_0\rangle \to* \langle true,s_1,b_1\rangle$
and e_1*-steps*: $P \vdash \langle e_1,\ s_1,b_1\rangle \to* \langle e',s_2,b_2\rangle$

shows $P \vdash \langle \mathit{if}\ (e)\ e_1\ \mathit{else}\ e_2,\ s_0,b_0\rangle \to* \langle e',s_2,b_2\rangle\langle \mathit{proof}\rangle$
lemma *CondReds2F*:
assumes *e-steps*: $P \vdash \langle e,s_0,b_0\rangle \to* \langle \mathit{false},s_1,b_1\rangle$
and e_2*-steps*: $P \vdash \langle e_2,\ s_1,b_1\rangle \to* \langle e',s_2,b_2\rangle$
shows $P \vdash \langle \mathit{if}\ (e)\ e_1\ \mathit{else}\ e_2,\ s_0,b_0\rangle \to* \langle e',s_2,b_2\rangle\langle \mathit{proof}\rangle$

While

lemma *WhileFReds*:
assumes *b-steps*: $P \vdash \langle b,s,b_0\rangle \to* \langle \mathit{false},s',b'\rangle$
shows $P \vdash \langle \mathit{while}\ (b)\ c,s,b_0\rangle \to* \langle \mathit{unit},s',b'\rangle\langle \mathit{proof}\rangle$
lemma *WhileRedsThrow*:
assumes *b-steps*: $P \vdash \langle b,s,b_0\rangle \to* \langle \mathit{throw}\ e,s',b'\rangle$
shows $P \vdash \langle \mathit{while}\ (b)\ c,s,b_0\rangle \to* \langle \mathit{throw}\ e,s',b'\rangle\langle \mathit{proof}\rangle$
lemma *WhileTReds*:
assumes *b-steps*: $P \vdash \langle b,s_0,b_0\rangle \to* \langle \mathit{true},s_1,b_1\rangle$
and *c-steps*: $P \vdash \langle c,s_1,b_1\rangle \to* \langle \mathit{Val}\ v_1,s_2,b_2\rangle$
and *while-steps*: $P \vdash \langle \mathit{while}\ (b)\ c,s_2,b_2\rangle \to* \langle e,s_3,b_3\rangle$
shows $P \vdash \langle \mathit{while}\ (b)\ c,s_0,b_0\rangle \to* \langle e,s_3,b_3\rangle\langle \mathit{proof}\rangle$
lemma *WhileTRedsThrow*:
assumes *b-steps*: $P \vdash \langle b,s_0,b_0\rangle \to* \langle \mathit{true},s_1,b_1\rangle$
and *c-steps*: $P \vdash \langle c,s_1,b_1\rangle \to* \langle \mathit{throw}\ e,s_2,b_2\rangle$
shows $P \vdash \langle \mathit{while}\ (b)\ c,s_0,b_0\rangle \to* \langle \mathit{throw}\ e,s_2,b_2\rangle\langle \mathit{proof}\rangle$

Throw

lemma *ThrowReds*:
$P \vdash \langle e,s,b\rangle \to* \langle e',s',b'\rangle \Longrightarrow P \vdash \langle \mathit{throw}\ e,s,b\rangle \to* \langle \mathit{throw}\ e',s',b'\rangle\langle \mathit{proof}\rangle$
lemma *ThrowRedsNull*:
$P \vdash \langle e,s,b\rangle \to* \langle \mathit{null},s',b'\rangle \Longrightarrow P \vdash \langle \mathit{throw}\ e,s,b\rangle \to* \langle \mathit{THROW}\ \mathit{NullPointer},s',b'\rangle\langle \mathit{proof}\rangle$
lemma *ThrowRedsThrow*:
$P \vdash \langle e,s,b\rangle \to* \langle \mathit{throw}\ a,s',b'\rangle \Longrightarrow P \vdash \langle \mathit{throw}\ e,s,b\rangle \to* \langle \mathit{throw}\ a,s',b'\rangle\langle \mathit{proof}\rangle$

InitBlock

lemma *InitBlockReds-aux*:
$P \vdash \langle e,s,b\rangle \to* \langle e',s',b'\rangle \Longrightarrow$
$\forall h\ l\ sh\ h'\ l'\ sh'\ v.\ s = (h,l(V\mapsto v),sh) \longrightarrow s' = (h',l',sh') \longrightarrow$
$P \vdash \langle\{V{:}T := \mathit{Val}\ v;\ e\},(h,l,sh),b\rangle \to* \langle\{V{:}T := \mathit{Val}(\mathit{the}(l'\ V));\ e'\},(h',l'(V:=(l\ V)),sh'),b'\rangle\langle \mathit{proof}\rangle$
lemma *InitBlockReds*:
$P \vdash \langle e,\ (h,l(V\mapsto v),sh),b\rangle \to* \langle e',\ (h',l',sh'),b'\rangle \Longrightarrow$
$P \vdash \langle\{V{:}T := \mathit{Val}\ v;\ e\},\ (h,l,sh),b\rangle \to* \langle\{V{:}T := \mathit{Val}(\mathit{the}(l'\ V));\ e'\},\ (h',l'(V:=(l\ V)),sh'),b'\rangle\langle \mathit{proof}\rangle$
lemma *InitBlockRedsFinal*:
$[\![\ P \vdash \langle e,(h,l(V\mapsto v),sh),b\rangle \to* \langle e',(h',l',sh'),b'\rangle;\ \mathit{final}\ e'\ ]\!] \Longrightarrow$
$P \vdash \langle\{V{:}T := \mathit{Val}\ v;\ e\},(h,l,sh),b\rangle \to* \langle e',(h',\ l'(V := l\ V),sh'),b'\rangle\langle \mathit{proof}\rangle$

Block

lemmas *converse-rtranclE3* = *converse-rtranclE* [*of* (*xa*,*xb*,*xc*) (*za*,*zb*,*zc*), *split-rule*]

lemma *BlockRedsFinal*:
assumes *reds*: $P \vdash \langle e_0,s_0,b_0\rangle \to* \langle e_2,(h_2,l_2,sh_2),b_2\rangle$ **and** *fin*: *final* e_2
shows $\bigwedge h_0\ l_0\ sh_0.\ s_0 = (h_0,l_0(V{:=}\mathit{None}),sh_0) \Longrightarrow P \vdash \langle\{V{:}T;\ e_0\},(h_0,l_0,sh_0),b_0\rangle \to* \langle e_2,(h_2,l_2(V:=l_0\ V),sh_2),b_2\rangle\langle \mathit{proof}\rangle$

try-catch

lemma *TryReds*:
$P \vdash \langle e,s,b\rangle \to* \langle e',s',b'\rangle \Longrightarrow P \vdash \langle try\ e\ catch(C\ V)\ e_2,s,b\rangle \to* \langle try\ e'\ catch(C\ V)\ e_2,s',b'\rangle\langle proof\rangle$
lemma *TryRedsVal*:
$P \vdash \langle e,s,b\rangle \to* \langle Val\ v,s',b'\rangle \Longrightarrow P \vdash \langle try\ e\ catch(C\ V)\ e_2,s,b\rangle \to* \langle Val\ v,s',b'\rangle\langle proof\rangle$
lemma *TryCatchRedsFinal*:
assumes e_1-*steps*: $P \vdash \langle e_1,s_0,b_0\rangle \to* \langle Throw\ a,(h_1,l_1,sh_1),b_1\rangle$
and $h_1 a$: $h_1\ a = Some(D,fs)$ **and** *sub*: $P \vdash D \preceq^* C$
and e_2-*steps*: $P \vdash \langle e_2,\ (h_1,\ l_1(V \mapsto Addr\ a),sh_1),b_1\rangle \to* \langle e_2',\ (h_2,l_2,sh_2),\ b_2\rangle$
and *final*: *final* e_2'
shows $P \vdash \langle try\ e_1\ catch(C\ V)\ e_2,\ s_0,\ b_0\rangle \to* \langle e_2',\ (h_2,\ (l_2{::}locals)(V := l_1\ V),sh_2),b_2\rangle\langle proof\rangle$
lemma *TryRedsFail*:
$[\![\ P \vdash \langle e_1,s,b\rangle \to* \langle Throw\ a,(h,l,sh),b'\rangle;\ h\ a = Some(D,fs);\ \neg\ P \vdash D \preceq^* C\ ]\!]$
$\Longrightarrow P \vdash \langle try\ e_1\ catch(C\ V)\ e_2,s,b\rangle \to* \langle Throw\ a,(h,l,sh),b'\rangle\langle proof\rangle$

List

lemma *ListReds1*:
$P \vdash \langle e,s,b\rangle \to* \langle e',s',b'\rangle \Longrightarrow P \vdash \langle e\#es,s,b\rangle\ [\to]*\ \langle e'\ \#\ es,s',b'\rangle\langle proof\rangle$
lemma *ListReds2*:
$P \vdash \langle es,s,b\rangle\ [\to]*\ \langle es',s',b'\rangle \Longrightarrow P \vdash \langle Val\ v\ \#\ es,s,b\rangle\ [\to]*\ \langle Val\ v\ \#\ es',s',b'\rangle\langle proof\rangle$
lemma *ListRedsVal*:
$[\![\ P \vdash \langle e,s_0,b_0\rangle \to* \langle Val\ v,s_1,b_1\rangle;\ P \vdash \langle es,s_1,b_1\rangle\ [\to]*\ \langle es',s_2,b_2\rangle\ ]\!]$
$\Longrightarrow P \vdash \langle e\#es,s_0,b_0\rangle\ [\to]*\ \langle Val\ v\ \#\ es',s_2,b_2\rangle\langle proof\rangle$

Call

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

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

lemma *Red-dom-lcl*:
$P \vdash \langle e,(h,l,sh),b\rangle \to \langle e',(h',l',sh'),b'\rangle \Longrightarrow dom\ l' \subseteq dom\ l \cup fv\ e$ **and**
$P \vdash \langle es,(h,l,sh),b\rangle\ [\to]\ \langle es',(h',l',sh'),b'\rangle \Longrightarrow dom\ l' \subseteq dom\ l \cup fvs\ es\langle proof\rangle$
lemma *Reds-dom-lcl*:
assumes *wf*: *wwf-J-prog P*
shows $P \vdash \langle e,(h,l,sh),b\rangle \to* \langle e',(h',l',sh'),b'\rangle \Longrightarrow dom\ l' \subseteq dom\ l \cup fv\ e\langle proof\rangle$

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

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

lemma *blocksReds*:
$\bigwedge l.\ [\![\ length\ Vs = length\ Ts;\ length\ vs = length\ Ts;\ distinct\ Vs;$
$P \vdash \langle e,\ (h,l(Vs\ [\mapsto]\ vs),sh),b\rangle \to* \langle e',\ (h',l',sh'),b'\rangle\ ]\!]$
$\Longrightarrow P \vdash \langle blocks(Vs,Ts,vs,e),\ (h,l,sh),b\rangle \to* \langle blocks(Vs,Ts,map\ (the \circ l')\ Vs,e'),\ (h',override\text{-}on\ l'\ l\ (set\ Vs),sh'),b'\rangle\langle proof\rangle$

lemma *blocksFinal*:
$\bigwedge l.\ [\![\ length\ Vs = length\ Ts;\ length\ vs = length\ Ts;\ final\ e\ ]\!] \Longrightarrow$
$P \vdash \langle blocks(Vs,Ts,vs,e),\ (h,l,sh),b\rangle \to* \langle e,\ (h,l,sh),b\rangle\langle proof\rangle$

lemma *blocksRedsFinal*:
assumes *wf*: *length Vs = length Ts length vs = length Ts distinct Vs*
and *reds*: $P \vdash \langle e, (h,l(Vs\ [\mapsto]\ vs),sh),b\rangle \to* \langle e', (h',l',sh'),b'\rangle$
and *fin*: *final e'* **and** *l''*: *l'' = override-on l' l (set Vs)*
shows $P \vdash \langle blocks(Vs,Ts,vs,e), (h,l,sh),b\rangle \to* \langle e', (h',l'',sh'),b'\rangle\langle proof\rangle$

An now the actual method call reduction lemmas.

lemma *CallRedsObj*:
$P \vdash \langle e,s,b\rangle \to* \langle e',s',b'\rangle \Longrightarrow P \vdash \langle e{\cdot}M(es),s,b\rangle \to* \langle e'{\cdot}M(es),s',b'\rangle\langle proof\rangle$

lemma *CallRedsParams*:
$P \vdash \langle es,s,b\rangle\ [\to]*\ \langle es',s',b'\rangle \Longrightarrow P \vdash \langle (Val\ v){\cdot}M(es),s,b\rangle \to* \langle (Val\ v){\cdot}M(es'),s',b'\rangle\langle proof\rangle$

lemma *CallRedsFinal*:
assumes *wwf*: *wwf-J-prog P*
and $P \vdash \langle e,s_0,b_0\rangle \to* \langle addr\ a,s_1,b_1\rangle$
$P \vdash \langle es,s_1,b_1\rangle\ [\to]*\ \langle map\ Val\ vs,(h_2,l_2,sh_2),b_2\rangle$
$h_2\ a = Some(C,fs)\ P \vdash C\ sees\ M,NonStatic{:}Ts\to T = (pns,body)\ in\ D$
size vs = size pns
and l_2': $l_2' = [this \mapsto Addr\ a,\ pns[\mapsto]vs]$
and *body*: $P \vdash \langle body,(h_2,l_2',sh_2),b_2\rangle \to* \langle ef,(h_3,l_3,sh_3),b_3\rangle$
and *final ef*
shows $P \vdash \langle e{\cdot}M(es),\ s_0,b_0\rangle \to* \langle ef,(h_3,l_2,sh_3),b_3\rangle\langle proof\rangle$

lemma *CallRedsThrowParams*:
assumes *e-steps*: $P \vdash \langle e,s_0,b_0\rangle \to* \langle Val\ v,s_1,b_1\rangle$
and *es-steps*: $P \vdash \langle es,s_1,b_1\rangle\ [\to]*\ \langle map\ Val\ vs_1\ @\ throw\ a\ \#\ es_2,s_2,b_2\rangle$
shows $P \vdash \langle e{\cdot}M(es),s_0,b_0\rangle \to* \langle throw\ a,s_2,b_2\rangle\langle proof\rangle$

lemma *CallRedsThrowObj*:
$P \vdash \langle e,s_0,b_0\rangle \to* \langle throw\ a,s_1,b_1\rangle \Longrightarrow P \vdash \langle e{\cdot}M(es),s_0,b_0\rangle \to* \langle throw\ a,s_1,b_1\rangle\langle proof\rangle$

lemma *CallRedsNull*:
assumes *e-steps*: $P \vdash \langle e,s_0,b_0\rangle \to* \langle null,s_1,b_1\rangle$
and *es-steps*: $P \vdash \langle es,s_1,b_1\rangle\ [\to]*\ \langle map\ Val\ vs,s_2,b_2\rangle$
shows $P \vdash \langle e{\cdot}M(es),s_0,b_0\rangle \to* \langle THROW\ NullPointer,s_2,b_2\rangle\langle proof\rangle$
lemma *CallRedsNone*:
assumes *e-steps*: $P \vdash \langle e,s,b\rangle \to* \langle addr\ a,s_1,b_1\rangle$
and *es-steps*: $P \vdash \langle es,s_1,b_1\rangle\ [\to]*\ \langle map\ Val\ vs,s_2,b_2\rangle$
and *hp$_2$a*: *hp s$_2$ a = Some(C,fs)*
and *ncM*: $\neg(\exists\ b\ Ts\ T\ m\ D.\ P \vdash C\ sees\ M,b{:}Ts\to T = m\ in\ D)$
shows $P \vdash \langle e{\cdot}M(es),s,b\rangle \to* \langle THROW\ NoSuchMethodError,s_2,b_2\rangle\langle proof\rangle$
lemma *CallRedsStatic*:
assumes *e-steps*: $P \vdash \langle e,s,b\rangle \to* \langle addr\ a,s_1,b_1\rangle$
and *es-steps*: $P \vdash \langle es,s_1,b_1\rangle\ [\to]*\ \langle map\ Val\ vs,s_2,b_2\rangle$
and *hp$_2$a*: *hp s$_2$ a = Some(C,fs)*
and *cM-Static*: $P \vdash C\ sees\ M,Static{:}Ts\to T = m\ in\ D$
shows $P \vdash \langle e{\cdot}M(es),s,b\rangle \to* \langle THROW\ IncompatibleClassChangeError,s_2,b_2\rangle\langle proof\rangle$

1.23.2 SCall

lemma *SCallRedsParams*:
$P \vdash \langle es,s,b\rangle\ [\to]*\ \langle es',s',b'\rangle \Longrightarrow P \vdash \langle C{\cdot}_sM(es),s,b\rangle \to* \langle C{\cdot}_sM(es'),s',b'\rangle\langle proof\rangle$
lemma *SCallRedsFinal*:

assumes *wwf*: *wwf-J-prog P*
and $P \vdash \langle es,s_0,b_0\rangle\ [\rightarrow]*\ \langle map\ Val\ vs,(h_2,l_2,sh_2),b_2\rangle$
$P \vdash C\ sees\ M,Static:Ts\rightarrow T = (pns,body)\ in\ D$
$sh_2\ D = Some(sfs,Done) \vee (M = clinit \wedge sh_2\ D = \lfloor(sfs,\ Processing)\rfloor)$
$size\ vs = size\ pns$
and l_2': $l_2' = [pns[\mapsto]vs]$
and *body*: $P \vdash \langle body,(h_2,l_2',sh_2),False\rangle \rightarrow* \langle ef,(h_3,l_3,sh_3),b_3\rangle$
and *final ef*
shows $P \vdash \langle C\bullet_s M(es),\ s_0,b_0\rangle \rightarrow* \langle ef,(h_3,l_2,sh_3),b_3\rangle\langle proof\rangle$
lemma *SCallRedsThrowParams*:
$[\![\ P \vdash \langle es,s_0,b_0\rangle\ [\rightarrow]*\ \langle map\ Val\ vs_1\ @\ throw\ a\ \#\ es_2,s_2,b_2\rangle\ ]\!]$
$\Longrightarrow P \vdash \langle C\bullet_s M(es),s_0,b_0\rangle \rightarrow* \langle throw\ a,s_2,b_2\rangle\langle proof\rangle$
lemma *SCallRedsNone*:
$[\![\ P \vdash \langle es,s,b\rangle\ [\rightarrow]*\ \langle map\ Val\ vs,s_2,False\rangle;$
$\neg(\exists\ b\ Ts\ T\ m\ D.\ P \vdash C\ sees\ M,b:Ts\rightarrow T = m\ in\ D)\ ]\!]$
$\Longrightarrow P \vdash \langle C\bullet_s M(es),s,b\rangle \rightarrow* \langle THROW\ NoSuchMethodError,s_2,False\rangle\langle proof\rangle$
lemma *SCallRedsNonStatic*:
$[\![\ P \vdash \langle es,s,b\rangle\ [\rightarrow]*\ \langle map\ Val\ vs,s_2,False\rangle;$
$P \vdash C\ sees\ M,NonStatic:Ts\rightarrow T = m\ in\ D\ ]\!]$
$\Longrightarrow P \vdash \langle C\bullet_s M(es),s,b\rangle \rightarrow* \langle THROW\ IncompatibleClassChangeError,s_2,False\rangle\langle proof\rangle$
lemma *SCallInitThrowReds*:
assumes *wwf*: *wwf-J-prog P*
and $P \vdash \langle es,s_0,b_0\rangle\ [\rightarrow]*\ \langle map\ Val\ vs,(h_1,l_1,sh_1),False\rangle$
$P \vdash C\ sees\ M,Static:Ts\rightarrow T = (pns,body)\ in\ D$
$\nexists sfs.\ sh_1\ D = Some(sfs,Done)$
$M \neq clinit$
and $P \vdash \langle INIT\ D\ ([D],False) \leftarrow unit,(h_1,l_1,sh_1),False\rangle \rightarrow* \langle throw\ a,(h_2,l_2,sh_2),b_2\rangle$
shows $P \vdash \langle C\bullet_s M(es),\ s_0,b_0\rangle \rightarrow* \langle throw\ a,(h_2,l_2,sh_2),b_2\rangle\langle proof\rangle$
lemma *SCallInitReds*:
assumes *wwf*: *wwf-J-prog P*
and $P \vdash \langle es,s_0,b_0\rangle\ [\rightarrow]*\ \langle map\ Val\ vs,(h_1,l_1,sh_1),False\rangle$
$P \vdash C\ sees\ M,Static:Ts\rightarrow T = (pns,body)\ in\ D$
$\nexists sfs.\ sh_1\ D = Some(sfs,Done)$
$M \neq clinit$
and $P \vdash \langle INIT\ D\ ([D],False) \leftarrow unit,(h_1,l_1,sh_1),False\rangle \rightarrow* \langle Val\ v',(h_2,l_2,sh_2),b_2\rangle$
and $size\ vs = size\ pns$
and l_2': $l_2' = [pns[\mapsto]vs]$
and *body*: $P \vdash \langle body,(h_2,l_2',sh_2),False\rangle \rightarrow* \langle ef,(h_3,l_3,sh_3),b_3\rangle$
and *final ef*
shows $P \vdash \langle C\bullet_s M(es),s_0,b_0\rangle \rightarrow* \langle ef,(h_3,l_2,sh_3),b_3\rangle\langle proof\rangle$
lemma *SCallInitProcessingReds*:
assumes *wwf*: *wwf-J-prog P*
and $P \vdash \langle es,s_0,b_0\rangle\ [\rightarrow]*\ \langle map\ Val\ vs,(h_2,l_2,sh_2),b_2\rangle$
$P \vdash C\ sees\ M,Static:Ts\rightarrow T = (pns,body)\ in\ D$
$sh_2\ D = Some(sfs,Processing)$
and $size\ vs = size\ pns$
and l_2': $l_2' = [pns[\mapsto]vs]$
and *body*: $P \vdash \langle body,(h_2,l_2',sh_2),False\rangle \rightarrow* \langle ef,(h_3,l_3,sh_3),b_3\rangle$
and *final ef*
shows $P \vdash \langle C\bullet_s M(es),s_0,b_0\rangle \rightarrow* \langle ef,(h_3,l_2,sh_3),b_3\rangle\langle proof\rangle$

The main Theorem

lemma assumes *wwf*: *wwf-J-prog P*

shows *big-by-small*: $P \vdash \langle e,s\rangle \Rightarrow \langle e',s'\rangle$
$\Longrightarrow (\bigwedge b.\ iconf\ (shp\ s)\ e \Longrightarrow P,shp\ s \vdash_b (e,b)\ \surd \Longrightarrow P \vdash \langle e,s,b\rangle \rightarrow* \langle e',s',False\rangle)$
and *bigs-by-smalls*: $P \vdash \langle es,s\rangle\ [\Rightarrow]\ \langle es',s'\rangle$
$\Longrightarrow (\bigwedge b.\ iconfs\ (shp\ s)\ es \Longrightarrow P,shp\ s \vdash_b (es,b)\ \surd \Longrightarrow P \vdash \langle es,s,b\rangle\ [\rightarrow]*\ \langle es',s',False\rangle)\langle proof\rangle$

1.23.3 Big steps simulates small step

This direction was carried out by Norbert Schirmer and Daniel Wasserrab (and modified to include statics and DCI by Susannah Mansky).

The big step equivalent of *RedWhile*:

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

lemma *blocksEval*:
$\bigwedge Ts\ vs\ l\ l'.\ [\![size\ ps = size\ Ts;\ size\ ps = size\ vs;\ P \vdash \langle blocks(ps,Ts,vs,e),(h,l,sh)\rangle \Rightarrow \langle e',(h',l',sh')\rangle\ ]\!]$
$\Longrightarrow \exists\ l''.\ P \vdash \langle e,(h,l(ps[\mapsto]vs),sh)\rangle \Rightarrow \langle e',(h',l'',sh')\rangle\langle proof\rangle$
lemma
assumes *wf*: *wwf-J-prog P*
shows *eval-restrict-lcl*:
$P \vdash \langle e,(h,l,sh)\rangle \Rightarrow \langle e',(h',l',sh')\rangle \Longrightarrow (\bigwedge W.\ fv\ e \subseteq W \Longrightarrow P \vdash \langle e,(h,l|`W,sh)\rangle \Rightarrow \langle e',(h',l'|`W,sh')\rangle)$
and $P \vdash \langle es,(h,l,sh)\rangle\ [\Rightarrow]\ \langle es',(h',l',sh')\rangle \Longrightarrow (\bigwedge W.\ fvs\ es \subseteq W \Longrightarrow P \vdash \langle es,(h,l|`W,sh)\rangle\ [\Rightarrow]$
$\langle es',(h',l'|`W,sh')\rangle)\langle proof\rangle$

lemma *eval-notfree-unchanged*:
$P \vdash \langle e,(h,l,sh)\rangle \Rightarrow \langle e',(h',l',sh')\rangle \Longrightarrow (\bigwedge V.\ V \notin fv\ e \Longrightarrow l'\ V = l\ V)$
and $P \vdash \langle es,(h,l,sh)\rangle\ [\Rightarrow]\ \langle es',(h',l',sh')\rangle \Longrightarrow (\bigwedge V.\ V \notin fvs\ es \Longrightarrow l'\ V = l\ V)\langle proof\rangle$

lemma *eval-closed-lcl-unchanged*:
$[\![\ P \vdash \langle e,(h,l,sh)\rangle \Rightarrow \langle e',(h',l',sh')\rangle;\ fv\ e = \{\}\ ]\!] \Longrightarrow l' = l\langle proof\rangle$

lemma *list-eval-Throw*:
assumes *eval-e*: $P \vdash \langle throw\ x,s\rangle \Rightarrow \langle e',s'\rangle$
shows $P \vdash \langle map\ Val\ vs\ @\ throw\ x\ \#\ es',s\rangle\ [\Rightarrow]\ \langle map\ Val\ vs\ @\ e'\ \#\ es',s'\rangle\langle proof\rangle$
lemma *seq-ext*:
assumes *IH*: $\bigwedge e'\ s'.\ P \vdash \langle e'',s''\rangle \Rightarrow \langle e',s'\rangle \Longrightarrow P \vdash \langle e,s\rangle \Rightarrow \langle e',s'\rangle$
and *seq*: $P \vdash \langle e''\ ;;\ e_0,s''\rangle \Rightarrow \langle e',s'\rangle$
shows $P \vdash \langle e\ ;;\ e_0,s\rangle \Rightarrow \langle e',s'\rangle$
$\langle proof\rangle$
lemma *rinit-Val-ext*:
assumes *ri*: $P \vdash \langle RI\ (C,e'')\ ;\ Cs \leftarrow e_0,s''\rangle \Rightarrow \langle Val\ v',s_1\rangle$
and *IH*: $\bigwedge e'\ s'.\ P \vdash \langle e'',s''\rangle \Rightarrow \langle e',s'\rangle \Longrightarrow P \vdash \langle e,s\rangle \Rightarrow \langle e',s'\rangle$
shows $P \vdash \langle RI\ (C,e)\ ;\ Cs \leftarrow e_0,s\rangle \Rightarrow \langle Val\ v',s_1\rangle$
$\langle proof\rangle$
lemma *rinit-throw-ext*:
assumes *ri*: $P \vdash \langle RI\ (C,e'')\ ;\ Cs \leftarrow e_0,s''\rangle \Rightarrow \langle throw\ e_t,s'\rangle$
and *IH*: $\bigwedge e'\ s'.\ P \vdash \langle e'',s''\rangle \Rightarrow \langle e',s'\rangle \Longrightarrow P \vdash \langle e,s\rangle \Rightarrow \langle e',s'\rangle$
shows $P \vdash \langle RI\ (C,e)\ ;\ Cs \leftarrow e_0,s\rangle \Rightarrow \langle throw\ e_t,s'\rangle$
$\langle proof\rangle$
lemma *rinit-ext*:
assumes *IH*: $\bigwedge e'\ s'.\ P \vdash \langle e'',s''\rangle \Rightarrow \langle e',s'\rangle \Longrightarrow P \vdash \langle e,s\rangle \Rightarrow \langle e',s'\rangle$
shows $\bigwedge e'\ s'.\ P \vdash \langle RI\ (C,e'')\ ;\ Cs \leftarrow e_0,s''\rangle \Rightarrow \langle e',s'\rangle$
$\Longrightarrow P \vdash \langle RI\ (C,e)\ ;\ Cs \leftarrow e_0,s\rangle \Rightarrow \langle e',s'\rangle$
$\langle proof\rangle$

lemma
shows *eval-init-return*: $P \vdash \langle e,s\rangle \Rightarrow \langle e',s'\rangle$
$\Longrightarrow$ *iconf* (*shp s*) *e*
$\Longrightarrow$ ($\exists$ *Cs b*. $e = INIT\ C'\ (Cs,b) \leftarrow unit$) $\vee$ ($\exists$ *C* e_0 *Cs* e_i. $e = RI(C,e_0);Cs@[C'] \leftarrow unit$)
$\vee$ ($\exists$ e_0. $e = RI(C',e_0);Nil \leftarrow unit$)
$\Longrightarrow$ (*val-of* $e' = Some\ v \longrightarrow$ ($\exists$ *sfs i*. *shp* $s'\ C' = \lfloor(sfs,i)\rfloor \wedge (i = Done \vee i = Processing)$))
$\wedge$ (*throw-of* $e' = Some\ a \longrightarrow$ ($\exists$ *sfs i*. *shp* $s'\ C' = \lfloor(sfs,Error)\rfloor$))
and $P \vdash \langle es,s\rangle\ [\Rightarrow]\ \langle es',s'\rangle \Longrightarrow True$
⟨*proof*⟩

lemma *init-Val-PD*: $P \vdash \langle INIT\ C'\ (Cs,b) \leftarrow unit,s\rangle \Rightarrow \langle Val\ v,s'\rangle$
$\Longrightarrow$ *iconf* (*shp s*) ($INIT\ C'\ (Cs,b) \leftarrow unit$)
$\Longrightarrow$ $\exists$ *sfs i*. *shp* $s'\ C' = \lfloor(sfs,i)\rfloor \wedge (i = Done \vee i = Processing)$
⟨*proof*⟩

lemma *init-throw-PD*: $P \vdash \langle INIT\ C'\ (Cs,b) \leftarrow unit,s\rangle \Rightarrow \langle throw\ a,s'\rangle$
$\Longrightarrow$ *iconf* (*shp s*) ($INIT\ C'\ (Cs,b) \leftarrow unit$)
$\Longrightarrow$ $\exists$ *sfs i*. *shp* $s'\ C' = \lfloor(sfs,Error)\rfloor$
⟨*proof*⟩

lemma *rinit-Val-PD*: $P \vdash \langle RI(C,e_0);Cs \leftarrow unit,s\rangle \Rightarrow \langle Val\ v,s'\rangle$
$\Longrightarrow$ *iconf* (*shp s*) ($RI(C,e_0);Cs \leftarrow unit$) $\Longrightarrow$ $last(C\#Cs) = C'$
$\Longrightarrow$ $\exists$ *sfs i*. *shp* $s'\ C' = \lfloor(sfs,i)\rfloor \wedge (i = Done \vee i = Processing)$
⟨*proof*⟩

lemma *rinit-throw-PD*: $P \vdash \langle RI(C,e_0);Cs \leftarrow unit,s\rangle \Rightarrow \langle throw\ a,s'\rangle$
$\Longrightarrow$ *iconf* (*shp s*) ($RI(C,e_0);Cs \leftarrow unit$) $\Longrightarrow$ $last(C\#Cs) = C'$
$\Longrightarrow$ $\exists$ *sfs i*. *shp* $s'\ C' = \lfloor(sfs,Error)\rfloor$
⟨*proof*⟩
lemma *eval-init-seq'*: $P \vdash \langle e,s\rangle \Rightarrow \langle e',s'\rangle$
$\Longrightarrow$ ($\exists$ *C Cs b* e_i. $e = INIT\ C\ (Cs,b) \leftarrow e_i$) $\vee$ ($\exists$ *C* e_0 *Cs* e_i. $e = RI(C,e_0);Cs \leftarrow e_i$)
$\Longrightarrow$ ($\exists$ *C Cs b* e_i. $e = INIT\ C\ (Cs,b) \leftarrow e_i \wedge P \vdash \langle(INIT\ C\ (Cs,b) \leftarrow unit);;\ e_i,s\rangle \Rightarrow \langle e',s'\rangle$)
$\vee$ ($\exists$ *C* e_0 *Cs* e_i. $e = RI(C,e_0);Cs \leftarrow e_i \wedge P \vdash \langle(RI(C,e_0);Cs \leftarrow unit);;\ e_i,s\rangle \Rightarrow \langle e',s'\rangle$)
and $P \vdash \langle es,s\rangle\ [\Rightarrow]\ \langle es',s'\rangle \Longrightarrow True$
⟨*proof*⟩

lemma *eval-init-seq*: $P \vdash \langle INIT\ C\ (Cs,b) \leftarrow e,(h,\ l,\ sh)\rangle \Rightarrow \langle e',s'\rangle$
$\Longrightarrow P \vdash \langle(INIT\ C\ (Cs,b) \leftarrow unit);;\ e,(h,\ l,\ sh)\rangle \Rightarrow \langle e',s'\rangle$
⟨*proof*⟩

The key lemma:

lemma
assumes *wf*: *wwf-J-prog P*
shows *extend-1-eval*: $P \vdash \langle e,s,b\rangle \rightarrow \langle e'',s'',b''\rangle \Longrightarrow P,shp\ s \vdash_b (e,b)\ \surd$
$\Longrightarrow$ ($\bigwedge s'\ e'.\ P \vdash \langle e'',s''\rangle \Rightarrow \langle e',s'\rangle \Longrightarrow P \vdash \langle e,s\rangle \Rightarrow \langle e',s'\rangle$)
and *extend-1-evals*: $P \vdash \langle es,s,b\rangle\ [\rightarrow]\ \langle es'',s'',b''\rangle \Longrightarrow P,shp\ s \vdash_b (es,b)\ \surd$
$\Longrightarrow$ ($\bigwedge s'\ es'.\ P \vdash \langle es'',s''\rangle\ [\Rightarrow]\ \langle es',s'\rangle \Longrightarrow P \vdash \langle es,s\rangle\ [\Rightarrow]\ \langle es',s'\rangle$)
⟨*proof*⟩

Its extension to $\rightarrow*$:

lemma *extend-eval*:
assumes *wf*: *wwf-J-prog P*
shows ⟦ $P \vdash \langle e,s,b\rangle \rightarrow* \langle e'',s'',b''\rangle$; $P \vdash \langle e'',s''\rangle \Rightarrow \langle e',s'\rangle$;
iconf (*shp s*) *e*; $P,shp\ s \vdash_b (e::expr,b)\ \surd$ ⟧

$\Longrightarrow P \vdash \langle e,s\rangle \Rightarrow \langle e',s'\rangle\langle proof\rangle$

lemma *extend-evals*:
assumes *wf*: *wwf-J-prog P*
shows $[\![P \vdash \langle es,s,b\rangle\ [\rightarrow]*\ \langle es'',s'',b''\rangle;\ P \vdash \langle es'',s''\rangle\ [\Rightarrow]\ \langle es',s'\rangle;$
iconfs (*shp s*) *es*; *P*,*shp s* $\vdash_b$ (*es*::*expr list*,*b*) $\surd$ $]\!]$
$\Longrightarrow P \vdash \langle es,s\rangle\ [\Rightarrow]\ \langle es',s'\rangle\langle proof\rangle$

Finally, small step semantics can be simulated by big step semantics:

theorem
assumes *wf*: *wwf-J-prog P*
shows *small-by-big*:
$[\![P \vdash \langle e,s,b\rangle \rightarrow* \langle e',s',b'\rangle;$ *iconf* (*shp s*) *e*; *P*,*shp s* $\vdash_b$ (*e*,*b*) $\surd$; *final e'*$]\!]$
$\Longrightarrow P \vdash \langle e,s\rangle \Rightarrow \langle e',s'\rangle$
and $[\![P \vdash \langle es,s,b\rangle\ [\rightarrow]*\ \langle es',s',b'\rangle;$ *iconfs* (*shp s*) *es*; *P*,*shp s* $\vdash_b$ (*es*,*b*) $\surd$; *finals es'*$]\!]$
$\Longrightarrow P \vdash \langle es,s\rangle\ [\Rightarrow]\ \langle es',s'\rangle\langle proof\rangle$

1.23.4 Equivalence

And now, the crowning achievement:

corollary *big-iff-small*:
$[\![$ *wwf-J-prog P*; *iconf* (*shp s*) *e*; *P*,*shp s* $\vdash_b$ (*e*::*expr*,*b*) $\surd$ $]\!]$
$\Longrightarrow P \vdash \langle e,s\rangle \Rightarrow \langle e',s'\rangle\ =\ (P \vdash \langle e,s,b\rangle \rightarrow* \langle e',s',False\rangle \wedge$ *final e'*$)\langle proof\rangle$
corollary *big-iff-small-WT*:
wwf-J-prog P $\Longrightarrow$ *P*,*E* $\vdash$ *e*::*T* $\Longrightarrow$ *P*,*shp s* $\vdash_b$ (*e*,*b*) $\surd$ $\Longrightarrow$
$P \vdash \langle e,s\rangle \Rightarrow \langle e',s'\rangle\ =\ (P \vdash \langle e,s,b\rangle \rightarrow* \langle e',s',False\rangle \wedge$ *final e'*$)\langle proof\rangle$

1.23.5 Lifting type safety to ⇒

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

lemma *eval-preserves-sconf*:
fixes *s*::*state* **and** *s'*::*state*
assumes *wf-J-prog P* **and** $P \vdash \langle e,s\rangle \Rightarrow \langle e',s'\rangle$ **and** *iconf* (*shp s*) *e*
and *P*,*E* $\vdash$ *e*::*T* **and** *P*,*E* $\vdash$ *s*$\surd$
shows *P*,*E* $\vdash$ *s'*$\surd\langle proof\rangle$

lemma *eval-preserves-type*:
fixes *s*::*state*
assumes *wf*: *wf-J-prog P*
and $P \vdash \langle e,s\rangle \Rightarrow \langle e',s'\rangle$ **and** *P*,*E* $\vdash$ *s*$\surd$ **and** *iconf* (*shp s*) *e* **and** *P*,*E* $\vdash$ *e*::*T*
shows $\exists\, T'.\ P \vdash T' \leq T \wedge$ *P*,*E*,*hp s'*,*shp s'* $\vdash$ *e'*:*T'*$\langle proof\rangle$

end

1.24 Program annotation

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

inductive
Anno :: [*J-prog*,*env*, *expr* , *expr*] ⇒ *bool*
(‹-,- ⊢ - ⇝ -› [*51*,*0*,*0*,*51*]*50*)
and *Annos* :: [*J-prog*,*env*, *expr list*, *expr list*] ⇒ *bool*
(‹-,- ⊢ - [⇝] -› [*51*,*0*,*0*,*51*]*50*)

for P :: *J-prog*
where

AnnoNew: $P,E \vdash new\ C \rightsquigarrow new\ C$
| *AnnoCast*: $P,E \vdash e \rightsquigarrow e' \Longrightarrow P,E \vdash Cast\ C\ e \rightsquigarrow Cast\ C\ e'$
| *AnnoVal*: $P,E \vdash Val\ v \rightsquigarrow Val\ v$
| *AnnoVarVar*: $E\ V = \lfloor T \rfloor \Longrightarrow P,E \vdash Var\ V \rightsquigarrow Var\ V$
| *AnnoVarField*: $[\![\ E\ V = None;\ E\ this = \lfloor Class\ C \rfloor;\ P \vdash C\ sees\ V,NonStatic{:}T\ in\ D\ ]\!]$
$\Longrightarrow P,E \vdash Var\ V \rightsquigarrow Var\ this{\cdot}V\{D\}$
| *AnnoBinOp*:
$[\![\ P,E \vdash e1 \rightsquigarrow e1';\ P,E \vdash e2 \rightsquigarrow e2'\ ]\!]$
$\Longrightarrow P,E \vdash e1\ «bop»\ e2 \rightsquigarrow e1'\ «bop»\ e2'$
| *AnnoLAssVar*:
$[\![\ E\ V = \lfloor T \rfloor;\ P,E \vdash e \rightsquigarrow e'\ ]\!] \Longrightarrow P,E \vdash V{:=}e \rightsquigarrow V{:=}e'$
| *AnnoLAssField*:
$[\![\ E\ V = None;\ E\ this = \lfloor Class\ C \rfloor;\ P \vdash C\ sees\ V,NonStatic{:}T\ in\ D;\ P,E \vdash e \rightsquigarrow e'\ ]\!]$
$\Longrightarrow P,E \vdash V{:=}e \rightsquigarrow Var\ this{\cdot}V\{D\} := e'$
| *AnnoFAcc*:
$[\![\ P,E \vdash e \rightsquigarrow e';\ P,E \vdash e' :: Class\ C;\ P \vdash C\ sees\ F,NonStatic{:}T\ in\ D\ ]\!]$
$\Longrightarrow P,E \vdash e{\cdot}F\{[]\} \rightsquigarrow e'{\cdot}F\{D\}$
| *AnnoSFAcc*:
$[\![\ P \vdash C\ sees\ F,Static{:}T\ in\ D\ ]\!]$
$\Longrightarrow P,E \vdash C{\cdot}_sF\{[]\} \rightsquigarrow C{\cdot}_sF\{D\}$
| *AnnoFAss*: $[\![\ P,E \vdash e1 \rightsquigarrow e1';\ P,E \vdash e2 \rightsquigarrow e2';$
$P,E \vdash e1' :: Class\ C;\ P \vdash C\ sees\ F,NonStatic{:}T\ in\ D\ ]\!]$
$\Longrightarrow P,E \vdash e1{\cdot}F\{[]\} := e2 \rightsquigarrow e1'{\cdot}F\{D\} := e2'$
| *AnnoSFAss*: $[\![\ P,E \vdash e2 \rightsquigarrow e2';\ P \vdash C\ sees\ F,Static{:}T\ in\ D\ ]\!]$
$\Longrightarrow P,E \vdash C{\cdot}_sF\{[]\} := e2 \rightsquigarrow C{\cdot}_sF\{D\} := e2'$
| *AnnoCall*:
$[\![\ P,E \vdash e \rightsquigarrow e';\ P,E \vdash es\ [\rightsquigarrow]\ es'\ ]\!]$
$\Longrightarrow P,E \vdash Call\ e\ M\ es \rightsquigarrow Call\ e'\ M\ es'$
| *AnnoSCall*:
$[\![\ P,E \vdash es\ [\rightsquigarrow]\ es'\ ]\!]$
$\Longrightarrow P,E \vdash SCall\ C\ M\ es \rightsquigarrow SCall\ C\ M\ es'$
| *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*: $P,E \vdash e \rightsquigarrow e' \Longrightarrow P,E \vdash throw\ e \rightsquigarrow throw\ e'$
| *AnnoTry*: $[\![\ P,E \vdash e1 \rightsquigarrow e1';\ P,E(V \mapsto Class\ C) \vdash e2 \rightsquigarrow e2'\ ]\!]$
$\Longrightarrow P,E \vdash try\ e1\ catch(C\ V)\ e2 \rightsquigarrow try\ e1'\ catch(C\ V)\ e2'$

| *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

Chapter 2

Jinja Virtual Machine

2.1 State of the JVM

theory *JVMState* **imports** *../Common/Objects* **begin**

type-synonym
pc = *nat*

abbreviation *start-sheap* :: *sheap*
where *start-sheap* ≡ (λ*x*. *None*)(*Start* ↦ (*Map.empty*,*Done*))

definition *start-sheap-preloaded* :: *'m prog* ⇒ *sheap*
where
start-sheap-preloaded P ≡ *fold* (λ(*C*,*cl*) *f*. *f*(*C* := *Some* (*sblank P C*, *Prepared*))) *P* (λ*x*. *None*)

2.1.1 Frame Stack

datatype *init-call-status* = *No-ics* | *Calling cname cname list*
| *Called cname list* | *Throwing cname list addr*
— *No-ics* = not currently calling or waiting for the result of an initialization procedure call
— *Calling C Cs* = current instruction is calling for initialization of classes *C*#*Cs* (last class is the original) – still collecting classes to be initialized, *C* most recently collected
— *Called Cs* = current instruction called initialization and is waiting for the result – now initializing classes in the list
— *Throwing Cs a* = frame threw or was thrown an error causing erroneous end of initialization procedure for classes *Cs*

type-synonym
frame = *val list* × *val list* × *cname* × *mname* × *pc* × *init-call-status*
— operand stack
— registers (including this pointer, method parameters, and local variables)
— name of class where current method is defined
— current method
— program counter within frame
— indicates frame's initialization call status

translations
(*type*) *frame* <= (*type*) *val list* × *val list* × *char list* × *char list* × *nat* × *init-call-status*

fun *curr-stk* :: *frame* ⇒ *val list* **where**
curr-stk (*stk*, *loc*, *C*, *M*, *pc*, *ics*) = *stk*

fun *curr-class* :: *frame* ⇒ *cname* **where**
curr-class (*stk*, *loc*, *C*, *M*, *pc*, *ics*) = *C*

fun *curr-method* :: *frame* ⇒ *mname* **where**
curr-method (*stk*, *loc*, *C*, *M*, *pc*, *ics*) = *M*

fun *curr-pc* :: *frame* ⇒ *nat* **where**
curr-pc (*stk*, *loc*, *C*, *M*, *pc*, *ics*) = *pc*

fun *init-status* :: *frame* ⇒ *init-call-status* **where**
init-status (*stk*, *loc*, *C*, *M*, *pc*, *ics*) = *ics*

fun *ics-of* :: *frame* ⇒ *init-call-status* **where**
ics-of fr = *snd*(*snd*(*snd*(*snd* (*snd fr*))))

2.1.2 Runtime State

type-synonym
jvm-state = *addr option* × *heap* × *frame list* × *sheap*
— exception flag, heap, frames, static heap

translations
(*type*) *jvm-state* <= (*type*) *nat option* × *heap* × *frame list* × *sheap*

fun *frames-of* :: *jvm-state* ⇒ *frame list* **where**
frames-of (*xp*, *h*, *frs*, *sh*) = *frs*

abbreviation *sheap* :: *jvm-state* ⇒ *sheap* **where**
sheap js ≡ *snd* (*snd* (*snd js*))

end

2.2 Instructions of the JVM

theory *JVMInstructions* **imports** *JVMState* **begin**

datatype
instr = *Load nat* — load from local variable
| *Store nat* — store into local variable
| *Push val* — push a value (constant)
| *New cname* — create object
| *Getfield vname cname* — Fetch field from object
| *Getstatic cname vname cname* — Fetch static field from class
| *Putfield vname cname* — Set field in object
| *Putstatic cname vname cname* — Set static field in class
| *Checkcast cname* — Check whether object is of given type
| *Invoke mname nat* — inv. instance meth of an object
| *Invokestatic cname mname nat* — inv. static method of a class
| *Return* — return from method
| *Pop* — pop top element from opstack

```
  | IAdd                  — integer addition
  | Goto int              — goto relative address
  | CmpEq                 — equality comparison
  | IfFalse int           — branch if top of stack false
  | Throw                 — throw top of stack as exception

type-synonym
  bytecode = instr list

type-synonym
  ex-entry = pc × pc × cname × pc × nat
  — start-pc, end-pc, exception type, handler-pc, remaining stack depth

type-synonym
  ex-table = ex-entry list

type-synonym
  jvm-method = nat × nat × bytecode × ex-table
   — max stacksize
   — number of local variables. Add 1 + no. of parameters to get no. of registers
   — instruction sequence
   — exception handler table

type-synonym
  jvm-prog = jvm-method prog

end
```

2.3 Exception handling in the JVM

```
theory JVMExceptions imports ../Common/Exceptions JVMInstructions
begin

definition matches-ex-entry :: 'm prog ⇒ cname ⇒ pc ⇒ ex-entry ⇒ bool
where
  matches-ex-entry P C pc xcp ≡
                let (s, e, C', h, d) = xcp in
                s ≤ pc ∧ pc < e ∧ P ⊢ C ⪯* C'


primrec match-ex-table :: 'm prog ⇒ cname ⇒ pc ⇒ ex-table ⇒ (pc × nat) option
where
  match-ex-table P C pc []     = None
| match-ex-table P C pc (e#es) = (if matches-ex-entry P C pc e
                                  then Some (snd(snd(snd e)))
                                  else match-ex-table P C pc es)

abbreviation
  ex-table-of :: jvm-prog ⇒ cname ⇒ mname ⇒ ex-table where
  ex-table-of P C M == snd (snd (snd (snd (snd (snd (snd (method P C M)))))))


fun find-handler :: jvm-prog ⇒ addr ⇒ heap ⇒ frame list ⇒ sheap ⇒ jvm-state
```

where
find-handler P a h [] sh = (Some a, h, [], sh)
| *find-handler P a h (fr#frs) sh =*
(let (stk,loc,C,M,pc,ics) = fr in
case match-ex-table P (cname-of h a) pc (ex-table-of P C M) of
None ⇒
(case M = clinit of
True ⇒ (case frs of (stk',loc',C',M',pc',ics')#frs'
⇒ (case ics' of Called Cs ⇒ (None, h, (stk',loc',C',M',pc',Throwing (C#Cs) a)#frs', sh)
| - ⇒ (None, h, (stk',loc',C',M',pc',ics')#frs', sh) — this won't happen in wf code
)
| [] ⇒ (Some a, h, [], sh)
)
| - ⇒ find-handler P a h frs sh
)
| Some pc-d ⇒ (None, h, (Addr a # drop (size stk − snd pc-d) stk, loc, C, M, fst pc-d, No-ics)#frs, sh))

lemma *find-handler-cases*:
find-handler P a h frs sh = js
⟹ (∃frs'. frs' ≠ [] ∧ js = (None, h, frs', sh)) ∨ (js = (Some a, h, [], sh))
⟨*proof*⟩

lemma *find-handler-heap*[*simp*]:
find-handler P a h frs sh = (xp',h',frs',sh') ⟹ h' = h
⟨*proof*⟩

lemma *find-handler-sheap*[*simp*]:
find-handler P a h frs sh = (xp',h',frs',sh') ⟹ sh' = sh
⟨*proof*⟩

lemma *find-handler-frames*[*simp*]:
find-handler P a h frs sh = (xp',h',frs',sh') ⟹ length frs' ≤ length frs
⟨*proof*⟩

lemma *find-handler-None*:
find-handler P a h frs sh = (None, h, frs', sh') ⟹ frs' ≠ []
⟨*proof*⟩

lemma *find-handler-Some*:
find-handler P a h frs sh = (Some x, h, frs', sh') ⟹ frs' = []
⟨*proof*⟩

lemma *find-handler-Some-same-error-same-heap*[*simp*]:
find-handler P a h frs sh = (Some x, h', frs', sh') ⟹ x = a ∧ h = h' ∧ sh = sh'
⟨*proof*⟩

lemma *find-handler-prealloc-pres*:
assumes *preallocated h*
and *fh*: *find-handler P a h frs sh = (xp',h',frs',sh')*
shows *preallocated h'*
⟨*proof*⟩

lemma *find-handler-frs-tl-neq*:
ics-of f ≠ *No-ics*
⟹ (*xp*, *h*, *f*#*frs*, *sh*) ≠ *find-handler P xa h'* (*f'* # *frs*) *sh'*
⟨*proof*⟩

end

2.4 Program Execution in the JVM

theory *JVMExecInstr*
imports *JVMInstructions JVMExceptions*
begin

— frame calling the class initialization method for the given class in the given program
fun *create-init-frame* :: [*jvm-prog*, *cname*] ⇒ *frame* **where**
create-init-frame P C =
(*let* (*D*,*b*,*Ts*,*T*,(*mxs*,*mxl*$_0$,*ins*,*xt*)) = *method P C clinit*
in ([],(*replicate mxl*$_0$ *undefined*),*D*,*clinit*,*0*,*No-ics*)
)

primrec *exec-instr* :: [*instr*, *jvm-prog*, *heap*, *val list*, *val list*,
cname, *mname*, *pc*, *init-call-status*, *frame list*, *sheap*] ⇒ *jvm-state*
where
exec-instr-Load:
exec-instr (*Load n*) *P h stk loc C*$_0$ *M*$_0$ *pc ics frs sh* =
(*None*, *h*, ((*loc* ! *n*) # *stk*, *loc*, *C*$_0$, *M*$_0$, *Suc pc*, *ics*)#*frs*, *sh*)

| *exec-instr-Store*:
exec-instr (*Store n*) *P h stk loc C*$_0$ *M*$_0$ *pc ics frs sh* =
(*None*, *h*, (*tl stk*, *loc*[*n*:=*hd stk*], *C*$_0$, *M*$_0$, *Suc pc*, *ics*)#*frs*, *sh*)

| *exec-instr-Push*:
exec-instr (*Push v*) *P h stk loc C*$_0$ *M*$_0$ *pc ics frs sh* =
(*None*, *h*, (*v* # *stk*, *loc*, *C*$_0$, *M*$_0$, *Suc pc*, *ics*)#*frs*, *sh*)

| *exec-instr-New*:
exec-instr (*New C*) *P h stk loc C*$_0$ *M*$_0$ *pc ics frs sh* =
(*case* (*ics*, *sh C*) *of*
(*Called Cs*, -) ⇒
(*case new-Addr h of*
None ⇒ (⌊*addr-of-sys-xcpt OutOfMemory*⌋, *h*, (*stk*, *loc*, *C*$_0$, *M*$_0$, *pc*, *No-ics*)#*frs*, *sh*)
| *Some a* ⇒ (*None*, *h*(*a*↦*blank P C*), (*Addr a*#*stk*, *loc*, *C*$_0$, *M*$_0$, *Suc pc*, *No-ics*)#*frs*, *sh*)
)
| (-, *Some*(*obj*, *Done*)) ⇒
(*case new-Addr h of*
None ⇒ (⌊*addr-of-sys-xcpt OutOfMemory*⌋, *h*, (*stk*, *loc*, *C*$_0$, *M*$_0$, *pc*, *ics*)#*frs*, *sh*)
| *Some a* ⇒ (*None*, *h*(*a*↦*blank P C*), (*Addr a*#*stk*, *loc*, *C*$_0$, *M*$_0$, *Suc pc*, *ics*)#*frs*, *sh*)
)
| - ⇒ (*None*, *h*, (*stk*, *loc*, *C*$_0$, *M*$_0$, *pc*, *Calling C* [])#*frs*, *sh*)
)

| *exec-instr-Getfield*:

```
exec-instr (Getfield F C) P h stk loc C0 M0 pc ics frs sh =
  (let v      = hd stk;
       (D,fs) = the(h(the-Addr v));
       (D',b,t) = field P C F;
       xp'    = if v=Null then ⌊addr-of-sys-xcpt NullPointer⌋
                else if ¬(∃ t b. P ⊢ D has F,b:t in C)
                     then ⌊addr-of-sys-xcpt NoSuchFieldError⌋
                     else case b of Static ⇒ ⌊addr-of-sys-xcpt IncompatibleClassChangeError⌋
                                  | NonStatic ⇒ None
   in case xp' of None ⇒ (xp', h, (the(fs(F,C))#(tl stk), loc, C0, M0, pc+1, ics)#frs, sh)
                | Some x ⇒ (xp', h, (stk, loc, C0, M0, pc, ics)#frs, sh))

| exec-instr-Getstatic:
exec-instr (Getstatic C F D) P h stk loc C0 M0 pc ics frs sh =
  (let (D',b,t) = field P D F;
       xp'    = if ¬(∃ t b. P ⊢ C has F,b:t in D)
                then ⌊addr-of-sys-xcpt NoSuchFieldError⌋
                else case b of NonStatic ⇒ ⌊addr-of-sys-xcpt IncompatibleClassChangeError⌋
                             | Static ⇒ None
   in (case (xp', ics, sh D') of
         (Some a, -) ⇒ (xp', h, (stk, loc, C0, M0, pc, ics)#frs, sh)
       | (-, Called Cs, -) ⇒ let (sfs, i) = the(sh D');
                                 v = the(sfs F)
                             in (xp', h, (v#stk, loc, C0, M0, Suc pc, No-ics)#frs, sh)
       | (-, -, Some (sfs, Done)) ⇒ let v = the (sfs F)
                                    in (xp', h, (v#stk, loc, C0, M0, Suc pc, ics)#frs, sh)
       | - ⇒ (xp', h, (stk, loc, C0, M0, pc, Calling D' [])#frs, sh)
    )
  )

| exec-instr-Putfield:
exec-instr (Putfield F C) P h stk loc C0 M0 pc ics frs sh =
  (let v    = hd stk;
       r    = hd (tl stk);
       a    = the-Addr r;
       (D,fs) = the (h a);
       (D',b,t) = field P C F;
       xp'    = if r=Null then ⌊addr-of-sys-xcpt NullPointer⌋
                else if ¬(∃ t b. P ⊢ D has F,b:t in C)
                     then ⌊addr-of-sys-xcpt NoSuchFieldError⌋
                     else case b of Static ⇒ ⌊addr-of-sys-xcpt IncompatibleClassChangeError⌋
                                  | NonStatic ⇒ None;
       h'   = h(a ↦ (D, fs((F,C) ↦ v)))
   in case xp' of None ⇒ (xp', h', (tl (tl stk)), loc, C0, M0, pc+1, ics)#frs, sh)
                | Some x ⇒ (xp', h, (stk, loc, C0, M0, pc, ics)#frs, sh)
  )

| exec-instr-Putstatic:
exec-instr (Putstatic C F D) P h stk loc C0 M0 pc ics frs sh =
  (let (D',b,t) = field P D F;
       xp'    = if ¬(∃ t b. P ⊢ C has F,b:t in D)
                then ⌊addr-of-sys-xcpt NoSuchFieldError⌋
                else case b of NonStatic ⇒ ⌊addr-of-sys-xcpt IncompatibleClassChangeError⌋
                             | Static ⇒ None
```

```
   in (case (xp', ics, sh D') of
         (Some a, -) ⇒ (xp', h, (stk, loc, C_0, M_0, pc, ics)#frs, sh)
       | (-, Called Cs, -)
      ⇒ let (sfs, i) = the(sh D')
        in (xp', h, (tl stk, loc, C_0, M_0, Suc pc, No-ics)#frs, sh(D':=Some ((sfs(F ↦ hd stk)), i)))
       | (-, -, Some (sfs, Done))
      ⇒ (xp', h, (tl stk, loc, C_0, M_0, Suc pc, ics)#frs, sh(D':=Some ((sfs(F ↦ hd stk)), Done)))
       | - ⇒ (xp', h, (stk, loc, C_0, M_0, pc, Calling D' [])#frs, sh)
     )
  )

| exec-instr-Checkcast:
exec-instr (Checkcast C) P h stk loc C_0 M_0 pc ics frs sh =
   (if cast-ok P C h (hd stk)
     then (None, h, (stk, loc, C_0, M_0, Suc pc, ics)#frs, sh)
     else (⌊addr-of-sys-xcpt ClassCast⌋, h, (stk, loc, C_0, M_0, pc, ics)#frs, sh)
   )

| exec-instr-Invoke:
exec-instr (Invoke M n) P h stk loc C_0 M_0 pc ics frs sh =
  (let ps  = take n stk;
       r   = stk!n;
       C   = fst(the(h(the-Addr r)));
       (D,b,Ts,T,mxs,mxl_0,ins,xt)= method P C M;
       xp' = if r=Null then ⌊addr-of-sys-xcpt NullPointer⌋
             else if ¬(∃ Ts T m D b. P ⊢ C sees M,b:Ts → T = m in D)
                  then ⌊addr-of-sys-xcpt NoSuchMethodError⌋
                  else case b of Static ⇒ ⌊addr-of-sys-xcpt IncompatibleClassChangeError⌋
                               | NonStatic ⇒ None;
       f'  = ([],[r]@(rev ps)@(replicate mxl_0 undefined),D,M,0,No-ics)
   in case xp' of None ⇒ (xp', h, f'#(stk, loc, C_0, M_0, pc, ics)#frs, sh)
                | Some a ⇒ (xp', h, (stk, loc, C_0, M_0, pc, ics)#frs, sh)
  )

| exec-instr-Invokestatic:
exec-instr (Invokestatic C M n) P h stk loc C_0 M_0 pc ics frs sh =
  (let ps  = take n stk;
       (D,b,Ts,T,mxs,mxl_0,ins,xt)= method P C M;
       xp' = if ¬(∃ Ts T m D b. P ⊢ C sees M,b:Ts → T = m in D)
             then ⌊addr-of-sys-xcpt NoSuchMethodError⌋
             else case b of NonStatic ⇒ ⌊addr-of-sys-xcpt IncompatibleClassChangeError⌋
                          | Static ⇒ None;
       f'  = ([],(rev ps)@(replicate mxl_0 undefined),D,M,0,No-ics)
   in (case (xp', ics, sh D) of
         (Some a, -) ⇒ (xp', h, (stk, loc, C_0, M_0, pc, ics)#frs, sh)
       | (-, Called Cs, -) ⇒ (xp', h, f'#(stk, loc, C_0, M_0, pc, No-ics)#frs, sh)
       | (-, -, Some (sfs, Done)) ⇒ (xp', h, f'#(stk, loc, C_0, M_0, pc, ics)#frs, sh)
       | - ⇒ (xp', h, (stk, loc, C_0, M_0, pc, Calling D [])#frs, sh)
     )
  )

| exec-instr-Return:
 exec-instr Return P h stk_0 loc_0 C_0 M_0 pc ics frs sh =
    (case frs of
```

```
      [] ⇒ let sh' =  (case M₀ = clinit of True ⇒ sh(C₀:=Some(fst(the(sh C₀)), Done))
                                              | - ⇒ sh
                      )
           in (None, h, [], sh')
    | (stk',loc',C',m',pc',ics')#frs'
        ⇒ let (D,b,Ts,T,(mxs,mxl₀,ins,xt)) = method P C₀ M₀;
              offset = case b of NonStatic ⇒ 1 | Static ⇒ 0;
              (sh'', stk'', pc'') = (case M₀ = clinit of True ⇒ (sh(C₀:=Some(fst(the(sh C₀)), Done)),
stk', pc')
                                                  | - ⇒ (sh, (hd stk₀)#(drop (length Ts + offset) stk'), Suc pc')
                                    )
          in (None, h, (stk'',loc',C',m',pc'',ics')#frs', sh'')
  )

| exec-instr-Pop:
exec-instr Pop P h stk loc C₀ M₀ pc ics frs sh = (None, h, (tl stk, loc, C₀, M₀, Suc pc, ics)#frs, sh)

| exec-instr-IAdd:
exec-instr IAdd P h stk loc C₀ M₀ pc ics frs sh =
    (None, h, (Intg (the-Intg (hd (tl stk)) + the-Intg (hd stk))#(tl (tl stk)), loc, C₀, M₀, Suc pc,
ics)#frs, sh)

| exec-instr-IfFalse:
exec-instr (IfFalse i) P h stk loc C₀ M₀ pc ics frs sh =
  (let pc' = if hd stk = Bool False then nat(int pc+i) else pc+1
   in (None, h, (tl stk, loc, C₀, M₀, pc', ics)#frs, sh))

| exec-instr-CmpEq:
exec-instr CmpEq P h stk loc C₀ M₀ pc ics frs sh =
   (None, h, (Bool (hd (tl stk) = hd stk) # tl (tl stk), loc, C₀, M₀, Suc pc, ics)#frs, sh)

| exec-instr-Goto:
exec-instr (Goto i) P h stk loc C₀ M₀ pc ics frs sh =
  (None, h, (stk, loc, C₀, M₀, nat(int pc+i), ics)#frs, sh)

| exec-instr-Throw:
exec-instr Throw P h stk loc C₀ M₀ pc ics frs sh =
  (let xp' = if hd stk = Null then ⌊addr-of-sys-xcpt NullPointer⌋
             else ⌊the-Addr(hd stk)⌋
   in (xp', h, (stk, loc, C₀, M₀, pc, ics)#frs, sh))
```

Given a preallocated heap, a thrown exception is either a system exception or thrown directly by *Throw*.

```
lemma exec-instr-xcpts:
assumes σ' = exec-instr i P h stk loc C M pc ics' frs sh
  and fst σ' = Some a
shows i = (JVMInstructions.Throw) ∨ a ∈ {a. ∃ x ∈ sys-xcpts. a = addr-of-sys-xcpt x}
⟨proof⟩

lemma exec-instr-prealloc-pres:
assumes preallocated h
  and exec-instr i P h stk loc C₀ M₀ pc ics frs sh = (xp',h',frs',sh')
shows preallocated h'
⟨proof⟩
```

end

2.5 Program Execution in the JVM in full small step style

theory *JVMExec*
imports *JVMExecInstr*
begin

abbreviation
instrs-of :: *jvm-prog* ⇒ *cname* ⇒ *mname* ⇒ *instr list* **where**
instrs-of P C M == *fst*(*snd*(*snd*(*snd*(*snd*(*snd*(*snd*(*method P C M*)))))))

fun *curr-instr* :: *jvm-prog* ⇒ *frame* ⇒ *instr* **where**
curr-instr P (*stk*,*loc*,*C*,*M*,*pc*,*ics*) = *instrs-of P C M* ! *pc*

— execution of single step of the initialization procedure
fun *exec-Calling* :: [*cname*, *cname list*, *jvm-prog*, *heap*, *val list*, *val list*,
cname, *mname*, *pc*, *frame list*, *sheap*] ⇒ *jvm-state*
where
exec-Calling C Cs P h stk loc C_0 M_0 *pc frs sh* =
(*case sh C of*
None ⇒ (*None*, *h*, (*stk*, *loc*, C_0, M_0, *pc*, *Calling C Cs*)#*frs*, *sh*(*C* := *Some* (*sblank P C*, *Prepared*)))
| *Some* (*obj*, *iflag*) ⇒
(*case iflag of*
Done ⇒ (*None*, *h*, (*stk*, *loc*, C_0, M_0, *pc*, *Called Cs*)#*frs*, *sh*)
| *Processing* ⇒ (*None*, *h*, (*stk*, *loc*, C_0, M_0, *pc*, *Called Cs*)#*frs*, *sh*)
| *Error* ⇒ (*None*, *h*, (*stk*, *loc*, C_0, M_0, *pc*,
Throwing Cs (*addr-of-sys-xcpt NoClassDefFoundError*))#*frs*, *sh*)
| *Prepared* ⇒
let sh' = *sh*(*C*:=*Some*(*fst*(*the*(*sh C*)), *Processing*));
D = *fst*(*the*(*class P C*))
in if C = *Object*
then (*None*, *h*, (*stk*, *loc*, C_0, M_0, *pc*, *Called* (*C*#*Cs*))#*frs*, *sh'*)
else (*None*, *h*, (*stk*, *loc*, C_0, M_0, *pc*, *Calling D* (*C*#*Cs*))#*frs*, *sh'*)
)
)

— single step of execution without error handling
fun *exec-step* :: [*jvm-prog*, *heap*, *val list*, *val list*,
cname, *mname*, *pc*, *init-call-status*, *frame list*, *sheap*] ⇒ *jvm-state*
where
exec-step P h stk loc C M pc (*Calling C' Cs*) *frs sh*
= *exec-Calling C' Cs P h stk loc C M pc frs sh* |
exec-step P h stk loc C M pc (*Called* (*C'*#*Cs*)) *frs sh*
= (*None*, *h*, *create-init-frame P C'*#(*stk*, *loc*, *C*, *M*, *pc*, *Called Cs*)#*frs*, *sh*) |
exec-step P h stk loc C M pc (*Throwing* (*C'*#*Cs*) *a*) *frs sh*
= (*None*, *h*, (*stk*,*loc*,*C*,*M*,*pc*,*Throwing Cs a*)#*frs*, *sh*(*C'*:=*Some*(*fst*(*the*(*sh C'*)), *Error*))) |
exec-step P h stk loc C M pc (*Throwing* [] *a*) *frs sh*
= (⌊*a*⌋, *h*, (*stk*,*loc*,*C*,*M*,*pc*,*No-ics*)#*frs*, *sh*) |
exec-step P h stk loc C M pc ics frs sh
= *exec-instr* (*instrs-of P C M* ! *pc*) *P h stk loc C M pc ics frs sh*

— execution including error handling
fun *exec* :: *jvm-prog* × *jvm-state* ⇒ *jvm-state option* — single step execution **where**
exec (*P*, *None*, *h*, (*stk*,*loc*,*C*,*M*,*pc*,*ics*)#*frs*, *sh*) =
  (*let* (*xp'*, *h'*, *frs'*, *sh'*) = *exec-step P h stk loc C M pc ics frs sh*
    in case xp' of
        None ⇒ *Some* (*None*,*h'*,*frs'*,*sh'*)
      | *Some x* ⇒ *Some* (*find-handler P x h* ((*stk*,*loc*,*C*,*M*,*pc*,*ics*)#*frs*) *sh*)
)
| *exec* - = *None*

— relational view
inductive-set
  exec-1 :: *jvm-prog* ⇒ (*jvm-state* × *jvm-state*) *set*
  and *exec-1'* :: *jvm-prog* ⇒ *jvm-state* ⇒ *jvm-state* ⇒ *bool*
    (‹- ⊢/ - −*jvm*→$_1$/ -› [*61*,*61*,*61*] *60*)
  for *P* :: *jvm-prog*
where
  P ⊢ *σ* −*jvm*→$_1$ *σ'* ≡ (*σ*,*σ'*) ∈ *exec-1 P*
| *exec-1I*: *exec* (*P*,*σ*) = *Some σ'* ⟹ *P* ⊢ *σ* −*jvm*→$_1$ *σ'*

— reflexive transitive closure:
definition *exec-all* :: *jvm-prog* ⇒ *jvm-state* ⇒ *jvm-state* ⇒ *bool*
    (‹(- ⊢/ - −*jvm*→/ -)› [*61*,*61*,*61*]*60*) **where**
  exec-all-def1: *P* ⊢ *σ* −*jvm*→ *σ'* ⟷ (*σ*,*σ'*) ∈ (*exec-1 P*)*

notation (*ASCII*)
  exec-all (‹- |−/ - −*jvm*−>/ -› [*61*,*61*,*61*]*60*)

lemma *exec-1-eq*:
  exec-1 P = {(*σ*,*σ'*). *exec* (*P*,*σ*) = *Some σ'*}⟨*proof*⟩
lemma *exec-1-iff*:
  P ⊢ *σ* −*jvm*→$_1$ *σ'* = (*exec* (*P*,*σ*) = *Some σ'*)⟨*proof*⟩
lemma *exec-all-def*:
  P ⊢ *σ* −*jvm*→ *σ'* = ((*σ*,*σ'*) ∈ {(*σ*,*σ'*). *exec* (*P*,*σ*) = *Some σ'*}*)⟨*proof*⟩
lemma *jvm-refl*[*iff*]: *P* ⊢ *σ* −*jvm*→ *σ*⟨*proof*⟩
lemma *jvm-trans*[*trans*]:
⟦ *P* ⊢ *σ* −*jvm*→ *σ'*; *P* ⊢ *σ'* −*jvm*→ *σ''* ⟧ ⟹ *P* ⊢ *σ* −*jvm*→ *σ''*⟨*proof*⟩
lemma *jvm-one-step1*[*trans*]:
⟦ *P* ⊢ *σ* −*jvm*→$_1$ *σ'*; *P* ⊢ *σ'* −*jvm*→ *σ''* ⟧ ⟹ *P* ⊢ *σ* −*jvm*→ *σ''*⟨*proof*⟩
lemma *jvm-one-step2*[*trans*]:
⟦ *P* ⊢ *σ* −*jvm*→ *σ'*; *P* ⊢ *σ'* −*jvm*→$_1$ *σ''* ⟧ ⟹ *P* ⊢ *σ* −*jvm*→ *σ''*⟨*proof*⟩
lemma *exec-all-conf*:
  ⟦ *P* ⊢ *σ* −*jvm*→ *σ'*; *P* ⊢ *σ* −*jvm*→ *σ''* ⟧
  ⟹ *P* ⊢ *σ'* −*jvm*→ *σ''* ∨ *P* ⊢ *σ''* −*jvm*→ *σ'*⟨*proof*⟩
lemma *exec-1-exec-all-conf*:
⟦ *exec* (*P*, *σ*) = *Some σ'*; *P* ⊢ *σ* −*jvm*→ *σ''*; *σ* ≠ *σ''* ⟧
⟹ *P* ⊢ *σ'* −*jvm*→ *σ''*
⟨*proof*⟩

lemma *exec-all-finalD*: *P* ⊢ (*x*, *h*, [], *sh*) −*jvm*→ *σ* ⟹ *σ* = (*x*, *h*, [], *sh*)⟨*proof*⟩
lemma *exec-all-deterministic*:
  ⟦ *P* ⊢ *σ* −*jvm*→ (*x*,*h*,[],*sh*); *P* ⊢ *σ* −*jvm*→ *σ'* ⟧ ⟹ *P* ⊢ *σ'* −*jvm*→ (*x*,*h*,[],*sh*)⟨*proof*⟩

2.5.1 Preservation of preallocated

lemma *exec-Calling-prealloc-pres*:
assumes *preallocated h*
and *exec-Calling C Cs P h stk loc* C_0 M_0 *pc frs sh* = (*xp'*,*h'*,*frs'*,*sh'*)
shows *preallocated h'*
⟨*proof*⟩

lemma *exec-step-prealloc-pres*:
assumes *pre*: *preallocated h*
and *exec-step P h stk loc C M pc ics frs sh* = (*xp'*,*h'*,*frs'*,*sh'*)
shows *preallocated h'*
⟨*proof*⟩

lemma *exec-prealloc-pres*:
assumes *pre*: *preallocated h*
and *exec* (*P*, *xp*, *h*, *frs*, *sh*) = *Some*(*xp'*,*h'*,*frs'*,*sh'*)
shows *preallocated h'*
⟨*proof*⟩

2.5.2 Start state

The *Start* class is defined based on a given initial class and method. It has two methods: one that calls the initial method in the initial class, which is called by the starting program, and *clinit*, as required for the class to be well-formed.

definition *start-method* :: *cname* ⇒ *mname* ⇒ *jvm-method mdecl* **where**
start-method C M = (*start-m*, *Static*, [], *Void*, (*1*,*0*,[*Invokestatic C M 0*,*Return*],[]))
abbreviation *start-clinit* :: *jvm-method mdecl* **where**
start-clinit ≡ (*clinit*, *Static*, [], *Void*, (*1*,*0*,[*Push Unit*,*Return*],[]))
definition *start-class* :: *cname* ⇒ *mname* ⇒ *jvm-method cdecl* **where**
start-class C M = (*Start*, *Object*, [], [*start-method C M*, *start-clinit*])

The start configuration of the JVM in program *P*: in the start heap, we call the "start" method of the "Start"; this method performs *Invokestatic* on the class and method given to create the start program's *Start* class. This allows the initialization procedure to be called on the initial class in a natural way before the initial method is performed. There is no *this* pointer of the frame as *start* is *Static*. The start sheap has every class pre-prepared; this decision is not necessary. The starting program includes the added *Start* class, given a method *M* of class *C*, added to program *P*.

definition *start-state* :: *jvm-prog* ⇒ *jvm-state* **where**
start-state P = (*None*, *start-heap P*, [([], [], *Start*, *start-m*, *0*, *No-ics*)], *start-sheap*)
abbreviation *start-prog* :: *jvm-prog* ⇒ *cname* ⇒ *mname* ⇒ *jvm-prog* **where**
start-prog P C M ≡ *start-class C M* # *P*

end

2.6 A Defensive JVM

theory *JVMDefensive*
imports *JVMExec ../Common/Conform*
begin

Extend the state space by one element indicating a type error (or other abnormal termination)

datatype *'a type-error = TypeError | Normal 'a*

fun *is-Addr :: val ⇒ bool* **where**
  is-Addr (Addr a) ⟷ True
| *is-Addr v ⟷ False*

fun *is-Intg :: val ⇒ bool* **where**
  is-Intg (Intg i) ⟷ True
| *is-Intg v ⟷ False*

fun *is-Bool :: val ⇒ bool* **where**
  is-Bool (Bool b) ⟷ True
| *is-Bool v ⟷ False*

definition *is-Ref :: val ⇒ bool* **where**
  is-Ref v ⟷ v = Null ∨ is-Addr v

primrec *check-instr :: [instr, jvm-prog, heap, val list, val list,*
cname, mname, pc, frame list, sheap] ⇒ bool **where**
  check-instr-Load:
    check-instr (Load n) P h stk loc C M₀ pc frs sh =
    (n < length loc)

| *check-instr-Store*:
    check-instr (Store n) P h stk loc C₀ M₀ pc frs sh =
    (0 < length stk ∧ n < length loc)

| *check-instr-Push*:
    check-instr (Push v) P h stk loc C₀ M₀ pc frs sh =
    (¬is-Addr v)

| *check-instr-New*:
    check-instr (New C) P h stk loc C₀ M₀ pc frs sh =
    is-class P C

| *check-instr-Getfield*:
    check-instr (Getfield F C) P h stk loc C₀ M₀ pc frs sh =
    (0 < length stk ∧ (∃ C′ T. P ⊢ C sees F,NonStatic:T in C′) ∧
    (let (C′, b, T) = field P C F; ref = hd stk in
      C′ = C ∧ is-Ref ref ∧ (ref ≠ Null ⟶
        h (the-Addr ref) ≠ None ∧
        (let (D,vs) = the (h (the-Addr ref)) in
          P ⊢ D ≼ C ∧ vs (F,C) ≠ None ∧ P,h ⊢ the (vs (F,C)) :≤ T))))*

| *check-instr-Getstatic*:
    check-instr (Getstatic C F D) P h stk loc C₀ M₀ pc frs sh =
    ((∃ T. P ⊢ C sees F,Static:T in D) ∧
    (let (C′, b, T) = field P C F in
      C′ = D ∧ (sh D ≠ None ⟶
        (let (sfs,i) = the (sh D) in
          sfs F ≠ None ∧ P,h ⊢ the (sfs F) :≤ T))))

| *check-instr-Putfield*:
  check-instr (*Putfield F C*) *P h stk loc* C_0 M_0 *pc frs sh* =
  (*1* < *length stk* ∧ (∃ *C′ T*. *P* ⊢ *C sees F*,*NonStatic*:*T in C′*) ∧
  (*let* (*C′*, *b*, *T*) = *field P C F*; *v* = *hd stk*; *ref* = *hd* (*tl stk*) *in*
    C′ = *C* ∧ *is-Ref ref* ∧ (*ref* ≠ *Null* ⟶
      h (*the-Addr ref*) ≠ *None* ∧
      (*let D* = *fst* (*the* (*h* (*the-Addr ref*))) *in*
        P ⊢ *D* ≼* *C* ∧ *P*,*h* ⊢ *v* :≤ *T*))))

| *check-instr-Putstatic*:
  check-instr (*Putstatic C F D*) *P h stk loc* C_0 M_0 *pc frs sh* =
  (*0* < *length stk* ∧ (∃ *T*. *P* ⊢ *C sees F*,*Static*:*T in D*) ∧
  (*let* (*C′*, *b*, *T*) = *field P C F*; *v* = *hd stk in*
    C′ = *D* ∧ *P*,*h* ⊢ *v* :≤ *T*))

| *check-instr-Checkcast*:
  check-instr (*Checkcast C*) *P h stk loc* C_0 M_0 *pc frs sh* =
  (*0* < *length stk* ∧ *is-class P C* ∧ *is-Ref* (*hd stk*))

| *check-instr-Invoke*:
  check-instr (*Invoke M n*) *P h stk loc* C_0 M_0 *pc frs sh* =
  (*n* < *length stk* ∧ *is-Ref* (*stk*!*n*) ∧
  (*stk*!*n* ≠ *Null* ⟶
    (*let a* = *the-Addr* (*stk*!*n*);
        C = *cname-of h a*;
        Ts = *fst* (*snd* (*snd* (*method P C M*)))
    in h a ≠ *None* ∧ *P* ⊢ *C has M*,*NonStatic* ∧
      P,*h* ⊢ *rev* (*take n stk*) [:≤] *Ts*)))

| *check-instr-Invokestatic*:
  check-instr (*Invokestatic C M n*) *P h stk loc* C_0 M_0 *pc frs sh* =
  (*n* ≤ *length stk* ∧
    (*let Ts* = *fst* (*snd* (*snd* (*method P C M*)))
    in P ⊢ *C has M*,*Static* ∧
      P,*h* ⊢ *rev* (*take n stk*) [:≤] *Ts*))

| *check-instr-Return*:
  check-instr Return P h stk loc C_0 M_0 *pc frs sh* =
  (*case* (*M*₀ = *clinit*) *of False* ⇒ (*0* < *length stk* ∧ ((*0* < *length frs*) ⟶
                                (∃ *b*. *P* ⊢ C_0 *has* M_0,*b*) ∧
                                (*let v* = *hd stk*;
                                    T = *fst* (*snd* (*snd* (*snd* (*method P* C_0 M_0))))
                                in P,*h* ⊢ *v* :≤ *T*)))
                | *True* ⇒ *P* ⊢ C_0 *has* M_0,*Static*)

| *check-instr-Pop*:
  check-instr Pop P h stk loc C_0 M_0 *pc frs sh* =
  (*0* < *length stk*)

| *check-instr-IAdd*:
  check-instr IAdd P h stk loc C_0 M_0 *pc frs sh* =
  (*1* < *length stk* ∧ *is-Intg* (*hd stk*) ∧ *is-Intg* (*hd* (*tl stk*)))

| *check-instr-IfFalse*:

check-instr (*IfFalse b*) *P h stk loc* C_0 M_0 *pc frs sh* =
(*0* < *length stk* ∧ *is-Bool* (*hd stk*) ∧ *0* ≤ *int pc*+*b*)

| *check-instr-CmpEq*:
check-instr CmpEq P h stk loc C_0 M_0 *pc frs sh* =
(*1* < *length stk*)

| *check-instr-Goto*:
check-instr (*Goto b*) *P h stk loc* C_0 M_0 *pc frs sh* =
(*0* ≤ *int pc*+*b*)

| *check-instr-Throw*:
check-instr Throw P h stk loc C_0 M_0 *pc frs sh* =
(*0* < *length stk* ∧ *is-Ref* (*hd stk*))

definition *check* :: *jvm-prog* ⇒ *jvm-state* ⇒ *bool* **where**
check P σ = (*let* (*xcpt*, *h*, *frs*, *sh*) = *σ in*
(*case frs of* [] ⇒ *True* | (*stk*,*loc*,*C*,*M*,*pc*,*ics*)#*frs′* ⇒
∃ *b*. *P* ⊢ *C has M*, *b* ∧
(*let* (*C′*,*b*,*Ts*,*T*,*mxs*,*mxl*$_0$,*ins*,*xt*) = *method P C M*; *i* = *ins*!*pc in*
pc < *size ins* ∧ *size stk* ≤ *mxs* ∧
check-instr i P h stk loc C M pc frs′ sh)))

definition *exec-d* :: *jvm-prog* ⇒ *jvm-state* ⇒ *jvm-state option type-error* **where**
exec-d P σ = (*if check P σ then Normal* (*exec* (*P*, *σ*)) *else TypeError*)

inductive-set
exec-1-d :: *jvm-prog* ⇒ (*jvm-state type-error* × *jvm-state type-error*) *set*
and *exec-1-d′* :: *jvm-prog* ⇒ *jvm-state type-error* ⇒ *jvm-state type-error* ⇒ *bool*
(‹- ⊢ - −*jvmd*→$_1$ -› [*61*,*61*,*61*]*60*)
for *P* :: *jvm-prog*
where
P ⊢ *σ* −*jvmd*→$_1$ *σ′* ≡ (*σ*,*σ′*) ∈ *exec-1-d P*
| *exec-1-d-ErrorI*: *exec-d P σ* = *TypeError* ⟹ *P* ⊢ *Normal σ* −*jvmd*→$_1$ *TypeError*
| *exec-1-d-NormalI*: *exec-d P σ* = *Normal* (*Some σ′*) ⟹ *P* ⊢ *Normal σ* −*jvmd*→$_1$ *Normal σ′*

— reflexive transitive closure:
definition *exec-all-d* :: *jvm-prog* ⇒ *jvm-state type-error* ⇒ *jvm-state type-error* ⇒ *bool*
(‹- ⊢ - −*jvmd*→ -› [*61*,*61*,*61*]*60*) **where**
exec-all-d-def1: *P* ⊢ *σ* −*jvmd*→ *σ′* ⟷ (*σ*,*σ′*) ∈ (*exec-1-d P*)*

notation (*ASCII*)
exec-all-d (‹- |− - −*jvmd*−> -› [*61*,*61*,*61*]*60*)

lemma *exec-1-d-eq*:
exec-1-d P = {(*s*,*t*). ∃*σ*. *s* = *Normal σ* ∧ *t* = *TypeError* ∧ *exec-d P σ* = *TypeError*} ∪
{(*s*,*t*). ∃*σ σ′*. *s* = *Normal σ* ∧ *t* = *Normal σ′* ∧ *exec-d P σ* = *Normal* (*Some σ′*)}
⟨*proof*⟩

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

lemma *if-neq* [*dest*!]:
(*if P then A else B*) ≠ *B* ⟹ *P*
⟨*proof*⟩

lemma *exec-d-no-errorI* [*intro*]:
check P σ ⟹ *exec-d P σ* ≠ *TypeError*
⟨*proof*⟩

theorem *no-type-error-commutes*:
exec-d P σ ≠ *TypeError* ⟹ *exec-d P σ* = *Normal* (*exec* (*P*, *σ*))
⟨*proof*⟩

lemma *defensive-imp-aggressive*:
P ⊢ (*Normal σ*) −*jvmd*→ (*Normal σ'*) ⟹ *P* ⊢ *σ* −*jvm*→ *σ'*⟨*proof*⟩
end

2.7 The Jinja Type System as a Semilattice

theory *SemiType*
imports *../Common/ WellForm Jinja.Semilattices*
begin

definition *super* :: *'a prog* ⇒ *cname* ⇒ *cname*
where *super P C* ≡ *fst* (*the* (*class P C*))

lemma *superI*:
(*C*,*D*) ∈ *subcls1 P* ⟹ *super P C* = *D*
⟨*proof*⟩

primrec *the-Class* :: *ty* ⇒ *cname*
where
the-Class (*Class C*) = *C*

definition *sup* :: *'c prog* ⇒ *ty* ⇒ *ty* ⇒ *ty err*
where
sup P T$_1$ *T*$_2$ ≡
if is-refT T$_1$ ∧ *is-refT T*$_2$ *then*
OK (*if T*$_1$ = *NT then T*$_2$ *else*
if T$_2$ = *NT then T*$_1$ *else*
(*Class* (*exec-lub* (*subcls1 P*) (*super P*) (*the-Class T*$_1$) (*the-Class T*$_2$))))
else
(*if T*$_1$ = *T*$_2$ *then OK T*$_1$ *else Err*)

lemma *sup-def'*:
sup P = (λ*T*$_1$ *T*$_2$.
if is-refT T$_1$ ∧ *is-refT T*$_2$ *then*
OK (*if T*$_1$ = *NT then T*$_2$ *else*
if T$_2$ = *NT then T*$_1$ *else*
(*Class* (*exec-lub* (*subcls1 P*) (*super P*) (*the-Class T*$_1$) (*the-Class T*$_2$))))
else

(*if* $T_1 = T_2$ *then OK* T_1 *else Err*))
⟨*proof*⟩

abbreviation
subtype :: *'c prog* ⇒ *ty* ⇒ *ty* ⇒ *bool*
where *subtype P* ≡ *widen P*

definition *esl* :: *'c prog* ⇒ *ty esl*
where
esl P ≡ (*types P*, *subtype P*, *sup P*)

lemma *is-class-is-subcls*:
wf-prog m P ⟹ *is-class P C* = *P* ⊢ *C* ≼* *Object*⟨*proof*⟩

lemma *subcls-antisym*:
⟦*wf-prog m P*; *P* ⊢ *C* ≼* *D*; *P* ⊢ *D* ≼* *C*⟧ ⟹ *C* = *D*
⟨*proof*⟩

lemma *widen-antisym*:
⟦ *wf-prog m P*; *P* ⊢ *T* ≤ *U*; *P* ⊢ *U* ≤ *T* ⟧ ⟹ *T* = *U*⟨*proof*⟩
lemma *order-widen* [*intro*,*simp*]:
wf-prog m P ⟹ *order* (*subtype P*) (*types P*)⟨*proof*⟩

lemma *NT-widen*:
P ⊢ *NT* ≤ *T* = (*T* = *NT* ∨ (∃ *C*. *T* = *Class C*))⟨*proof*⟩

lemma *Class-widen2*: *P* ⊢ *Class C* ≤ *T* = (∃ *D*. *T* = *Class D* ∧ *P* ⊢ *C* ≼* *D*)⟨*proof*⟩
lemma *wf-converse-subcls1-impl-acc-subtype*:
wf ((*subcls1 P*)^−1) ⟹ *acc* (*subtype P*)⟨*proof*⟩
lemma *wf-subtype-acc* [*intro*, *simp*]:
wf-prog wf-mb P ⟹ *acc* (*subtype P*)⟨*proof*⟩
lemma *exec-lub-refl* [*simp*]: *exec-lub r f T T* = *T*⟨*proof*⟩
lemma *closed-err-types*:
wf-prog wf-mb P ⟹ *closed* (*err* (*types P*)) (*lift2* (*sup P*))⟨*proof*⟩

lemma *sup-subtype-greater*:
⟦ *wf-prog wf-mb P*; *is-type P t1*; *is-type P t2*; *sup P t1 t2* = *OK s* ⟧
⟹ *subtype P t1 s* ∧ *subtype P t2 s*⟨*proof*⟩
lemma *sup-subtype-smallest*:
⟦ *wf-prog wf-mb P*; *is-type P a*; *is-type P b*; *is-type P c*;
subtype P a c; *subtype P b c*; *sup P a b* = *OK d* ⟧
⟹ *subtype P d c*⟨*proof*⟩
lemma *sup-exists*:
⟦ *subtype P a c*; *subtype P b c* ⟧ ⟹ ∃ *T*. *sup P a b* = *OK T*⟨*proof*⟩
lemma *err-semilat-JType-esl*:
wf-prog wf-mb P ⟹ *err-semilat* (*esl P*)⟨*proof*⟩

end

2.8 The JVM Type System as Semilattice

theory *JVM-SemiType* **imports** *SemiType* **begin**

type-synonym $ty_l = ty\ err\ list$
type-synonym $ty_s = ty\ list$
type-synonym $ty_i = ty_s \times ty_l$
type-synonym $ty_i' = ty_i\ option$
type-synonym $ty_m = ty_i'\ list$
type-synonym $ty_P = mname \Rightarrow cname \Rightarrow ty_m$

definition *stk-esl* :: $'c\ prog \Rightarrow nat \Rightarrow ty_s\ esl$
where
stk-esl P mxs ≡ *upto-esl mxs* (*SemiType.esl P*)

definition *loc-sl* :: $'c\ prog \Rightarrow nat \Rightarrow ty_l\ sl$
where
loc-sl P mxl ≡ *Listn.sl mxl* (*Err.sl* (*SemiType.esl P*))

definition *sl* :: $'c\ prog \Rightarrow nat \Rightarrow nat \Rightarrow ty_i'\ err\ sl$
where
sl P mxs mxl ≡
Err.sl(*Opt.esl*(*Product.esl* (*stk-esl P mxs*) (*Err.esl*(*loc-sl P mxl*))))

definition *states* :: $'c\ prog \Rightarrow nat \Rightarrow nat \Rightarrow ty_i'\ err\ set$
where *states P mxs mxl* ≡ *fst*(*sl P mxs mxl*)

definition *le* :: $'c\ prog \Rightarrow nat \Rightarrow nat \Rightarrow ty_i'\ err\ ord$
where
le P mxs mxl ≡ *fst*(*snd*(*sl P mxs mxl*))

definition *sup* :: $'c\ prog \Rightarrow nat \Rightarrow nat \Rightarrow ty_i'\ err\ binop$
where
sup P mxs mxl ≡ *snd*(*snd*(*sl P mxs mxl*))

definition *sup-ty-opt* :: [$'c\ prog$,*ty err*,*ty err*] ⇒ *bool*
(‹- ⊢ - ≤⊤ -› [71,71,71] 70)
where
sup-ty-opt P ≡ *Err.le* (*subtype P*)

definition *sup-state* :: [$'c\ prog$,ty_i,ty_i] ⇒ *bool*
(‹- ⊢ - ≤$_i$ -› [71,71,71] 70)
where
sup-state P ≡ *Product.le* (*Listn.le* (*subtype P*)) (*Listn.le* (*sup-ty-opt P*))

definition *sup-state-opt* :: [$'c\ prog$,ty_i',ty_i'] ⇒ *bool*
(‹- ⊢ - ≤'' -› [71,71,71] 70)
where
sup-state-opt P ≡ *Opt.le* (*sup-state P*)

abbreviation

sup-loc :: ['*c prog*,*ty*$_l$,*ty*$_l$] ⇒ *bool* (‹- ⊢ - [≤$_\top$] -› [71,71,71] 70)
where *P* ⊢ *LT* [≤$_\top$] *LT'* ≡ *list-all2* (*sup-ty-opt P*) *LT LT'*

notation (*ASCII*)
sup-ty-opt (‹- |- - <=*T* -› [71,71,71] 70) **and**
sup-state (‹- |- - <=*i* -› [71,71,71] 70) **and**
sup-state-opt (‹- |- - <=' -› [71,71,71] 70) **and**
sup-loc (‹- |- - [<=*T*] -› [71,71,71] 70)

2.8.1 Unfolding

lemma *JVM-states-unfold*:
states P mxs mxl ≡ *err*(*opt*((*Union* {*nlists n* (*types P*) |*n*. *n* <= *mxs*}) ×
nlists mxl (*err*(*types P*)))⟨*proof*⟩
lemma *JVM-le-unfold*:
le P m n ≡
Err.le(*Opt.le*(*Product.le*(*Listn.le*(*subtype P*))(*Listn.le*(*Err.le*(*subtype P*)))))⟨*proof*⟩
lemma *sl-def2*:
JVM-SemiType.sl P mxs mxl ≡
(*states P mxs mxl*, *JVM-SemiType.le P mxs mxl*, *JVM-SemiType.sup P mxs mxl*)⟨*proof*⟩

lemma *JVM-le-conv*:
le P m n (*OK t1*) (*OK t2*) = *P* ⊢ *t1* ≤' *t2*⟨*proof*⟩
lemma *JVM-le-Err-conv*:
le P m n = *Err.le* (*sup-state-opt P*)⟨*proof*⟩
lemma *err-le-unfold* [*iff*]:
Err.le r (*OK a*) (*OK b*) = *r a b*⟨*proof*⟩

2.8.2 Semilattice

lemma *order-sup-state-opt'* [*intro*, *simp*]:
wf-prog wf-mb P ⟹
order (*sup-state-opt P*) (*opt* ((⋃ {*nlists n* (*types P*) |*n*. *n* ≤ *mxs*}) × *nlists* (*Suc* (*length Ts* +
mxl$_0$)) (*err* (*types P*))))⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*pro*

2.9 Effect of Instructions on the State Type

theory *Effect*
imports *JVM-SemiType ../JVM/JVMExceptions*
begin

— FIXME
locale *prog* =
fixes *P* :: *'a prog*

locale *jvm-method* = *prog* +
fixes *mxs* :: *nat*
fixes *mxl*$_0$ :: *nat*
fixes *Ts* :: *ty list*
fixes *T*$_r$ :: *ty*
fixes *is* :: *instr list*
fixes *xt* :: *ex-table*

fixes *mxl* :: *nat*

defines *mxl-def*: $mxl \equiv 1+size\ Ts+mxl_0$

Program counter of successor instructions:

```
primrec succs :: instr ⇒ ty_i ⇒ pc ⇒ pc list where
  succs (Load idx) τ pc     = [pc+1]
| succs (Store idx) τ pc    = [pc+1]
| succs (Push v) τ pc       = [pc+1]
| succs (Getfield F C) τ pc = [pc+1]
| succs (Getstatic C F D) τ pc = [pc+1]
| succs (Putfield F C) τ pc = [pc+1]
| succs (Putstatic C F D) τ pc = [pc+1]
| succs (New C) τ pc        = [pc+1]
| succs (Checkcast C) τ pc  = [pc+1]
| succs Pop τ pc            = [pc+1]
| succs IAdd τ pc           = [pc+1]
| succs CmpEq τ pc          = [pc+1]
| succs-IfFalse:
    succs (IfFalse b) τ pc    = [pc+1, nat (int pc + b)]
| succs-Goto:
    succs (Goto b) τ pc       = [nat (int pc + b)]
| succs-Return:
    succs Return τ pc         = []
| succs-Invoke:
    succs (Invoke M n) τ pc   = (if (fst τ)!n = NT then [] else [pc+1])
| succs-Invokestatic:
    succs (Invokestatic C M n) τ pc   = [pc+1]
| succs-Throw:
    succs Throw τ pc          = []
```

Effect of instruction on the state type:

```
fun the-class:: ty ⇒ cname where
  the-class (Class C) = C

fun eff_i :: instr × 'm prog × ty_i ⇒ ty_i where
  eff_i-Load:
    eff_i (Load n,  P, (ST, LT))          = (ok-val (LT ! n) # ST, LT)
| eff_i-Store:
    eff_i (Store n, P, (T#ST, LT))        = (ST, LT[n:= OK T])
| eff_i-Push:
    eff_i (Push v, P, (ST, LT))           = (the (typeof v) # ST, LT)
| eff_i-Getfield:
    eff_i (Getfield F C, P, (T#ST, LT))   = (snd (snd (field P C F)) # ST, LT)
| eff_i-Getstatic:
    eff_i (Getstatic C F D, P, (ST, LT))  = (snd (snd (field P C F)) # ST, LT)
| eff_i-Putfield:
    eff_i (Putfield F C, P, (T1#T2#ST, LT)) = (ST,LT)
| eff_i-Putstatic:
    eff_i (Putstatic C F D, P, (T#ST, LT)) = (ST,LT)
| eff_i-New:
    eff_i (New C, P, (ST,LT))             = (Class C # ST, LT)
| eff_i-Checkcast:
    eff_i (Checkcast C, P, (T#ST,LT))     = (Class C # ST,LT)
| eff_i-Pop:
    eff_i (Pop, P, (T#ST,LT))             = (ST,LT)
```

| eff_i-*IAdd*:
  eff_i (*IAdd*, *P*,(T_1#T_2#*ST*,*LT*)) = (*Integer*#*ST*,*LT*)
| eff_i-*CmpEq*:
  eff_i (*CmpEq*, *P*, (T_1#T_2#*ST*,*LT*)) = (*Boolean*#*ST*,*LT*)
| eff_i-*IfFalse*:
  eff_i (*IfFalse b*, *P*, (T_1#*ST*,*LT*)) = (*ST*,*LT*)
| eff_i-*Invoke*:
  eff_i (*Invoke M n*, *P*, (*ST*,*LT*)) =
  (*let C* = *the-class* (*ST*!*n*); (*D*,*b*,*Ts*,T_r,*m*) = *method P C M*
   in (T_r # *drop* (*n*+*1*) *ST*, *LT*))
| eff_i-*Invokestatic*:
  eff_i (*Invokestatic C M n*, *P*, (*ST*,*LT*)) =
  (*let* (*D*,*b*,*Ts*,T_r,*m*) = *method P C M*
   in (T_r # *drop n ST*, *LT*))
| eff_i-*Goto*:
  eff_i (*Goto n*, *P*, *s*) = *s*

fun *is-relevant-class* :: *instr* ⇒ '*m prog* ⇒ *cname* ⇒ *bool* **where**
  rel-Getfield:
    is-relevant-class (*Getfield F D*)
    = (λ*P C*. *P* ⊢ *NullPointer* ⪯* *C* ∨ *P* ⊢ *NoSuchFieldError* ⪯* *C*
        ∨ *P* ⊢ *IncompatibleClassChangeError* ⪯* *C*)
| *rel-Getstatic*:
    is-relevant-class (*Getstatic C F D*)
    = (λ*P C*. *True*)
| *rel-Putfield*:
    is-relevant-class (*Putfield F D*)
    = (λ*P C*. *P* ⊢ *NullPointer* ⪯* *C* ∨ *P* ⊢ *NoSuchFieldError* ⪯* *C*
        ∨ *P* ⊢ *IncompatibleClassChangeError* ⪯* *C*)
| *rel-Putstatic*:
    is-relevant-class (*Putstatic C F D*)
    = (λ*P C*. *True*)
| *rel-Checkcast*:
    is-relevant-class (*Checkcast D*) = (λ*P C*. *P* ⊢ *ClassCast* ⪯* *C*)
| *rel-New*:
    is-relevant-class (*New D*) = (λ*P C*. *True*)
| *rel-Throw*:
    is-relevant-class Throw = (λ*P C*. *True*)
| *rel-Invoke*:
    is-relevant-class (*Invoke M n*) = (λ*P C*. *True*)
| *rel-Invokestatic*:
    is-relevant-class (*Invokestatic C M n*) = (λ*P C*. *True*)
| *rel-default*:
    is-relevant-class i = (λ*P C*. *False*)

definition *is-relevant-entry* :: '*m prog* ⇒ *instr* ⇒ *pc* ⇒ *ex-entry* ⇒ *bool* **where**
  is-relevant-entry P i pc e ⟷ (*let* (*f*,*t*,*C*,*h*,*d*) = *e in is-relevant-class i P C* ∧ *pc* ∈ {*f*..<*t*})

definition *relevant-entries* :: '*m prog* ⇒ *instr* ⇒ *pc* ⇒ *ex-table* ⇒ *ex-table* **where**
  relevant-entries P i pc = *filter* (*is-relevant-entry P i pc*)

definition *xcpt-eff* :: *instr* ⇒ '*m prog* ⇒ *pc* ⇒ ty_i
    ⇒ *ex-table* ⇒ (*pc* × ty_i') *list* **where**
  xcpt-eff i P pc τ *et* = (*let* (*ST*,*LT*) = τ *in*

map (λ(*f*,*t*,*C*,*h*,*d*). (*h*, *Some* (*Class C*#*drop* (*size ST* − *d*) *ST*, *LT*))) (*relevant-entries P i pc et*))

definition *norm-eff* :: *instr* ⇒ '*m prog* ⇒ *nat* ⇒ ty_i ⇒ (*pc* × ty_i') *list* **where**
norm-eff i P pc τ = *map* (λ*pc'*. (*pc'*,*Some* (*eff*$_i$ (*i*,*P*,*τ*)))) (*succs i τ pc*)

definition *eff* :: *instr* ⇒ '*m prog* ⇒ *pc* ⇒ *ex-table* ⇒ ty_i' ⇒ (*pc* × ty_i') *list* **where**
eff i P pc et t = (*case t of*
None ⇒ []
| *Some τ* ⇒ (*norm-eff i P pc τ*) @ (*xcpt-eff i P pc τ et*))

lemma *eff-None*:
eff i P pc xt None = []
⟨*proof*⟩

lemma *eff-Some*:
eff i P pc xt (*Some τ*) = *norm-eff i P pc τ* @ *xcpt-eff i P pc τ xt*
⟨*proof*⟩

Conditions under which eff is applicable:

fun *app*$_i$ :: *instr* × '*m prog* × *pc* × *nat* × *ty* × ty_i ⇒ *bool* **where**
app$_i$-*Load*:
app$_i$ (*Load n*, *P*, *pc*, *mxs*, T_r, (*ST*,*LT*)) =
(*n* < *length LT* ∧ *LT* ! *n* ≠ *Err* ∧ *length ST* < *mxs*)
| *app*$_i$-*Store*:
app$_i$ (*Store n*, *P*, *pc*, *mxs*, T_r, (*T*#*ST*, *LT*)) =
(*n* < *length LT*)
| *app*$_i$-*Push*:
app$_i$ (*Push v*, *P*, *pc*, *mxs*, T_r, (*ST*,*LT*)) =
(*length ST* < *mxs* ∧ *typeof v* ≠ *None*)
| *app*$_i$-*Getfield*:
app$_i$ (*Getfield F C*, *P*, *pc*, *mxs*, T_r, (*T*#*ST*, *LT*)) =
(∃ T_f. *P* ⊢ *C sees F*,*NonStatic*:T_f *in C* ∧ *P* ⊢ *T* ≤ *Class C*)
| *app*$_i$-*Getstatic*:
app$_i$ (*Getstatic C F D*, *P*, *pc*, *mxs*, T_r, (*ST*, *LT*)) =
(*length ST* < *mxs* ∧ (∃ T_f. *P* ⊢ *C sees F*,*Static*:T_f *in D*))
| *app*$_i$-*Putfield*:
app$_i$ (*Putfield F C*, *P*, *pc*, *mxs*, T_r, (T_1#T_2#*ST*, *LT*)) =
(∃ T_f. *P* ⊢ *C sees F*,*NonStatic*:T_f *in C* ∧ *P* ⊢ T_2 ≤ (*Class C*) ∧ *P* ⊢ T_1 ≤ T_f)
| *app*$_i$-*Putstatic*:
app$_i$ (*Putstatic C F D*, *P*, *pc*, *mxs*, T_r, (*T*#*ST*, *LT*)) =
(∃ T_f. *P* ⊢ *C sees F*,*Static*:T_f *in D* ∧ *P* ⊢ *T* ≤ T_f)
| *app*$_i$-*New*:
app$_i$ (*New C*, *P*, *pc*, *mxs*, T_r, (*ST*,*LT*)) =
(*is-class P C* ∧ *length ST* < *mxs*)
| *app*$_i$-*Checkcast*:
app$_i$ (*Checkcast C*, *P*, *pc*, *mxs*, T_r, (*T*#*ST*,*LT*)) =
(*is-class P C* ∧ *is-refT T*)
| *app*$_i$-*Pop*:
app$_i$ (*Pop*, *P*, *pc*, *mxs*, T_r, (*T*#*ST*,*LT*)) =
True
| *app*$_i$-*IAdd*:
app$_i$ (*IAdd*, *P*, *pc*, *mxs*, T_r, (T_1#T_2#*ST*,*LT*)) = (T_1 = T_2 ∧ T_1 = *Integer*)
| *app*$_i$-*CmpEq*:

app_i (*CmpEq*, P, pc, mxs, T_r, (T_1#T_2#ST,LT)) =
($T_1 = T_2 \vee$ *is-refT* $T_1 \wedge$ *is-refT* T_2)
| app_i-*IfFalse*:
app_i (*IfFalse* b, P, pc, mxs, T_r, (*Boolean*#ST,LT)) =
($0 \leq$ *int* $pc + b$)
| app_i-*Goto*:
app_i (*Goto* b, P, pc, mxs, T_r, s) =
($0 \leq$ *int* $pc + b$)
| app_i-*Return*:
app_i (*Return*, P, pc, mxs, T_r, (T#ST,LT)) =
($P \vdash T \leq T_r$)
| app_i-*Throw*:
app_i (*Throw*, P, pc, mxs, T_r, (T#ST,LT)) =
is-refT T
| app_i-*Invoke*:
app_i (*Invoke* M n, P, pc, mxs, T_r, (ST,LT)) =
($n <$ *length* $ST \wedge$
($ST!n \neq NT \longrightarrow$
($\exists$ C D Ts T m. $ST!n =$ *Class* $C \wedge P \vdash C$ *sees* M,*NonStatic*:$Ts \rightarrow T = m$ *in* $D \wedge$
$P \vdash$ *rev* (*take* n ST) [$\leq$] Ts)))
| app_i-*Invokestatic*:
app_i (*Invokestatic* C M n, P, pc, mxs, T_r, (ST,LT)) =
(*length* $ST - n < mxs \wedge n \leq$ *length* $ST \wedge M \neq$ *clinit* $\wedge$
($\exists$ D Ts T m. $P \vdash C$ *sees* M,*Static*:$Ts \rightarrow T = m$ *in* $D \wedge$
$P \vdash$ *rev* (*take* n ST) [$\leq$] Ts))

| app_i-*default*:
app_i (i,P, pc,mxs,T_r,s) = *False*

definition *xcpt-app* :: *instr* $\Rightarrow$ '*m prog* $\Rightarrow$ *pc* $\Rightarrow$ *nat* $\Rightarrow$ *ex-table* $\Rightarrow$ ty_i $\Rightarrow$ *bool* **where**
xcpt-app i P pc mxs xt $\tau \longleftrightarrow$ ($\forall$ (f,t,C,h,d) $\in$ *set* (*relevant-entries P i pc xt*). *is-class P C* $\wedge$ $d \leq$ *size* (*fst* τ) $\wedge$ $d <$ *mxs*)

definition *app* :: *instr* $\Rightarrow$ '*m prog* $\Rightarrow$ *nat* $\Rightarrow$ *ty* $\Rightarrow$ *nat* $\Rightarrow$ *nat* $\Rightarrow$ *ex-table* $\Rightarrow$ ty_i' $\Rightarrow$ *bool* **where**
app i P mxs T_r *pc mpc xt t* = (*case t of None* $\Rightarrow$ *True* | *Some* $\tau \Rightarrow$
app_i (i,P,pc,mxs,T_r,τ) $\wedge$ *xcpt-app i P pc mxs xt* $\tau \wedge$
($\forall$ (pc',τ') $\in$ *set* (*eff i P pc xt t*). $pc' <$ *mpc*))

lemma *app-Some*:
app i P mxs T_r *pc mpc xt* (*Some* τ) =
(app_i (i,P,pc,mxs,T_r,τ) $\wedge$ *xcpt-app i P pc mxs xt* $\tau \wedge$
($\forall$ (pc',s') $\in$ *set* (*eff i P pc xt* (*Some* τ)). $pc' <$ *mpc*))
⟨*proof*⟩

locale *eff* = *jvm-method* +
fixes eff_i **and** app_i **and** *eff* **and** *app*
fixes *norm-eff* **and** *xcpt-app* **and** *xcpt-eff*

fixes *mpc*
defines *mpc* ≡ *size is*

defines eff_i i τ ≡ *Effect*.eff_i (i,P,τ)

notes eff_i*-simps* [*simp*] = *Effect.*eff_i*.simps* [**where** $P = P$, *folded* eff_i*-def*]

defines app_i *i pc* $\tau \equiv$ *Effect.*app_i (*i*, *P*, *pc*, *mxs*, T_r, τ)
notes app_i*-simps* [*simp*] = *Effect.*app_i*.simps* [**where** $P{=}P$ **and** *mxs*=*mxs* **and** $T_r{=}T_r$, *folded* app_i*-def*]

defines *xcpt-eff i pc* $\tau \equiv$ *Effect.xcpt-eff i P pc* τ *xt*
notes *xcpt-eff* = *Effect.xcpt-eff-def* [*of - P - - xt*, *folded xcpt-eff-def*]

defines *norm-eff i pc* $\tau \equiv$ *Effect.norm-eff i P pc* τ
notes *norm-eff* = *Effect.norm-eff-def* [*of - P*, *folded norm-eff-def* eff_i*-def*]

defines *eff i pc* ≡ *Effect.eff i P pc xt*
notes *eff* = *Effect.eff-def* [*of - P - xt*, *folded eff-def norm-eff-def xcpt-eff-def*]

defines *xcpt-app i pc* $\tau \equiv$ *Effect.xcpt-app i P pc mxs xt* τ
notes *xcpt-app* = *Effect.xcpt-app-def* [*of - P - mxs xt*, *folded xcpt-app-def*]

defines *app i pc* ≡ *Effect.app i P mxs* T_r *pc mpc xt*
notes *app* = *Effect.app-def* [*of - P mxs* T_r *- mpc xt*, *folded app-def xcpt-app-def* app_i*-def eff-def*]

lemma *length-cases2*:
assumes $\bigwedge LT.\ P\ ([],LT)$
assumes $\bigwedge l\ ST\ LT.\ P\ (l\#ST,LT)$
shows *P s*
⟨*proof*⟩

lemma *length-cases3*:
assumes $\bigwedge LT.\ P\ ([],LT)$
assumes $\bigwedge l\ LT.\ P\ ([l],LT)$
assumes $\bigwedge l\ ST\ LT.\ P\ (l\#ST,LT)$
shows *P s*⟨*proof*⟩
lemma *length-cases4*:
assumes $\bigwedge LT.\ P\ ([],LT)$
assumes $\bigwedge l\ LT.\ P\ ([l],LT)$
assumes $\bigwedge l\ l'\ LT.\ P\ ([l,l'],LT)$
assumes $\bigwedge l\ l'\ ST\ LT.\ P\ (l\#l'\#ST,LT)$
shows *P s*⟨*proof*⟩

simp rules for *app*

lemma *appNone*[*simp*]: *app i P mxs* T_r *pc mpc et None* = *True*
⟨*proof*⟩

lemma *appLoad*[*simp*]:
app_i (*Load idx*, *P*, T_r, *mxs*, *pc*, *s*) = ($\exists\, ST\ LT.\ s = (ST,LT) \wedge idx < length\ LT \wedge LT!idx \neq Err \wedge length\ ST < mxs$)
⟨*proof*⟩

lemma *appStore*[*simp*]:
app_i (*Store idx*,*P*,*pc*,*mxs*,T_r,*s*) = ($\exists\, ts\ ST\ LT.\ s = (ts\#ST,LT) \wedge idx < length\ LT$)
⟨*proof*⟩

lemma *appPush*[*simp*]:
app_i (*Push* v,P,pc,mxs,T_r,s) =
(∃ *ST LT*. s = (*ST*,*LT*) ∧ *length ST* < *mxs* ∧ *typeof* v ≠ *None*)
⟨*proof*⟩

lemma *appGetField*[*simp*]:
app_i (*Getfield F C*,P,pc,mxs,T_r,s) =
(∃ *oT vT ST LT*. s = (*oT*#*ST*, *LT*) ∧
P ⊢ *C sees F*,*NonStatic*:*vT in C* ∧ P ⊢ *oT* ≤ (*Class C*))
⟨*proof*⟩

lemma *appGetStatic*[*simp*]:
app_i (*Getstatic C F D*,P,pc,mxs,T_r,s) =
(∃ *vT ST LT*. s = (*ST*, *LT*) ∧ *length ST* < *mxs* ∧ P ⊢ *C sees F*,*Static*:*vT in D*)
⟨*proof*⟩

lemma *appPutField*[*simp*]:
app_i (*Putfield F C*,P,pc,mxs,T_r,s) =
(∃ *vT vT*′ *oT ST LT*. s = (*vT*#*oT*#*ST*, *LT*) ∧
P ⊢ *C sees F*,*NonStatic*:*vT*′ *in C* ∧ P ⊢ *oT* ≤ (*Class C*) ∧ P ⊢ *vT* ≤ *vT*′)
⟨*proof*⟩

lemma *appPutstatic*[*simp*]:
app_i (*Putstatic C F D*,P,pc,mxs,T_r,s) =
(∃ *vT vT*′ *ST LT*. s = (*vT*#*ST*, *LT*) ∧
P ⊢ *C sees F*,*Static*:*vT*′ *in D* ∧ P ⊢ *vT* ≤ *vT*′)
⟨*proof*⟩

lemma *appNew*[*simp*]:
app_i (*New C*,P,pc,mxs,T_r,s) =
(∃ *ST LT*. s=(*ST*,*LT*) ∧ *is-class P C* ∧ *length ST* < *mxs*)
⟨*proof*⟩

lemma *appCheckcast*[*simp*]:
app_i (*Checkcast C*,P,pc,mxs,T_r,s) =
(∃ *T ST LT*. s = (*T*#*ST*,*LT*) ∧ *is-class P C* ∧ *is-refT T*)
⟨*proof*⟩

lemma *app*$_i$*Pop*[*simp*]:
app_i (*Pop*,P,pc,mxs,T_r,s) = (∃ *ts ST LT*. s = (*ts*#*ST*,*LT*))
⟨*proof*⟩

lemma *appIAdd*[*simp*]:
app_i (*IAdd*,P,pc,mxs,T_r,s) = (∃ *ST LT*. s = (*Integer*#*Integer*#*ST*,*LT*))⟨*proof*⟩

lemma *appIfFalse* [*simp*]:
app_i (*IfFalse b*,P,pc,mxs,T_r,s) =
(∃ *ST LT*. s = (*Boolean*#*ST*,*LT*) ∧ 0 ≤ *int pc* + *b*)⟨*proof*⟩
lemma *appCmpEq*[*simp*]:
app_i (*CmpEq*,P,pc,mxs,T_r,s) =
(∃ T_1 T_2 *ST LT*. s = (T_1#T_2#*ST*,*LT*) ∧ (¬*is-refT* T_1 ∧ T_2 = T_1 ∨ *is-refT* T_1 ∧ *is-refT* T_2))
⟨*proof*⟩

lemma *appReturn*[*simp*]:
app_i (*Return*,*P*,*pc*,*mxs*,T_r,*s*) = (∃ *T ST LT*. *s* = (*T*#*ST*,*LT*) ∧ *P* ⊢ *T* ≤ T_r)
⟨*proof*⟩

lemma *appThrow*[*simp*]:
app_i (*Throw*,*P*,*pc*,*mxs*,T_r,*s*) = (∃ *T ST LT*. *s*=(*T*#*ST*,*LT*) ∧ *is-refT T*)
⟨*proof*⟩

lemma *effNone*:
(*pc*′, *s*′) ∈ *set* (*eff i P pc et None*) ⟹ *s*′ = *None*
⟨*proof*⟩

some helpers to make the specification directly executable:

lemma *relevant-entries-append* [*simp*]:
relevant-entries P i pc (*xt* @ *xt*′) = *relevant-entries P i pc xt* @ *relevant-entries P i pc xt*′
⟨*proof*⟩

lemma *xcpt-app-append* [*iff*]:
xcpt-app i P pc mxs (*xt*@*xt*′) *τ* = (*xcpt-app i P pc mxs xt τ* ∧ *xcpt-app i P pc mxs xt*′ *τ*)
⟨*proof*⟩

lemma *xcpt-eff-append* [*simp*]:
xcpt-eff i P pc τ (*xt*@*xt*′) = *xcpt-eff i P pc τ xt* @ *xcpt-eff i P pc τ xt*′
⟨*proof*⟩

lemma *app-append* [*simp*]:
app i P pc T mxs mpc (*xt*@*xt*′) *τ* = (*app i P pc T mxs mpc xt τ* ∧ *app i P pc T mxs mpc xt*′ *τ*)
⟨*proof*⟩

end

2.10 Monotonicity of eff and app

theory *EffectMono* **imports** *Effect* **begin**

declare *not-Err-eq* [*iff*]

lemma app_i-*mono*:
assumes *wf*: *wf-prog p P*
assumes *less*: *P* ⊢ *τ* ≤$_i$ *τ*′
shows app_i (*i*,*P*,*mxs*,*mpc*,*rT*,*τ*′) ⟹ app_i (*i*,*P*,*mxs*,*mpc*,*rT*,*τ*)⟨*proof*⟩
lemma *succs-mono*:
assumes *wf*: *wf-prog p P* **and** app_i: app_i (*i*,*P*,*mxs*,*mpc*,*rT*,*τ*′)
shows *P* ⊢ *τ* ≤$_i$ *τ*′ ⟹ *set* (*succs i τ pc*) ⊆ *set* (*succs i τ*′ *pc*)⟨*proof*⟩

lemma *app-mono*:
assumes *wf*: *wf-prog p P*
assumes *less*′: *P* ⊢ *τ* ≤′ *τ*′
shows *app i P m rT pc mpc xt τ*′ ⟹ *app i P m rT pc mpc xt τ*⟨*proof*⟩

lemma *eff*$_i$-*mono*:
assumes *wf*: *wf-prog p P*
assumes *less*: *P* ⊢ *τ* ≤$_i$ *τ*′
assumes app_i: *app i P m rT pc mpc xt* (*Some τ*′)

assumes *succs*: *succs i τ pc* ≠ [] *succs i τ′ pc* ≠ []
shows $P \vdash eff_i\ (i,P,\tau) \leq_i eff_i\ (i,P,\tau')\langle proof\rangle$
end

2.11 The Bytecode Verifier

theory *BVSpec*
imports *Effect*
begin

This theory contains a specification of the BV. The specification describes correct typings of method bodies; it corresponds to type *checking*.

definition
— The method type only contains declared classes:
check-types :: *′m prog* ⇒ *nat* ⇒ *nat* ⇒ ty_i' *err list* ⇒ *bool*
where
check-types P mxs mxl τs ≡ *set τs* ⊆ *states P mxs mxl*

— An instruction is welltyped if it is applicable and its effect
— is compatible with the type at all successor instructions:
definition
wt-instr :: [*′m prog*,*ty*,*nat*,*pc*,*ex-table*,*instr*,*pc*,ty_m] ⇒ *bool*
(‹-,-,-,-,- ⊢ -,- :: -› [*60*,*0*,*0*,*0*,*0*,*0*,*0*,*61*] *60*)
where
P,*T*,*mxs*,*mpc*,*xt* ⊢ *i*,*pc* :: *τs* ≡
app i P mxs T pc mpc xt (*τs*!*pc*) ∧
(∀ (*pc′*,*τ′*) ∈ *set* (*eff i P pc xt* (*τs*!*pc*)). *P* ⊢ *τ′* ≤′ *τs*!*pc′*)

— The type at *pc=0* conforms to the method calling convention:
definition *wt-start* :: [*′m prog*,*cname*,*staticb*,*ty list*,*nat*,ty_m] ⇒ *bool*
where
wt-start P C b Ts mxl_0 *τs* ≡
case b of NonStatic ⇒ *P* ⊢ *Some* ([],*OK* (*Class C*)#*map OK Ts*@*replicate* mxl_0 *Err*) ≤′ *τs*!*0*
| *Static* ⇒ *P* ⊢ *Some* ([],*map OK Ts*@*replicate* mxl_0 *Err*) ≤′ *τs*!*0*

— A method is welltyped if the body is not empty,
— if the method type covers all instructions and mentions
— declared classes only, if the method calling convention is respected, and
— if all instructions are welltyped.
definition *wt-method* :: [*′m prog*,*cname*,*staticb*,*ty list*,*ty*,*nat*,*nat*,*instr list*,
ex-table,ty_m] ⇒ *bool*
where
wt-method P C b Ts T_r *mxs* mxl_0 *is xt τs* ≡ (*b* = *Static* ∨ *b* = *NonStatic*) ∧
0 < *size is* ∧ *size τs* = *size is* ∧
check-types P mxs ((*case b of Static* ⇒ *0* | *NonStatic* ⇒ *1*)+*size Ts*+mxl_0) (*map OK τs*) ∧
wt-start P C b Ts mxl_0 *τs* ∧
(∀ *pc* < *size is*. *P*,T_r,*mxs*,*size is*,*xt* ⊢ *is*!*pc*,*pc* :: *τs*)

— A program is welltyped if it is wellformed and all methods are welltyped
definition *wf-jvm-prog-phi* :: ty_P ⇒ *jvm-prog* ⇒ *bool* (‹*wf′-jvm′-prog*_›)
where
$wf\text{-}jvm\text{-}prog_\Phi$ ≡
wf-prog (λ*P C* (*M*,*b*,*Ts*,T_r,(*mxs*,mxl_0,*is*,*xt*)).

wt-method P C b Ts T_r *mxs* mxl_0 *is xt* (Φ *C M*))

definition *wf-jvm-prog* :: *jvm-prog* ⇒ *bool*
where
wf-jvm-prog P ≡ ∃ Φ. *wf-jvm-prog*$_\Phi$ *P*

lemma *wt-jvm-progD*:
wf-jvm-prog$_\Phi$ *P* ⟹ ∃ *wt*. *wf-prog wt P*⟨*proof*⟩
lemma *wt-jvm-prog-impl-wt-instr*:
assumes *wf*: *wf-jvm-prog*$_\Phi$ *P* **and**
sees: *P* ⊢ *C sees M*,*b*:*Ts* → *T* = (*mxs*,mxl_0,*ins*,*xt*) *in C* **and**
pc: *pc* < *size ins*
shows *P*,*T*,*mxs*,*size ins*,*xt* ⊢ *ins*!*pc*,*pc* :: Φ *C M*⟨*proof*⟩
lemma *wt-jvm-prog-impl-wt-start*:
assumes *wf*: *wf-jvm-prog*$_\Phi$ *P* **and**
sees: *P* ⊢ *C sees M*,*b*:*Ts* → *T* = (*mxs*,mxl_0,*ins*,*xt*) *in C*
shows *0* < *size ins* ∧ *wt-start P C b Ts* mxl_0 (Φ *C M*)⟨*proof*⟩
lemma *wf-jvm-prog-nclinit*:
assumes *wtp*: *wf-jvm-prog*$_\Phi$ *P*
and *meth*: *P* ⊢ *C sees M*, *b* : *Ts*→*T* = (*mxs*, mxl_0, *ins*, *xt*) *in D*
and *wt*: *P*,*T*,*mxs*,*size ins*,*xt* ⊢ *ins*!*pc*,*pc* :: Φ *C M*
and *pc*: *pc* < *length ins* **and** Φ: Φ *C M* ! *pc* = *Some*(*ST*,*LT*)
and *ins*: *ins* ! *pc* = *Invokestatic* C_0 M_0 *n*
shows M_0 ≠ *clinit*
⟨*proof*⟩

end

2.12 The Typing Framework for the JVM

theory *TF-JVM*
imports *Jinja.Typing-Framework-err EffectMono BVSpec*
begin

definition *exec* :: *jvm-prog* ⇒ *nat* ⇒ *ty* ⇒ *ex-table* ⇒ *instr list* ⇒ ty_i' *err step-type*
where
exec G maxs rT et bs ≡
err-step (*size bs*) (λ*pc*. *app* (*bs*!*pc*) *G maxs rT pc* (*size bs*) *et*)
(λ*pc*. *eff* (*bs*!*pc*) *G pc et*)

locale *JVM-sl* =
fixes *P* :: *jvm-prog* **and** *mxs* **and** mxl_0 **and** *n*
fixes *b* **and** *Ts* :: *ty list* **and** *is* **and** *xt* **and** T_r

fixes *mxl* **and** *A* **and** *r* **and** *f* **and** *app* **and** *eff* **and** *step*
defines [*simp*]: *mxl* ≡ (*case b of Static* ⇒ *0* | *NonStatic* ⇒ *1*)+*size Ts*+mxl_0
defines [*simp*]: *A* ≡ *states P mxs mxl*
defines [*simp*]: *r* ≡ *JVM-SemiType.le P mxs mxl*
defines [*simp*]: *f* ≡ *JVM-SemiType.sup P mxs mxl*

defines [*simp*]: *app* ≡ λ*pc*. *Effect.app* (*is*!*pc*) *P mxs* T_r *pc* (*size is*) *xt*
defines [*simp*]: *eff* ≡ λ*pc*. *Effect.eff* (*is*!*pc*) *P pc xt*
defines [*simp*]: *step* ≡ *err-step* (*size is*) *app eff*

defines [*simp*]: $n \equiv size\ is$
assumes *staticb*: $b = Static \vee b = NonStatic$

locale *start-context* = *JVM-sl* +
fixes p **and** C

assumes *wf*: *wf-prog p P*
assumes *C*: *is-class P C*
assumes *Ts*: *set Ts* $\subseteq$ *types P*

fixes *first* :: ty_i' **and** *start*
defines [*simp*]:
first $\equiv$ *Some* ([],(*case b of Static* $\Rightarrow$ [] | *NonStatic* $\Rightarrow$ [*OK* (*Class C*)]) @ *map OK Ts* @ *replicate* mxl_0 *Err*)
defines [*simp*]:
start $\equiv$ (*OK first*) # *replicate* (*size is* $-$ *1*) (*OK None*)
thm *start-context.intro*

lemma *start-context-intro-auxi*:
fixes *P b Ts p C*
assumes $b = Static \vee b = NonStatic$
and *wf-prog p P*
and *is-class P C*
and *set Ts* $\subseteq$ *types P*
shows *start-context P b Ts p C*
⟨*proof*⟩

2.12.1 Connecting JVM and Framework

lemma (**in** *start-context*) *semi*: *semilat* (*A*, *r*, *f*)
⟨*proof*⟩

lemma (**in** *JVM-sl*) *step-def-exec*: *step* $\equiv$ *exec P mxs* T_r *xt is*
⟨*proof*⟩

lemma *special-ex-swap-lemma* [*iff*]:
(*?* X. (*?* n. $X = A\ n$ & $P\ n$) & $Q\ X$) = (*?* n. $Q(A\ n)$ & $P\ n$)
⟨*proof*⟩

lemma *ex-in-list* [*iff*]:
($\exists\, n$. $ST \in nlists\ n\ A \wedge n \leq mxs$) = (*set ST* $\subseteq A \wedge$ *size ST* $\leq mxs$)
⟨*proof*⟩

lemma *singleton-nlists*:
($\exists\, n$. [*Class C*] $\in$ *nlists n* (*types P*) $\wedge\ n \leq mxs$) = (*is-class P C* $\wedge$ $0 < mxs$)
⟨*proof*⟩

lemma *set-drop-subset*:
set xs $\subseteq A \Longrightarrow$ *set* (*drop n xs*) $\subseteq A$
⟨*proof*⟩

lemma *Suc-minus-minus-le*:
$n < mxs \Longrightarrow Suc\ (n - (n - b)) \leq mxs$
⟨*proof*⟩

lemma *in-nlistsE*:
$[\![\ xs \in nlists\ n\ A;\ [\![size\ xs = n;\ set\ xs \subseteq A]\!] \Longrightarrow P\ ]\!] \Longrightarrow P$
⟨*proof*⟩

declare *is-relevant-entry-def* [*simp*]
declare *set-drop-subset* [*simp*]

theorem (**in** *start-context*) *exec-pres-type*:
pres-type step (*size is*) *A*⟨*proof*⟩
declare *is-relevant-entry-def* [*simp del*]
declare *set-drop-subset* [*simp del*]

lemma *lesubstep-type-simple*:
$xs\ [\sqsubseteq_{Product.le\ (=)\ r}]\ ys \Longrightarrow set\ xs\ \{\sqsubseteq_r\}\ set\ ys$⟨*proof*⟩
declare *is-relevant-entry-def* [*simp del*]

lemma *conjI2*: $[\![\ A;\ A \Longrightarrow B\ ]\!] \Longrightarrow A \wedge B$ ⟨*proof*⟩

lemma (**in** *JVM-sl*) *eff-mono*:
assumes *wf*: *wf-prog p P* **and** *pc* < *length is* **and**
lesub: $s \sqsubseteq_{sup\text{-}state\text{-}opt\ P} t$ **and** *app*: *app pc t*
shows *set* (*eff pc s*) $\{\sqsubseteq_{sup\text{-}state\text{-}opt\ P}\}$ *set* (*eff pc t*)⟨*proof*⟩
lemma (**in** *JVM-sl*) *bounded-step*: *bounded step* (*size is*)⟨*proof*⟩
theorem (**in** *JVM-sl*) *step-mono*:
wf-prog wf-mb P ⟹ *mono r step* (*size is*) *A*⟨*proof*⟩

lemma (**in** *start-context*) *first-in-A* [*iff*]: *OK first* ∈ *A*
⟨*proof*⟩

lemma (**in** *JVM-sl*) *wt-method-def2*:
wt-method P C' b Ts T_r *mxs* mxl_0 *is xt* τs =
(*is* ≠ [] ∧
(*b* = *Static* ∨ *b* = *NonStatic*) ∧
size τs = *size is* ∧
OK ' *set* $\tau s \subseteq$ *states P mxs mxl* ∧
wt-start P C' b Ts mxl_0 τs ∧
wt-app-eff (*sup-state-opt P*) *app eff* τs)⟨*proof*⟩

end

2.13 Kildall for the JVM

theory *BVExec*
imports *Jinja.Abstract-BV TF-JVM Jinja.Typing-Framework-2*
begin

definition *kiljvm* :: *jvm-prog* ⇒ *nat* ⇒ *nat* ⇒ *ty* ⇒

instr list ⇒ *ex-table* ⇒ ty_i' *err list* ⇒ ty_i' *err list*
where
kiljvm P mxs mxl T_r *is xt* ≡
kildall (*JVM-SemiType.le P mxs mxl*) (*JVM-SemiType.sup P mxs mxl*)
(*exec P mxs* T_r *xt is*)

definition *wt-kildall* :: *jvm-prog* ⇒ *cname* ⇒ *staticb* ⇒ *ty list* ⇒ *ty* ⇒ *nat* ⇒ *nat* ⇒
instr list ⇒ *ex-table* ⇒ *bool*
where
wt-kildall P C' b Ts T_r *mxs* mxl_0 *is xt* ≡ (*b* = *Static* ∨ *b* = *NonStatic*) ∧
0 < *size is* ∧
(*let first* = *Some* ([],(*case b of Static* ⇒ [] | *NonStatic* ⇒ [*OK* (*Class C'*)])
@(*map OK Ts*)@(*replicate* mxl_0 *Err*));
start = (*OK first*)#(*replicate* (*size is* − *1*) (*OK None*));
result = *kiljvm P mxs*
((*case b of Static* ⇒ *0* | *NonStatic* ⇒ *1*)+*size Ts*+mxl_0)
T_r *is xt start*
in ∀ *n* < *size is*. *result*!*n* ≠ *Err*)

definition $wf\text{-}jvm\text{-}prog_k$ :: *jvm-prog* ⇒ *bool*
where
$wf\text{-}jvm\text{-}prog_k$ *P* ≡
wf-prog (λ*P C'* (*M*,*b*,*Ts*,T_r,(*mxs*,mxl_0,*is*,*xt*)). *wt-kildall P C' b Ts* T_r *mxs* mxl_0 *is xt*) *P*

context *start-context*
begin

lemma *Cons-less-Conss3* [*simp*]:
x#*xs* [⊏$_r$] *y*#*ys* = (*x* ⊏$_r$ *y* ∧ *xs* [⊑$_r$] *ys* ∨ *x* = *y* ∧ *xs* [⊏$_r$] *ys*)
⟨*proof*⟩

lemma *acc-le-listI3* [*intro*!]:
acc r ⟹ *acc* (*Listn.le r*)
⟨*proof*⟩

lemma *wf-jvm*: *wf* {(*ss'*, *ss*). *ss* [⊏$_r$] *ss'*}
⟨*proof*⟩

lemma *iter-properties-bv*[*rule-format*]:
shows ⟦ ∀ *p*∈*w0*. *p* < *n*; *ss0* ∈ *nlists n A*; ∀ *p*<*n*. *p* ∉ *w0* ⟶ *stable r step ss0 p* ⟧ ⟹
iter f step ss0 w0 = (*ss'*,*w'*) ⟶
ss' ∈ *nlists n A* ∧ *stables r step ss'* ∧ *ss0* [⊑$_r$] *ss'* ∧
(∀ *ts*∈*nlists n A*. *ss0* [⊑$_r$] *ts* ∧ *stables r step ts* ⟶ *ss'* [⊑$_r$] *ts*)⟨*proof*⟩

lemma *kildall-properties-bv*:
shows ⟦ *ss0* ∈ *nlists n A* ⟧ ⟹
kildall r f step ss0 ∈ *nlists n A* ∧
stables r step (*kildall r f step ss0*) ∧
ss0 [⊑$_r$] *kildall r f step ss0* ∧
(∀ *ts*∈*nlists n A*. *ss0* [⊑$_r$] *ts* ∧ *stables r step ts* ⟶
kildall r f step ss0 [⊑$_r$] *ts*)⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩

2.14 LBV for the JVM

theory *LBVJVM*
imports *Jinja.Abstract-BV TF-JVM*
begin

type-synonym *prog-cert = cname ⇒ mname ⇒ ty$_i$' err list*

definition *check-cert :: jvm-prog ⇒ nat ⇒ nat ⇒ nat ⇒ ty$_i$' err list ⇒ bool*
where
check-cert P mxs mxl n cert ≡ check-types P mxs mxl cert ∧ size cert = n+1 ∧
(∀ i<n. cert!i ≠ Err) ∧ cert!n = OK None

definition *lbvjvm :: jvm-prog ⇒ nat ⇒ nat ⇒ ty ⇒ ex-table ⇒*
ty$_i$' err list ⇒ instr list ⇒ ty$_i$' err ⇒ ty$_i$' err
where
lbvjvm P mxs maxr T$_r$ et cert bs ≡
wtl-inst-list bs cert (JVM-SemiType.sup P mxs maxr) (JVM-SemiType.le P mxs maxr) Err (OK None) (exec P mxs T$_r$ et bs) 0

definition *wt-lbv :: jvm-prog ⇒ cname ⇒ staticb ⇒ ty list ⇒ ty ⇒ nat ⇒ nat ⇒*
ex-table ⇒ ty$_i$' err list ⇒ instr list ⇒ bool
where
wt-lbv P C b Ts T$_r$ mxs mxl$_0$ et cert ins ≡ (b = Static ∨ b = NonStatic) ∧
check-cert P mxs ((case b of Static ⇒ 0 | NonStatic ⇒ 1)+size Ts+mxl$_0$) (size ins) cert ∧
0 < size ins ∧
(let start = Some ([],(case b of Static ⇒ [] | NonStatic ⇒ [OK (Class C)])
@((map OK Ts))@(replicate mxl$_0$ Err));
result = lbvjvm P mxs ((case b of Static ⇒ 0 | NonStatic ⇒ 1)+size Ts+mxl$_0$) T$_r$ et cert ins (OK start)
in result ≠ Err)

definition *wt-jvm-prog-lbv :: jvm-prog ⇒ prog-cert ⇒ bool*
where
wt-jvm-prog-lbv P cert ≡
wf-prog (λP C (mn,b,Ts,T$_r$,(mxs,mxl$_0$,ins,et)). wt-lbv P C b Ts T$_r$ mxs mxl$_0$ et (cert C mn) ins) P

definition *mk-cert :: jvm-prog ⇒ nat ⇒ ty ⇒ ex-table ⇒ instr list*
⇒ ty$_m$ ⇒ ty$_i$' err list
where
mk-cert P mxs T$_r$ et bs phi ≡ make-cert (exec P mxs T$_r$ et bs) (map OK phi) (OK None)

definition *prg-cert :: jvm-prog ⇒ ty$_P$ ⇒ prog-cert*
where
prg-cert P phi C mn ≡ let (C,b,Ts,T$_r$,(mxs,mxl$_0$,ins,et)) = method P C mn
in mk-cert P mxs T$_r$ et ins (phi C mn)

lemma *check-certD [intro?]*:
check-cert P mxs mxl n cert ⟹ cert-ok cert n Err (OK None) (states P mxs mxl)
⟨*proof*⟩

lemma (**in** *start-context*) *wt-lbv-wt-step*:
assumes *lbv*: *wt-lbv P C b Ts T$_r$ mxs mxl$_0$ xt cert is*

shows $\exists \tau s \in nlists\ (size\ is)\ A.\ wt\text{-}step\ r\ Err\ step\ \tau s \wedge OK\ first \sqsubseteq_r \tau s!0 \langle proof \rangle$

lemma (**in** *start-context*) *wt-lbv-wt-method*:
assumes *lbv*: *wt-lbv P C b Ts* T_r *mxs* mxl_0 *xt cert is*
shows $\exists \tau s.$ *wt-method P C b Ts* T_r *mxs* mxl_0 *is xt* $\tau s \langle proof \rangle$

lemma (**in** *start-context*) *wt-method-wt-lbv*:
assumes *wt*: *wt-method P C b Ts* T_r *mxs* mxl_0 *is xt* τs
defines [*simp*]: *cert* $\equiv$ *mk-cert P mxs* T_r *xt is* τs

shows *wt-lbv P C b Ts* T_r *mxs* mxl_0 *xt cert is*$\langle proof \rangle$

theorem *jvm-lbv-correct*:
wt-jvm-prog-lbv P Cert $\Longrightarrow$ *wf-jvm-prog P*$\langle proof \rangle$
theorem *jvm-lbv-complete*:
assumes *wt*: *wf-jvm-prog*$_\Phi$ *P*
shows *wt-jvm-prog-lbv P* (*prg-cert P* Φ)$\langle proof \rangle$
end

2.15 BV Type Safety Invariant

theory *BVConform*
imports *BVSpec ../JVM/JVMExec ../Common/Conform*
begin

2.15.1 *correct-state* definitions

definition *confT* :: *'c prog* $\Rightarrow$ *heap* $\Rightarrow$ *val* $\Rightarrow$ *ty err* $\Rightarrow$ *bool*
(‹-,- ⊢ - :≤⊤ -› [51,51,51,51] 50)
where
P,h $\vdash$ *v* $:\leq_\top$ *E* $\equiv$ *case E of Err* $\Rightarrow$ *True* | *OK T* $\Rightarrow$ *P,h* $\vdash$ *v* $:\leq$ *T*

notation (*ASCII*)
confT (‹-,- |- - :<=T -› [51,51,51,51] 50)

abbreviation
confTs :: *'c prog* $\Rightarrow$ *heap* $\Rightarrow$ *val list* $\Rightarrow$ *ty*$_l$ $\Rightarrow$ *bool*
(‹-,- ⊢ - [:≤⊤] -› [51,51,51,51] 50) **where**
P,h $\vdash$ *vs* $[:\leq_\top]$ *Ts* $\equiv$ *list-all2* (*confT P h*) *vs Ts*

notation (*ASCII*)
confTs (‹-,- |- - [:<=T] -› [51,51,51,51] 50)

fun *Called-context* :: *jvm-prog* $\Rightarrow$ *cname* $\Rightarrow$ *instr* $\Rightarrow$ *bool* **where**
Called-context P C_0 (*New C'*) = (C_0=*C'*) |
Called-context P C_0 (*Getstatic C F D*) = ((C_0=*D*) $\wedge$ ($\exists$ *t. P* $\vdash$ *C has F,Static:t in D*)) |
Called-context P C_0 (*Putstatic C F D*) = ((C_0=*D*) $\wedge$ ($\exists$ *t. P* $\vdash$ *C has F,Static:t in D*)) |
Called-context P C_0 (*Invokestatic C M n*)
= ($\exists$ *Ts T m D.* (C_0=*D*) $\wedge$ *P* $\vdash$ *C sees M,Static:Ts* $\rightarrow$ *T* = *m in D*) |
Called-context P - - = False

abbreviation *Called-set* :: *instr set* **where**
Called-set $\equiv$ {*i*. $\exists$ *C. i* = *New C*} $\cup$ {*i*. $\exists$ *C M n. i* = *Invokestatic C M n*}

$$\cup \{i.\ \exists\, C\ F\ D.\ i = \mathit{Getstatic}\ C\ F\ D\} \cup \{i.\ \exists\, C\ F\ D.\ i = \mathit{Putstatic}\ C\ F\ D\}$$

lemma *Called-context-Called-set*:
Called-context P D i $\Longrightarrow$ *i* $\in$ *Called-set* ⟨*proof*⟩

fun *valid-ics* :: *jvm-prog* ⇒ *heap* ⇒ *sheap* ⇒ *cname* × *mname* × *pc* × *init-call-status* ⇒ *bool*
(‹-,-,- ⊢$_i$ -› [51,51,51,51] 50) **where**
valid-ics P h sh (*C*,*M*,*pc*,*Calling C′ Cs*)
= (*let ins* = *instrs-of P C M in Called-context P* (*last* (*C′*#*Cs*)) (*ins*!*pc*)
∧ *is-class P C′*) |
valid-ics P h sh (*C*,*M*,*pc*,*Throwing Cs a*)
=(*let ins* = *instrs-of P C M in* ∃ *C1*. *Called-context P C1* (*ins*!*pc*)
∧ (∃ *obj*. *h a* = *Some obj*)) |
valid-ics P h sh (*C*,*M*,*pc*,*Called Cs*)
= (*let ins* = *instrs-of P C M*
in ∃ *C1 sobj*. *Called-context P C1* (*ins*!*pc*) ∧ *sh C1* = *Some sobj*) |
valid-ics P - - - = *True*

definition *conf-f* :: *jvm-prog* ⇒ *heap* ⇒ *sheap* ⇒ ty_i ⇒ *bytecode* ⇒ *frame* ⇒ *bool*
where
conf-f P h sh ≡ λ(*ST*,*LT*) *is* (*stk*,*loc*,*C*,*M*,*pc*,*ics*).
P,*h* ⊢ *stk* [:≤] *ST* ∧ *P*,*h* ⊢ *loc* [:≤$_\top$] *LT* ∧ *pc* < *size is* ∧ *P*,*h*,*sh* ⊢$_i$ (*C*,*M*,*pc*,*ics*)

lemma *conf-f-def2*:
conf-f P h sh (*ST*,*LT*) *is* (*stk*,*loc*,*C*,*M*,*pc*,*ics*) ≡
P,*h* ⊢ *stk* [:≤] *ST* ∧ *P*,*h* ⊢ *loc* [:≤$_\top$] *LT* ∧ *pc* < *size is* ∧ *P*,*h*,*sh* ⊢$_i$ (*C*,*M*,*pc*,*ics*)
⟨*proof*⟩

primrec *conf-fs* :: [*jvm-prog*,*heap*,*sheap*,ty_P,*cname*,*mname*,*nat*,*ty*,*frame list*] ⇒ *bool*
where
conf-fs P h sh Φ *C*$_0$ *M*$_0$ *n*$_0$ *T*$_0$ [] = *True*
| *conf-fs P h sh* Φ *C*$_0$ *M*$_0$ *n*$_0$ *T*$_0$ (*f*#*frs*) =
(*let* (*stk*,*loc*,*C*,*M*,*pc*,*ics*) = *f in*
(∃ *ST LT b Ts T mxs mxl*$_0$ *is xt*.
Φ *C M* ! *pc* = *Some* (*ST*,*LT*) ∧
(*P* ⊢ *C sees M*,*b*:*Ts* → *T* = (*mxs*,*mxl*$_0$,*is*,*xt*) *in C*) ∧
((∃ *D Ts′ T′ m D′*. *M*$_0$ ≠ *clinit* ∧ *ics* = *No-ics* ∧
is!*pc* = *Invoke M*$_0$ *n*$_0$ ∧ *ST*!*n*$_0$ = *Class D* ∧
P ⊢ *D sees M*$_0$,*NonStatic*:*Ts′* → *T′* = *m in D′* ∧ *P* ⊢ *C*$_0$ ≼* *D′* ∧ *P* ⊢ *T*$_0$ ≤ *T′*) ∨
(∃ *D Ts′ T′ m*. *M*$_0$ ≠ *clinit* ∧ *ics* = *No-ics* ∧
is!*pc* = *Invokestatic D M*$_0$ *n*$_0$ ∧
P ⊢ *D sees M*$_0$,*Static*:*Ts′* → *T′* = *m in C*$_0$ ∧ *P* ⊢ *T*$_0$ ≤ *T′*) ∨
(*M*$_0$ = *clinit* ∧ (∃ *Cs*. *ics* = *Called Cs*))) ∧
conf-f P h sh (*ST*, *LT*) *is f* ∧ *conf-fs P h sh* Φ *C M* (*size Ts*) *T frs*))

fun *ics-classes* :: *init-call-status* ⇒ *cname list* **where**
ics-classes (*Calling C Cs*) = *Cs* |
ics-classes (*Throwing Cs a*) = *Cs* |
ics-classes (*Called Cs*) = *Cs* |
ics-classes - = []

fun *frame-clinit-classes* :: *frame* ⇒ *cname list* **where**
frame-clinit-classes (*stk*,*loc*,*C*,*M*,*pc*,*ics*) = (*if M*=*clinit then* [*C*] *else* []) @ *ics-classes ics*

abbreviation *clinit-classes* :: *frame list* ⇒ *cname list* **where**
clinit-classes frs ≡ *concat* (*map frame-clinit-classes frs*)

definition *distinct-clinit* :: *frame list* ⇒ *bool* **where**
distinct-clinit frs ≡ *distinct* (*clinit-classes frs*)

definition *conf-clinit* :: *jvm-prog* ⇒ *sheap* ⇒ *frame list* ⇒ *bool* **where**
conf-clinit P sh frs
≡ *distinct-clinit frs* ∧
(∀ *C* ∈ *set*(*clinit-classes frs*). *is-class P C* ∧ (∃ *sfs*. *sh C* = *Some*(*sfs*, *Processing*)))

definition *correct-state* :: [*jvm-prog*,*ty*$_P$,*jvm-state*] ⇒ *bool* (‹-,- ⊢ - √› [61,0,0] 61)
where
correct-state P Φ ≡ λ(*xp*,*h*,*frs*,*sh*).
case xp of
None ⇒ (*case frs of*
[] ⇒ *True*
| (*f*#*fs*) ⇒ *P*⊢ *h*√ ∧ *P*,*h*⊢$_s$ *sh*√ ∧ *conf-clinit P sh frs* ∧
(*let* (*stk*,*loc*,*C*,*M*,*pc*,*ics*) = *f*
in ∃ *b Ts T mxs mxl*$_0$ *is xt* τ.
(*P* ⊢ *C sees M*,*b*:*Ts*→*T* = (*mxs*,*mxl*$_0$,*is*,*xt*) *in C*) ∧
Φ *C M* ! *pc* = *Some* τ ∧
conf-f P h sh τ *is f* ∧ *conf-fs P h sh* Φ *C M* (*size Ts*) *T fs*))
| *Some x* ⇒ *frs* = []

notation
correct-state (‹-,- |− - [*ok*]› [61,0,0] 61)

2.15.2 Values and ⊤

lemma *confT-Err* [*iff*]: *P*,*h* ⊢ *x* :≤$_\top$ *Err*
⟨*proof*⟩

lemma *confT-OK* [*iff*]: *P*,*h* ⊢ *x* :≤$_\top$ *OK T* = (*P*,*h* ⊢ *x* :≤ *T*)
⟨*proof*⟩

lemma *confT-cases*:
P,*h* ⊢ *x* :≤$_\top$ *X* = (*X* = *Err* ∨ (∃ *T*. *X* = *OK T* ∧ *P*,*h* ⊢ *x* :≤ *T*))
⟨*proof*⟩

lemma *confT-hext* [*intro?*, *trans*]:
⟦ *P*,*h* ⊢ *x* :≤$_\top$ *T*; *h* ⊴ *h*′ ⟧ ⟹ *P*,*h*′ ⊢ *x* :≤$_\top$ *T*
⟨*proof*⟩

lemma *confT-widen* [*intro?*, *trans*]:
⟦ *P*,*h* ⊢ *x* :≤$_\top$ *T*; *P* ⊢ *T* ≤$_\top$ *T*′ ⟧ ⟹ *P*,*h* ⊢ *x* :≤$_\top$ *T*′
⟨*proof*⟩

2.15.3 Stack and Registers

lemmas *confTs-Cons1* [*iff*] = *list-all2-Cons1* [*of confT P h*] **for** *P h*

lemma *confTs-confT-sup*:
assumes *confTs*: $P,h \vdash loc\ [:\leq_\top]\ LT$ **and** n: $n < size\ LT$ **and**
LTn: $LT!n = OK\ T$ **and** *subtype*: $P \vdash T \leq T'$
shows $P,h \vdash (loc!n) :\leq T'\langle proof\rangle$
lemma *confTs-hext* [*intro?*]:
$P,h \vdash loc\ [:\leq_\top]\ LT \Longrightarrow h \trianglelefteq h' \Longrightarrow P,h' \vdash loc\ [:\leq_\top]\ LT$
$\langle proof\rangle$

lemma *confTs-widen* [*intro?*, *trans*]:
$P,h \vdash loc\ [:\leq_\top]\ LT \Longrightarrow P \vdash LT\ [\leq_\top]\ LT' \Longrightarrow P,h \vdash loc\ [:\leq_\top]\ LT'$
$\langle proof\rangle$

lemma *confTs-map* [*iff*]:
$\bigwedge vs.\ (P,h \vdash vs\ [:\leq_\top]\ map\ OK\ Ts) = (P,h \vdash vs\ [:\leq]\ Ts)$
$\langle proof\rangle$

lemma *reg-widen-Err* [*iff*]:
$\bigwedge LT.\ (P \vdash replicate\ n\ Err\ [\leq_\top]\ LT) = (LT = replicate\ n\ Err)$
$\langle proof\rangle$

lemma *confTs-Err* [*iff*]:
$P,h \vdash replicate\ n\ v\ [:\leq_\top]\ replicate\ n\ Err$
$\langle proof\rangle$

2.15.4 valid *init-call-status*

lemma *valid-ics-shupd*:
assumes $P,h,sh \vdash_i (C,\ M,\ pc,\ ics)$ **and** $distinct\ (C'\#ics\text{-}classes\ ics)$
shows $P,h,sh(C' \mapsto (sfs,\ i')) \vdash_i (C,\ M,\ pc,\ ics)$
$\langle proof\rangle$

2.15.5 correct-frame

lemma *conf-f-Throwing*:
assumes *conf-f P h sh* $(ST,\ LT)$ *is* $(stk,\ loc,\ C,\ M,\ pc,\ Called\ Cs)$
and *is-class P C'* **and** $h\ xcp = Some\ obj$ **and** $sh\ C' = Some(sfs,Processing)$
shows *conf-f P h sh* $(ST,\ LT)$ *is* $(stk,\ loc,\ C,\ M,\ pc,\ Throwing\ (C'\ \#\ Cs)\ xcp)$
$\langle proof\rangle$

lemma *conf-f-shupd*:
assumes *conf-f P h sh* (ST,LT) *ins f*
and $i = Processing$
$\vee\ (distinct\ (C\#ics\text{-}classes\ (ics\text{-}of\ f)) \wedge (curr\text{-}method\ f = clinit \longrightarrow C \neq curr\text{-}class\ f))$
shows *conf-f P h* $(sh(C \mapsto (sfs,\ i)))$ (ST,LT) *ins f*
$\langle proof\rangle$

lemma *conf-f-shupd'*:
assumes *conf-f P h sh* (ST,LT) *ins f*
and $sh\ C = Some(sfs,i)$
shows *conf-f P h* $(sh(C \mapsto (sfs',\ i)))$ (ST,LT) *ins f*
$\langle proof\rangle$

2.15.6 correct-frames

lemmas [*simp del*] = *fun-upd-apply*

lemma *conf-fs-hext*:
$\bigwedge C\ M\ n\ T_r$.
$[\![\ conf\text{-}fs\ P\ h\ sh\ \Phi\ C\ M\ n\ T_r\ frs;\ h \trianglelefteq h'\ ]\!] \Longrightarrow conf\text{-}fs\ P\ h'\ sh\ \Phi\ C\ M\ n\ T_r\ frs\langle proof\rangle$

lemma *conf-fs-shupd*:
assumes *conf-fs P h sh* Φ C_0 *M n T frs*
and *dist*: *distinct* (*C#clinit-classes frs*)
shows *conf-fs P h* (*sh*($C \mapsto$ (*sfs*, *i*))) Φ C_0 *M n T frs*
⟨*proof*⟩

lemma *conf-fs-shupd'*:
assumes *conf-fs P h sh* Φ C_0 *M n T frs*
and *shC*: *sh C* = *Some*(*sfs*,*i*)
shows *conf-fs P h* (*sh*($C \mapsto$ (*sfs'*, *i*))) Φ C_0 *M n T frs*
⟨*proof*⟩

2.15.7 correctness wrt *clinit* use

lemma *conf-clinit-Cons*:
assumes *conf-clinit P sh* (*f*#*frs*)
shows *conf-clinit P sh frs*
⟨*proof*⟩

lemma *conf-clinit-Cons-Cons*:
conf-clinit P sh (*f'*#*f*#*frs*) $\Longrightarrow$ *conf-clinit P sh* (*f'*#*frs*)
⟨*proof*⟩

lemma *conf-clinit-diff*:
assumes *conf-clinit P sh* ((*stk*,*loc*,*C*,*M*,*pc*,*ics*)#*frs*)
shows *conf-clinit P sh* ((*stk'*,*loc'*,*C*,*M*,*pc'*,*ics*)#*frs*)
⟨*proof*⟩

lemma *conf-clinit-diff'*:
assumes *conf-clinit P sh* ((*stk*,*loc*,*C*,*M*,*pc*,*ics*)#*frs*)
shows *conf-clinit P sh* ((*stk'*,*loc'*,*C*,*M*,*pc'*,*No-ics*)#*frs*)
⟨*proof*⟩

lemma *conf-clinit-Called-Throwing*:
conf-clinit P sh ((*stk'*, *loc'*, *C'*, *clinit*, *pc'*, *ics'*) # (*stk*, *loc*, *C*, *M*, *pc*, *Called Cs*) # *fs*)
$\Longrightarrow$ *conf-clinit P sh* ((*stk*, *loc*, *C*, *M*, *pc*, *Throwing* (*C'* # *Cs*) *xcp*) # *fs*)
⟨*proof*⟩

lemma *conf-clinit-Throwing*:
conf-clinit P sh ((*stk*, *loc*, *C*, *M*, *pc*, *Throwing* (*C'*#*Cs*) *xcp*) # *fs*)
$\Longrightarrow$ *conf-clinit P sh* ((*stk*, *loc*, *C*, *M*, *pc*, *Throwing Cs xcp*) # *fs*)
⟨*proof*⟩

lemma *conf-clinit-Called*:
$[\![$ *conf-clinit P sh* ((*stk*, *loc*, *C*, *M*, *pc*, *Called* (*C'*#*Cs*)) # *frs*);
P ⊢ *C'* *sees clinit*,*Static*: [] → *Void*=(*mxs'*,*mxl'*,*ins'*,*xt'*) *in C'* $]\!]$
$\Longrightarrow$ *conf-clinit P sh* (*create-init-frame P C'* # (*stk*, *loc*, *C*, *M*, *pc*, *Called Cs*) # *frs*)
⟨*proof*⟩

lemma *conf-clinit-Cons-nclinit*:
assumes *conf-clinit P sh frs* **and** *nclinit*: $M \neq clinit$
shows *conf-clinit P sh* ((*stk*, *loc*, *C*, *M*, *pc*, *No-ics*) # *frs*)
⟨*proof*⟩

lemma *conf-clinit-Invoke*:
assumes *conf-clinit P sh* ((*stk*, *loc*, *C*, *M*, *pc*, *ics*) # *frs*) **and** $M' \neq clinit$
shows *conf-clinit P sh* ((*stk'*, *loc'*, *C'*, *M'*, *pc'*, *No-ics*) # (*stk*, *loc*, *C*, *M*, *pc*, *No-ics*) # *frs*)
⟨*proof*⟩

lemma *conf-clinit-nProc-dist*:
assumes *conf-clinit P sh frs*
and $\forall$ *sfs*. *sh C* $\neq$ *Some*(*sfs*,*Processing*)
shows *distinct* (*C* # *clinit-classes frs*)
⟨*proof*⟩

lemma *conf-clinit-shupd*:
assumes *conf-clinit P sh frs*
and *dist*: *distinct* (*C*#*clinit-classes frs*)
shows *conf-clinit P* (*sh*(*C* ↦ (*sfs*, *i*))) *frs*
⟨*proof*⟩

lemma *conf-clinit-shupd'*:
assumes *conf-clinit P sh frs*
and *sh C* = *Some*(*sfs*,*i*)
shows *conf-clinit P* (*sh*(*C* ↦ (*sfs'*, *i*))) *frs*
⟨*proof*⟩

lemma *conf-clinit-shupd-Called*:
assumes *conf-clinit P sh* ((*stk*,*loc*,*C*,*M*,*pc*,*Calling C' Cs*)#*frs*)
and *dist*: *distinct* (*C'*#*clinit-classes* ((*stk*,*loc*,*C*,*M*,*pc*,*Calling C' Cs*)#*frs*))
and *cls*: *is-class P C'*
shows *conf-clinit P* (*sh*(*C'* ↦ (*sfs*, *Processing*))) ((*stk*,*loc*,*C*,*M*,*pc*,*Called* (*C'*#*Cs*))#*frs*)
⟨*proof*⟩

lemma *conf-clinit-shupd-Calling*:
assumes *conf-clinit P sh* ((*stk*,*loc*,*C*,*M*,*pc*,*Calling C' Cs*)#*frs*)
and *dist*: *distinct* (*C'*#*clinit-classes* ((*stk*,*loc*,*C*,*M*,*pc*,*Calling C' Cs*)#*frs*))
and *cls*: *is-class P C'*
shows *conf-clinit P* (*sh*(*C'* ↦ (*sfs*, *Processing*)))
((*stk*,*loc*,*C*,*M*,*pc*,*Calling* (*fst*(*the*(*class P C'*))) (*C'*#*Cs*))#*frs*)
⟨*proof*⟩

2.15.8 correct state

lemma *correct-state-Cons*:
assumes *cr*: *P*,Φ |− (*xp*,*h*,*f*#*frs*,*sh*) [*ok*]
shows *P*,Φ |− (*xp*,*h*,*frs*,*sh*) [*ok*]
⟨*proof*⟩

lemma *correct-state-shupd*:
assumes *cs*: *P*,Φ |− (*xp*,*h*,*frs*,*sh*) [*ok*] **and** *shC*: *sh C* = *Some*(*sfs*,*i*)
and *dist*: *distinct* (*C*#*clinit-classes frs*)

shows $P,\Phi \vdash (xp,h,frs,sh(C \mapsto (sfs,\ i'))) \ [ok]$
⟨*proof*⟩

lemma *correct-state-Throwing-ex*:
assumes *correct*: $P,\Phi \vdash (xp,h,(stk,loc,C,M,pc,ics)\#frs,sh)\surd$
shows $\bigwedge Cs\ a.\ ics = Throwing\ Cs\ a \Longrightarrow \exists\, obj.\ h\ a = Some\ obj$
⟨*proof*⟩

end

2.16 Property preservation under *class-add*

theory *ClassAdd*
imports *BVConform*
begin

lemma *err-mono*: $A \subseteq B \Longrightarrow err\ A \subseteq err\ B$
⟨*proof*⟩

lemma *opt-mono*: $A \subseteq B \Longrightarrow opt\ A \subseteq opt\ B$
⟨*proof*⟩
abbreviation *class-add* :: *jvm-prog* ⇒ *jvm-method cdecl* ⇒ *jvm-prog* **where**
class-add P cd ≡ *cd*#*P*

2.16.1 Fields

lemma *class-add-has-fields*:
assumes *fs*: $P \vdash D\ has\text{-}fields\ FDTs$ **and** *nc*: $\neg is\text{-}class\ P\ C$
shows $class\text{-}add\ P\ (C,\ cdec) \vdash D\ has\text{-}fields\ FDTs$
⟨*proof*⟩

lemma *class-add-has-fields-rev*:
⟦ $class\text{-}add\ P\ (C,\ cdec) \vdash D\ has\text{-}fields\ FDTs$; $\neg P \vdash D \preceq^*\ C$ ⟧
⟹ $P \vdash D\ has\text{-}fields\ FDTs$
⟨*proof*⟩

lemma *class-add-has-field*:
assumes $P \vdash C_0\ has\ F,b{:}T\ in\ D$ **and** $\neg\ is\text{-}class\ P\ C$
shows $class\text{-}add\ P\ (C,\ cdec) \vdash C_0\ has\ F,b{:}T\ in\ D$
⟨*proof*⟩

lemma *class-add-has-field-rev*:
assumes *has*: $class\text{-}add\ P\ (C,\ cdec) \vdash C_0\ has\ F,b{:}T\ in\ D$
and *ncp*: $\bigwedge D'.\ P \vdash C_0 \preceq^*\ D' \Longrightarrow D' \neq C$
shows $P \vdash C_0\ has\ F,b{:}T\ in\ D$
⟨*proof*⟩

lemma *class-add-sees-field*:
assumes $P \vdash C_0\ sees\ F,b{:}T\ in\ D$ **and** $\neg\ is\text{-}class\ P\ C$
shows $class\text{-}add\ P\ (C,\ cdec) \vdash C_0\ sees\ F,b{:}T\ in\ D$
⟨*proof*⟩

lemma *class-add-sees-field-rev*:

assumes *has*: *class-add P* (*C*, *cdec*) ⊢ C_0 *sees F*,*b*:*T in D*
and *ncp*: ⋀*D'*. *P* ⊢ C_0 ≼* *D'* ⟹ *D'* ≠ *C*
shows *P* ⊢ C_0 *sees F*,*b*:*T in D*
⟨*proof*⟩

lemma *class-add-field*:
assumes *fd*: *P* ⊢ C_0 *sees F*,*b*:*T in D* **and** ¬ *is-class P C*
shows *field P* C_0 *F* = *field* (*class-add P* (*C*, *cdec*)) C_0 *F*
⟨*proof*⟩

2.16.2 Methods

lemma *class-add-sees-methods*:
assumes *ms*: *P* ⊢ *D sees-methods Mm* **and** *nc*: ¬*is-class P C*
shows *class-add P* (*C*, *cdec*) ⊢ *D sees-methods Mm*
⟨*proof*⟩

lemma *class-add-sees-methods-rev*:
⟦ *class-add P* (*C*, *cdec*) ⊢ *D sees-methods Mm*;
⋀*D'*. *P* ⊢ *D* ≼* *D'* ⟹ *D'* ≠ *C* ⟧
⟹ *P* ⊢ *D sees-methods Mm*
⟨*proof*⟩

lemma *class-add-sees-methods-Obj*:
assumes *P* ⊢ *Object sees-methods Mm* **and** *nObj*: *C* ≠ *Object*
shows *class-add P* (*C*, *cdec*) ⊢ *Object sees-methods Mm*
⟨*proof*⟩

lemma *class-add-sees-methods-rev-Obj*:
assumes *class-add P* (*C*, *cdec*) ⊢ *Object sees-methods Mm* **and** *nObj*: *C* ≠ *Object*
shows *P* ⊢ *Object sees-methods Mm*
⟨*proof*⟩

lemma *class-add-sees-method*:
assumes *P* ⊢ C_0 *sees* M_0, *b* : *Ts*→*T* = *m in D* **and** ¬ *is-class P C*
shows *class-add P* (*C*, *cdec*) ⊢ C_0 *sees* M_0, *b* : *Ts*→*T* = *m in D*
⟨*proof*⟩

lemma *class-add-method*:
assumes *md*: *P* ⊢ C_0 *sees* M_0, *b* : *Ts*→*T* = *m in D* **and** ¬ *is-class P C*
shows *method P* C_0 M_0 = *method* (*class-add P* (*C*, *cdec*)) C_0 M_0
⟨*proof*⟩

lemma *class-add-sees-method-rev*:
⟦ *class-add P* (*C*, *cdec*) ⊢ C_0 *sees* M_0, *b* : *Ts*→*T* = *m in D*;
¬ *P* ⊢ C_0 ≼* *C* ⟧
⟹ *P* ⊢ C_0 *sees* M_0, *b* : *Ts*→*T* = *m in D*
⟨*proof*⟩

lemma *class-add-sees-method-Obj*:
⟦ *P* ⊢ *Object sees* M_0, *b* : *Ts*→*T* = *m in D*; *C* ≠ *Object* ⟧
⟹ *class-add P* (*C*, *cdec*) ⊢ *Object sees* M_0, *b* : *Ts*→*T* = *m in D*
⟨*proof*⟩

lemma *class-add-sees-method-rev-Obj*:
⟦ *class-add P* (*C*, *cdec*) ⊢ *Object sees* M_0, *b* : *Ts*→*T* = *m in D*; *C* ≠ *Object* ⟧
⟹ *P* ⊢ *Object sees* M_0, *b* : *Ts*→*T* = *m in D*
⟨*proof*⟩

2.16.3 Types and states

lemma *class-add-is-type*:
is-type P T ⟹ *is-type* (*class-add P* (*C*, *cdec*)) *T*
⟨*proof*⟩

lemma *class-add-types*:
types P ⊆ *types* (*class-add P* (*C*, *cdec*))
⟨*proof*⟩

lemma *class-add-states*:
states P mxs mxl ⊆ *states* (*class-add P* (*C*, *cdec*)) *mxs mxl*
⟨*proof*⟩

lemma *class-add-check-types*:
check-types P mxs mxl τs ⟹ *check-types* (*class-add P* (*C*, *cdec*)) *mxs mxl τs*
⟨*proof*⟩

2.16.4 Subclasses and subtypes

lemma *class-add-subcls*:
⟦ *P* ⊢ *D* ≼* *D*′; ¬ *is-class P C* ⟧
⟹ *class-add P* (*C*, *cdec*) ⊢ *D* ≼* *D*′
⟨*proof*⟩

lemma *class-add-subcls-rev*:
⟦ *class-add P* (*C*, *cdec*) ⊢ *D* ≼* *D*′; ¬*P* ⊢ *D* ≼* *C* ⟧
⟹ *P* ⊢ *D* ≼* *D*′
⟨*proof*⟩

lemma *class-add-subtype*:
⟦ *subtype P x y*; ¬ *is-class P C* ⟧
⟹ *subtype* (*class-add P* (*C*, *cdec*)) *x y*
⟨*proof*⟩

lemma *class-add-widens*:
⟦ *P* ⊢ *Ts* [≤] *Ts*′; ¬ *is-class P C* ⟧
⟹ (*class-add P* (*C*, *cdec*)) ⊢ *Ts* [≤] *Ts*′
⟨*proof*⟩

lemma *class-add-sup-ty-opt*:
⟦ *P* ⊢ *l1* ≤⊤ *l2*; ¬ *is-class P C* ⟧
⟹ *class-add P* (*C*, *cdec*) ⊢ *l1* ≤⊤ *l2*
⟨*proof*⟩

lemma *class-add-sup-loc*:
⟦ *P* ⊢ *LT* [≤⊤] *LT*′; ¬ *is-class P C* ⟧
⟹ *class-add P* (*C*, *cdec*) ⊢ *LT* [≤⊤] *LT*′
⟨*proof*⟩

lemma *class-add-sup-state*:
⟦ $P \vdash \tau \leq_i \tau'$; ¬ *is-class* P C ⟧
⟹ *class-add* P (C, *cdec*) $\vdash \tau \leq_i \tau'$
⟨*proof*⟩

lemma *class-add-sup-state-opt*:
⟦ $P \vdash \tau \leq' \tau'$; ¬ *is-class* P C ⟧
⟹ *class-add* P (C, *cdec*) $\vdash \tau \leq' \tau'$
⟨*proof*⟩

2.16.5 Effect

lemma *class-add-is-relevant-class*:
⟦ *is-relevant-class* i P C_0; ¬ *is-class* P C ⟧
⟹ *is-relevant-class* i (*class-add* P (C, *cdec*)) C_0
⟨*proof*⟩

lemma *class-add-is-relevant-class-rev*:
assumes *irc*: *is-relevant-class* i (*class-add* P (C, *cdec*)) C_0
and *ncp*: ⋀*cd* D'. *cd* ∈ *set* P ⟹ ¬$P \vdash$ *fst cd* $\preceq^*$ C
and *wfxp*: *wf-syscls* P
shows *is-relevant-class* i P C_0
⟨*proof*⟩

lemma *class-add-is-relevant-entry*:
⟦ *is-relevant-entry* P i *pc* e; ¬ *is-class* P C ⟧
⟹ *is-relevant-entry* (*class-add* P (C, *cdec*)) i *pc* e
⟨*proof*⟩

lemma *class-add-is-relevant-entry-rev*:
⟦ *is-relevant-entry* (*class-add* P (C, *cdec*)) i *pc* e;
⋀*cd* D'. *cd* ∈ *set* P ⟹ ¬$P \vdash$ *fst cd* $\preceq^*$ C;
wf-syscls P ⟧
⟹ *is-relevant-entry* P i *pc* e
⟨*proof*⟩

lemma *class-add-relevant-entries*:
¬ *is-class* P C
⟹ *set* (*relevant-entries* P i *pc* *xt*) ⊆ *set* (*relevant-entries* (*class-add* P (C, *cdec*)) i *pc* *xt*)
⟨*proof*⟩

lemma *class-add-relevant-entries-eq*:
assumes *wf*: *wf-prog wf-md* P **and** *nclass*: ¬ *is-class* P C
shows *relevant-entries* P i *pc* *xt* = *relevant-entries* (*class-add* P (C, *cdec*)) i *pc* *xt*
⟨*proof*⟩

lemma *class-add-norm-eff-pc*:
assumes *ne*: ∀ (pc',τ') ∈ *set* (*norm-eff* i P *pc* τ). $pc' < mpc$
shows ∀ (pc',τ') ∈ *set* (*norm-eff* i (*class-add* P (C, *cdec*)) *pc* τ). $pc' < mpc$
⟨*proof*⟩

lemma *class-add-norm-eff-sup-state-opt*:
assumes *ne*: ∀ (pc',τ') ∈ *set* (*norm-eff* i P *pc* τ). $P \vdash \tau' \leq' \tau s!pc'$

and *nclass*: $\neg$ *is-class P C* **and** *app*: app_i (*i*, *P*, *pc*, *mxs*, *T*, τ)
shows $\forall (pc',\tau') \in set\ (norm\text{-}eff\ i\ (class\text{-}add\ P\ (C,\ cdec))\ pc\ \tau).\ (class\text{-}add\ P\ (C,\ cdec)) \vdash \tau' \leq' \tau s!pc'$
⟨*proof*⟩

lemma *class-add-xcpt-eff-eq*:
assumes *wf*: *wf-prog wf-md P* **and** *nclass*: $\neg$ *is-class P C*
shows *xcpt-eff i P pc* τ *xt* = *xcpt-eff i* (*class-add P* (*C*, *cdec*)) *pc* τ *xt*
⟨*proof*⟩

lemma *class-add-eff-pc*:
assumes *eff*: $\forall (pc',\tau') \in set\ (eff\ i\ P\ pc\ xt\ (Some\ \tau)).\ pc' < mpc$
and *wf*: *wf-prog wf-md P* **and** *nclass*: $\neg$ *is-class P C*
shows $\forall (pc',\tau') \in set\ (eff\ i\ (class\text{-}add\ P\ (C,\ cdec))\ pc\ xt\ (Some\ \tau)).\ pc' < mpc$
⟨*proof*⟩

lemma *class-add-eff-sup-state-opt*:
assumes *eff*: $\forall (pc',\tau') \in set\ (eff\ i\ P\ pc\ xt\ (Some\ \tau)).\ P \vdash \tau' \leq' \tau s!pc'$
and *wf*: *wf-prog wf-md P***and** *nclass*: $\neg$ *is-class P C*
and *app*: app_i (*i*, *P*, *pc*, *mxs*, *T*, τ)
shows $\forall (pc',\tau') \in set\ (eff\ i\ (class\text{-}add\ P\ (C,\ cdec))\ pc\ xt\ (Some\ \tau)).$
$(class\text{-}add\ P\ (C,\ cdec)) \vdash \tau' \leq' \tau s!pc'$
⟨*proof*⟩

lemma *class-add-*app_i:
assumes app_i (*i*, *P*, *pc*, *mxs*, T_r, *ST*, *LT*) **and** $\neg$ *is-class P C*
shows app_i (*i*, *class-add P* (*C*, *cdec*), *pc*, *mxs*, T_r, *ST*, *LT*)
⟨*proof*⟩

lemma *class-add-xcpt-app*:
assumes *xa*: *xcpt-app i P pc mxs xt* τ
and *wf*: *wf-prog wf-md P* **and** *nclass*: $\neg$ *is-class P C*
shows *xcpt-app i* (*class-add P* (*C*, *cdec*)) *pc mxs xt* τ
⟨*proof*⟩

lemma *class-add-app*:
assumes *app*: *app i P mxs T pc mpc xt t*
and *wf*: *wf-prog wf-md P* **and** *nclass*: $\neg$ *is-class P C*
shows *app i* (*class-add P* (*C*, *cdec*)) *mxs T pc mpc xt t*
⟨*proof*⟩

2.16.6 Well-formedness and well-typedness

lemma *class-add-wf-mdecl*:
$[\![$ *wf-mdecl wf-md P* C_0 *md*;
$\bigwedge C_0$ *md*. *wf-md P* C_0 *md* $\Longrightarrow$ *wf-md* (*class-add P* (*C*, *cdec*)) C_0 *md* $]\!]$
$\Longrightarrow$ *wf-mdecl wf-md* (*class-add P* (*C*, *cdec*)) C_0 *md*
⟨*proof*⟩

lemma *class-add-wf-mdecl'*:
assumes *wfd*: *wf-mdecl wf-md P* C_0 *md*
and *ms*: (C_0,*S*,*fs*,*ms*) $\in$ *set P* **and** *md*: *md* $\in$ *set ms*
and *wf-md'*: $\bigwedge C_0$ *S fs ms m*.$[\![(C_0,S,fs,ms) \in set\ P;\ m \in set\ ms]\!] \Longrightarrow$ *wf-md'* (*class-add P* (*C*, *cdec*)) C_0 *m*

shows *wf-mdecl wf-md′ (class-add P (C, cdec)) C₀ md*
⟨*proof*⟩

lemma *class-add-wf-cdecl*:
assumes *wfcd*: *wf-cdecl wf-md P cd* **and** *cdP*: *cd ∈ set P*
and *ncp*: *¬ P ⊢ fst cd ≼* C* **and** *dist*: *distinct-fst P*
and *wfmd*: *⋀C₀ md. wf-md P C₀ md ⟹ wf-md (class-add P (C, cdec)) C₀ md*
and *nclass*: *¬ is-class P C*
shows *wf-cdecl wf-md (class-add P (C, cdec)) cd*
⟨*proof*⟩

lemma *class-add-wf-cdecl′*:
assumes *wfcd*: *wf-cdecl wf-md P cd* **and** *cdP*: *cd ∈ set P*
and *ncp*: *¬P ⊢ fst cd ≼* C* **and** *dist*: *distinct-fst P*
and *wfmd*: *⋀C₀ S fs ms m.⟦(C₀,S,fs,ms) ∈ set P; m ∈ set ms⟧ ⟹ wf-md′ (class-add P (C, cdec)) C₀ m*
and *nclass*: *¬ is-class P C*
shows *wf-cdecl wf-md′ (class-add P (C, cdec)) cd*
⟨*proof*⟩

lemma *class-add-wt-start*:
⟦ wt-start P C₀ b Ts mxl τs; ¬ is-class P C ⟧
⟹ wt-start (class-add P (C, cdec)) C₀ b Ts mxl τs
⟨*proof*⟩

lemma *class-add-wt-instr*:
assumes *wti*: *P,T,mxs,mpc,xt ⊢ i,pc :: τs*
and *wf*: *wf-prog wf-md P* **and** *nclass*: *¬ is-class P C*
shows *class-add P (C, cdec),T,mxs,mpc,xt ⊢ i,pc :: τs*
⟨*proof*⟩

lemma *class-add-wt-method*:
assumes *wtm*: *wt-method P C₀ b Ts T_r mxs mxl₀ is xt (Φ C₀ M₀)*
and *wf*: *wf-prog wf-md P* **and** *nclass*: *¬ is-class P C*
shows *wt-method (class-add P (C, cdec)) C₀ b Ts T_r mxs mxl₀ is xt (Φ C₀ M₀)*
⟨*proof*⟩

lemma *class-add-wt-method′*:
⟦ (λP C (M,b,Ts,T_r,(mxs,mxl₀,is,xt)). wt-method P C b Ts T_r mxs mxl₀ is xt (Φ C M)) P C₀ md;
wf-prog wf-md P; ¬ is-class P C ⟧
⟹ (λP C (M,b,Ts,T_r,(mxs,mxl₀,is,xt)). wt-method P C b Ts T_r mxs mxl₀ is xt (Φ C M))
(class-add P (C, cdec)) C₀ md
⟨*proof*⟩

2.16.7 *distinct-fst*

lemma *class-add-distinct-fst*:
⟦ distinct-fst P; ¬ is-class P C ⟧
⟹ distinct-fst (class-add P (C, cdec))
⟨*proof*⟩

2.16.8 Conformance

lemma *class-add-conf*:

⟦ P,$h \vdash v :\leq T$; $\neg$ *is-class* P C ⟧
$\Longrightarrow$ *class-add* P (C, *cdec*),$h \vdash v :\leq T$
⟨*proof*⟩

lemma *class-add-oconf*:
fixes *obj*::*obj*
assumes *oc*: P,$h \vdash$ *obj* $\surd$ **and** *ns*: $\neg$ *is-class* P C
and *ncp*: $\bigwedge D'$. $P \vdash$ *fst*(*obj*) $\preceq^*$ $D' \Longrightarrow D' \neq C$
shows (*class-add* P (C, *cdec*)),$h \vdash$ *obj* $\surd$
⟨*proof*⟩

lemma *class-add-soconf*:
assumes *soc*: P,h,$C_0 \vdash_s$ *sfs* $\surd$ **and** *ns*: $\neg$ *is-class* P C
and *ncp*: $\bigwedge D'$. $P \vdash C_0 \preceq^* D' \Longrightarrow D' \neq C$
shows (*class-add* P (C, *cdec*)),h,$C_0 \vdash_s$ *sfs* $\surd$
⟨*proof*⟩

lemma *class-add-hconf*:
assumes $P \vdash h$ $\surd$ **and** $\neg$ *is-class* P C
and $\bigwedge a$ *obj* D'. h a = *Some obj* $\Longrightarrow P \vdash$ *fst*(*obj*) $\preceq^*$ $D' \Longrightarrow D' \neq C$
shows *class-add* P (C, *cdec*) $\vdash h$ $\surd$
⟨*proof*⟩

lemma *class-add-hconf-wf*:
assumes *wf*: *wf-prog wf-md* P **and** $P \vdash h$ $\surd$ **and** $\neg$ *is-class* P C
and $\bigwedge a$ *obj*. h a = *Some obj* $\Longrightarrow$ *fst*(*obj*) $\neq C$
shows *class-add* P (C, *cdec*) $\vdash h$ $\surd$
⟨*proof*⟩

lemma *class-add-shconf*:
assumes P,$h \vdash_s$ *sh* $\surd$ **and** *ns*: $\neg$ *is-class* P C
and $\bigwedge C$ *sobj* D'. *sh* C = *Some sobj* $\Longrightarrow P \vdash C \preceq^* D' \Longrightarrow D' \neq C$
shows *class-add* P (C, *cdec*),$h \vdash_s$ *sh* $\surd$
⟨*proof*⟩

lemma *class-add-shconf-wf*:
assumes *wf*: *wf-prog wf-md* P **and** P,$h \vdash_s$ *sh* $\surd$ **and** $\neg$ *is-class* P C
and $\bigwedge C$ *sobj*. *sh* C = *Some sobj* $\Longrightarrow C \neq C$
shows *class-add* P (C, *cdec*),$h \vdash_s$ *sh* $\surd$
⟨*proof*⟩

end

2.17 Properties and types of the starting program

theory *StartProg*
imports *ClassAdd*
begin

lemmas *wt-defs* = *correct-state-def conf-f-def wt-instr-def eff-def norm-eff-def app-def xcpt-app-def*

declare *wt-defs* [*simp*] — removed from *simp* at the end of file

declare *start-class-def* [*simp*]

2.17.1 Types

abbreviation *start-φ_m* :: *ty_m* **where**
start-φ_m ≡ [*Some*([],[]),*Some*([*Void*],[])]

fun Φ-*start* :: *ty_P* ⇒ *ty_P* **where**
Φ-*start* Φ *C M* = (*if C*=*Start* ∧ (*M*=*start-m* ∨ *M*=*clinit*) *then start-φ_m else* Φ *C M*)

lemma Φ-*start*: ⋀*C*. *C* ≠ *Start* ⟹ Φ-*start* Φ *C* = Φ *C*
Φ-*start* Φ *Start start-m* = *start-φ_m* Φ-*start* Φ *Start clinit* = *start-φ_m*
⟨*proof*⟩

lemma *check-types-φ_m*: *check-types* (*start-prog P C M*) *1 0* (*map OK start-φ_m*)
⟨*proof*⟩

2.17.2 Some simple properties

lemma *preallocated-start-state*: *start-state P* = *σ* ⟹ *preallocated* (*fst*(*snd σ*))
⟨*proof*⟩

lemma *start-prog-Start-super*: *start-prog P C M* ⊢ *Start* ≺^{1} *Object*
⟨*proof*⟩

lemma *start-prog-Start-fields*:
start-prog P C M ⊢ *Start has-fields FDTs* ⟹ *map-of FDTs* (*F*, *Start*) = *None*
⟨*proof*⟩

lemma *start-prog-Start-soconf*:
(*start-prog P C M*),*h*,*Start* ⊢$_s$ *Map.empty* √
⟨*proof*⟩

lemma *start-prog-start-shconf*:
start-prog P C M,*start-heap P* ⊢$_s$ *start-sheap* √⟨*proof*⟩

2.17.3 Well-typed and well-formed

lemma *start-wt-method*:
assumes *P* ⊢ *C sees M*, *Static* : []→*Void* = *m in D* **and** *M* ≠ *clinit* **and** ¬ *is-class P Start*
shows *wt-method* (*start-prog P C M*) *Start Static* [] *Void 1 0* [*Invokestatic C M 0*, *Return*] [] *start-φ_m*
(**is** *wt-method ?P ?C ?b ?Ts ?T_r ?mxs ?mxl_0 ?is ?xt ?τs*)
⟨*proof*⟩

lemma *start-clinit-wt-method*:
assumes *P* ⊢ *C sees M*, *Static* : []→*Void* = *m in D* **and** *M* ≠ *clinit* **and** ¬ *is-class P Start*
shows *wt-method* (*start-prog P C M*) *Start Static* [] *Void 1 0* [*Push Unit*,*Return*] [] *start-φ_m*
(**is** *wt-method ?P ?C ?b ?Ts ?T_r ?mxs ?mxl_0 ?is ?xt ?τs*)
⟨*proof*⟩

lemma *start-class-wf*:
assumes *P* ⊢ *C sees M*, *Static* : []→*Void* = *m in D*
and *M* ≠ *clinit* **and** ¬ *is-class P Start*
and Φ *Start start-m* = *start-φ_m* **and** Φ *Start clinit* = *start-φ_m*
and *is-class P Object*

and $\bigwedge b'\ Ts'\ T'\ m'\ D'.\ P \vdash Object\ sees\ start\text{-}m,\ b' :\ Ts' \to T' = m'\ in\ D'$
$\Longrightarrow b' = Static \wedge Ts' = [] \wedge T' = Void$
and $\bigwedge b'\ Ts'\ T'\ m'\ D'.\ P \vdash Object\ sees\ clinit,\ b' :\ Ts' \to T' = m'\ in\ D'$
$\Longrightarrow b' = Static \wedge Ts' = [] \wedge T' = Void$
shows *wf-cdecl* $(\lambda P\ C\ (M,b,Ts,T_r,(mxs,mxl_0,is,xt)).\ wt\text{-}method\ P\ C\ b\ Ts\ T_r\ mxs\ mxl_0\ is\ xt\ (\Phi\ C\ M))$
$(start\text{-}prog\ P\ C\ M)\ (start\text{-}class\ C\ M)$
⟨*proof*⟩

lemma *start-prog-wf-jvm-prog-phi*:
assumes *wtp*: $wf\text{-}jvm\text{-}prog_\Phi\ P$
and *nstart*: $\neg\ is\text{-}class\ P\ Start$
and *meth*: $P \vdash C\ sees\ M,\ Static :\ [] \to Void = m\ in\ D$ **and** *nclinit*: $M \neq clinit$
and Φ: $\bigwedge C.\ C \neq Start \Longrightarrow \Phi'\ C = \Phi\ C$
and Φ': $\Phi'\ Start\ start\text{-}m = start\text{-}\varphi_m\ \Phi'\ Start\ clinit = start\text{-}\varphi_m$
and *Obj-start-m*: $\bigwedge b'\ Ts'\ T'\ m'\ D'.\ P \vdash Object\ sees\ start\text{-}m,\ b' :\ Ts' \to T' = m'\ in\ D'$
$\Longrightarrow b' = Static \wedge Ts' = [] \wedge T' = Void$
shows $wf\text{-}jvm\text{-}prog_{\Phi'}\ (start\text{-}prog\ P\ C\ M)$
⟨*proof*⟩

lemma *start-prog-wf-jvm-prog*:
assumes *wf*: *wf-jvm-prog P*
and *nstart*: $\neg\ is\text{-}class\ P\ Start$
and *meth*: $P \vdash C\ sees\ M,\ Static :\ [] \to Void = m\ in\ D$ **and** *nclinit*: $M \neq clinit$
and *Obj-start-m*: $\bigwedge b'\ Ts'\ T'\ m'\ D'.\ P \vdash Object\ sees\ start\text{-}m,\ b' :\ Ts' \to T' = m'\ in\ D'$
$\Longrightarrow b' = Static \wedge Ts' = [] \wedge T' = Void$
shows *wf-jvm-prog* $(start\text{-}prog\ P\ C\ M)$
⟨*proof*⟩

2.17.4 Methods and instructions

lemma *start-prog-Start-sees-methods*:
$P \vdash Object\ sees\text{-}methods\ Mm$
$\Longrightarrow start\text{-}prog\ P\ C\ M \vdash$
$Start\ sees\text{-}methods\ Mm\ ++\ (map\text{-}option\ (\lambda m.\ (m,Start)) \circ map\text{-}of\ [start\text{-}method\ C\ M,\ start\text{-}clinit])$
⟨*proof*⟩

lemma *start-prog-Start-sees-start-method*:
$P \vdash Object\ sees\text{-}methods\ Mm$
$\Longrightarrow start\text{-}prog\ P\ C\ M \vdash$
$Start\ sees\ start\text{-}m,\ Static : [] \to Void = (1,\ 0,\ [Invokestatic\ C\ M\ 0, Return],\ [])\ in\ Start$
⟨*proof*⟩

lemma *wf-start-prog-Start-sees-start-method*:
assumes *wf*: *wf-prog wf-md P*
shows $start\text{-}prog\ P\ C\ M \vdash$
$Start\ sees\ start\text{-}m,\ Static : [] \to Void = (1,\ 0,\ [Invokestatic\ C\ M\ 0, Return],\ [])\ in\ Start$
⟨*proof*⟩

lemma *start-prog-start-m-instrs*:
assumes *wf*: *wf-prog wf-md P*
shows $(instrs\text{-}of\ (start\text{-}prog\ P\ C\ M)\ Start\ start\text{-}m) = [Invokestatic\ C\ M\ 0,\ Return]$
⟨*proof*⟩

declare *wt-defs* [*simp del*]

end

2.18 BV Type Safety Proof

theory *BVSpecTypeSafe*
imports *BVConform StartProg*
begin

This theory contains proof that the specification of the bytecode verifier only admits type safe programs.

2.18.1 Preliminaries

Simp and intro setup for the type safety proof:

lemmas *defs1 = correct-state-def conf-f-def wt-instr-def eff-def norm-eff-def app-def xcpt-app-def*

lemmas *widen-rules* [*intro*] = *conf-widen confT-widen confs-widens confTs-widen*

2.18.2 Exception Handling

For the *Invoke* instruction the BV has checked all handlers that guard the current *pc*.

lemma *Invoke-handlers*:
match-ex-table P C pc xt = Some (pc',d') $\Longrightarrow$
$\exists$ *(f,t,D,h,d)* $\in$ *set (relevant-entries P (Invoke n M) pc xt).*
P $\vdash$ *C* $\preceq^*$ *D* $\wedge$ *pc* $\in$ *{f..<t}* $\wedge$ *pc' = h* $\wedge$ *d' = d*
⟨*proof*⟩

For the *Invokestatic* instruction the BV has checked all handlers that guard the current *pc*.

lemma *Invokestatic-handlers*:
match-ex-table P C pc xt = Some (pc',d') $\Longrightarrow$
$\exists$ *(f,t,D,h,d)* $\in$ *set (relevant-entries P (Invokestatic* C_0 *n M) pc xt).*
P $\vdash$ *C* $\preceq^*$ *D* $\wedge$ *pc* $\in$ *{f..<t}* $\wedge$ *pc' = h* $\wedge$ *d' = d*
⟨*proof*⟩

For the instrs in *Called-set* the BV has checked all handlers that guard the current *pc*.

lemma *Called-set-handlers*:
match-ex-table P C pc xt = Some (pc',d') $\Longrightarrow$ *i* $\in$ *Called-set* $\Longrightarrow$
$\exists$ *(f,t,D,h,d)* $\in$ *set (relevant-entries P i pc xt).*
P $\vdash$ *C* $\preceq^*$ *D* $\wedge$ *pc* $\in$ *{f..<t}* $\wedge$ *pc' = h* $\wedge$ *d' = d*
⟨*proof*⟩

We can prove separately that the recursive search for exception handlers (*find-handler*) in the frame stack results in a conforming state (if there was no matching exception handler in the current frame). We require that the exception is a valid heap address, and that the state before the exception occurred conforms.

lemma *uncaught-xcpt-correct*:
assumes *wt*: *wf-jvm-prog*$_\Phi$ *P*

assumes *h*: *h xcp = Some obj*
shows $\bigwedge f$. *P*,$\Phi \vdash$ (*None*, *h*, *f#frs*, *sh*)$\surd$
$\Longrightarrow$ *curr-method f* $\neq$ *clinit* $\Longrightarrow$ *P*,$\Phi \vdash$ *find-handler P xcp h frs sh* $\surd$
(**is** $\bigwedge f$. *?correct* (*None*, *h*, *f#frs*, *sh*) $\Longrightarrow$ *?prem f* $\Longrightarrow$ *?correct* (*?find frs*))$\langle$*proof*$\rangle$

The requirement of lemma *uncaught-xcpt-correct* (that the exception is a valid reference on the heap) is always met for welltyped instructions and conformant states:

lemma *exec-instr-xcpt-h*:
$[\![$ *fst* (*exec-instr* (*ins*!*pc*) *P h stk vars C M pc ics frs sh*) = *Some xcp*;
P,*T*,*mxs*,*size ins*,*xt* $\vdash$ *ins*!*pc*,*pc* :: Φ *C M*;
P,$\Phi \vdash$ (*None*, *h*, (*stk*,*loc*,*C*,*M*,*pc*,*ics*)#*frs*, *sh*)$\surd$ $]\!]$
$\Longrightarrow \exists$ *obj*. *h xcp = Some obj*
(**is** $[\![$ *?xcpt*; *?wt*; *?correct* $]\!] \Longrightarrow$ *?thesis*)$\langle$*proof*$\rangle$
lemma *exec-step-xcpt-h*:
assumes *xcpt*: *fst* (*exec-step P h stk vars C M pc ics frs sh*) = *Some xcp*
and *ins*: *instrs-of P C M = ins*
and *wti*: *P*,*T*,*mxs*,*size ins*,*xt* $\vdash$ *ins*!*pc*,*pc* :: Φ *C M*
and *correct*: *P*,$\Phi \vdash$ (*None*, *h*, (*stk*,*loc*,*C*,*M*,*pc*,*ics*)#*frs*, *sh*)$\surd$
shows $\exists$ *obj*. *h xcp = Some obj*
$\langle$*proof*$\rangle$

lemma *conf-sys-xcpt*:
$[\![$*preallocated h*; *C* $\in$ *sys-xcpts*$]\!] \Longrightarrow$ *P*,*h* $\vdash$ *Addr* (*addr-of-sys-xcpt C*) $:\leq$ *Class C*
$\langle$*proof*$\rangle$

lemma *match-ex-table-SomeD*:
match-ex-table P C pc xt = Some (*pc*′,*d*′) $\Longrightarrow$
$\exists$ (*f*,*t*,*D*,*h*,*d*) $\in$ *set xt*. *matches-ex-entry P C pc* (*f*,*t*,*D*,*h*,*d*) $\wedge$ *h = pc*′ $\wedge$ *d*=*d*′
$\langle$*proof*$\rangle$

Finally we can state that, whenever an exception occurs, the next state always conforms:

lemma *xcpt-correct*:
fixes σ' :: *jvm-state*
assumes *wtp*: *wf-jvm-prog*$_\Phi$ *P*
assumes *meth*: *P* $\vdash$ *C sees M*,*b*:*Ts*$\rightarrow$*T*=(*mxs*,*mxl*$_0$,*ins*,*xt*) *in C*
assumes *wt*: *P*,*T*,*mxs*,*size ins*,*xt* $\vdash$ *ins*!*pc*,*pc* :: Φ *C M*
assumes *xp*: *fst* (*exec-step P h stk loc C M pc ics frs sh*) = *Some xcp*
assumes *s*′: *Some* σ' = *exec* (*P*, *None*, *h*, (*stk*,*loc*,*C*,*M*,*pc*,*ics*)#*frs*, *sh*)
assumes *correct*: *P*,$\Phi \vdash$ (*None*, *h*, (*stk*,*loc*,*C*,*M*,*pc*,*ics*)#*frs*, *sh*)$\surd$
shows *P*,$\Phi \vdash \sigma'\surd\langle$*proof*$\rangle$

declare *defs1* [*simp*]

2.18.3 Initialization procedure steps

In this section we prove that, for states that result in a step of the initialization procedure rather than an instruction execution, the state after execution of the step still conforms.

lemma *Calling-correct*:
fixes σ' :: *jvm-state*
assumes *wtprog*: *wf-jvm-prog*$_\Phi$ *P*
assumes *mC*: *P* $\vdash$ *C sees M*,*b*:*Ts*$\rightarrow$*T*=(*mxs*,*mxl*$_0$,*ins*,*xt*) *in C*
assumes *s*′: *Some* σ' = *exec* (*P*, *None*, *h*, (*stk*,*loc*,*C*,*M*,*pc*,*ics*)#*frs*, *sh*)
assumes *cf*: *P*,$\Phi \vdash$ (*None*, *h*, (*stk*,*loc*,*C*,*M*,*pc*,*ics*)#*frs*, *sh*)$\surd$

assumes *xc*: *fst* (*exec-step P h stk loc C M pc ics frs sh*) = *None*
assumes *ics*: *ics* = *Calling C′ Cs*

shows *P*,Φ ⊢ *σ′*√
⟨*proof*⟩

lemma *Throwing-correct*:
fixes *σ′* :: *jvm-state*
assumes *wtprog*: *wf-jvm-prog*$_\Phi$ *P*
assumes *mC*: *P* ⊢ *C sees M*,*b*:*Ts*→*T*=(*mxs*,*mxl*$_0$,*ins*,*xt*) *in C*
assumes *s′*: *Some σ′* = *exec* (*P*, *None*, *h*, (*stk*,*loc*,*C*,*M*,*pc*,*ics*)#*frs*, *sh*)
assumes *cf*: *P*,Φ ⊢ (*None*, *h*, (*stk*,*loc*,*C*,*M*,*pc*,*ics*)#*frs*, *sh*)√
assumes *xc*: *fst* (*exec-step P h stk loc C M pc ics frs sh*) = *None*
assumes *ics*: *ics* = *Throwing* (*C′*#*Cs*) *a*

shows *P*,Φ ⊢ *σ′*√
⟨*proof*⟩

lemma *Called-correct*:
fixes *σ′* :: *jvm-state*
assumes *wtprog*: *wf-jvm-prog*$_\Phi$ *P*
assumes *mC*: *P* ⊢ *C sees M*,*b*:*Ts*→*T*=(*mxs*,*mxl*$_0$,*ins*,*xt*) *in C*
assumes *s′*: *Some σ′* = *exec* (*P*, *None*, *h*, (*stk*,*loc*,*C*,*M*,*pc*,*ics*)#*frs*, *sh*)
assumes *cf*: *P*,Φ ⊢ (*None*, *h*, (*stk*,*loc*,*C*,*M*,*pc*,*ics*)#*frs*, *sh*)√
assumes *xc*: *fst* (*exec-step P h stk loc C M pc ics frs sh*) = *None*
assumes *ics*[*simp*]: *ics* = *Called* (*C′*#*Cs*)

shows *P*,Φ ⊢ *σ′*√
⟨*proof*⟩

2.18.4 Single Instructions

In this section we prove for each single (welltyped) instruction that the state after execution of the instruction still conforms. Since we have already handled exceptions above, we can now assume that no exception occurs in this step. For instructions that may call the initialization procedure, we cover the calling and non-calling cases separately.

lemma *Invoke-correct*:
fixes *σ′* :: *jvm-state*
assumes *wtprog*: *wf-jvm-prog*$_\Phi$ *P*
assumes *meth-C*: *P* ⊢ *C sees M*,*b*:*Ts*→*T*=(*mxs*,*mxl*$_0$,*ins*,*xt*) *in C*
assumes *ins*: *ins* ! *pc* = *Invoke M′ n*
assumes *wti*: *P*,*T*,*mxs*,*size ins*,*xt* ⊢ *ins*!*pc*,*pc* :: Φ *C M*
assumes *σ′*: *Some σ′* = *exec* (*P*, *None*, *h*, (*stk*,*loc*,*C*,*M*,*pc*,*ics*)#*frs*, *sh*)
assumes *approx*: *P*,Φ ⊢ (*None*, *h*, (*stk*,*loc*,*C*,*M*,*pc*,*ics*)#*frs*, *sh*)√
assumes *no-xcp*: *fst* (*exec-step P h stk loc C M pc ics frs sh*) = *None*
shows *P*,Φ ⊢ *σ′*√⟨*proof*⟩
lemma *Invokestatic-nInit-correct*:
fixes *σ′* :: *jvm-state*
assumes *wtprog*: *wf-jvm-prog*$_\Phi$ *P*
assumes *meth-C*: *P* ⊢ *C sees M*,*b*:*Ts*→*T*=(*mxs*,*mxl*$_0$,*ins*,*xt*) *in C*
assumes *ins*: *ins* ! *pc* = *Invokestatic D M′ n* **and** *nclinit*: *M′* ≠ *clinit*
assumes *wti*: *P*,*T*,*mxs*,*size ins*,*xt* ⊢ *ins*!*pc*,*pc* :: Φ *C M*
assumes *σ′*: *Some σ′* = *exec* (*P*, *None*, *h*, (*stk*,*loc*,*C*,*M*,*pc*,*ics*)#*frs*, *sh*)

assumes *approx*: $P,\Phi \vdash (None,\ h,\ (stk,loc,C,M,pc,ics)\#frs,\ sh)\surd$
assumes *no-xcp*: *fst* (*exec-step P h stk loc C M pc ics frs sh*) = *None*
assumes *cs*: $ics = Called\ [] \vee (ics = No\text{-}ics \wedge (\exists\, sfs.\ sh\ (fst(method\ P\ D\ M')) = Some(sfs,\ Done)))$
shows $P,\Phi \vdash \sigma'\surd\langle proof\rangle$
lemma *Invokestatic-Init-correct*:
fixes σ' :: *jvm-state*
assumes *wtprog*: $wf\text{-}jvm\text{-}prog_\Phi\ P$
assumes *meth-C*: $P \vdash C\ sees\ M,b:Ts\rightarrow T=(mxs,mxl_0,ins,xt)\ in\ C$
assumes *ins*: $ins\ !\ pc = Invokestatic\ D\ M'\ n$ **and** *nclinit*: $M' \neq clinit$
assumes *wti*: $P,T,mxs,size\ ins,xt \vdash ins!pc,pc :: \Phi\ C\ M$
assumes σ': $Some\ \sigma' = exec\ (P,\ None,\ h,\ (stk,loc,C,M,pc,No\text{-}ics)\#frs,\ sh)$
assumes *approx*: $P,\Phi \vdash (None,\ h,\ (stk,loc,C,M,pc,No\text{-}ics)\#frs,\ sh)\surd$
assumes *no-xcp*: *fst* (*exec-step P h stk loc C M pc No-ics frs sh*) = *None*
assumes *nDone*: $\forall\, sfs.\ sh\ (fst(method\ P\ D\ M')) \neq Some(sfs,\ Done)$
shows $P,\Phi \vdash \sigma'\surd\langle proof\rangle$
declare *list-all2-Cons2* [*iff*]

lemma *Return-correct*:
fixes σ' :: *jvm-state*
assumes *wt-prog*: $wf\text{-}jvm\text{-}prog_\Phi\ P$
assumes *meth*: $P \vdash C\ sees\ M,b:Ts\rightarrow T=(mxs,mxl_0,ins,xt)\ in\ C$
assumes *ins*: $ins\ !\ pc = Return$
assumes *wt*: $P,T,mxs,size\ ins,xt \vdash ins!pc,pc :: \Phi\ C\ M$
assumes s': $Some\ \sigma' = exec\ (P,\ None,\ h,\ (stk,loc,C,M,pc,ics)\#frs,\ sh)$
assumes *correct*: $P,\Phi \vdash (None,\ h,\ (stk,loc,C,M,pc,ics)\#frs,\ sh)\surd$

shows $P,\Phi \vdash \sigma'\surd\langle proof\rangle$
declare *sup-state-opt-any-Some* [*iff*]
declare *not-Err-eq* [*iff*]

lemma *Load-correct*:
assumes *wf-prog wt P* **and**
mC: $P \vdash C\ sees\ M,b:Ts\rightarrow T=(mxs,mxl_0,ins,xt)\ in\ C$ **and**
i: $ins!pc = Load\ idx$ **and**
$P,T,mxs,size\ ins,xt \vdash ins!pc,pc :: \Phi\ C\ M$ **and**
$Some\ \sigma' = exec\ (P,\ None,\ h,\ (stk,loc,C,M,pc,ics)\#frs,\ sh)$ **and**
cf: $P,\Phi \vdash (None,\ h,\ (stk,loc,C,M,pc,ics)\#frs,\ sh)\surd$
shows $P,\Phi \vdash \sigma'\ [ok]\langle proof\rangle$
declare [[*simproc del*: *list-to-set-comprehension*]]

lemma *Store-correct*:
assumes *wf-prog wt P* **and**
mC: $P \vdash C\ sees\ M,b:Ts\rightarrow T=(mxs,mxl_0,ins,xt)\ in\ C$ **and**
i: $ins!pc = Store\ idx$ **and**
$P,T,mxs,size\ ins,xt \vdash ins!pc,pc :: \Phi\ C\ M$ **and**
$Some\ \sigma' = exec\ (P,\ None,\ h,\ (stk,loc,C,M,pc,ics)\#frs,\ sh)$ **and**
cf: $P,\Phi \vdash (None,\ h,\ (stk,loc,C,M,pc,ics)\#frs,\ sh)\surd$
shows $P,\Phi \vdash \sigma'\ [ok]\langle proof\rangle$

lemma *Push-correct*:
assumes *wf-prog wt P* **and**
mC: $P \vdash C\ sees\ M,b:Ts\rightarrow T=(mxs,mxl_0,ins,xt)\ in\ C$ **and**
i: $ins!pc = Push\ v$ **and**
$P,T,mxs,size\ ins,xt \vdash ins!pc,pc :: \Phi\ C\ M$ **and**

Some $\sigma' = exec\ (P,\ None,\ h,\ (stk,loc,C,M,pc,ics)\#frs,\ sh)$ **and**
cf: $P,\Phi \vdash (None,\ h,\ (stk,loc,C,M,pc,ics)\#frs,\ sh)\surd$
shows $P,\Phi \mid\!- \sigma'\ [ok]\langle proof\rangle\langle proof\rangle\langle proof\rangle\langle proof\rangle\langle proof\rangle\langle proof\rangle\langle proof\rangle\langle proof\rangle\langle proof\rangle\langle proof\rangle\langle proof\rangle\langle proof\rangle\langle proof\rangle\langle proof\rangle\langle proof\rangle\langle proof\rangle\langle proof\rangle\langle pr$

2.19 Welltyped Programs produce no Type Errors

theory *BVNoTypeError*
imports *../JVM/JVMDefensive BVSpecTypeSafe*
begin

lemma *has-methodI*:
$P \vdash C$ *sees* $M,b:Ts\to T = m$ *in* $D \Longrightarrow P \vdash C$ *has* M,b
$\langle proof\rangle$

Some simple lemmas about the type testing functions of the defensive JVM:

lemma *typeof-NoneD* [*simp,dest*]: *typeof* $v = Some\ x \Longrightarrow \neg$*is-Addr* v
$\langle proof\rangle$

lemma *is-Ref-def2*:
is-Ref $v = (v = Null \vee (\exists\, a.\ v = Addr\ a))$
$\langle proof\rangle$

lemma [*iff*]: *is-Ref Null* $\langle proof\rangle$

lemma *is-RefI* [*intro, simp*]: $P,h \vdash v :\leq T \Longrightarrow$ *is-refT* $T \Longrightarrow$ *is-Ref* $v\langle proof\rangle$
lemma *is-IntgI* [*intro, simp*]: $P,h \vdash v :\leq Integer \Longrightarrow$ *is-Intg* $v\langle proof\rangle$
lemma *is-BoolI* [*intro, simp*]: $P,h \vdash v :\leq Boolean \Longrightarrow$ *is-Bool* $v\langle proof\rangle$
declare *defs1* [*simp del*]

lemma *wt-jvm-prog-states-NonStatic*:
assumes *wf*: *wf-jvm-prog*$_\Phi$ P
and *mC*: $P \vdash C$ *sees* M,*NonStatic*: $Ts\to T = (mxs,\ mxl,\ ins,\ et)$ *in* C
and Φ: $\Phi\ C\ M\ !\ pc = \tau$ **and** *pc*: $pc <$ *size ins*
shows $OK\ \tau \in states\ P\ mxs\ (1+size\ Ts+mxl)\langle proof\rangle$
lemma *wt-jvm-prog-states-Static*:
assumes *wf*: *wf-jvm-prog*$_\Phi$ P
and *mC*: $P \vdash C$ *sees* M,*Static*: $Ts\to T = (mxs,\ mxl,\ ins,\ et)$ *in* C
and Φ: $\Phi\ C\ M\ !\ pc = \tau$ **and** *pc*: $pc <$ *size ins*
shows $OK\ \tau \in states\ P\ mxs\ (size\ Ts+mxl)\langle proof\rangle$

The main theorem: welltyped programs do not produce type errors if they are started in a conformant state.

theorem *no-type-error*:
fixes σ :: *jvm-state*
assumes *welltyped*: *wf-jvm-prog*$_\Phi$ P **and** *conforms*: $P,\Phi \vdash \sigma\ \surd$
shows *exec-d* $P\ \sigma \neq$ *TypeError*$\langle proof\rangle$

The theorem above tells us that, in welltyped programs, the defensive machine reaches the same result as the aggressive one (after arbitrarily many steps).

theorem *welltyped-aggressive-imp-defensive*:
wf-jvm-prog$_\Phi$ $P \Longrightarrow P,\Phi \vdash \sigma\ \surd \Longrightarrow P \vdash \sigma\ -jvm\to\ \sigma'$
$\Longrightarrow P \vdash (Normal\ \sigma)\ -jvmd\to\ (Normal\ \sigma')\langle proof\rangle$

As corollary we get that the aggressive and the defensive machine are equivalent for well-typed programs (if started in a conformant state or in the canonical start state)

corollary *welltyped-commutes*:
fixes σ :: *jvm-state*
assumes *wf*: *wf-jvm-prog*$_\Phi$ P **and** *conforms*: $P,\Phi \vdash \sigma \surd$
shows $P \vdash (Normal\ \sigma)\ -jvmd\rightarrow\ (Normal\ \sigma') = P \vdash \sigma\ -jvm\rightarrow\ \sigma'$
⟨*proof*⟩

corollary *welltyped-initial-commutes*:
assumes *wf*: *wf-jvm-prog* P
assumes *nstart*: $\neg$ *is-class* P *Start*
assumes *meth*: $P \vdash C$ *sees* M,*Static*:$[]\rightarrow$*Void* = b *in* C
assumes *nclinit*: $M \neq$ *clinit*
assumes *Obj-start-m*:
$(\bigwedge b'\ Ts'\ T'\ m'\ D'.\ P \vdash Object\ sees\ start\text{-}m,\ b' :\ Ts'\rightarrow T' = m'\ in\ D'$
$\Longrightarrow b' = Static \wedge Ts' = [] \wedge T' = Void)$
defines *start*: $\sigma \equiv$ *start-state* P
shows *start-prog* $P\ C\ M \vdash (Normal\ \sigma)\ -jvmd\rightarrow\ (Normal\ \sigma') =$ *start-prog* $P\ C\ M \vdash \sigma\ -jvm\rightarrow\ \sigma'$
⟨*proof*⟩

lemma *not-TypeError-eq* [*iff*]:
$x \neq TypeError = (\exists\, t.\ x = Normal\ t)$
⟨*proof*⟩

locale *cnf* =
fixes P **and** Φ **and** σ
assumes *wf*: *wf-jvm-prog*$_\Phi$ P
assumes *cnf*: *correct-state* $P\ \Phi\ \sigma$

theorem (**in** *cnf*) *no-type-errors*:
$P \vdash (Normal\ \sigma)\ -jvmd\rightarrow\ \sigma' \Longrightarrow \sigma' \neq TypeError$
⟨*proof*⟩

locale *start* =
fixes P **and** C **and** M **and** σ **and** T **and** b **and** P_0
assumes *wf*: *wf-jvm-prog* P
assumes *nstart*: $\neg$ *is-class* P *Start*
assumes *sees*: $P \vdash C$ *sees* M,*Static*:$[]\rightarrow$*Void* = b *in* C
assumes *nclinit*: $M \neq$ *clinit*
assumes *Obj-start-m*: $(\bigwedge b'\ Ts'\ T'\ m'\ D'.\ P \vdash Object\ sees\ start\text{-}m,\ b' :\ Ts'\rightarrow T' = m'\ in\ D'$
$\Longrightarrow b' = Static \wedge Ts' = [] \wedge T' = Void)$
defines $\sigma \equiv Normal\ (start\text{-}state\ P)$
defines [*simp*]: $P_0 \equiv$ *start-prog* $P\ C\ M$

corollary (**in** *start*) *bv-no-type-error*:
shows $P_0 \vdash \sigma\ -jvmd\rightarrow\ \sigma' \Longrightarrow \sigma' \neq TypeError$
⟨*proof*⟩

end

Chapter 3

Compilation

3.1 An Intermediate Language

theory *J1* **imports** *../J/BigStep* **begin**

type-synonym $expr_1 = nat\ exp$
type-synonym $J_1\text{-}prog = expr_1\ prog$
type-synonym $state_1 = heap \times (val\ list) \times sheap$

definition $hp_1 :: state_1 \Rightarrow heap$
where
$hp_1 \equiv fst$
definition $lcl_1 :: state_1 \Rightarrow val\ list$
where
$lcl_1 \equiv fst \circ snd$
definition $shp_1 :: state_1 \Rightarrow sheap$
where
$shp_1 \equiv snd \circ snd$

primrec
max-vars :: *'a exp* ⇒ *nat*
and *max-varss* :: *'a exp list* ⇒ *nat*
where
max-vars(*new C*) = *0*
| *max-vars*(*Cast C e*) = *max-vars e*
| *max-vars*(*Val v*) = *0*
| *max-vars*(e_1 «*bop*» e_2) = *max* (*max-vars* e_1) (*max-vars* e_2)
| *max-vars*(*Var V*) = *0*
| *max-vars*(*V*:=*e*) = *max-vars e*
| *max-vars*(*e*·*F*{*D*}) = *max-vars e*
| *max-vars*($C\cdot_s F\{D\}$) = *0*
| *max-vars*(*FAss* e_1 *F D* e_2) = *max* (*max-vars* e_1) (*max-vars* e_2)
| *max-vars*(*SFAss C F D* e_2) = *max-vars* e_2
| *max-vars*(*e*·*M*(*es*)) = *max* (*max-vars e*) (*max-varss es*)
| *max-vars*($C\cdot_s M(es)$) = *max-varss es*
| *max-vars*({*V*:*T*; *e*}) = *max-vars e* + *1*
| *max-vars*(e_1;;e_2) = *max* (*max-vars* e_1) (*max-vars* e_2)
| *max-vars*(*if* (*e*) e_1 *else* e_2) =
max (*max-vars e*) (*max* (*max-vars* e_1) (*max-vars* e_2))
| *max-vars*(*while* (*b*) *e*) = *max* (*max-vars b*) (*max-vars e*)

| *max-vars*(*throw e*) = *max-vars e*
| *max-vars*(*try* e_1 *catch*(*C V*) e_2) = *max* (*max-vars* e_1) (*max-vars* e_2 + *1*)
| *max-vars*(*INIT C* (*Cs*,*b*) ← *e*) = *max-vars e*
| *max-vars*(*RI*(*C*,*e*);*Cs* ← *e'*) = *max* (*max-vars e*) (*max-vars e'*)

| *max-varss* [] = *0*
| *max-varss* (*e*#*es*) = *max* (*max-vars e*) (*max-varss es*)

inductive
eval$_1$ :: J_1-*prog* ⇒ *expr*$_1$ ⇒ *state*$_1$ ⇒ *expr*$_1$ ⇒ *state*$_1$ ⇒ *bool*
(‹- ⊢$_1$ ((*1*⟨-,/-⟩) ⇒/ (*1*⟨-,/-⟩))› [*51*,*0*,*0*,*0*,*0*] *81*)
and *evals*$_1$ :: J_1-*prog* ⇒ *expr*$_1$ *list* ⇒ *state*$_1$ ⇒ *expr*$_1$ *list* ⇒ *state*$_1$ ⇒ *bool*
(‹- ⊢$_1$ ((*1*⟨-,/-⟩) [⇒]/ (*1*⟨-,/-⟩))› [*51*,*0*,*0*,*0*,*0*] *81*)
for *P* :: J_1-*prog*
where

New$_1$:
⟦ *sh C* = *Some* (*sfs*, *Done*); *new-Addr h* = *Some a*;
P ⊢ *C has-fields FDTs*; *h'* = *h*(*a*↦*blank P C*) ⟧
⟹ *P* ⊢$_1$ ⟨*new C*,(*h*,*l*,*sh*)⟩ ⇒ ⟨*addr a*,(*h'*,*l*,*sh*)⟩
| *NewFail*$_1$:
⟦ *sh C* = *Some* (*sfs*, *Done*); *new-Addr h* = *None* ⟧ ⟹
P ⊢$_1$ ⟨*new C*, (*h*,*l*,*sh*)⟩ ⇒ ⟨*THROW OutOfMemory*,(*h*,*l*,*sh*)⟩
| *NewInit*$_1$:
⟦ ∄*sfs*. *sh C* = *Some* (*sfs*, *Done*); *P* ⊢$_1$ ⟨*INIT C* ([*C*],*False*) ← *unit*,(*h*,*l*,*sh*)⟩ ⇒ ⟨*Val v'*,(*h'*,*l'*,*sh'*)⟩;
new-Addr h' = *Some a*; *P* ⊢ *C has-fields FDTs*; *h''* = *h'*(*a*↦*blank P C*) ⟧
⟹ *P* ⊢$_1$ ⟨*new C*,(*h*,*l*,*sh*)⟩ ⇒ ⟨*addr a*,(*h''*,*l'*,*sh'*)⟩
| *NewInitOOM*$_1$:
⟦ ∄*sfs*. *sh C* = *Some* (*sfs*, *Done*); *P* ⊢$_1$ ⟨*INIT C* ([*C*],*False*) ← *unit*,(*h*,*l*,*sh*)⟩ ⇒ ⟨*Val v'*,(*h'*,*l'*,*sh'*)⟩;
new-Addr h' = *None*; *is-class P C* ⟧
⟹ *P* ⊢$_1$ ⟨*new C*,(*h*,*l*,*sh*)⟩ ⇒ ⟨*THROW OutOfMemory*,(*h'*,*l'*,*sh'*)⟩
| *NewInitThrow*$_1$:
⟦ ∄*sfs*. *sh C* = *Some* (*sfs*, *Done*); *P* ⊢$_1$ ⟨*INIT C* ([*C*],*False*) ← *unit*,(*h*,*l*,*sh*)⟩ ⇒ ⟨*throw a*,*s'*⟩;
is-class P C ⟧
⟹ *P* ⊢$_1$ ⟨*new C*,(*h*,*l*,*sh*)⟩ ⇒ ⟨*throw a*,*s'*⟩

| *Cast*$_1$:
⟦ *P* ⊢$_1$ ⟨*e*,*s*$_0$⟩ ⇒ ⟨*addr a*,(*h*,*l*,*sh*)⟩; *h a* = *Some*(*D*,*fs*); *P* ⊢ *D* ≼* *C* ⟧
⟹ *P* ⊢$_1$ ⟨*Cast C e*,*s*$_0$⟩ ⇒ ⟨*addr a*,(*h*,*l*,*sh*)⟩
| *CastNull*$_1$:
P ⊢$_1$ ⟨*e*,*s*$_0$⟩ ⇒ ⟨*null*,*s*$_1$⟩ ⟹
P ⊢$_1$ ⟨*Cast C e*,*s*$_0$⟩ ⇒ ⟨*null*,*s*$_1$⟩
| *CastFail*$_1$:
⟦ *P* ⊢$_1$ ⟨*e*,*s*$_0$⟩ ⇒ ⟨*addr a*,(*h*,*l*,*sh*)⟩; *h a* = *Some*(*D*,*fs*); ¬ *P* ⊢ *D* ≼* *C* ⟧
⟹ *P* ⊢$_1$ ⟨*Cast C e*,*s*$_0$⟩ ⇒ ⟨*THROW ClassCast*,(*h*,*l*,*sh*)⟩
| *CastThrow*$_1$:
P ⊢$_1$ ⟨*e*,*s*$_0$⟩ ⇒ ⟨*throw e'*,*s*$_1$⟩ ⟹
P ⊢$_1$ ⟨*Cast C e*,*s*$_0$⟩ ⇒ ⟨*throw e'*,*s*$_1$⟩

| *Val*$_1$:
P ⊢$_1$ ⟨*Val v*,*s*⟩ ⇒ ⟨*Val v*,*s*⟩

| *BinOp*$_1$:
⟦ *P* ⊢$_1$ ⟨*e*$_1$,*s*$_0$⟩ ⇒ ⟨*Val v*$_1$,*s*$_1$⟩; *P* ⊢$_1$ ⟨*e*$_2$,*s*$_1$⟩ ⇒ ⟨*Val v*$_2$,*s*$_2$⟩; *binop*(*bop*,*v*$_1$,*v*$_2$) = *Some v* ⟧

$\Longrightarrow P \vdash_1 \langle e_1\ «bop»\ e_2,s_0\rangle \Rightarrow \langle Val\ v,s_2\rangle$
| *BinOpThrow*$_{11}$:
$P \vdash_1 \langle e_1,s_0\rangle \Rightarrow \langle throw\ e,s_1\rangle \Longrightarrow$
$P \vdash_1 \langle e_1\ «bop»\ e_2,\ s_0\rangle \Rightarrow \langle throw\ e,s_1\rangle$
| *BinOpThrow*$_{21}$:
$[\![\ P \vdash_1 \langle e_1,s_0\rangle \Rightarrow \langle Val\ v_1,s_1\rangle;\ P \vdash_1 \langle e_2,s_1\rangle \Rightarrow \langle throw\ e,s_2\rangle\ ]\!]$
$\Longrightarrow P \vdash_1 \langle e_1\ «bop»\ e_2,s_0\rangle \Rightarrow \langle throw\ e,s_2\rangle$

| *Var*$_1$:
$[\![\ ls!i = v;\ i < size\ ls\ ]\!] \Longrightarrow$
$P \vdash_1 \langle Var\ i,(h,ls,sh)\rangle \Rightarrow \langle Val\ v,(h,ls,sh)\rangle$

| *LAss*$_1$:
$[\![\ P \vdash_1 \langle e,s_0\rangle \Rightarrow \langle Val\ v,(h,ls,sh)\rangle;\ i < size\ ls;\ ls' = ls[i := v]\ ]\!]$
$\Longrightarrow P \vdash_1 \langle i:= e,s_0\rangle \Rightarrow \langle unit,(h,ls',sh)\rangle$
| *LAssThrow*$_1$:
$P \vdash_1 \langle e,s_0\rangle \Rightarrow \langle throw\ e',s_1\rangle \Longrightarrow$
$P \vdash_1 \langle i:= e,s_0\rangle \Rightarrow \langle throw\ e',s_1\rangle$

| *FAcc*$_1$:
$[\![\ P \vdash_1 \langle e,s_0\rangle \Rightarrow \langle addr\ a,(h,ls,sh)\rangle;\ h\ a = Some(C,fs);$
$P \vdash C\ has\ F,NonStatic{:}t\ in\ D;$
$fs(F,D) = Some\ v\ ]\!]$
$\Longrightarrow P \vdash_1 \langle e{\cdot}F\{D\},s_0\rangle \Rightarrow \langle Val\ v,(h,ls,sh)\rangle$
| *FAccNull*$_1$:
$P \vdash_1 \langle e,s_0\rangle \Rightarrow \langle null,s_1\rangle \Longrightarrow$
$P \vdash_1 \langle e{\cdot}F\{D\},s_0\rangle \Rightarrow \langle THROW\ NullPointer,s_1\rangle$
| *FAccThrow*$_1$:
$P \vdash_1 \langle e,s_0\rangle \Rightarrow \langle throw\ e',s_1\rangle \Longrightarrow$
$P \vdash_1 \langle e{\cdot}F\{D\},s_0\rangle \Rightarrow \langle throw\ e',s_1\rangle$
| *FAccNone*$_1$:
$[\![\ P \vdash_1 \langle e,s_0\rangle \Rightarrow \langle addr\ a,(h,ls,sh)\rangle;\ h\ a = Some(C,fs);$
$\neg(\exists\ b\ t.\ P \vdash C\ has\ F,b{:}t\ in\ D)\ ]\!]$
$\Longrightarrow P \vdash_1 \langle e{\cdot}F\{D\},s_0\rangle \Rightarrow \langle THROW\ NoSuchFieldError,(h,ls,sh)\rangle$
| *FAccStatic*$_1$:
$[\![\ P \vdash_1 \langle e,s_0\rangle \Rightarrow \langle addr\ a,(h,ls,sh)\rangle;\ h\ a = Some(C,fs);$
$P \vdash C\ has\ F,Static{:}t\ in\ D\ ]\!]$
$\Longrightarrow P \vdash_1 \langle e{\cdot}F\{D\},s_0\rangle \Rightarrow \langle THROW\ IncompatibleClassChangeError,(h,ls,sh)\rangle$

| *SFAcc*$_1$:
$[\![\ P \vdash C\ has\ F,Static{:}t\ in\ D;$
$sh\ D = Some\ (sfs,Done);$
$sfs\ F = Some\ v\ ]\!]$
$\Longrightarrow P \vdash_1 \langle C{\cdot}_s F\{D\},(h,ls,sh)\rangle \Rightarrow \langle Val\ v,(h,ls,sh)\rangle$
| *SFAccInit*$_1$:
$[\![\ P \vdash C\ has\ F,Static{:}t\ in\ D;$
$\nexists sfs.\ sh\ D = Some\ (sfs,Done);\ P \vdash_1 \langle INIT\ D\ ([D],False) \leftarrow unit,(h,ls,sh)\rangle \Rightarrow \langle Val\ v',(h',ls',sh')\rangle;$
$sh'\ D = Some\ (sfs,i);$
$sfs\ F = Some\ v\ ]\!]$
$\Longrightarrow P \vdash_1 \langle C{\cdot}_s F\{D\},(h,ls,sh)\rangle \Rightarrow \langle Val\ v,(h',ls',sh')\rangle$
| *SFAccInitThrow*$_1$:
$[\![\ P \vdash C\ has\ F,Static{:}t\ in\ D;$
$\nexists sfs.\ sh\ D = Some\ (sfs,Done);\ P \vdash_1 \langle INIT\ D\ ([D],False) \leftarrow unit,(h,ls,sh)\rangle \Rightarrow \langle throw\ a,s'\rangle\ ]\!]$
$\Longrightarrow P \vdash_1 \langle C{\cdot}_s F\{D\},(h,ls,sh)\rangle \Rightarrow \langle throw\ a,s'\rangle$

| $SFAccNone_1$:
$[\![\neg(\exists\, b\ t.\ P \vdash C\ has\ F,b{:}t\ in\ D)]\!]$
$\Longrightarrow P \vdash_1 \langle C\cdot_s F\{D\},s\rangle \Rightarrow \langle THROW\ NoSuchFieldError,s\rangle$
| $SFAccNonStatic_1$:
$[\![P \vdash C\ has\ F,NonStatic{:}t\ in\ D]\!]$
$\Longrightarrow P \vdash_1 \langle C\cdot_s F\{D\},s\rangle \Rightarrow \langle THROW\ IncompatibleClassChangeError,s\rangle$

| $FAss_1$:
$[\![P \vdash_1 \langle e_1,s_0\rangle \Rightarrow \langle addr\ a,s_1\rangle;\ P \vdash_1 \langle e_2,s_1\rangle \Rightarrow \langle Val\ v,(h_2,l_2,sh_2)\rangle;$
$h_2\ a = Some(C,fs);\ P \vdash C\ has\ F,NonStatic{:}t\ in\ D;$
$fs' = fs((F,D)\mapsto v);\ h_2' = h_2(a\mapsto(C,fs'))]\!]$
$\Longrightarrow P \vdash_1 \langle e_1\cdot F\{D\}{:=}e_2,s_0\rangle \Rightarrow \langle unit,(h_2',l_2,sh_2)\rangle$
| $FAssNull_1$:
$[\![P \vdash_1 \langle e_1,s_0\rangle \Rightarrow \langle null,s_1\rangle;\ \ P \vdash_1 \langle e_2,s_1\rangle \Rightarrow \langle Val\ v,s_2\rangle]\!] \Longrightarrow$
$P \vdash_1 \langle e_1\cdot F\{D\}{:=}e_2,s_0\rangle \Rightarrow \langle THROW\ NullPointer,s_2\rangle$
| $FAssThrow_{11}$:
$P \vdash_1 \langle e_1,s_0\rangle \Rightarrow \langle throw\ e',s_1\rangle \Longrightarrow$
$P \vdash_1 \langle e_1\cdot F\{D\}{:=}e_2,s_0\rangle \Rightarrow \langle throw\ e',s_1\rangle$
| $FAssThrow_{21}$:
$[\![P \vdash_1 \langle e_1,s_0\rangle \Rightarrow \langle Val\ v,s_1\rangle;\ P \vdash_1 \langle e_2,s_1\rangle \Rightarrow \langle throw\ e',s_2\rangle]\!]$
$\Longrightarrow P \vdash_1 \langle e_1\cdot F\{D\}{:=}e_2,s_0\rangle \Rightarrow \langle throw\ e',s_2\rangle$
| $FAssNone_1$:
$[\![P \vdash_1 \langle e_1,s_0\rangle \Rightarrow \langle addr\ a,s_1\rangle;\ P \vdash_1 \langle e_2,s_1\rangle \Rightarrow \langle Val\ v,(h_2,l_2,sh_2)\rangle;$
$h_2\ a = Some(C,fs);\ \neg(\exists\, b\ t.\ P \vdash C\ has\ F,b{:}t\ in\ D)]\!]$
$\Longrightarrow P \vdash_1 \langle e_1\cdot F\{D\}{:=}e_2,s_0\rangle \Rightarrow \langle THROW\ NoSuchFieldError,(h_2,l_2,sh_2)\rangle$
| $FAssStatic_1$:
$[\![P \vdash_1 \langle e_1,s_0\rangle \Rightarrow \langle addr\ a,s_1\rangle;\ P \vdash_1 \langle e_2,s_1\rangle \Rightarrow \langle Val\ v,(h_2,l_2,sh_2)\rangle;$
$h_2\ a = Some(C,fs);\ P \vdash C\ has\ F,Static{:}t\ in\ D]\!]$
$\Longrightarrow P \vdash_1 \langle e_1\cdot F\{D\}{:=}e_2,s_0\rangle \Rightarrow \langle THROW\ IncompatibleClassChangeError,(h_2,l_2,sh_2)\rangle$

| $SFAss_1$:
$[\![P \vdash_1 \langle e_2,s_0\rangle \Rightarrow \langle Val\ v,(h_1,l_1,sh_1)\rangle;$
$P \vdash C\ has\ F,Static{:}t\ in\ D;$
$sh_1\ D = Some(sfs,Done);\ sfs' = sfs(F\mapsto v);\ sh_1' = sh_1(D\mapsto(sfs',Done))]\!]$
$\Longrightarrow P \vdash_1 \langle C\cdot_s F\{D\}{:=}e_2,s_0\rangle \Rightarrow \langle unit,(h_1,l_1,sh_1')\rangle$
| $SFAssInit_1$:
$[\![P \vdash_1 \langle e_2,s_0\rangle \Rightarrow \langle Val\ v,(h_1,l_1,sh_1)\rangle;$
$P \vdash C\ has\ F,Static{:}t\ in\ D;$
$\nexists\, sfs.\ sh_1\ D = Some(sfs,Done);\ P \vdash_1 \langle INIT\ D\ ([D],False) \leftarrow unit,(h_1,l_1,sh_1)\rangle \Rightarrow \langle Val\ v',(h',l',sh')\rangle;$
$sh'\ D = Some(sfs,i);$
$sfs' = sfs(F\mapsto v);\ sh'' = sh'(D\mapsto(sfs',i))]\!]$
$\Longrightarrow P \vdash_1 \langle C\cdot_s F\{D\}{:=}e_2,s_0\rangle \Rightarrow \langle unit,(h',l',sh'')\rangle$
| $SFAssInitThrow_1$:
$[\![P \vdash_1 \langle e_2,s_0\rangle \Rightarrow \langle Val\ v,(h_1,l_1,sh_1)\rangle;$
$P \vdash C\ has\ F,Static{:}t\ in\ D;$
$\nexists\, sfs.\ sh_1\ D = Some(sfs,Done);\ P \vdash_1 \langle INIT\ D\ ([D],False) \leftarrow unit,(h_1,l_1,sh_1)\rangle \Rightarrow \langle throw\ a,s'\rangle]\!]$
$\Longrightarrow P \vdash_1 \langle C\cdot_s F\{D\}{:=}e_2,s_0\rangle \Rightarrow \langle throw\ a,s'\rangle$
| $SFAssThrow_1$:
$P \vdash_1 \langle e_2,s_0\rangle \Rightarrow \langle throw\ e',s_2\rangle$
$\Longrightarrow P \vdash_1 \langle C\cdot_s F\{D\}{:=}e_2,s_0\rangle \Rightarrow \langle throw\ e',s_2\rangle$
| $SFAssNone_1$:
$[\![P \vdash_1 \langle e_2,s_0\rangle \Rightarrow \langle Val\ v,(h_2,l_2,sh_2)\rangle;$
$\neg(\exists\, b\ t.\ P \vdash C\ has\ F,b{:}t\ in\ D)]\!]$

$\Longrightarrow P \vdash_1 \langle C\cdot_s F\{D\}:=e_2,s_0\rangle \Rightarrow \langle THROW\ NoSuchFieldError,(h_2,l_2,sh_2)\rangle$

| *SFAssNonStatic*$_1$:
$[\![P \vdash_1 \langle e_2,s_0\rangle \Rightarrow \langle Val\ v,(h_2,l_2,sh_2)\rangle;$
$P \vdash C\ has\ F,NonStatic{:}t\ in\ D]\!]$
$\Longrightarrow P \vdash_1 \langle C\cdot_s F\{D\}:=e_2,s_0\rangle \Rightarrow \langle THROW\ IncompatibleClassChangeError,(h_2,l_2,sh_2)\rangle$

| *CallObjThrow*$_1$:
$P \vdash_1 \langle e,s_0\rangle \Rightarrow \langle throw\ e',s_1\rangle \Longrightarrow$
$P \vdash_1 \langle e\cdot M(es),s_0\rangle \Rightarrow \langle throw\ e',s_1\rangle$

| *CallNull*$_1$:
$[\![P \vdash_1 \langle e,s_0\rangle \Rightarrow \langle null,s_1\rangle;\ P \vdash_1 \langle es,s_1\rangle\ [\Rightarrow]\ \langle map\ Val\ vs,s_2\rangle]\!]$
$\Longrightarrow P \vdash_1 \langle e\cdot M(es),s_0\rangle \Rightarrow \langle THROW\ NullPointer,s_2\rangle$

| *Call*$_1$:
$[\![P \vdash_1 \langle e,s_0\rangle \Rightarrow \langle addr\ a,s_1\rangle;\ P \vdash_1 \langle es,s_1\rangle\ [\Rightarrow]\ \langle map\ Val\ vs,(h_2,ls_2,sh_2)\rangle;$
$h_2\ a = Some(C,fs);\ P \vdash C\ sees\ M,NonStatic{:}Ts\to T = body\ in\ D;$
$size\ vs = size\ Ts;\ ls_2' = (Addr\ a)\ \#\ vs\ @\ replicate\ (max\text{-}vars\ body)\ undefined;$
$P \vdash_1 \langle body,(h_2,ls_2',sh_2)\rangle \Rightarrow \langle e',(h_3,ls_3,sh_3)\rangle]\!]$
$\Longrightarrow P \vdash_1 \langle e\cdot M(es),s_0\rangle \Rightarrow \langle e',(h_3,ls_2,sh_3)\rangle$

| *CallParamsThrow*$_1$:
$[\![P \vdash_1 \langle e,s_0\rangle \Rightarrow \langle Val\ v,s_1\rangle;\ P \vdash_1 \langle es,s_1\rangle\ [\Rightarrow]\ \langle es',s_2\rangle;$
$es' = map\ Val\ vs\ @\ throw\ ex\ \#\ es_2]\!]$
$\Longrightarrow P \vdash_1 \langle e\cdot M(es),s_0\rangle \Rightarrow \langle throw\ ex,s_2\rangle$

| *CallNone*$_1$:
$[\![P \vdash_1 \langle e,s_0\rangle \Rightarrow \langle addr\ a,s_1\rangle;\ \ P \vdash_1 \langle ps,s_1\rangle\ [\Rightarrow]\ \langle map\ Val\ vs,(h_2,ls_2,sh_2)\rangle;$
$h_2\ a = Some(C,fs);\ \neg(\exists\ b\ Ts\ T\ body\ D.\ P \vdash C\ sees\ M,b{:}Ts\to T = body\ in\ D)]\!]$
$\Longrightarrow P \vdash_1 \langle e\cdot M(ps),s_0\rangle \Rightarrow \langle THROW\ NoSuchMethodError,(h_2,ls_2,sh_2)\rangle$

| *CallStatic*$_1$:
$[\![P \vdash_1 \langle e,s_0\rangle \Rightarrow \langle addr\ a,s_1\rangle;\ \ P \vdash_1 \langle ps,s_1\rangle\ [\Rightarrow]\ \langle map\ Val\ vs,(h_2,ls_2,sh_2)\rangle;$
$h_2\ a = Some(C,fs);\ P \vdash C\ sees\ M,Static{:}Ts\to T = body\ in\ D]\!]$
$\Longrightarrow P \vdash_1 \langle e\cdot M(ps),s_0\rangle \Rightarrow \langle THROW\ IncompatibleClassChangeError,(h_2,ls_2,sh_2)\rangle$

| *SCallParamsThrow*$_1$:
$[\![P \vdash_1 \langle es,s_0\rangle\ [\Rightarrow]\ \langle es',s_2\rangle;\ es' = map\ Val\ vs\ @\ throw\ ex\ \#\ es_2]\!]$
$\Longrightarrow P \vdash_1 \langle C\cdot_s M(es),s_0\rangle \Rightarrow \langle throw\ ex,s_2\rangle$

| *SCallNone*$_1$:
$[\![P \vdash_1 \langle ps,s_0\rangle\ [\Rightarrow]\ \langle map\ Val\ vs,s_2\rangle;$
$\neg(\exists\ b\ Ts\ T\ body\ D.\ P \vdash C\ sees\ M,b{:}Ts\to T = body\ in\ D)]\!]$
$\Longrightarrow P \vdash_1 \langle C\cdot_s M(ps),s_0\rangle \Rightarrow \langle THROW\ NoSuchMethodError,s_2\rangle$

| *SCallNonStatic*$_1$:
$[\![P \vdash_1 \langle ps,s_0\rangle\ [\Rightarrow]\ \langle map\ Val\ vs,s_2\rangle;$
$P \vdash C\ sees\ M,NonStatic{:}Ts\to T = body\ in\ D]\!]$
$\Longrightarrow P \vdash_1 \langle C\cdot_s M(ps),s_0\rangle \Rightarrow \langle THROW\ IncompatibleClassChangeError,s_2\rangle$

| *SCallInitThrow*$_1$:
$[\![P \vdash_1 \langle ps,s_0\rangle\ [\Rightarrow]\ \langle map\ Val\ vs,(h_1,ls_1,sh_1)\rangle;$
$P \vdash C\ sees\ M,Static{:}Ts\to T = body\ in\ D;$
$\nexists sfs.\ sh_1\ D = Some(sfs,Done);\ M \neq clinit;$
$P \vdash_1 \langle INIT\ D\ ([D],False) \leftarrow unit,(h_1,ls_1,sh_1)\rangle \Rightarrow \langle throw\ a,s'\rangle]\!]$
$\Longrightarrow P \vdash_1 \langle C\cdot_s M(ps),s_0\rangle \Rightarrow \langle throw\ a,s'\rangle$

| *SCallInit*$_1$:
$[\![P \vdash_1 \langle ps,s_0\rangle\ [\Rightarrow]\ \langle map\ Val\ vs,(h_1,ls_1,sh_1)\rangle;$
$P \vdash C\ sees\ M,Static{:}Ts\to T = body\ in\ D;$
$\nexists sfs.\ sh_1\ D = Some(sfs,Done);\ M \neq clinit;$
$P \vdash_1 \langle INIT\ D\ ([D],False) \leftarrow unit,(h_1,ls_1,sh_1)\rangle \Rightarrow \langle Val\ v',(h_2,ls_2,sh_2)\rangle;$
$size\ vs = size\ Ts;\ ls_2' = vs\ @\ replicate\ (max\text{-}vars\ body)\ undefined;$

$P \vdash_1 \langle body,(h_2,ls_2{}',sh_2)\rangle \Rightarrow \langle e',(h_3,ls_3,sh_3)\rangle \rrbracket$
$\Longrightarrow P \vdash_1 \langle C\cdot_s M(ps),s_0\rangle \Rightarrow \langle e',(h_3,ls_2,sh_3)\rangle$
| $SCall_1$:
$\llbracket P \vdash_1 \langle ps,s_0\rangle [\Rightarrow] \langle map\ Val\ vs,(h_2,ls_2,sh_2)\rangle;$
$P \vdash C\ sees\ M,Static{:}Ts\to T = body\ in\ D;$
$sh_2\ D = Some(sfs,Done) \vee (M = clinit \wedge sh_2\ D = \lfloor(sfs,\ Processing)\rfloor);$
$size\ vs = size\ Ts;\ ls_2{}' = vs\ @\ replicate\ (max\text{-}vars\ body)\ undefined;$
$P \vdash_1 \langle body,(h_2,ls_2{}',sh_2)\rangle \Rightarrow \langle e',(h_3,ls_3,sh_3)\rangle \rrbracket$
$\Longrightarrow P \vdash_1 \langle C\cdot_s M(ps),s_0\rangle \Rightarrow \langle e',(h_3,ls_2,sh_3)\rangle$

| $Block_1$:
$P \vdash_1 \langle e,s_0\rangle \Rightarrow \langle e',s_1\rangle \Longrightarrow P \vdash_1 \langle Block\ i\ T\ e,s_0\rangle \Rightarrow \langle e',s_1\rangle$

| Seq_1:
$\llbracket P \vdash_1 \langle e_0,s_0\rangle \Rightarrow \langle Val\ v,s_1\rangle;\ P \vdash_1 \langle e_1,s_1\rangle \Rightarrow \langle e_2,s_2\rangle \rrbracket$
$\Longrightarrow P \vdash_1 \langle e_0;;e_1,s_0\rangle \Rightarrow \langle e_2,s_2\rangle$
| $SeqThrow_1$:
$P \vdash_1 \langle e_0,s_0\rangle \Rightarrow \langle throw\ e,s_1\rangle \Longrightarrow$
$P \vdash_1 \langle e_0;;e_1,s_0\rangle \Rightarrow \langle throw\ e,s_1\rangle$

| $CondT_1$:
$\llbracket P \vdash_1 \langle e,s_0\rangle \Rightarrow \langle true,s_1\rangle;\ P \vdash_1 \langle e_1,s_1\rangle \Rightarrow \langle e',s_2\rangle \rrbracket$
$\Longrightarrow P \vdash_1 \langle if\ (e)\ e_1\ else\ e_2,s_0\rangle \Rightarrow \langle e',s_2\rangle$
| $CondF_1$:
$\llbracket P \vdash_1 \langle e,s_0\rangle \Rightarrow \langle false,s_1\rangle;\ P \vdash_1 \langle e_2,s_1\rangle \Rightarrow \langle e',s_2\rangle \rrbracket$
$\Longrightarrow P \vdash_1 \langle if\ (e)\ e_1\ else\ e_2,s_0\rangle \Rightarrow \langle e',s_2\rangle$
| $CondThrow_1$:
$P \vdash_1 \langle e,s_0\rangle \Rightarrow \langle throw\ e',s_1\rangle \Longrightarrow$
$P \vdash_1 \langle if\ (e)\ e_1\ else\ e_2,\ s_0\rangle \Rightarrow \langle throw\ e',s_1\rangle$

| $WhileF_1$:
$P \vdash_1 \langle e,s_0\rangle \Rightarrow \langle false,s_1\rangle \Longrightarrow$
$P \vdash_1 \langle while\ (e)\ c,s_0\rangle \Rightarrow \langle unit,s_1\rangle$
| $WhileT_1$:
$\llbracket P \vdash_1 \langle e,s_0\rangle \Rightarrow \langle true,s_1\rangle;\ P \vdash_1 \langle c,s_1\rangle \Rightarrow \langle Val\ v_1,s_2\rangle;$
$P \vdash_1 \langle while\ (e)\ c,s_2\rangle \Rightarrow \langle e_3,s_3\rangle \rrbracket$
$\Longrightarrow P \vdash_1 \langle while\ (e)\ c,s_0\rangle \Rightarrow \langle e_3,s_3\rangle$
| $WhileCondThrow_1$:
$P \vdash_1 \langle e,s_0\rangle \Rightarrow \langle throw\ e',s_1\rangle \Longrightarrow$
$P \vdash_1 \langle while\ (e)\ c,s_0\rangle \Rightarrow \langle throw\ e',s_1\rangle$
| $WhileBodyThrow_1$:
$\llbracket P \vdash_1 \langle e,s_0\rangle \Rightarrow \langle true,s_1\rangle;\ P \vdash_1 \langle c,s_1\rangle \Rightarrow \langle throw\ e',s_2\rangle \rrbracket$
$\Longrightarrow P \vdash_1 \langle while\ (e)\ c,s_0\rangle \Rightarrow \langle throw\ e',s_2\rangle$

| $Throw_1$:
$P \vdash_1 \langle e,s_0\rangle \Rightarrow \langle addr\ a,s_1\rangle \Longrightarrow$
$P \vdash_1 \langle throw\ e,s_0\rangle \Rightarrow \langle Throw\ a,s_1\rangle$
| $ThrowNull_1$:
$P \vdash_1 \langle e,s_0\rangle \Rightarrow \langle null,s_1\rangle \Longrightarrow$
$P \vdash_1 \langle throw\ e,s_0\rangle \Rightarrow \langle THROW\ NullPointer,s_1\rangle$
| $ThrowThrow_1$:
$P \vdash_1 \langle e,s_0\rangle \Rightarrow \langle throw\ e',s_1\rangle \Longrightarrow$
$P \vdash_1 \langle throw\ e,s_0\rangle \Rightarrow \langle throw\ e',s_1\rangle$

| Try_1:
$P \vdash_1 \langle e_1,s_0\rangle \Rightarrow \langle Val\ v_1,s_1\rangle \Longrightarrow$
$P \vdash_1 \langle try\ e_1\ catch(C\ i)\ e_2,s_0\rangle \Rightarrow \langle Val\ v_1,s_1\rangle$
| $TryCatch_1$:
$[\![P \vdash_1 \langle e_1,s_0\rangle \Rightarrow \langle Throw\ a,(h_1,ls_1,sh_1)\rangle;$
$h_1\ a = Some(D,fs);\ P \vdash D \preceq^* C;\ i < length\ ls_1;$
$P \vdash_1 \langle e_2,(h_1,ls_1[i:=Addr\ a],sh_1)\rangle \Rightarrow \langle e_2',(h_2,ls_2,sh_2)\rangle\ ]\!]$
$\Longrightarrow P \vdash_1 \langle try\ e_1\ catch(C\ i)\ e_2,s_0\rangle \Rightarrow \langle e_2',(h_2,ls_2,sh_2)\rangle$
| $TryThrow_1$:
$[\![P \vdash_1 \langle e_1,s_0\rangle \Rightarrow \langle Throw\ a,(h_1,ls_1,sh_1)\rangle;\ h_1\ a = Some(D,fs);\ \neg\ P \vdash D \preceq^* C\ ]\!]$
$\Longrightarrow P \vdash_1 \langle try\ e_1\ catch(C\ i)\ e_2,s_0\rangle \Rightarrow \langle Throw\ a,(h_1,ls_1,sh_1)\rangle$

| Nil_1:
$P \vdash_1 \langle[],s\rangle\ [\Rightarrow]\ \langle[],s\rangle$

| $Cons_1$:
$[\![P \vdash_1 \langle e,s_0\rangle \Rightarrow \langle Val\ v,s_1\rangle;\ P \vdash_1 \langle es,s_1\rangle\ [\Rightarrow]\ \langle es',s_2\rangle\ ]\!]$
$\Longrightarrow P \vdash_1 \langle e\#es,s_0\rangle\ [\Rightarrow]\ \langle Val\ v\ \#\ es',s_2\rangle$
| $ConsThrow_1$:
$P \vdash_1 \langle e,s_0\rangle \Rightarrow \langle throw\ e',s_1\rangle \Longrightarrow$
$P \vdash_1 \langle e\#es,s_0\rangle\ [\Rightarrow]\ \langle throw\ e'\ \#\ es,\ s_1\rangle$

— init rules

| $InitFinal_1$:
$P \vdash_1 \langle e,s\rangle \Rightarrow \langle e',s'\rangle \Longrightarrow P \vdash_1 \langle INIT\ C\ (Nil,b) \leftarrow e,s\rangle \Rightarrow \langle e',s'\rangle$
| $InitNone_1$:
$[\![sh\ C = None;\ P \vdash_1 \langle INIT\ C'\ (C\#Cs,False) \leftarrow e,(h,l,sh(C \mapsto (sblank\ P\ C,\ Prepared)))\rangle \Rightarrow \langle e',s'\rangle$
$]\!]$
$\Longrightarrow P \vdash_1 \langle INIT\ C'\ (C\#Cs,False) \leftarrow e,(h,l,sh)\rangle \Rightarrow \langle e',s'\rangle$
| $InitDone_1$:
$[\![sh\ C = Some(sfs,Done);\ P \vdash_1 \langle INIT\ C'\ (Cs,True) \leftarrow e,(h,l,sh)\rangle \Rightarrow \langle e',s'\rangle\ ]\!]$
$\Longrightarrow P \vdash_1 \langle INIT\ C'\ (C\#Cs,False) \leftarrow e,(h,l,sh)\rangle \Rightarrow \langle e',s'\rangle$
| $InitProcessing_1$:
$[\![sh\ C = Some(sfs,Processing);\ P \vdash_1 \langle INIT\ C'\ (Cs,True) \leftarrow e,(h,l,sh)\rangle \Rightarrow \langle e',s'\rangle\ ]\!]$
$\Longrightarrow P \vdash_1 \langle INIT\ C'\ (C\#Cs,False) \leftarrow e,(h,l,sh)\rangle \Rightarrow \langle e',s'\rangle$
| $InitError_1$:
$[\![sh\ C = Some(sfs,Error);$
$P \vdash_1 \langle RI\ (C,\ THROW\ NoClassDefFoundError);Cs \leftarrow e,(h,l,sh)\rangle \Rightarrow \langle e',s'\rangle\ ]\!]$
$\Longrightarrow P \vdash_1 \langle INIT\ C'\ (C\#Cs,False) \leftarrow e,(h,l,sh)\rangle \Rightarrow \langle e',s'\rangle$
| $InitObject_1$:
$[\![sh\ C = Some(sfs,Prepared);$
$C = Object;$
$sh' = sh(C \mapsto (sfs,Processing));$
$P \vdash_1 \langle INIT\ C'\ (C\#Cs,True) \leftarrow e,(h,l,sh')\rangle \Rightarrow \langle e',s'\rangle\ ]\!]$
$\Longrightarrow P \vdash_1 \langle INIT\ C'\ (C\#Cs,False) \leftarrow e,(h,l,sh)\rangle \Rightarrow \langle e',s'\rangle$
| $InitNonObject_1$:
$[\![sh\ C = Some(sfs,Prepared);$
$C \neq Object;$
$class\ P\ C = Some\ (D,r);$
$sh' = sh(C \mapsto (sfs,Processing));$
$P \vdash_1 \langle INIT\ C'\ (D\#C\#Cs,False) \leftarrow e,(h,l,sh')\rangle \Rightarrow \langle e',s'\rangle\ ]\!]$
$\Longrightarrow P \vdash_1 \langle INIT\ C'\ (C\#Cs,False) \leftarrow e,(h,l,sh)\rangle \Rightarrow \langle e',s'\rangle$
| $InitRInit_1$:

$P \vdash_1 \langle RI\ (C,C\cdot_s clinit([]));Cs \leftarrow e,(h,l,sh)\rangle \Rightarrow \langle e',s'\rangle$
$\Longrightarrow P \vdash_1 \langle INIT\ C'\ (C\#Cs,True) \leftarrow e,(h,l,sh)\rangle \Rightarrow \langle e',s'\rangle$

| $RInit_1$:
$[\![P \vdash_1 \langle e,s\rangle \Rightarrow \langle Val\ v,\ (h',l',sh')\rangle;$
$sh'\ C = Some(sfs,\ i);\ sh'' = sh'(C \mapsto (sfs,\ Done));$
$C' = last(C\#Cs);$
$P \vdash_1 \langle INIT\ C'\ (Cs,True) \leftarrow e',\ (h',l',sh'')\rangle \Rightarrow \langle e_1,s_1\rangle\ ]\!]$
$\Longrightarrow P \vdash_1 \langle RI\ (C,e);Cs \leftarrow e',s\rangle \Rightarrow \langle e_1,s_1\rangle$
| $RInitInitFail_1$:
$[\![P \vdash_1 \langle e,s\rangle \Rightarrow \langle throw\ a,\ (h',l',sh')\rangle;$
$sh'\ C = Some(sfs,\ i);\ sh'' = sh'(C \mapsto (sfs,\ Error));$
$P \vdash_1 \langle RI\ (D,throw\ a);Cs \leftarrow e',\ (h',l',sh'')\rangle \Rightarrow \langle e_1,s_1\rangle\ ]\!]$
$\Longrightarrow P \vdash_1 \langle RI\ (C,e);D\#Cs \leftarrow e',s\rangle \Rightarrow \langle e_1,s_1\rangle$
| $RInitFailFinal_1$:
$[\![P \vdash_1 \langle e,s\rangle \Rightarrow \langle throw\ a,\ (h',l',sh')\rangle;$
$sh'\ C = Some(sfs,\ i);\ sh'' = sh'(C \mapsto (sfs,\ Error))\ ]\!]$
$\Longrightarrow P \vdash_1 \langle RI\ (C,e);Nil \leftarrow e',s\rangle \Rightarrow \langle throw\ a,\ (h',l',sh'')\rangle$

inductive-cases $eval_1$-*cases* [*cases set*]:
$P \vdash_1 \langle new\ C,s\rangle \Rightarrow \langle e',s'\rangle$
$P \vdash_1 \langle Cast\ C\ e,s\rangle \Rightarrow \langle e',s'\rangle$
$P \vdash_1 \langle Val\ v,s\rangle \Rightarrow \langle e',s'\rangle$
$P \vdash_1 \langle e_1\ «bop»\ e_2,s\rangle \Rightarrow \langle e',s'\rangle$
$P \vdash_1 \langle Var\ v,s\rangle \Rightarrow \langle e',s'\rangle$
$P \vdash_1 \langle V:=e,s\rangle \Rightarrow \langle e',s'\rangle$
$P \vdash_1 \langle e\cdot F\{D\},s\rangle \Rightarrow \langle e',s'\rangle$
$P \vdash_1 \langle C\cdot_s F\{D\},s\rangle \Rightarrow \langle e',s'\rangle$
$P \vdash_1 \langle e_1\cdot F\{D\}:=e_2,s\rangle \Rightarrow \langle e',s'\rangle$
$P \vdash_1 \langle C\cdot_s F\{D\}:=e_2,s\rangle \Rightarrow \langle e',s'\rangle$
$P \vdash_1 \langle e\cdot M(es),s\rangle \Rightarrow \langle e',s'\rangle$
$P \vdash_1 \langle C\cdot_s M(es),s\rangle \Rightarrow \langle e',s'\rangle$
$P \vdash_1 \langle \{V:T;e_1\},s\rangle \Rightarrow \langle e',s'\rangle$
$P \vdash_1 \langle e_1;;e_2,s\rangle \Rightarrow \langle e',s'\rangle$
$P \vdash_1 \langle if\ (e)\ e_1\ else\ e_2,s\rangle \Rightarrow \langle e',s'\rangle$
$P \vdash_1 \langle while\ (b)\ c,s\rangle \Rightarrow \langle e',s'\rangle$
$P \vdash_1 \langle throw\ e,s\rangle \Rightarrow \langle e',s'\rangle$
$P \vdash_1 \langle try\ e_1\ catch(C\ V)\ e_2,s\rangle \Rightarrow \langle e',s'\rangle$
$P \vdash_1 \langle INIT\ C\ (Cs,b) \leftarrow e,s\rangle \Rightarrow \langle e',s'\rangle$
$P \vdash_1 \langle RI\ (C,e);Cs \leftarrow e_0,s\rangle \Rightarrow \langle e',s'\rangle$

inductive-cases $evals_1$-*cases* [*cases set*]:
$P \vdash_1 \langle [],s\rangle\ [\Rightarrow]\ \langle e',s'\rangle$
$P \vdash_1 \langle e\#es,s\rangle\ [\Rightarrow]\ \langle e',s'\rangle$

lemma $eval_1$-*final*: $P \vdash_1 \langle e,s\rangle \Rightarrow \langle e',s'\rangle \Longrightarrow final\ e'$
and $evals_1$-*final*: $P \vdash_1 \langle es,s\rangle\ [\Rightarrow]\ \langle es',s'\rangle \Longrightarrow finals\ es'\langle proof\rangle$

lemma $eval_1$-*final-same*:
assumes *eval*: $P \vdash_1 \langle e,s\rangle \Rightarrow \langle e',s'\rangle$ **shows** $final\ e \Longrightarrow e = e' \wedge s = s'\langle proof\rangle$

3.1.1 Property preservation

lemma *eval$_1$-preserves-len*:
$P \vdash_1 \langle e_0,(h_0,ls_0,sh_0)\rangle \Rightarrow \langle e_1,(h_1,ls_1,sh_1)\rangle \Longrightarrow length\ ls_0 = length\ ls_1$
and *evals$_1$-preserves-len*:
$P \vdash_1 \langle es_0,(h_0,ls_0,sh_0)\rangle\ [\Rightarrow]\ \langle es_1,(h_1,ls_1,sh_1)\rangle \Longrightarrow length\ ls_0 = length\ ls_1\langle proof\rangle$

lemma *evals$_1$-preserves-elen*:
$\bigwedge es'\ s\ s'.\ P \vdash_1 \langle es,s\rangle\ [\Rightarrow]\ \langle es',s'\rangle \Longrightarrow length\ es = length\ es'\langle proof\rangle$

lemma *clinit$_1$-loc-pres*:
$P \vdash_1 \langle C_0{\cdot_s}clinit([]),(h,l,sh)\rangle \Rightarrow \langle e',(h',l',sh')\rangle \Longrightarrow l = l'$
$\langle proof\rangle$

lemma
shows *init$_1$-ri$_1$-same-loc*: $P \vdash_1 \langle e,(h,l,sh)\rangle \Rightarrow \langle e',(h',l',sh')\rangle$
$\Longrightarrow (\bigwedge C\ Cs\ b\ M\ a.\ e = INIT\ C\ (Cs,b) \leftarrow unit \vee e = C{\cdot_s}M([]) \vee e = RI\ (C,Throw\ a)\ ;\ Cs \leftarrow unit$
$\vee\ e = RI\ (C,C{\cdot_s}clinit([]))\ ;\ Cs \leftarrow unit$
$\Longrightarrow l = l')$
and $P \vdash_1 \langle es,(h,l,sh)\rangle\ [\Rightarrow]\ \langle es',(h',l',sh')\rangle \Longrightarrow True$
$\langle proof\rangle$

lemma *init$_1$-same-loc*: $P \vdash_1 \langle INIT\ C\ (Cs,b) \leftarrow unit,(h,l,sh)\rangle \Rightarrow \langle e',(h',l',sh')\rangle \Longrightarrow l = l'$
$\langle proof\rangle$

theorem *eval$_1$-hext*: $P \vdash_1 \langle e,(h,l,sh)\rangle \Rightarrow \langle e',(h',l',sh')\rangle \Longrightarrow h \trianglelefteq h'$
and *evals$_1$-hext*: $P \vdash_1 \langle es,(h,l,sh)\rangle\ [\Rightarrow]\ \langle es',(h',l',sh')\rangle \Longrightarrow h \trianglelefteq h'\langle proof\rangle$

3.1.2 Initialization

lemma *rinit$_1$-throw*:
$P_1 \vdash_1 \langle RI\ (D,Throw\ xa)\ ;\ Cs \leftarrow e,(h,\ l,\ sh)\rangle \Rightarrow \langle e',(h',\ l',\ sh')\rangle$
$\Longrightarrow e' = Throw\ xa$
$\langle proof\rangle$

lemma *rinit$_1$-throwE*:
$P \vdash_1 \langle RI\ (C,throw\ e)\ ;\ Cs \leftarrow e_0,s\rangle \Rightarrow \langle e',s'\rangle$
$\Longrightarrow \exists a\ s_t.\ e' = throw\ a \wedge P \vdash_1 \langle throw\ e,s\rangle \Rightarrow \langle throw\ a,s_t\rangle$
$\langle proof\rangle$

end

3.2 Well-Formedness of Intermediate Language

theory *J1WellForm*
imports *../J/JWellForm J1*
begin

3.2.1 Well-Typedness

type-synonym
$env_1 = ty\ list$ — type environment indexed by variable number

inductive
$WT_1 :: [J_1\text{-}prog, env_1, expr_1, ty] \Rightarrow bool$
$(\langle(\text{-},\text{-} \vdash_1 / \text{ - :: -})\rangle \quad [51,51,51]50)$
and $WTs_1 :: [J_1\text{-}prog, env_1, expr_1\ list, ty\ list] \Rightarrow bool$
$(\langle(\text{-},\text{-} \vdash_1 / \text{ - [::] -})\rangle\ [51,51,51]50)$
for $P :: J_1\text{-}prog$
where

$WTNew_1$:
$is\text{-}class\ P\ C \Longrightarrow$
$P,E \vdash_1 new\ C :: Class\ C$

| $WTCast_1$:
$[\![\ P,E \vdash_1 e :: Class\ D;\ \ is\text{-}class\ P\ C;\ \ P \vdash C \preceq^* D \vee P \vdash D \preceq^* C\ ]\!]$
$\Longrightarrow P,E \vdash_1 Cast\ C\ e :: Class\ C$

| $WTVal_1$:
$typeof\ v = Some\ T \Longrightarrow$
$P,E \vdash_1 Val\ v :: T$

| $WTVar_1$:
$[\![\ E!i = T;\ i < size\ E\ ]\!]$
$\Longrightarrow P,E \vdash_1 Var\ i :: T$

| $WTBinOp_1$:
$[\![\ P,E \vdash_1 e_1 :: T_1;\ \ P,E \vdash_1 e_2 :: T_2;$
$case\ bop\ of\ Eq \Rightarrow (P \vdash T_1 \le T_2 \vee P \vdash T_2 \le T_1) \wedge T = Boolean$
$\quad | Add \Rightarrow T_1 = Integer \wedge T_2 = Integer \wedge T = Integer\ ]\!]$
$\Longrightarrow P,E \vdash_1 e_1\ «bop»\ e_2 :: T$

| $WTLAss_1$:
$[\![\ E!i = T;\ \ i < size\ E;\ P,E \vdash_1 e :: T';\ \ P \vdash T' \le T\ ]\!]$
$\Longrightarrow P,E \vdash_1 i{:=}e :: Void$

| $WTFAcc_1$:
$[\![\ P,E \vdash_1 e :: Class\ C;\ \ P \vdash C\ sees\ F,NonStatic{:}T\ in\ D\ ]\!]$
$\Longrightarrow P,E \vdash_1 e{\cdot}F\{D\} :: T$

| $WTSFAcc_1$:
$[\![\ P \vdash C\ sees\ F,Static{:}T\ in\ D\ ]\!]$
$\Longrightarrow P,E \vdash_1 C{\cdot}_s F\{D\} :: T$

| $WTFAss_1$:
$[\![\ P,E \vdash_1 e_1 :: Class\ C;\ \ P \vdash C\ sees\ F,NonStatic{:}T\ in\ D;\ \ P,E \vdash_1 e_2 :: T';\ \ P \vdash T' \le T\ ]\!]$
$\Longrightarrow P,E \vdash_1 e_1{\cdot}F\{D\} := e_2 :: Void$

| $WTSFAss_1$:
$[\![\ \ P \vdash C\ sees\ F,Static{:}T\ in\ D;\ \ P,E \vdash_1 e_2 :: T';\ \ P \vdash T' \le T\ ]\!]$
$\Longrightarrow P,E \vdash_1 C{\cdot}_s F\{D\}{:=}e_2 :: Void$

| $WTCall_1$:
$[\![\ P,E \vdash_1 e :: Class\ C;\ P \vdash C\ sees\ M,NonStatic{:}Ts' \to T = m\ in\ D;$
$\ P,E \vdash_1 es\ [::]\ Ts;\ \ P \vdash Ts\ [\le]\ Ts'\ ]\!]$
$\Longrightarrow P,E \vdash_1 e{\cdot}M(es) :: T$

| $WTSCall_1$:
$[\![P \vdash C\ sees\ M,Static{:}Ts \to T = m\ in\ D;$
$P,E \vdash_1 es\ [::]\ Ts';\ P \vdash Ts'\ [\leq]\ Ts;\ M \neq clinit\ ]\!]$
$\Longrightarrow P,E \vdash_1 C{\cdot_s}M(es) :: T$

| $WTBlock_1$:
$[\![is\text{-}type\ P\ T;\ P,E@[T] \vdash_1 e::T'\ ]\!]$
$\Longrightarrow\ P,E \vdash_1 \{i{:}T;\ e\} :: T'$

| $WTSeq_1$:
$[\![P,E \vdash_1 e_1::T_1;\ P,E \vdash_1 e_2::T_2\ ]\!]$
$\Longrightarrow\ P,E \vdash_1 e_1;;e_2 :: T_2$

| $WTCond_1$:
$[\![P,E \vdash_1 e :: Boolean;\ P,E \vdash_1 e_1::T_1;\ P,E \vdash_1 e_2::T_2;$
$P \vdash T_1 \leq T_2 \vee P \vdash T_2 \leq T_1;\ P \vdash T_1 \leq T_2 \longrightarrow T = T_2;\ P \vdash T_2 \leq T_1 \longrightarrow T = T_1\ ]\!]$
$\Longrightarrow P,E \vdash_1 if\ (e)\ e_1\ else\ e_2 :: T$

| $WTWhile_1$:
$[\![P,E \vdash_1 e :: Boolean;\ P,E \vdash_1 c::T\ ]\!]$
$\Longrightarrow P,E \vdash_1 while\ (e)\ c :: Void$

| $WTThrow_1$:
$P,E \vdash_1 e :: Class\ C \Longrightarrow$
$P,E \vdash_1 throw\ e :: Void$

| $WTTry_1$:
$[\![P,E \vdash_1 e_1 :: T;\ P,E@[Class\ C] \vdash_1 e_2 :: T;\ is\text{-}class\ P\ C\ ]\!]$
$\Longrightarrow P,E \vdash_1 try\ e_1\ catch(C\ i)\ e_2 :: T$

| $WTNil_1$:
$P,E \vdash_1 [] [::] []$

| $WTCons_1$:
$[\![P,E \vdash_1 e :: T;\ P,E \vdash_1 es\ [::]\ Ts\ ]\!]$
$\Longrightarrow\ P,E \vdash_1 e\#es\ [::]\ T\#Ts$

lemma *init-nWT*$_1$ [*simp*]:$\neg P,E \vdash_1 INIT\ C\ (Cs,b) \leftarrow e :: T$
⟨*proof*⟩
lemma *rinit-nWT*$_1$ [*simp*]:$\neg P,E \vdash_1 RI(C,e);Cs \leftarrow e' :: T$
⟨*proof*⟩

lemma *WTs*$_1$*-same-size*: $\bigwedge Ts.\ P,E \vdash_1 es\ [::]\ Ts \Longrightarrow size\ es = size\ Ts$⟨*proof*⟩

lemma *WT*$_1$*-unique*:
$P,E \vdash_1 e :: T_1 \Longrightarrow (\bigwedge T_2.\ P,E \vdash_1 e :: T_2 \Longrightarrow T_1 = T_2)$ **and**
WTs$_1$*-unique*: $P,E \vdash_1 es\ [::]\ Ts_1 \Longrightarrow (\bigwedge Ts_2.\ P,E \vdash_1 es\ [::]\ Ts_2 \Longrightarrow Ts_1 = Ts_2)$⟨*proof*⟩

lemma assumes *wf*: *wf-prog p P*
shows *WT*$_1$*-is-type*: $P,E \vdash_1 e :: T \Longrightarrow set\ E \subseteq types\ P \Longrightarrow is\text{-}type\ P\ T$
and $P,E \vdash_1 es\ [::]\ Ts \Longrightarrow True$⟨*proof*⟩
lemma *WT*$_1$*-nsub-RI*: $P,E \vdash_1 e :: T \Longrightarrow \neg sub\text{-}RI\ e$
and *WTs*$_1$*-nsub-RIs*: $P,E \vdash_1 es\ [::]\ Ts \Longrightarrow \neg sub\text{-}RIs\ es$

⟨*proof*⟩

3.2.2 Runtime Well-Typedness

inductive
  $WTrt_1 :: J_1\text{-}prog \Rightarrow heap \Rightarrow sheap \Rightarrow env_1 \Rightarrow expr_1 \Rightarrow ty \Rightarrow bool$
  and $WTrts_1 :: J_1\text{-}prog \Rightarrow heap \Rightarrow sheap \Rightarrow env_1 \Rightarrow expr_1\ list \Rightarrow ty\ list \Rightarrow bool$
  and $WTrt2_1 :: [J_1\text{-}prog, env_1, heap, sheap, expr_1, ty] \Rightarrow bool$
      $(\langle\text{-},\text{-},\text{-},\text{-} \vdash_1 \text{-} : \text{-}\rangle \quad [51,51,51,51]50)$
  and $WTrts2_1 :: [J_1\text{-}prog, env_1, heap, sheap, expr_1\ list,\ ty\ list] \Rightarrow bool$
      $(\langle\text{-},\text{-},\text{-},\text{-} \vdash_1 \text{-} [:] \text{-}\rangle\ [51,51,51,51]50)$
  for $P :: J_1\text{-}prog$ **and** $h :: heap$ **and** $sh :: sheap$
where

  $P,E,h,sh \vdash_1 e : T \equiv WTrt_1\ P\ h\ sh\ E\ e\ T$
| $P,E,h,sh \vdash_1 es[:]Ts \equiv WTrts_1\ P\ h\ sh\ E\ es\ Ts$

| $WTrtNew_1$:
  $is\text{-}class\ P\ C \Longrightarrow$
  $P,E,h,sh \vdash_1 new\ C : Class\ C$

| $WTrtCast_1$:
  $[\![\ P,E,h,sh \vdash_1 e : T;\ is\text{-}refT\ T;\ is\text{-}class\ P\ C\ ]\!]$
  $\Longrightarrow P,E,h,sh \vdash_1 Cast\ C\ e : Class\ C$

| $WTrtVal_1$:
  $typeof_h\ v = Some\ T \Longrightarrow$
  $P,E,h,sh \vdash_1 Val\ v : T$

| $WTrtVar_1$:
  $[\![\ E!i = T;\ i < size\ E\ ]\!] \Longrightarrow$
  $P,E,h,sh \vdash_1 Var\ i : T$

| $WTrtBinOpEq_1$:
  $[\![\ P,E,h,sh \vdash_1 e_1 : T_1;\ \ P,E,h,sh \vdash_1 e_2 : T_2\ ]\!]$
  $\Longrightarrow P,E,h,sh \vdash_1 e_1\ «Eq»\ e_2 : Boolean$

| $WTrtBinOpAdd_1$:
  $[\![\ P,E,h,sh \vdash_1 e_1 : Integer;\ \ P,E,h,sh \vdash_1 e_2 : Integer\ ]\!]$
  $\Longrightarrow P,E,h,sh \vdash_1 e_1\ «Add»\ e_2 : Integer$

| $WTrtLAss_1$:
  $[\![\ E!i = T;\ i < size\ E;\ P,E,h,sh \vdash_1 e : T';\ \ P \vdash T' \leq T\ ]\!]$
  $\Longrightarrow P,E,h,sh \vdash_1 i{:=}e : Void$

| $WTrtFAcc_1$:
  $[\![\ P,E,h,sh \vdash_1 e : Class\ C;\ P \vdash C\ has\ F,NonStatic{:}T\ in\ D\ ]\!] \Longrightarrow$
  $P,E,h,sh \vdash_1 e{\cdot}F\{D\} : T$

| $WTrtFAccNT_1$:
  $P,E,h,sh \vdash_1 e : NT \Longrightarrow$
  $P,E,h,sh \vdash_1 e{\cdot}F\{D\} : T$

| $WTrtSFAcc_1$:

$[\![P \vdash C\ has\ F,Static{:}T\ in\ D]\!] \Longrightarrow$
$P,E,h,sh \vdash_1 C{\cdot_s}F\{D\} : T$

| *WTrtFAss*$_1$:
$[\![P,E,h,sh \vdash_1 e_1 : Class\ C;\ \ P \vdash C\ has\ F,NonStatic{:}T\ in\ D;\ P,E,h,sh \vdash_1 e_2 : T_2;\ \ P \vdash T_2 \leq T]\!]$
$\Longrightarrow P,E,h,sh \vdash_1 e_1{\cdot}F\{D\}{:=}e_2 : Void$

| *WTrtFAssNT*$_1$:
$[\![P,E,h,sh \vdash_1 e_1{:}NT;\ P,E,h,sh \vdash_1 e_2 : T_2]\!]$
$\Longrightarrow P,E,h,sh \vdash_1 e_1{\cdot}F\{D\}{:=}e_2 : Void$

| *WTrtSFAss*$_1$:
$[\![P,E,h,sh \vdash_1 e_2 : T_2;\ P \vdash C\ has\ F,Static{:}T\ in\ D;\ P \vdash T_2 \leq T]\!]$
$\Longrightarrow P,E,h,sh \vdash_1 C{\cdot_s}F\{D\}{:=}e_2 : Void$

| *WTrtCall*$_1$:
$[\![P,E,h,sh \vdash_1 e : Class\ C;\ P \vdash C\ sees\ M,NonStatic{:}Ts \to T = m\ in\ D;$
$\quad P,E,h,sh \vdash_1 es\ [:]\ Ts';\ P \vdash Ts'\ [\leq]\ Ts]\!]$
$\Longrightarrow P,E,h,sh \vdash_1 e{\cdot}M(es) : T$

| *WTrtCallNT*$_1$:
$[\![P,E,h,sh \vdash_1 e : NT;\ P,E,h,sh \vdash_1 es\ [:]\ Ts]\!]$
$\Longrightarrow P,E,h,sh \vdash_1 e{\cdot}M(es) : T$

| *WTrtSCall*$_1$:
$[\![P \vdash C\ sees\ M,Static{:}Ts \to T = m\ in\ D;$
$\quad P,E,h,sh \vdash_1 es\ [:]\ Ts';\ P \vdash Ts'\ [\leq]\ Ts;$
$\quad M = clinit \longrightarrow sh\ D = \lfloor(sfs,Processing)\rfloor \wedge es = map\ Val\ vs]\!]$
$\Longrightarrow P,E,h,sh \vdash_1 C{\cdot_s}M(es) : T$

| *WTrtBlock*$_1$:
$P,E@[T],h,sh \vdash_1 e : T' \Longrightarrow$
$P,E,h,sh \vdash_1 \{i{:}T;\ e\} : T'$

| *WTrtSeq*$_1$:
$[\![P,E,h,sh \vdash_1 e_1{:}T_1;\ \ P,E,h,sh \vdash_1 e_2{:}T_2]\!]$
$\Longrightarrow P,E,h,sh \vdash_1 e_1;;e_2 : T_2$

| *WTrtCond*$_1$:
$[\![P,E,h,sh \vdash_1 e : Boolean;\ \ P,E,h,sh \vdash_1 e_1{:}T_1;\ \ P,E,h,sh \vdash_1 e_2{:}T_2;$
$\quad P \vdash T_1 \leq T_2 \vee P \vdash T_2 \leq T_1;\ P \vdash T_1 \leq T_2 \longrightarrow T = T_2;\ P \vdash T_2 \leq T_1 \longrightarrow T = T_1]\!]$
$\Longrightarrow P,E,h,sh \vdash_1 if\ (e)\ e_1\ else\ e_2 : T$

| *WTrtWhile*$_1$:
$[\![P,E,h,sh \vdash_1 e : Boolean;\ \ P,E,h,sh \vdash_1 c{:}T]\!]$
$\Longrightarrow\ \ P,E,h,sh \vdash_1 while(e)\ c : Void$

| *WTrtThrow*$_1$:
$[\![P,E,h,sh \vdash_1 e : T_r;\ is\text{-}refT\ T_r]\!] \Longrightarrow$
$P,E,h,sh \vdash_1 throw\ e : T$

| *WTrtTry*$_1$:
$[\![P,E,h,sh \vdash_1 e_1 : T_1;\ \ P,E@[Class\ C],h,sh \vdash_1 e_2 : T_2;\ P \vdash T_1 \leq T_2]\!]$
$\Longrightarrow P,E,h,sh \vdash_1 try\ e_1\ catch(C\ i)\ e_2 : T_2$

| $WTrtInit_1$:
$[\![P,E,h,sh \vdash_1 e : T; \forall C' \in set\ (C\#Cs).\ is\text{-}class\ P\ C'; \neg sub\text{-}RI\ e;$
$\forall C' \in set\ (tl\ Cs).\ \exists sfs.\ sh\ C' = \lfloor(sfs,Processing)\rfloor;$
$b \longrightarrow (\forall C' \in set\ Cs.\ \exists sfs.\ sh\ C' = \lfloor(sfs,Processing)\rfloor);$
$distinct\ Cs; supercls\text{-}lst\ P\ Cs\ ]\!]$
$\Longrightarrow P,E,h,sh \vdash_1 INIT\ C\ (Cs,\ b) \leftarrow e : T$

| $WTrtRI_1$:
$[\![P,E,h,sh \vdash_1 e : T; P,E,h,sh \vdash_1 e' : T'; \forall C' \in set\ (C\#Cs).\ is\text{-}class\ P\ C'; \neg sub\text{-}RI\ e';$
$\forall C' \in set\ (C\#Cs).\ not\text{-}init\ C'\ e;$
$\forall C' \in set\ Cs.\ \exists sfs.\ sh\ C' = \lfloor(sfs,Processing)\rfloor;$
$\exists sfs.\ sh\ C = \lfloor(sfs,\ Processing)\rfloor \vee (sh\ C = \lfloor(sfs,\ Error)\rfloor \wedge e = THROW\ NoClassDefFoundError);$
$distinct\ (C\#Cs); supercls\text{-}lst\ P\ (C\#Cs)\ ]\!]$
$\Longrightarrow P,E,h,sh \vdash_1 RI(C,\ e);Cs \leftarrow e' : T'$

— well-typed expression lists

| $WTrtNil_1$:
$P,E,h,sh \vdash_1 [] [:] []$

| $WTrtCons_1$:
$[\![P,E,h,sh \vdash_1 e : T;\ \ P,E,h,sh \vdash_1 es\ [:]\ Ts\ ]\!]$
$\Longrightarrow\ \ P,E,h,sh \vdash_1 e\#es\ [:]\ T\#Ts$

lemma $WT_1\text{-}implies\text{-}WTrt_1$: $P,E \vdash_1 e :: T \Longrightarrow P,E,h,sh \vdash_1 e : T$
and $WTs_1\text{-}implies\text{-}WTrts_1$: $P,E \vdash_1 es\ [::]\ Ts \Longrightarrow P,E,h,sh \vdash_1 es\ [:]\ Ts\langle proof\rangle$

3.2.3 Well-formedness

primrec $\mathcal{B} :: expr_1 \Rightarrow nat \Rightarrow bool$
and $\mathcal{B}s :: expr_1\ list \Rightarrow nat \Rightarrow bool$ **where**
$\mathcal{B}\ (new\ C)\ i = True\ |$
$\mathcal{B}\ (Cast\ C\ e)\ i = \mathcal{B}\ e\ i\ |$
$\mathcal{B}\ (Val\ v)\ i = True\ |$
$\mathcal{B}\ (e_1\ «bop»\ e_2)\ i = (\mathcal{B}\ e_1\ i \wedge \mathcal{B}\ e_2\ i)\ |$
$\mathcal{B}\ (Var\ j)\ i = True\ |$
$\mathcal{B}\ (e{\cdot}F\{D\})\ i = \mathcal{B}\ e\ i\ |$
$\mathcal{B}\ (C{\cdot}_sF\{D\})\ i = True\ |$
$\mathcal{B}\ (j{:=}e)\ i = \mathcal{B}\ e\ i\ |$
$\mathcal{B}\ (e_1{\cdot}F\{D\} := e_2)\ i = (\mathcal{B}\ e_1\ i \wedge \mathcal{B}\ e_2\ i)\ |$
$\mathcal{B}\ (C{\cdot}_sF\{D\} := e_2)\ i = \mathcal{B}\ e_2\ i\ |$
$\mathcal{B}\ (e{\cdot}M(es))\ i = (\mathcal{B}\ e\ i \wedge \mathcal{B}s\ es\ i)\ |$
$\mathcal{B}\ (C{\cdot}_sM(es))\ i = \mathcal{B}s\ es\ i\ |$
$\mathcal{B}\ (\{j{:}T\ ;\ e\})\ i = (i = j \wedge \mathcal{B}\ e\ (i{+}1))\ |$
$\mathcal{B}\ (e_1;;e_2)\ i = (\mathcal{B}\ e_1\ i \wedge \mathcal{B}\ e_2\ i)\ |$
$\mathcal{B}\ (if\ (e)\ e_1\ else\ e_2)\ i = (\mathcal{B}\ e\ i \wedge \mathcal{B}\ e_1\ i \wedge \mathcal{B}\ e_2\ i)\ |$
$\mathcal{B}\ (throw\ e)\ i = \mathcal{B}\ e\ i\ |$
$\mathcal{B}\ (while\ (e)\ c)\ i = (\mathcal{B}\ e\ i \wedge \mathcal{B}\ c\ i)\ |$
$\mathcal{B}\ (try\ e_1\ catch(C\ j)\ e_2)\ i = (\mathcal{B}\ e_1\ i \wedge i{=}j \wedge \mathcal{B}\ e_2\ (i{+}1))\ |$
$\mathcal{B}\ (INIT\ C\ (Cs,b) \leftarrow e)\ i = \mathcal{B}\ e\ i\ |$
$\mathcal{B}\ (RI(C,e);Cs \leftarrow e')\ i = (\mathcal{B}\ e\ i \wedge \mathcal{B}\ e'\ i)\ |$

$\mathcal{B}s\ []\ i = True\ |$

$\mathcal{B}s\ (e\#es)\ i = (\mathcal{B}\ e\ i \wedge \mathcal{B}s\ es\ i)$

definition *wf-J$_1$-mdecl* :: *J$_1$-prog* $\Rightarrow$ *cname* $\Rightarrow$ *expr$_1$ mdecl* $\Rightarrow$ *bool*
where
wf-J$_1$-mdecl P C $\equiv$ $\lambda(M,b,Ts,T,body)$.
$\neg$*sub-RI body* $\wedge$
(*case b of*
NonStatic $\Rightarrow$
($\exists\, T'$. *P,Class C*#*Ts* $\vdash_1$ *body* :: $T' \wedge P \vdash T' \leq T$) $\wedge$
$\mathcal{D}$ *body* $\lfloor\{..size\ Ts\}\rfloor$ $\wedge$ $\mathcal{B}$ *body* (*size Ts* + *1*)
| *Static* $\Rightarrow$ ($\exists\, T'$. *P,Ts* $\vdash_1$ *body* :: $T' \wedge P \vdash T' \leq T$) $\wedge$
$\mathcal{D}$ *body* $\lfloor\{..<size\ Ts\}\rfloor$ $\wedge$ $\mathcal{B}$ *body* (*size Ts*))

lemma *wf-J$_1$-mdecl-NonStatic*[*simp*]:
wf-J$_1$-mdecl P C (*M*,*NonStatic*,*Ts*,*T*,*body*) $\equiv$
($\neg$*sub-RI body* $\wedge$
($\exists\, T'$. *P,Class C*#*Ts* $\vdash_1$ *body* :: $T' \wedge P \vdash T' \leq T$) $\wedge$
$\mathcal{D}$ *body* $\lfloor\{..size\ Ts\}\rfloor$ $\wedge$ $\mathcal{B}$ *body* (*size Ts* + *1*))$\langle$*proof*$\rangle$
lemma *wf-J$_1$-mdecl-Static*[*simp*]:
wf-J$_1$-mdecl P C (*M*,*Static*,*Ts*,*T*,*body*) $\equiv$
($\neg$*sub-RI body* $\wedge$
($\exists\, T'$. *P,Ts* $\vdash_1$ *body* :: $T' \wedge P \vdash T' \leq T$) $\wedge$
$\mathcal{D}$ *body* $\lfloor\{..<size\ Ts\}\rfloor$ $\wedge$ $\mathcal{B}$ *body* (*size Ts*))$\langle$*proof*$\rangle$
abbreviation *wf-J$_1$-prog* == *wf-prog wf-J$_1$-mdecl*

lemma *sees-wf$_1$-nsub-RI*:
assumes *wf*: *wf-J$_1$-prog P* **and** *cM*: *P* $\vdash$ *C sees M*,*b* : *Ts*$\rightarrow$*T* = *body in D*
shows $\neg$*sub-RI body*
$\langle$*proof*$\rangle$

lemma *wf$_1$-types-clinit*:
assumes *wf*:*wf-prog wf-md P* **and** *ex*: *class P C* = *Some a* **and** *proc*: *sh C* = $\lfloor$(*sfs*, *Processing*)$\rfloor$
shows *P*,*E*,*h*,*sh* $\vdash_1$ *C*$\cdot_s$*clinit*([]) : *Void*
$\langle$*proof*$\rangle$

lemma assumes *wf*: *wf-J$_1$-prog P*
shows *eval$_1$-proc-pres*: *P* $\vdash_1$ $\langle$*e*,(*h*,*l*,*sh*)$\rangle$ $\Rightarrow$ $\langle$*e'*,(*h'*,*l'*,*sh'*)$\rangle$
$\Longrightarrow$ *not-init C e* $\Longrightarrow$ $\exists$ *sfs*. *sh C* = $\lfloor$(*sfs*, *Processing*)$\rfloor$ $\Longrightarrow$ $\exists$ *sfs'*. *sh' C* = $\lfloor$(*sfs'*, *Processing*)$\rfloor$
and *evals$_1$-proc-pres*: *P* $\vdash_1$ $\langle$*es*,(*h*,*l*,*sh*)$\rangle$ [$\Rightarrow$] $\langle$*es'*,(*h'*,*l'*,*sh'*)$\rangle$
$\Longrightarrow$ *not-inits C es* $\Longrightarrow$ $\exists$ *sfs*. *sh C* = $\lfloor$(*sfs*, *Processing*)$\rfloor$ $\Longrightarrow$ $\exists$ *sfs'*. *sh' C* = $\lfloor$(*sfs'*, *Processing*)$\rfloor\langle$*proof*$\rangle$
lemma *clinit$_1$-proc-pres*:
$[\![$ *wf-J$_1$-prog P*; *P* $\vdash_1$ $\langle$*C*$_0\cdot_s$*clinit*([]),(*h*,*l*,*sh*)$\rangle$ $\Rightarrow$ $\langle$*e'*,(*h'*,*l'*,*sh'*)$\rangle$;
sh C' = $\lfloor$(*sfs*,*Processing*)$\rfloor$ $]\!]$
$\Longrightarrow$ $\exists$ *sfs*. *sh' C'* = $\lfloor$(*sfs*,*Processing*)$\rfloor$
$\langle$*proof*$\rangle$

end

3.3 Program Compilation

theory *PCompiler*

imports *../Common/WellForm*
begin

definition *compM* :: (*staticb* ⇒ *'a* ⇒ *'b*) ⇒ *'a mdecl* ⇒ *'b mdecl*
where
compM f ≡ λ(*M*, *b*, *Ts*, *T*, *m*). (*M*, *b*, *Ts*, *T*, *f b m*)

definition *compC* :: (*staticb* ⇒ *'a* ⇒ *'b*) ⇒ *'a cdecl* ⇒ *'b cdecl*
where
compC f ≡ λ(*C*,*D*,*Fdecls*,*Mdecls*). (*C*,*D*,*Fdecls*, *map* (*compM f*) *Mdecls*)

definition *compP* :: (*staticb* ⇒ *'a* ⇒ *'b*) ⇒ *'a prog* ⇒ *'b prog*
where
compP f ≡ *map* (*compC f*)

Compilation preserves the program structure. Therefore lookup functions either commute with compilation (like method lookup) or are preserved by it (like the subclass relation).

lemma *map-of-map4*:
map-of (*map* (λ(*x*,*a*,*b*,*c*).(*x*,*a*,*b*,*f c*)) *ts*) =
map-option (λ(*a*,*b*,*c*).(*a*,*b*,*f c*)) ∘ (*map-of ts*)⟨*proof*⟩
lemma *map-of-map245*:
map-of (*map* (λ(*x*,*a*,*b*,*c*,*d*).(*x*,*a*,*b*,*c*,*f a c d*)) *ts*) =
map-option (λ(*a*,*b*,*c*,*d*).(*a*,*b*,*c*,*f a c d*)) ∘ (*map-of ts*)⟨*proof*⟩

lemma *class-compP*:
class P C = *Some* (*D*, *fs*, *ms*)
⟹ *class* (*compP f P*) *C* = *Some* (*D*, *fs*, *map* (*compM f*) *ms*)⟨*proof*⟩

lemma *class-compPD*:
class (*compP f P*) *C* = *Some* (*D*, *fs*, *cms*)
⟹ ∃ *ms*. *class P C* = *Some*(*D*,*fs*,*ms*) ∧ *cms* = *map* (*compM f*) *ms*⟨*proof*⟩

lemma [*simp*]: *is-class* (*compP f P*) *C* = *is-class P C*⟨*proof*⟩

lemma [*simp*]: *class* (*compP f P*) *C* = *map-option* (λ*c*. *snd*(*compC f* (*C*,*c*))) (*class P C*)⟨*proof*⟩

lemma *sees-methods-compP*:
P ⊢ *C sees-methods Mm* ⟹
compP f P ⊢ *C sees-methods* (*map-option* (λ((*b*,*Ts*,*T*,*m*),*D*). ((*b*,*Ts*,*T*,*f b m*),*D*)) ∘ *Mm*)⟨*proof*⟩

lemma *sees-method-compP*:
P ⊢ *C sees M*,*b*: *Ts*→*T* = *m in D* ⟹
compP f P ⊢ *C sees M*,*b*: *Ts*→*T* = (*f b m*) *in D*⟨*proof*⟩

lemma [*simp*]:
P ⊢ *C sees M*,*b*: *Ts*→*T* = *m in D* ⟹
method (*compP f P*) *C M* = (*D*,*b*,*Ts*,*T*,*f b m*)⟨*proof*⟩

lemma *sees-methods-compPD*:
⟦ *cP* ⊢ *C sees-methods Mm'*; *cP* = *compP f P* ⟧ ⟹
∃ *Mm*. *P* ⊢ *C sees-methods Mm* ∧
Mm' = (*map-option* (λ((*b*,*Ts*,*T*,*m*),*D*). ((*b*,*Ts*,*T*,*f b m*),*D*)) ∘ *Mm*)⟨*proof*⟩

lemma *sees-method-compPD*:

compP f P $\vdash$ *C sees M,b: Ts→T = fm in D* $\Longrightarrow$
$\exists$ *m. P* $\vdash$ *C sees M,b: Ts→T = m in D* $\wedge$ *f b m = fm*⟨*proof*⟩

lemma [*simp*]: *subcls1*(*compP f P*) = *subcls1 P*⟨*proof*⟩

lemma *compP-widen*[*simp*]: (*compP f P* $\vdash$ *T* $\leq$ *T'*) = (*P* $\vdash$ *T* $\leq$ *T'*)⟨*proof*⟩

lemma [*simp*]: (*compP f P* $\vdash$ *Ts* [$\leq$] *Ts'*) = (*P* $\vdash$ *Ts* [$\leq$] *Ts'*)⟨*proof*⟩

lemma [*simp*]: *is-type* (*compP f P*) *T* = *is-type P T*⟨*proof*⟩

lemma [*simp*]: (*compP* (*f*::*staticb*⇒'*a*⇒'*b*) *P* $\vdash$ *C has-fields FDTs*) = (*P* $\vdash$ *C has-fields FDTs*)⟨*proof*⟩

lemma *fields-compP* [*simp*]: *fields* (*compP f P*) *C* = *fields P C*⟨*proof*⟩
lemma *ifields-compP* [*simp*]: *ifields* (*compP f P*) *C* = *ifields P C*⟨*proof*⟩
lemma *blank-compP* [*simp*]: *blank* (*compP f P*) *C* = *blank P C*⟨*proof*⟩
lemma *isfields-compP* [*simp*]: *isfields* (*compP f P*) *C* = *isfields P C*⟨*proof*⟩
lemma *sblank-compP* [*simp*]: *sblank* (*compP f P*) *C* = *sblank P C*⟨*proof*⟩
lemma *sees-fields-compP* [*simp*]: (*compP f P* $\vdash$ *C sees F,b:T in D*) = (*P* $\vdash$ *C sees F,b:T in D*)⟨*proof*⟩
lemma *has-field-compP* [*simp*]: (*compP f P* $\vdash$ *C has F,b:T in D*) = (*P* $\vdash$ *C has F,b:T in D*)⟨*proof*⟩
lemma *field-compP* [*simp*]: *field* (*compP f P*) *F D* = *field P F D*⟨*proof*⟩

3.3.1 Invariance of *wf-prog* under compilation

lemma [*iff*]: *distinct-fst* (*compP f P*) = *distinct-fst P*⟨*proof*⟩

lemma [*iff*]: *distinct-fst* (*map* (*compM f*) *ms*) = *distinct-fst ms*⟨*proof*⟩

lemma [*iff*]: *wf-syscls* (*compP f P*) = *wf-syscls P*⟨*proof*⟩

lemma [*iff*]: *wf-fdecl* (*compP f P*) = *wf-fdecl P*⟨*proof*⟩

lemma *wf-clinit-compM* [*iff*]: *wf-clinit* (*map* (*compM f*) *ms*) = *wf-clinit ms*⟨*proof*⟩
lemma *set-compP*:
((*C,D,fs,ms'*) $\in$ *set*(*compP f P*)) =
(∃ *ms*. (*C,D,fs,ms*) $\in$ *set P* $\wedge$ *ms'* = *map* (*compM f*) *ms*)⟨*proof*⟩
lemma *wf-cdecl-compPI*:
⟦ $\bigwedge$*C M b Ts T m*.
⟦ *wf-mdecl wf*$_1$ *P C* (*M,b,Ts,T,m*); *P* $\vdash$ *C sees M,b:Ts→T = m in C* ⟧
$\Longrightarrow$ *wf-mdecl wf*$_2$ (*compP f P*) *C* (*M,b,Ts,T, f b m*);
$\forall$ *x*$\in$*set P*. *wf-cdecl wf*$_1$ *P x*; *x* $\in$ *set* (*compP f P*); *wf-prog p P* ⟧
$\Longrightarrow$ *wf-cdecl wf*$_2$ (*compP f P*) *x*⟨*proof*⟩

lemma *wf-prog-compPI*:
assumes *lift*:
$\bigwedge$*C M b Ts T m*.
⟦ *P* $\vdash$ *C sees M,b:Ts→T = m in C*; *wf-mdecl wf*$_1$ *P C* (*M,b,Ts,T,m*) ⟧
$\Longrightarrow$ *wf-mdecl wf*$_2$ (*compP f P*) *C* (*M,b,Ts,T, f b m*)
and *wf*: *wf-prog wf*$_1$ *P*
shows *wf-prog wf*$_2$ (*compP f P*)⟨*proof*⟩

end
theory *Hidden*
imports *List−Index.List-Index*

begin

definition *hidden* :: *'a list ⇒ nat ⇒ bool* **where**
hidden xs i ≡ i < size xs ∧ xs!i ∈ set(drop (i+1) xs)

lemma *hidden-last-index*: *x ∈ set xs ⟹ hidden (xs @ [x]) (last-index xs x)*
⟨*proof*⟩

lemma *hidden-inacc*: *hidden xs i ⟹ last-index xs x ≠ i*
⟨*proof*⟩

lemma [*simp*]: *hidden xs i ⟹ hidden (xs@[x]) i*
⟨*proof*⟩

lemma *fun-upds-apply*:
(m(xs[↦]ys)) x =
(let xs' = take (size ys) xs
in if x ∈ set xs' then Some(ys ! last-index xs' x) else m x)
⟨*proof*⟩

lemma *map-upds-apply-eq-Some*:
((m(xs[↦]ys)) x = Some y) =
(let xs' = take (size ys) xs
in if x ∈ set xs' then ys ! last-index xs' x = y else m x = Some y)
⟨*proof*⟩

lemma *map-upds-upd-conv-last-index*:
⟦x ∈ set xs; size xs ≤ size ys ⟧
⟹ m(xs[↦]ys, x↦y) = m(xs[↦]ys[last-index xs x := y])
⟨*proof*⟩

end

3.4 Compilation Stage 1

theory *Compiler1* **imports** *PCompiler J1 Hidden* **begin**

Replacing variable names by indices.

primrec *compE*$_1$ :: *vname list ⇒ expr ⇒ expr*$_1$
and *compEs*$_1$ :: *vname list ⇒ expr list ⇒ expr*$_1$ *list* **where**
compE$_1$ *Vs (new C) = new C*
| *compE*$_1$ *Vs (Cast C e) = Cast C (compE*$_1$ *Vs e)*
| *compE*$_1$ *Vs (Val v) = Val v*
| *compE*$_1$ *Vs (e*$_1$ *«bop» e*$_2$*) = (compE*$_1$ *Vs e*$_1$*) «bop» (compE*$_1$ *Vs e*$_2$*)*
| *compE*$_1$ *Vs (Var V) = Var(last-index Vs V)*
| *compE*$_1$ *Vs (V:=e) = (last-index Vs V):= (compE*$_1$ *Vs e)*
| *compE*$_1$ *Vs (e·F{D}) = (compE*$_1$ *Vs e)·F{D}*
| *compE*$_1$ *Vs (C·*$_s$*F{D}) = C·*$_s$*F{D}*

| $compE_1\ Vs\ (e_1{\cdot}F\{D\}{:=}e_2) = (compE_1\ Vs\ e_1){\cdot}F\{D\} := (compE_1\ Vs\ e_2)$
| $compE_1\ Vs\ (C{\cdot}_sF\{D\}{:=}e_2) = C{\cdot}_sF\{D\} := (compE_1\ Vs\ e_2)$
| $compE_1\ Vs\ (e{\cdot}M(es)) = (compE_1\ Vs\ e){\cdot}M(compEs_1\ Vs\ es)$
| $compE_1\ Vs\ (C{\cdot}_sM(es)) = C{\cdot}_sM(compEs_1\ Vs\ es)$
| $compE_1\ Vs\ \{V{:}T;\ e\} = \{(size\ Vs){:}T;\ compE_1\ (Vs@[V])\ e\}$
| $compE_1\ Vs\ (e_1;;e_2) = (compE_1\ Vs\ e_1);;(compE_1\ Vs\ e_2)$
| $compE_1\ Vs\ (if\ (e)\ e_1\ else\ e_2) = if\ (compE_1\ Vs\ e)\ (compE_1\ Vs\ e_1)\ else\ (compE_1\ Vs\ e_2)$
| $compE_1\ Vs\ (while\ (e)\ c) = while\ (compE_1\ Vs\ e)\ (compE_1\ Vs\ c)$
| $compE_1\ Vs\ (throw\ e) = throw\ (compE_1\ Vs\ e)$
| $compE_1\ Vs\ (try\ e_1\ catch(C\ V)\ e_2) =$
  $try(compE_1\ Vs\ e_1)\ catch(C\ (size\ Vs))\ (compE_1\ (Vs@[V])\ e_2)$
| $compE_1\ Vs\ (INIT\ C\ (Cs,b) \leftarrow e) = INIT\ C\ (Cs,b) \leftarrow (compE_1\ Vs\ e)$
| $compE_1\ Vs\ (RI(C,e);Cs \leftarrow e') = RI(C,(compE_1\ Vs\ e));Cs \leftarrow (compE_1\ Vs\ e')$

| $compEs_1\ Vs\ [] = []$
| $compEs_1\ Vs\ (e\#es) = compE_1\ Vs\ e\ \#\ compEs_1\ Vs\ es$

lemma [*simp*]: $compEs_1\ Vs\ es = map\ (compE_1\ Vs)\ es\langle proof\rangle$
lemma [*simp*]: $\bigwedge Vs.\ sub\text{-}RI\ (compE_1\ Vs\ e) = sub\text{-}RI\ e$
and [*simp*]: $\bigwedge Vs.\ sub\text{-}RIs\ (compEs_1\ Vs\ es) = sub\text{-}RIs\ es$
$\langle proof\rangle$

primrec $fin_1{::}\ expr \Rightarrow expr_1$ **where**
$fin_1(Val\ v) = Val\ v$
| $fin_1(throw\ e) = throw(fin_1\ e)$

lemma *comp-final*: $final\ e \Longrightarrow compE_1\ Vs\ e = fin_1\ e\langle proof\rangle$

lemma [*simp*]:
$\bigwedge Vs.\ max\text{-}vars\ (compE_1\ Vs\ e) = max\text{-}vars\ e$
and $\bigwedge Vs.\ max\text{-}varss\ (compEs_1\ Vs\ es) = max\text{-}varss\ es\langle proof\rangle$

Compiling programs:

definition $compP_1 :: J\text{-}prog \Rightarrow J_1\text{-}prog$
where
$compP_1 \equiv compP\ (\lambda b\ (pns,body).\ compE_1\ (case\ b\ of\ NonStatic \Rightarrow this\#pns \mid Static \Rightarrow pns)\ body)$

end

3.5 Correctness of Stage 1

theory *Correctness1*
imports *J1WellForm Compiler1*
begin

3.5.1 Correctness of program compilation

primrec $unmod :: expr_1 \Rightarrow nat \Rightarrow bool$
and $unmods :: expr_1\ list \Rightarrow nat \Rightarrow bool$ **where**
$unmod\ (new\ C)\ i = True$ |
$unmod\ (Cast\ C\ e)\ i = unmod\ e\ i$ |
$unmod\ (Val\ v)\ i = True$ |
$unmod\ (e_1\ «bop»\ e_2)\ i = (unmod\ e_1\ i \wedge unmod\ e_2\ i)$ |
$unmod\ (Var\ i)\ j = True$ |

unmod (*i*:=*e*) *j* = (*i* ≠ *j* ∧ *unmod e j*) |
unmod (*e*·*F*{*D*}) *i* = *unmod e i* |
unmod (*C*·$_s$*F*{*D*}) *i* = *True* |
unmod (*e*$_1$·*F*{*D*}:=*e*$_2$) *i* = (*unmod e*$_1$ *i* ∧ *unmod e*$_2$ *i*) |
unmod (*C*·$_s$*F*{*D*}:=*e*$_2$) *i* = *unmod e*$_2$ *i* |
unmod (*e*·*M*(*es*)) *i* = (*unmod e i* ∧ *unmods es i*) |
unmod (*C*·$_s$*M*(*es*)) *i* = *unmods es i* |
unmod {*j*:*T*; *e*} *i* = *unmod e i* |
unmod (*e*$_1$;;*e*$_2$) *i* = (*unmod e*$_1$ *i* ∧ *unmod e*$_2$ *i*) |
unmod (*if* (*e*) *e*$_1$ *else e*$_2$) *i* = (*unmod e i* ∧ *unmod e*$_1$ *i* ∧ *unmod e*$_2$ *i*) |
unmod (*while* (*e*) *c*) *i* = (*unmod e i* ∧ *unmod c i*) |
unmod (*throw e*) *i* = *unmod e i* |
unmod (*try e*$_1$ *catch*(*C i*) *e*$_2$) *j* = (*unmod e*$_1$ *j* ∧ (*if i*=*j then False else unmod e*$_2$ *j*)) |
unmod (*INIT C* (*Cs*,*b*) ← *e*) *i* = *unmod e i* |
unmod (*RI*(*C*,*e*);*Cs* ← *e*′) *i* = (*unmod e i* ∧ *unmod e*′ *i*) |

unmods ([]) *i* = *True* |
unmods (*e*#*es*) *i* = (*unmod e i* ∧ *unmods es i*)

lemma *hidden-unmod*: ⋀*Vs*. *hidden Vs i* ⟹ *unmod* (*compE*$_1$ *Vs e*) *i* **and**
⋀*Vs*. *hidden Vs i* ⟹ *unmods* (*compEs*$_1$ *Vs es*) *i*⟨*proof*⟩

lemma *eval*$_1$*-preserves-unmod*:
⟦ *P* ⊢$_1$ ⟨*e*,(*h*,*ls*,*sh*)⟩ ⇒ ⟨*e*′,(*h*′,*ls*′,*sh*′)⟩; *unmod e i*; *i* < *size ls* ⟧
⟹ *ls* ! *i* = *ls*′ ! *i*
and ⟦ *P* ⊢$_1$ ⟨*es*,(*h*,*ls*,*sh*)⟩ [⇒] ⟨*es*′,(*h*′,*ls*′,*sh*′)⟩; *unmods es i*; *i* < *size ls* ⟧
⟹ *ls* ! *i* = *ls*′ ! *i*⟨*proof*⟩

lemma *LAss-lem*:
⟦*x* ∈ *set xs*; *size xs* ≤ *size ys* ⟧
⟹ *m*$_1$ ⊆$_m$ *m*$_2$(*xs*[↦]*ys*) ⟹ *m*$_1$(*x*↦*y*) ⊆$_m$ *m*$_2$(*xs*[↦]*ys*[*last-index xs x* := *y*])⟨*proof*⟩**lemma** *Block-lem*:
fixes *l* :: *'a* ⇀ *'b*
assumes *0*: *l* ⊆$_m$ [*Vs* [↦] *ls*]
and *1*: *l*′ ⊆$_m$ [*Vs* [↦] *ls*′, *V*↦*v*]
and *hidden*: *V* ∈ *set Vs* ⟹ *ls* ! *last-index Vs V* = *ls*′ ! *last-index Vs V*
and *size*: *size ls* = *size ls*′ *size Vs* < *size ls*′
shows *l*′(*V* := *l V*) ⊆$_m$ [*Vs* [↦] *ls*′]⟨*proof*⟩

The main theorem:

theorem assumes *wf*: *wwf-J-prog P*
shows *eval*$_1$*-eval*: *P* ⊢ ⟨*e*,(*h*,*l*,*sh*)⟩ ⇒ ⟨*e*′,(*h*′,*l*′,*sh*′)⟩
⟹ (⋀*Vs ls*. ⟦ *fv e* ⊆ *set Vs*; *l* ⊆$_m$ [*Vs*[↦]*ls*]; *size Vs* + *max-vars e* ≤ *size ls* ⟧
⟹ ∃ *ls*′. *compP*$_1$ *P* ⊢$_1$ ⟨*compE*$_1$ *Vs e*,(*h*,*ls*,*sh*)⟩ ⇒ ⟨*fin*$_1$ *e*′,(*h*′,*ls*′,*sh*′)⟩ ∧ *l*′ ⊆$_m$ [*Vs*[↦]*ls*′])
and *evals*$_1$*-evals*: *P* ⊢ ⟨*es*,(*h*,*l*,*sh*)⟩ [⇒] ⟨*es*′,(*h*′,*l*′,*sh*′)⟩
⟹ (⋀*Vs ls*. ⟦ *fvs es* ⊆ *set Vs*; *l* ⊆$_m$ [*Vs*[↦]*ls*]; *size Vs* + *max-varss es* ≤ *size ls* ⟧
⟹ ∃ *ls*′. *compP*$_1$ *P* ⊢$_1$ ⟨*compEs*$_1$ *Vs es*,(*h*,*ls*,*sh*)⟩ [⇒] ⟨*compEs*$_1$ *Vs es*′,(*h*′,*ls*′,*sh*′)⟩ ∧
l′ ⊆$_m$ [*Vs*[↦]*ls*′])⟨*proof*⟩

3.5.2 Preservation of well-formedness

The compiler preserves well-formedness. Is less trivial than it may appear. We start with two simple properties: preservation of well-typedness

lemma *compE*$_1$*-pres-wt*: ⋀*Vs Ts U*.
⟦ *P*,[*Vs*[↦]*Ts*] ⊢ *e* :: *U*; *size Ts* = *size Vs* ⟧

$\Longrightarrow$ *compP f P*,*Ts* $\vdash_1$ *compE*$_1$ *Vs e* :: *U*
and $\bigwedge$*Vs Ts Us*.
$[\![$ *P*,$[$*Vs*$[\mapsto]$*Ts*$]$ $\vdash$ *es* [::] *Us*; *size Ts* = *size Vs* $]\!]$
$\Longrightarrow$ *compP f P*,*Ts* $\vdash_1$ *compEs*$_1$ *Vs es* [::] *Us*$\langle$*proof*$\rangle$

and the correct block numbering:

lemma $\mathcal{B}$: $\bigwedge$*Vs n*. *size Vs* = *n* $\Longrightarrow$ $\mathcal{B}$ (*compE*$_1$ *Vs e*) *n*
and $\mathcal{B}s$: $\bigwedge$*Vs n*. *size Vs* = *n* $\Longrightarrow$ $\mathcal{B}s$ (*compEs*$_1$ *Vs es*) *n*$\langle$*proof*$\rangle$

The main complication is preservation of definite assignment $\mathcal{D}$.

lemma *image-last-index*: *A* $\subseteq$ *set*(*xs*@[*x*]) $\Longrightarrow$ *last-index* (*xs* @ [*x*]) ‘ *A* =
(*if x* $\in$ *A then insert* (*size xs*) (*last-index xs* ‘ (*A*$-${*x*})) *else last-index xs* ‘ *A*)$\langle$*proof*$\rangle$

lemma *A-compE*$_1$*-None*[*simp*]:
$\bigwedge$*Vs*. $\mathcal{A}$ *e* = *None* $\Longrightarrow$ $\mathcal{A}$ (*compE*$_1$ *Vs e*) = *None*
and $\bigwedge$*Vs*. $\mathcal{A}s$ *es* = *None* $\Longrightarrow$ $\mathcal{A}s$ (*compEs*$_1$ *Vs es*) = *None*$\langle$*proof*$\rangle$

lemma *A-compE*$_1$:
$\bigwedge$*A Vs*. $[\![$ $\mathcal{A}$ *e* = $\lfloor$*A*$\rfloor$; *fv e* $\subseteq$ *set Vs* $]\!]$ $\Longrightarrow$ $\mathcal{A}$ (*compE*$_1$ *Vs e*) = $\lfloor$*last-index Vs* ‘ *A*$\rfloor$
and $\bigwedge$*A Vs*. $[\![$ $\mathcal{A}s$ *es* = $\lfloor$*A*$\rfloor$; *fvs es* $\subseteq$ *set Vs* $]\!]$ $\Longrightarrow$ $\mathcal{A}s$ (*compEs*$_1$ *Vs es*) = $\lfloor$*last-index Vs* ‘ *A*$\rfloor\langle$*proof*$\rangle$

lemma *D-None*[*iff*]: $\mathcal{D}$ (*e*::*'a exp*) *None* **and** [*iff*]: $\mathcal{D}s$ (*es*::*'a exp list*) *None*$\langle$*proof*$\rangle$

lemma *D-last-index-compE*$_1$:
$\bigwedge$*A Vs*. $[\![$ *A* $\subseteq$ *set Vs*; *fv e* $\subseteq$ *set Vs* $]\!]$ $\Longrightarrow$
$\mathcal{D}$ *e* $\lfloor$*A*$\rfloor$ $\Longrightarrow$ $\mathcal{D}$ (*compE*$_1$ *Vs e*) $\lfloor$*last-index Vs* ‘ *A*$\rfloor$
and $\bigwedge$*A Vs*. $[\![$ *A* $\subseteq$ *set Vs*; *fvs es* $\subseteq$ *set Vs* $]\!]$ $\Longrightarrow$
$\mathcal{D}s$ *es* $\lfloor$*A*$\rfloor$ $\Longrightarrow$ $\mathcal{D}s$ (*compEs*$_1$ *Vs es*) $\lfloor$*last-index Vs* ‘ *A*$\rfloor\langle$*proof*$\rangle$

lemma *last-index-image-set*: *distinct xs* $\Longrightarrow$ *last-index xs* ‘ *set xs* = {..<*size xs*}$\langle$*proof*$\rangle$

lemma *D-compE*$_1$:
$[\![$ $\mathcal{D}$ *e* $\lfloor$*set Vs*$\rfloor$; *fv e* $\subseteq$ *set Vs*; *distinct Vs* $]\!]$ $\Longrightarrow$ $\mathcal{D}$ (*compE*$_1$ *Vs e*) $\lfloor${..<*length Vs*}$\rfloor\langle$*proof*$\rangle$

lemma *D-compE*$_1$*'*:
assumes $\mathcal{D}$ *e* $\lfloor$*set*(*V*#*Vs*)$\rfloor$ **and** *fv e* $\subseteq$ *set*(*V*#*Vs*) **and** *distinct*(*V*#*Vs*)
shows $\mathcal{D}$ (*compE*$_1$ (*V*#*Vs*) *e*) $\lfloor${..*length Vs*}$\rfloor\langle$*proof*$\rangle$
lemma *compP*$_1$*-pres-wf*: *wf-J-prog P* $\Longrightarrow$ *wf-J*$_1$*-prog* (*compP*$_1$ *P*)$\langle$*proof*$\rangle$

end

3.6 Compilation Stage 2

theory *Compiler2*
imports *PCompiler J1* ../*JVM*/*JVMExec*
begin

lemma *bop-expr-length-aux* [*simp*]:
length (*case bop of Eq* $\Rightarrow$ [*CmpEq*] | *Add* $\Rightarrow$ [*IAdd*]) = *Suc 0*
$\langle$*proof*$\rangle$

primrec *compE*$_2$ :: *expr*$_1$ $\Rightarrow$ *instr list*
and *compEs*$_2$ :: *expr*$_1$ *list* $\Rightarrow$ *instr list* **where**
compE$_2$ (*new C*) = [*New C*]

```
| compE₂ (Cast C e) = compE₂ e @ [Checkcast C]
| compE₂ (Val v) = [Push v]
| compE₂ (e₁ «bop» e₂) = compE₂ e₁ @ compE₂ e₂ @
  (case bop of Eq ⇒ [CmpEq]
             | Add ⇒ [IAdd])
| compE₂ (Var i) = [Load i]
| compE₂ (i:=e) = compE₂ e @ [Store i, Push Unit]
| compE₂ (e·F{D}) = compE₂ e @ [Getfield F D]
| compE₂ (C·ₛF{D}) = [Getstatic C F D]
| compE₂ (e₁·F{D} := e₂) =
      compE₂ e₁ @ compE₂ e₂ @ [Putfield F D, Push Unit]
| compE₂ (C·ₛF{D} := e₂) =
      compE₂ e₂ @ [Putstatic C F D, Push Unit]
| compE₂ (e·M(es)) = compE₂ e @ compEs₂ es @ [Invoke M (size es)]
| compE₂ (C·ₛM(es)) = compEs₂ es @ [Invokestatic C M (size es)]
| compE₂ ({i:T; e}) = compE₂ e
| compE₂ (e₁;;e₂) = compE₂ e₁ @ [Pop] @ compE₂ e₂
| compE₂ (if (e) e₁ else e₂) =
      (let cnd  = compE₂ e;
           thn  = compE₂ e₁;
           els  = compE₂ e₂;
           test = IfFalse (int(size thn + 2));
           thnex = Goto (int(size els + 1))
       in cnd @ [test] @ thn @ [thnex] @ els)
| compE₂ (while (e) c) =
      (let cnd  = compE₂ e;
           bdy  = compE₂ c;
           test = IfFalse (int(size bdy + 3));
           loop = Goto (−int(size bdy + size cnd + 2))
       in cnd @ [test] @ bdy @ [Pop] @ [loop] @ [Push Unit])
| compE₂ (throw e) = compE₂ e @ [instr.Throw]
| compE₂ (try e₁ catch(C i) e₂) =
  (let catch = compE₂ e₂
   in compE₂ e₁ @ [Goto (int(size catch)+2), Store i] @ catch)
| compE₂ (INIT C (Cs,b) ← e) = []
| compE₂ (RI(C,e);Cs ← e′) = []

| compEs₂ []     = []
| compEs₂ (e#es) = compE₂ e @ compEs₂ es
```

Compilation of exception table. Is given start address of code to compute absolute addresses necessary in exception table.

```
primrec compxE₂  :: expr₁       ⇒ pc ⇒ nat ⇒ ex-table
  and compxEs₂ :: expr₁ list ⇒ pc ⇒ nat ⇒ ex-table where
  compxE₂ (new C) pc d = []
| compxE₂ (Cast C e) pc d = compxE₂ e pc d
| compxE₂ (Val v) pc d = []
| compxE₂ (e₁ «bop» e₂) pc d =
   compxE₂ e₁ pc d @ compxE₂ e₂ (pc + size(compE₂ e₁)) (d+1)
| compxE₂ (Var i) pc d = []
| compxE₂ (i:=e) pc d = compxE₂ e pc d
| compxE₂ (e·F{D}) pc d = compxE₂ e pc d
| compxE₂ (C·ₛF{D}) pc d = []
| compxE₂ (e₁·F{D} := e₂) pc d =
```

$$
\begin{array}{l}
\quad compxE_2\ e_1\ pc\ d\ @\ compxE_2\ e_2\ (pc + size(compE_2\ e_1))\ (d{+}1) \\
\mid compxE_2\ (C{\cdot_s}F\{D\} := e_2)\ pc\ d = compxE_2\ e_2\ pc\ d \\
\mid compxE_2\ (e{\cdot}M(es))\ pc\ d = \\
\quad compxE_2\ e\ pc\ d\ @\ compxEs_2\ es\ (pc + size(compE_2\ e))\ (d{+}1) \\
\mid compxE_2\ (C{\cdot_s}M(es))\ pc\ d = compxEs_2\ es\ pc\ d \\
\mid compxE_2\ (\{i{:}T;\ e\})\ pc\ d = compxE_2\ e\ pc\ d \\
\mid compxE_2\ (e_1;;e_2)\ pc\ d = \\
\quad compxE_2\ e_1\ pc\ d\ @\ compxE_2\ e_2\ (pc{+}size(compE_2\ e_1){+}1)\ d \\
\mid compxE_2\ (if\ (e)\ e_1\ else\ e_2)\ pc\ d = \\
\qquad (let\ pc_1\ \ = pc + size(compE_2\ e) + 1; \\
\qquad\quad pc_2\ \ = pc_1 + size(compE_2\ e_1) + 1 \\
\qquad\ in\ compxE_2\ e\ pc\ d\ @\ compxE_2\ e_1\ pc_1\ d\ @\ compxE_2\ e_2\ pc_2\ d) \\
\mid compxE_2\ (while\ (b)\ e)\ pc\ d = \\
\quad compxE_2\ b\ pc\ d\ @\ compxE_2\ e\ (pc{+}size(compE_2\ b){+}1)\ d \\
\mid compxE_2\ (throw\ e)\ pc\ d = compxE_2\ e\ pc\ d \\
\mid compxE_2\ (try\ e_1\ catch(C\ i)\ e_2)\ pc\ d = \\
\quad (let\ pc_1 = pc + size(compE_2\ e_1) \\
\quad\ in\ compxE_2\ e_1\ pc\ d\ @\ compxE_2\ e_2\ (pc_1{+}2)\ d\ @\ [(pc,pc_1,C,pc_1{+}1,d)]) \\
\mid compxE_2\ (INIT\ C\ (Cs,\ b) \leftarrow e)\ pc\ d = [] \\
\mid compxE_2\ (RI(C,\ e);Cs \leftarrow e')\ pc\ d = []
\end{array}
$$

$$
\begin{array}{l}
\mid compxEs_2\ []\ pc\ d\quad = [] \\
\mid compxEs_2\ (e\#es)\ pc\ d = compxE_2\ e\ pc\ d\ @\ compxEs_2\ es\ (pc{+}size(compE_2\ e))\ (d{+}1)
\end{array}
$$

primrec *max-stack* :: $expr_1 \Rightarrow nat$
and *max-stacks* :: $expr_1\ list \Rightarrow nat$ **where**

$$
\begin{array}{l}
max\text{-}stack\ (new\ C) = 1 \\
\mid max\text{-}stack\ (Cast\ C\ e) = max\text{-}stack\ e \\
\mid max\text{-}stack\ (Val\ v) = 1 \\
\mid max\text{-}stack\ (e_1\ «bop»\ e_2) = max\ (max\text{-}stack\ e_1)\ (max\text{-}stack\ e_2) + 1 \\
\mid max\text{-}stack\ (Var\ i) = 1 \\
\mid max\text{-}stack\ (i{:=}e) = max\text{-}stack\ e \\
\mid max\text{-}stack\ (e{\cdot}F\{D\}) = max\text{-}stack\ e \\
\mid max\text{-}stack\ (C{\cdot_s}F\{D\}) = 1 \\
\mid max\text{-}stack\ (e_1{\cdot}F\{D\} := e_2) = max\ (max\text{-}stack\ e_1)\ (max\text{-}stack\ e_2) + 1 \\
\mid max\text{-}stack\ (C{\cdot_s}F\{D\} := e_2) = max\text{-}stack\ e_2 \\
\mid max\text{-}stack\ (e{\cdot}M(es)) = max\ (max\text{-}stack\ e)\ (max\text{-}stacks\ es) + 1 \\
\mid max\text{-}stack\ (C{\cdot_s}M(es)) = max\text{-}stacks\ es + 1 \\
\mid max\text{-}stack\ (\{i{:}T;\ e\}) = max\text{-}stack\ e \\
\mid max\text{-}stack\ (e_1;;e_2) = max\ (max\text{-}stack\ e_1)\ (max\text{-}stack\ e_2) \\
\mid max\text{-}stack\ (if\ (e)\ e_1\ else\ e_2) = \\
\quad max\ (max\text{-}stack\ e)\ (max\ (max\text{-}stack\ e_1)\ (max\text{-}stack\ e_2)) \\
\mid max\text{-}stack\ (while\ (e)\ c) = max\ (max\text{-}stack\ e)\ (max\text{-}stack\ c) \\
\mid max\text{-}stack\ (throw\ e) = max\text{-}stack\ e \\
\mid max\text{-}stack\ (try\ e_1\ catch(C\ i)\ e_2) = max\ (max\text{-}stack\ e_1)\ (max\text{-}stack\ e_2)
\end{array}
$$

$$
\begin{array}{l}
\mid max\text{-}stacks\ [] = 0 \\
\mid max\text{-}stacks\ (e\#es) = max\ (max\text{-}stack\ e)\ (1 + max\text{-}stacks\ es)
\end{array}
$$

lemma *max-stack1'*: $\neg sub\text{-}RI\ e \Longrightarrow 1 \leq max\text{-}stack\ e \langle proof \rangle$
lemma *compE$_2$-not-Nil'*: $\neg sub\text{-}RI\ e \Longrightarrow compE_2\ e \neq [] \langle proof \rangle$
lemma *compE$_2$-nRet*: $\bigwedge i.\ i \in set\ (compE_2\ e_1) \Longrightarrow i \neq Return$
and $\bigwedge i.\ i \in set\ (compEs_2\ es_1) \Longrightarrow i \neq Return$
$\langle proof \rangle$

definition *compMb*$_2$ :: *staticb* ⇒ *expr*$_1$ ⇒ *jvm-method*
where
compMb$_2$ ≡ λ*b body*.
let ins = *compE*$_2$ *body* @ [*Return*];
xt = *compxE*$_2$ *body 0 0*
in (*max-stack body*, *max-vars body*, *ins*, *xt*)

definition *compP*$_2$ :: *J*$_1$*-prog* ⇒ *jvm-prog*
where
compP$_2$ ≡ *compP compMb*$_2$

lemma *compMb*$_2$ [*simp*]:
compMb$_2$ *b e* = (*max-stack e*, *max-vars e*,
compE$_2$ *e* @ [*Return*], *compxE*$_2$ *e 0 0*)⟨*proof*⟩
end

3.7 Correctness of Stage 2

theory *Correctness2*
imports *HOL−Library.Sublist Compiler2 J1WellForm ../J/EConform*
begin

3.7.1 Instruction sequences

How to select individual instructions and subsequences of instructions from a program given the class, method and program counter.

definition *before* :: *jvm-prog* ⇒ *cname* ⇒ *mname* ⇒ *nat* ⇒ *instr list* ⇒ *bool*
(‹(-,-,-,-/ ▷ -)› [*51*,*0*,*0*,*0*,*51*] *50*) **where**
P,*C*,*M*,*pc* ▷ *is* ⟷ *prefix is* (*drop pc* (*instrs-of P C M*))

definition *at* :: *jvm-prog* ⇒ *cname* ⇒ *mname* ⇒ *nat* ⇒ *instr* ⇒ *bool*
(‹(-,-,-,-/ ▷ -)› [*51*,*0*,*0*,*0*,*51*] *50*) **where**
P,*C*,*M*,*pc* ▷ *i* ⟷ (∃ *is*. *drop pc* (*instrs-of P C M*) = *i*#*is*)

lemma [*simp*]: *P*,*C*,*M*,*pc* ▷ []⟨*proof*⟩

lemma [*simp*]: *P*,*C*,*M*,*pc* ▷ (*i*#*is*) = (*P*,*C*,*M*,*pc* ▷ *i* ∧ *P*,*C*,*M*,*pc* + *1* ▷ *is*)⟨*proof*⟩

lemma [*simp*]: *P*,*C*,*M*,*pc* ▷ (*is*$_1$ @ *is*$_2$) = (*P*,*C*,*M*,*pc* ▷ *is*$_1$ ∧ *P*,*C*,*M*,*pc* + *size is*$_1$ ▷ *is*$_2$)⟨*proof*⟩

lemma [*simp*]: *P*,*C*,*M*,*pc* ▷ *i* ⟹ *instrs-of P C M* ! *pc* = *i*⟨*proof*⟩
lemma *beforeM*:
P ⊢ *C sees M*,*b*: *Ts*→*T* = *body in D* ⟹
compP$_2$ *P*,*D*,*M*,*0* ▷ *compE*$_2$ *body* @ [*Return*]⟨*proof*⟩

This lemma executes a single instruction by rewriting:

lemma [*simp*]:
P,*C*,*M*,*pc* ▷ *instr* ⟹
(*P* ⊢ (*None*, *h*, (*vs*,*ls*,*C*,*M*,*pc*,*ics*) # *frs*, *sh*) −*jvm*→ *σ*′) =
((*None*, *h*, (*vs*,*ls*,*C*,*M*,*pc*,*ics*) # *frs*, *sh*) = *σ*′ ∨

$(\exists\,\sigma.\ exec(P,(None,\ h,\ (vs,ls,C,M,pc,ics)\ \#\ frs,\ sh)) = Some\ \sigma\ \wedge\ P \vdash \sigma\ -jvm\rightarrow\ \sigma')\langle proof\rangle$

3.7.2 Exception tables

definition *pcs* :: *ex-table* ⇒ *nat set*
where
pcs xt ≡ $\bigcup(f,t,C,h,d) \in set\ xt.\ \{f\ ..<\ t\}$

lemma *pcs-subset*:
shows $(\bigwedge pc\ d.\ pcs(compxE_2\ e\ pc\ d) \subseteq \{pc..<pc+size(compE_2\ e)\})$
and $(\bigwedge pc\ d.\ pcs(compxEs_2\ es\ pc\ d) \subseteq \{pc..<pc+size(compEs_2\ es)\})\langle proof\rangle$

lemma [*simp*]: *pcs* [] = {}⟨*proof*⟩

lemma [*simp*]: *pcs* (*x*#*xt*) = {*fst x* ..< *fst*(*snd x*)} ∪ *pcs xt*⟨*proof*⟩

lemma [*simp*]: $pcs(xt_1\ @\ xt_2) = pcs\ xt_1 \cup pcs\ xt_2\langle proof\rangle$

lemma [*simp*]: $pc < pc_0 \vee pc_0+size(compE_2\ e) \leq pc \Longrightarrow pc \notin pcs(compxE_2\ e\ pc_0\ d)\langle proof\rangle$

lemma [*simp*]: $pc < pc_0 \vee pc_0+size(compEs_2\ es) \leq pc \Longrightarrow pc \notin pcs(compxEs_2\ es\ pc_0\ d)\langle proof\rangle$

lemma [*simp*]: $pc_1 + size(compE_2\ e_1) \leq pc_2 \Longrightarrow pcs(compxE_2\ e_1\ pc_1\ d_1) \cap pcs(compxE_2\ e_2\ pc_2\ d_2)$
= {}⟨*proof*⟩

lemma [*simp*]: $pc_1 + size(compE_2\ e) \leq pc_2 \Longrightarrow pcs(compxE_2\ e\ pc_1\ d_1) \cap pcs(compxEs_2\ es\ pc_2\ d_2)$
= {}⟨*proof*⟩

lemma [*simp*]:
pc ∉ *pcs xt*$_0$ ⟹ *match-ex-table P C pc* (*xt*$_0$ @ *xt*$_1$) = *match-ex-table P C pc xt*$_1$⟨*proof*⟩

lemma [*simp*]: ⟦ *x* ∈ *set xt*; *pc* ∉ *pcs xt* ⟧ ⟹ ¬ *matches-ex-entry P D pc x*⟨*proof*⟩

lemma [*simp*]:
assumes *xe*: *xe* ∈ *set*(*compxE*$_2$ *e pc d*) **and** *outside*: *pc*′ < *pc* ∨ *pc*+*size*(*compE*$_2$ *e*) ≤ *pc*′
shows ¬ *matches-ex-entry P C pc*′ *xe*⟨*proof*⟩

lemma [*simp*]:
assumes *xe*: *xe* ∈ *set*(*compxEs*$_2$ *es pc d*) **and** *outside*: *pc*′ < *pc* ∨ *pc*+*size*(*compEs*$_2$ *es*) ≤ *pc*′
shows ¬ *matches-ex-entry P C pc*′ *xe*⟨*proof*⟩

lemma *match-ex-table-app*[*simp*]:
∀ *xte* ∈ *set xt*$_1$. ¬ *matches-ex-entry P D pc xte* ⟹
match-ex-table P D pc (*xt*$_1$ @ *xt*) = *match-ex-table P D pc xt*⟨*proof*⟩

lemma [*simp*]:
∀ *x* ∈ *set xtab*. ¬ *matches-ex-entry P C pc x* ⟹
match-ex-table P C pc xtab = *None*⟨*proof*⟩

lemma *match-ex-entry*:
matches-ex-entry P C pc (*start*, *end*, *catch-type*, *handler*) =
(*start* ≤ *pc* ∧ *pc* < *end* ∧ *P* ⊢ *C* ⪯* *catch-type*)⟨*proof*⟩

definition *caught* :: *jvm-prog* ⇒ *pc* ⇒ *heap* ⇒ *addr* ⇒ *ex-table* ⇒ *bool* **where**

caught P pc h a xt $\longleftrightarrow$
$(\exists$ *entry* $\in$ *set xt. matches-ex-entry P* (*cname-of h a*) *pc entry*)

definition *beforex* :: *jvm-prog* $\Rightarrow$ *cname* $\Rightarrow$ *mname* $\Rightarrow$ *ex-table* $\Rightarrow$ *nat set* $\Rightarrow$ *nat* $\Rightarrow$ *bool*
($\langle$(*2*-,/-,/- $\triangleright$/ - /'/ -,/-)$\rangle$ [*51*,*0*,*0*,*0*,*0*,*51*] *50*) **where**
P,*C*,*M* $\triangleright$ *xt* / *I*,*d* $\longleftrightarrow$
($\exists$ *xt*$_0$ *xt*$_1$. *ex-table-of P C M* = *xt*$_0$ @ *xt* @ *xt*$_1$ $\wedge$ *pcs xt*$_0$ $\cap$ *I* = {} $\wedge$ *pcs xt* $\subseteq$ *I* $\wedge$
($\forall$ *pc* $\in$ *I*. $\forall$ *C pc*' *d*'. *match-ex-table P C pc xt*$_1$ = $\lfloor$(*pc*',*d*')$\rfloor$ $\longrightarrow$ *d*' $\leq$ *d*))

definition *dummyx* :: *jvm-prog* $\Rightarrow$ *cname* $\Rightarrow$ *mname* $\Rightarrow$ *ex-table* $\Rightarrow$ *nat set* $\Rightarrow$ *nat* $\Rightarrow$ *bool* ($\langle$(*2*-,-,- $\triangleright$/ - /'-,-)$\rangle$ [*51*,*0*,*0*,*0*,*0*,*51*] *50*) **where**
P,*C*,*M* $\triangleright$ *xt*/*I*,*d* $\longleftrightarrow$ *P*,*C*,*M* $\triangleright$ *xt*/*I*,*d*

abbreviation
beforex$_0$ *P C M d I xt xt*$_0$ *xt*$_1$
$\equiv$ *ex-table-of P C M* = *xt*$_0$ @ *xt* @ *xt*$_1$ $\wedge$ *pcs xt*$_0$ $\cap$ *I* = {}
$\wedge$ *pcs xt* $\subseteq$ *I* $\wedge$ ($\forall$ *pc* $\in$ *I*. $\forall$ *C pc*' *d*'. *match-ex-table P C pc xt*$_1$ = $\lfloor$(*pc*',*d*')$\rfloor$ $\longrightarrow$ *d*' $\leq$ *d*)

lemma *beforex-beforex*$_0$*-eq*:
P,*C*,*M* $\triangleright$ *xt* / *I*,*d* $\equiv$ $\exists$ *xt*$_0$ *xt*$_1$. *beforex*$_0$ *P C M d I xt xt*$_0$ *xt*$_1$
$\langle$*proof*$\rangle$

lemma *beforexD1*: *P*,*C*,*M* $\triangleright$ *xt* / *I*,*d* $\Longrightarrow$ *pcs xt* $\subseteq$ *I*$\langle$*proof*$\rangle$

lemma *beforex-mono*: $[\![$ *P*,*C*,*M* $\triangleright$ *xt*/*I*,*d*'; *d*' $\leq$ *d* $]\!]$ $\Longrightarrow$ *P*,*C*,*M* $\triangleright$ *xt*/*I*,*d*$\langle$*proof*$\rangle$

lemma [*simp*]: *P*,*C*,*M* $\triangleright$ *xt*/*I*,*d* $\Longrightarrow$ *P*,*C*,*M* $\triangleright$ *xt*/*I*,*Suc d*$\langle$*proof*$\rangle$

lemma *beforex-append*[*simp*]:
pcs xt$_1$ $\cap$ *pcs xt*$_2$ = {} $\Longrightarrow$
P,*C*,*M* $\triangleright$ *xt*$_1$ @ *xt*$_2$/*I*,*d* =
(*P*,*C*,*M* $\triangleright$ *xt*$_1$/*I*$-$*pcs xt*$_2$,*d* $\wedge$ *P*,*C*,*M* $\triangleright$ *xt*$_2$/*I*$-$*pcs xt*$_1$,*d* $\wedge$ *P*,*C*,*M* $\triangleright$ *xt*$_1$@*xt*$_2$/*I*,*d*)$\langle$*proof*$\rangle$

lemma *beforex-appendD1*:
assumes *bx*: *P*,*C*,*M* $\triangleright$ *xt*$_1$ @ *xt*$_2$ @ [(*f*,*t*,*D*,*h*,*d*)] / *I*,*d*
and *pcs*: *pcs xt*$_1$ $\subseteq$ *J* **and** *JI*: *J* $\subseteq$ *I* **and** *Jpcs*: *J* $\cap$ *pcs xt*$_2$ = {}
shows *P*,*C*,*M* $\triangleright$ *xt*$_1$ / *J*,*d*$\langle$*proof*$\rangle$

lemma *beforex-appendD2*:
assumes *bx*: *P*,*C*,*M* $\triangleright$ *xt*$_1$ @ *xt*$_2$ @ [(*f*,*t*,*D*,*h*,*d*)] / *I*,*d*
and *pcs*: *pcs xt*$_2$ $\subseteq$ *J* **and** *JI*: *J* $\subseteq$ *I* **and** *Jpcs*: *J* $\cap$ *pcs xt*$_1$ = {}
shows *P*,*C*,*M* $\triangleright$ *xt*$_2$ / *J*,*d*$\langle$*proof*$\rangle$

lemma *beforexM*:
P $\vdash$ *C sees M*,*b*: *Ts*$\rightarrow$*T* = *body in D* $\Longrightarrow$ *compP*$_2$ *P*,*D*,*M* $\triangleright$ *compxE*$_2$ *body 0 0*/{..<*size*(*compE*$_2$ *body*)},*0*$\langle$*proof*$\rangle$

lemma *match-ex-table-SomeD2*:
assumes *met*: *match-ex-table P D pc* (*ex-table-of P C M*) = $\lfloor$(*pc*',*d*')$\rfloor$
and *bx*: *P*,*C*,*M* $\triangleright$ *xt*/*I*,*d*
and *nmet*: $\forall$ *x* $\in$ *set xt*. $\neg$ *matches-ex-entry P D pc x* **and** *pcI*: *pc* $\in$ *I*
shows *d*' $\leq$ *d*$\langle$*proof*$\rangle$

lemma *match-ex-table-SomeD1*:

⟦ *match-ex-table P D pc* (*ex-table-of P C M*) = ⌊(*pc'*,*d'*)⌋;
P,*C*,*M* ▷ *xt* / *I*,*d*; *pc* ∈ *I*; *pc* ∉ *pcs xt* ⟧ ⟹ *d'* ≤ *d*⟨*proof*⟩

3.7.3 The correctness proof

definition
handle :: *jvm-prog* ⇒ *cname* ⇒ *mname* ⇒ *addr* ⇒ *heap* ⇒ *val list* ⇒ *val list* ⇒ *nat* ⇒ *init-call-status* ⇒ *frame list* ⇒ *sheap*
⇒ *jvm-state* **where**
handle P C M a h vs ls pc ics frs sh = *find-handler P a h* ((*vs*,*ls*,*C*,*M*,*pc*,*ics*) # *frs*) *sh*

lemma *aux-isin*[*simp*]: ⟦ *B* ⊆ *A*; *a* ∈ *B* ⟧ ⟹ *a* ∈ *A*⟨*proof*⟩
lemma *handle-frs-tl-neq*:
ics-of f ≠ *No-ics*
⟹ (*xp*, *h*, *f*#*frs*, *sh*) ≠ *handle P C M xa h' vs l pc ics frs sh'*
⟨*proof*⟩

Correctness proof inductive hypothesis

fun *calling-to-called* :: *frame* ⇒ *frame* **where**
calling-to-called (*stk*,*loc*,*D*,*M*,*pc*,*ics*) = (*stk*,*loc*,*D*,*M*,*pc*,*case ics of Calling C Cs* ⇒ *Called* (*C*#*Cs*))

fun *calling-to-scalled* :: *frame* ⇒ *frame* **where**
calling-to-scalled (*stk*,*loc*,*D*,*M*,*pc*,*ics*) = (*stk*,*loc*,*D*,*M*,*pc*,*case ics of Calling C Cs* ⇒ *Called Cs*)

fun *calling-to-calling* :: *frame* ⇒ *cname* ⇒ *frame* **where**
calling-to-calling (*stk*,*loc*,*D*,*M*,*pc*,*ics*) *C'* = (*stk*,*loc*,*D*,*M*,*pc*,*case ics of Calling C Cs* ⇒ *Calling C'* (*C*#*Cs*))

fun *calling-to-throwing* :: *frame* ⇒ *addr* ⇒ *frame* **where**
calling-to-throwing (*stk*,*loc*,*D*,*M*,*pc*,*ics*) *a* = (*stk*,*loc*,*D*,*M*,*pc*,*case ics of Calling C Cs* ⇒ *Throwing* (*C*#*Cs*) *a*)

fun *calling-to-sthrowing* :: *frame* ⇒ *addr* ⇒ *frame* **where**
calling-to-sthrowing (*stk*,*loc*,*D*,*M*,*pc*,*ics*) *a* = (*stk*,*loc*,*D*,*M*,*pc*,*case ics of Calling C Cs* ⇒ *Throwing Cs a*)

— pieces of the correctness proof's inductive hypothesis, which depend on the expression being compiled)

fun *Jcc-cond* :: J_1-*prog* ⇒ *ty list* ⇒ *cname* ⇒ *mname* ⇒ *val list* ⇒ *pc* ⇒ *init-call-status*
⇒ *nat set* ⇒ *heap* ⇒ *sheap* ⇒ *expr*$_1$ ⇒ *bool* **where**
Jcc-cond P E C M vs pc ics I h sh (*INIT* C_0 (*Cs*,*b*) ← *e'*)
= ((∃ *T*. *P*,*E*,*h*,*sh* ⊢$_1$ *INIT* C_0 (*Cs*,*b*) ← *e'* : *T*) ∧ *unit* = *e'* ∧ *ics* = *No-ics*) |
Jcc-cond P E C M vs pc ics I h sh (*RI*(*C'*,e_0);*Cs* ← *e'*)
= (((*e*$_0$ = *C'*•$_s$*clinit*([]) ∧ (∃ *T*. *P*,*E*,*h*,*sh* ⊢$_1$ *RI*(*C'*,e_0);*Cs* ← *e'*:*T*))
∨ ((∃ *a*. *e*$_0$ = *Throw a*) ∧ (∀ *C* ∈ *set*(*C'*#*Cs*). *is-class P C*)))
∧ *unit* = *e'* ∧ *ics* = *No-ics*) |
Jcc-cond P E C M vs pc ics I h sh (*C'*•$_s$*M'*(*es*))
= (*let e* = (*C'*•$_s$*M'*(*es*))
in if M' = *clinit* ∧ *es* = [] *then* (∃ *T*. *P*,*E*,*h*,*sh* ⊢$_1$ *e*:*T*) ∧ (∃ *Cs*. *ics* = *Called Cs*)
else (*compP*$_2$ *P*,*C*,*M*,*pc* ▷ *compE*$_2$ *e* ∧ *compP*$_2$ *P*,*C*,*M* ▷ *compxE*$_2$ *e pc* (*size vs*)/*I*,*size vs*

```
                ∧ {pc..<pc+size(compE2 e)} ⊆ I ∧ ¬sub-RI e ∧ ics = No-ics)
  ) |
Jcc-cond P E C M vs pc ics I h sh e
 = (compP2 P,C,M,pc ▷ compE2 e ∧ compP2 P,C,M ▷ compxE2 e pc (size vs)/I,size vs
                ∧ {pc..<pc+size(compE2 e)} ⊆ I ∧ ¬sub-RI e ∧ ics = No-ics)


fun Jcc-frames :: jvm-prog ⇒ cname ⇒ mname ⇒ val list ⇒ val list ⇒ pc ⇒ init-call-status
 ⇒ frame list ⇒ expr1 ⇒ frame list where
Jcc-frames P C M vs ls pc ics frs (INIT C0 (C'#Cs,b) ← e')
 = (case b of False ⇒ (vs,ls,C,M,pc,Calling C' Cs) # frs
          | True ⇒ (vs,ls,C,M,pc,Called (C'#Cs)) # frs
  ) |
Jcc-frames P C M vs ls pc ics frs (INIT C0 (Nil,b) ← e')
 = (vs,ls,C,M,pc,Called [])#frs |
Jcc-frames P C M vs ls pc ics frs (RI(C',e0);Cs ← e')
 = (case e0 of Throw a ⇒ (vs,ls,C,M,pc,Throwing (C'#Cs) a) # frs
          | - ⇒ (vs,ls,C,M,pc,Called (C'#Cs)) # frs ) |
Jcc-frames P C M vs ls pc ics frs (C'•sM'(es))
 = (if M' = clinit ∧ es = []
   then create-init-frame P C'#(vs,ls,C,M,pc,ics)#frs
   else (vs,ls,C,M,pc,ics)#frs
  ) |
Jcc-frames P C M vs ls pc ics frs e
 = (vs,ls,C,M,pc,ics)#frs

fun Jcc-rhs :: J1-prog ⇒ ty list ⇒ cname ⇒ mname ⇒ val list ⇒ val list ⇒ pc ⇒ init-call-status
 ⇒ frame list ⇒ heap ⇒ val list ⇒ sheap ⇒ val ⇒ expr1 ⇒ jvm-state where
Jcc-rhs P E C M vs ls pc ics frs h' ls' sh' v (INIT C0 (Cs,b) ← e')
 = (None,h',(vs,ls,C,M,pc,Called [])#frs,sh') |
Jcc-rhs P E C M vs ls pc ics frs h' ls' sh' v (RI(C',e0);Cs ← e')
 = (None,h',(vs,ls,C,M,pc,Called [])#frs,sh') |
Jcc-rhs P E C M vs ls pc ics frs h' ls' sh' v (C'•sM'(es))
 = (let e = (C'•sM'(es))
   in if M' = clinit ∧ es = []
      then (None,h',(vs,ls,C,M,pc,ics)#frs,sh'(C'↦(fst(the(sh' C')),Done)))
      else (None,h',(v#vs,ls',C,M,pc+size(compE2 e),ics)#frs,sh')
  ) |
Jcc-rhs P E C M vs ls pc ics frs h' ls' sh' v e
 = (None,h',(v#vs,ls',C,M,pc+size(compE2 e),ics)#frs,sh')

fun Jcc-err :: jvm-prog ⇒ cname ⇒ mname ⇒ heap ⇒ val list ⇒ val list ⇒ pc ⇒ init-call-status
 ⇒ frame list ⇒ sheap ⇒ nat set ⇒ heap ⇒ val list ⇒ sheap ⇒ addr ⇒ expr1
 ⇒ bool where
Jcc-err P C M h vs ls pc ics frs sh I h' ls' sh' xa (INIT C0 (Cs,b) ← e')
 = (∃ vs'. P ⊢ (None,h,Jcc-frames P C M vs ls pc ics frs (INIT C0 (Cs,b) ← e'),sh)
        −jvm→ handle P C M xa h' (vs'@vs) ls pc ics frs sh') |
Jcc-err P C M h vs ls pc ics frs sh I h' ls' sh' xa (RI(C',e0);Cs ← e')
 = (∃ vs'. P ⊢ (None,h,Jcc-frames P C M vs ls pc ics frs (RI(C',e0);Cs ← e'),sh)
        −jvm→ handle P C M xa h' (vs'@vs) ls pc ics frs sh') |
Jcc-err P C M h vs ls pc ics frs sh I h' ls' sh' xa (C'•sM'(es))
 = (let e = (C'•sM'(es))
   in if M' = clinit ∧ es = []
      then case ics of
```

Called Cs ⇒ *P* ⊢ (*None*,*h*,*Jcc-frames P C M vs ls pc ics frs e*,*sh*)
−*jvm*→ (*None*,*h'*,(*vs*,*ls*,*C*,*M*,*pc*,*Throwing Cs xa*)#*frs*,(*sh'*(*C'* ↦ (*fst*(*the*(*sh'* *C'*)),*Error*))))
else (∃ *pc*₁. *pc* ≤ *pc*₁ ∧ *pc*₁ < *pc* + *size*(*compE*₂ *e*) ∧
¬ *caught P pc*₁ *h' xa* (*compxE*₂ *e pc* (*size vs*)) ∧
(∃ *vs'*. *P* ⊢ (*None*,*h*,*Jcc-frames P C M vs ls pc ics frs e*,*sh*)
−*jvm*→ *handle P C M xa h'* (*vs'*@*vs*) *ls' pc*₁ *ics frs sh'*))
) |
Jcc-err P C M h vs ls pc ics frs sh I h' ls' sh' xa e
= (∃ *pc*₁. *pc* ≤ *pc*₁ ∧ *pc*₁ < *pc* + *size*(*compE*₂ *e*) ∧
¬ *caught P pc*₁ *h' xa* (*compxE*₂ *e pc* (*size vs*)) ∧
(∃ *vs'*. *P* ⊢ (*None*,*h*,*Jcc-frames P C M vs ls pc ics frs e*,*sh*)
−*jvm*→ *handle P C M xa h'* (*vs'*@*vs*) *ls' pc*₁ *ics frs sh'*))

fun *Jcc-pieces* :: *J*₁*-prog* ⇒ *ty list* ⇒ *cname* ⇒ *mname* ⇒ *heap* ⇒ *val list* ⇒ *val list* ⇒ *pc* ⇒ *init-call-status*
⇒ *frame list* ⇒ *sheap* ⇒ *nat set* ⇒ *heap* ⇒ *val list* ⇒ *sheap* ⇒ *val* ⇒ *addr* ⇒ *expr*₁
⇒ *bool* × *frame list* × *jvm-state* × *bool* **where**
Jcc-pieces P E C M h vs ls pc ics frs sh I h' ls' sh' v xa e
= (*Jcc-cond P E C M vs pc ics I h sh e*, *Jcc-frames* (*compP*₂ *P*) *C M vs ls pc ics frs e*,
Jcc-rhs P E C M vs ls pc ics frs h' ls' sh' v e,
Jcc-err (*compP*₂ *P*) *C M h vs ls pc ics frs sh I h' ls' sh' xa e*)

— *Jcc-pieces* lemmas

lemma *nsub-RI-Jcc-pieces*:
assumes [*simp*]: *P* ≡ *compP*₂ *P*₁
and *nsub*: ¬*sub-RI e*
shows *Jcc-pieces P*₁ *E C M h vs ls pc ics frs sh I h' ls' sh' v xa e*
= (*let cond* = *P*,*C*,*M*,*pc* ▷ *compE*₂ *e* ∧ *P*,*C*,*M* ▷ *compxE*₂ *e pc* (*size vs*)/*I*,*size vs*
∧ {*pc*..<*pc*+*size*(*compE*₂ *e*)} ⊆ *I* ∧ *ics* = *No-ics*;
frs' = (*vs*,*ls*,*C*,*M*,*pc*,*ics*)#*frs*;
rhs = (*None*,*h'*,(*v*#*vs*,*ls'*,*C*,*M*,*pc*+*size*(*compE*₂ *e*),*ics*)#*frs*,*sh'*);
err = (∃ *pc*₁. *pc* ≤ *pc*₁ ∧ *pc*₁ < *pc* + *size*(*compE*₂ *e*) ∧
¬ *caught P pc*₁ *h' xa* (*compxE*₂ *e pc* (*size vs*)) ∧
(∃ *vs'*. *P* ⊢ (*None*,*h*,*frs'*,*sh*) −*jvm*→ *handle P C M xa h'* (*vs'*@*vs*) *ls' pc*₁ *ics frs sh'*))
in (*cond*, *frs'*,*rhs*, *err*)
)
⟨*proof*⟩

lemma *Jcc-pieces-Cast*:
assumes [*simp*]: *P* ≡ *compP*₂ *P*₁
and *Jcc-pieces P*₁ *E C M h*₀ *vs ls*₀ *pc ics frs sh*₀ *I h*₁ *ls*₁ *sh*₁ *v xa* (*Cast C' e*)
= (*True*, *frs*₀, (*xp'*,*h'*,(*v*#*vs'*,*ls'*,*C*₀,*M'*,*pc'*,*ics'*)#*frs'*,*sh'*), *err*)
shows *Jcc-pieces P*₁ *E C M h*₀ *vs ls*₀ *pc ics frs sh*₀ *I h*₁ *ls*₁ *sh*₁ *v' xa e*
= (*True*, *frs*₀, (*xp'*,*h'*,(*v'*#*vs'*,*ls'*,*C*₀,*M'*,*pc'* − 1,*ics'*)#*frs'*,*sh'*),
(∃ *pc*₁. *pc* ≤ *pc*₁ ∧ *pc*₁ < *pc* + *size*(*compE*₂ *e*) ∧
¬ *caught P pc*₁ *h*₁ *xa* (*compxE*₂ *e pc* (*size vs*)) ∧
(∃ *vs'*. *P* ⊢ (*None*,*h*₀,*frs*₀,*sh*₀) −*jvm*→ *handle P C M xa h*₁ (*vs'*@*vs*) *ls*₁ *pc*₁ *ics frs sh*₁)))
⟨*proof*⟩

lemma *Jcc-pieces-BinOp1*:
assumes
*Jcc-pieces P E C M h*₀ *vs ls*₀ *pc ics frs sh*₀ *I h*₂ *ls*₂ *sh*₂ *v xa* (*e* «*bop*» *e'*)

$= (True, frs_0, (xp',h',(v\#vs',ls',C_0,M',pc',ics')\#frs',sh'), err)$
shows $\exists\, err.\ Jcc\text{-}pieces\ P\ E\ C\ M\ h_0\ vs\ ls_0\ pc\ ics\ frs\ sh_0$
$(I - pcs\ (compxE_2\ e'\ (pc + length\ (compE_2\ e))\ (Suc\ (length\ vs')))))\ h_1\ ls_1\ sh_1\ v'\ xa\ e$
$= (True, frs_0, (xp',h_1,(v'\#vs',ls_1,C_0,M',pc' - size\ (compE_2\ e') - 1,ics')\#frs',sh_1), err)$
⟨*proof*⟩

lemma *Jcc-pieces-BinOp2*:
assumes [*simp*]: $P \equiv compP_2\ P_1$
and $Jcc\text{-}pieces\ P_1\ E\ C\ M\ h_0\ vs\ ls_0\ pc\ ics\ frs\ sh_0\ I\ h_2\ ls_2\ sh_2\ v\ xa\ (e\ «bop»\ e')$
$= (True, frs_0, (xp',h',(v\#vs',ls',C_0,M',pc',ics')\#frs',sh'), err)$
shows $\exists\, err.\ Jcc\text{-}pieces\ P_1\ E\ C\ M\ h_1\ (v_1\#vs)\ ls_1\ (pc + size\ (compE_2\ e))\ ics\ frs\ sh_1$
$(I - pcs\ (compxE_2\ e\ pc\ (length\ vs')))\ h_2\ ls_2\ sh_2\ v'\ xa\ e'$
$= (True, (v_1\#vs,ls_1,C,M,pc + size\ (compE_2\ e),ics)\#frs,$
$(xp',h',(v'\#v_1\#vs',ls',C_0,M',pc' - 1,ics')\#frs',sh'),$
$(\exists\, pc_1.\ pc + size\ (compE_2\ e) \le pc_1 \wedge pc_1 < pc + size\ (compE_2\ e) + length\ (compE_2\ e') \wedge$
$\neg\ caught\ P\ pc_1\ h_2\ xa\ (compxE_2\ e'\ (pc + size\ (compE_2\ e))\ (Suc\ (length\ vs))) \wedge$
$(\exists\, vs'.\ P \vdash (None,h_1,(v_1\#vs,ls_1,C,M,pc + size\ (compE_2\ e),ics)\#frs,sh_1)$
$-jvm\rightarrow handle\ P\ C\ M\ xa\ h_2\ (vs'@v_1\#vs)\ ls_2\ pc_1\ ics\ frs\ sh_2)))$
⟨*proof*⟩

lemma *Jcc-pieces-FAcc*:
assumes
$Jcc\text{-}pieces\ P\ E\ C\ M\ h_0\ vs\ ls_0\ pc\ ics\ frs\ sh_0\ I\ h_1\ ls_1\ sh_1\ v\ xa\ (e{\cdot}F\{D\})$
$= (True, frs_0, (xp',h',(v\#vs',ls',C_0,M',pc',ics')\#frs',sh'), err)$
shows $\exists\, err.\ Jcc\text{-}pieces\ P\ E\ C\ M\ h_0\ vs\ ls_0\ pc\ ics\ frs\ sh_0\ I\ h_1\ ls_1\ sh_1\ v'\ xa\ e$
$= (True, frs_0, (xp',h',(v'\#vs',ls',C_0,M',pc' - 1,ics')\#frs',sh'), err)$
⟨*proof*⟩

lemma *Jcc-pieces-LAss*:
assumes [*simp*]: $P \equiv compP_2\ P_1$
and $Jcc\text{-}pieces\ P_1\ E\ C\ M\ h_0\ vs\ ls_0\ pc\ ics\ frs\ sh_0\ I\ h_1\ ls_1\ sh_1\ v\ xa\ (i{:=}e)$
$= (True, frs_0, (xp',h',(v\#vs',ls',C_0,M',pc',ics')\#frs',sh'), err)$
shows $Jcc\text{-}pieces\ P_1\ E\ C\ M\ h_0\ vs\ ls_0\ pc\ ics\ frs\ sh_0\ I\ h_1\ ls_1\ sh_1\ v'\ xa\ e$
$= (True, frs_0, (xp',h',(v'\#vs',ls',C_0,M',pc' - 2,ics')\#frs',sh'),$
$(\exists\, pc_1.\ pc \le pc_1 \wedge pc_1 < pc + size(compE_2\ e) \wedge$
$\neg\ caught\ P\ pc_1\ h_1\ xa\ (compxE_2\ e\ pc\ (size\ vs)) \wedge$
$(\exists\, vs'.\ P \vdash (None,h_0,frs_0,sh_0) -jvm\rightarrow handle\ P\ C\ M\ xa\ h_1\ (vs'@vs)\ ls_1\ pc_1\ ics\ frs\ sh_1)))$
⟨*proof*⟩

lemma *Jcc-pieces-FAss1*:
assumes
$Jcc\text{-}pieces\ P\ E\ C\ M\ h_0\ vs\ ls_0\ pc\ ics\ frs\ sh_0\ I\ h_2\ ls_2\ sh_2\ v\ xa\ (e{\cdot}F\{D\}{:=}e')$
$= (True, frs_0, (xp',h',(v\#vs',ls',C_0,M',pc',ics')\#frs',sh'), err)$
shows $\exists\, err.\ Jcc\text{-}pieces\ P\ E\ C\ M\ h_0\ vs\ ls_0\ pc\ ics\ frs\ sh_0$
$(I - pcs\ (compxE_2\ e'\ (pc + length\ (compE_2\ e))\ (Suc\ (length\ vs'))))\ h_1\ ls_1\ sh_1\ v'\ xa\ e$
$= (True, frs_0, (xp',h_1,(v'\#vs',ls_1,C_0,M',pc' - size\ (compE_2\ e') - 2,ics')\#frs',sh_1), err)$
⟨*proof*⟩

lemma *Jcc-pieces-FAss2*:
assumes
$Jcc\text{-}pieces\ P\ E\ C\ M\ h_0\ vs\ ls_0\ pc\ ics\ frs\ sh_0\ I\ h_2\ ls_2\ sh_2\ v\ xa\ (e{\cdot}F\{D\}{:=}e')$
$= (True, frs_0, (xp',h',(v\#vs',ls',C_0,M',pc',ics')\#frs',sh'), err)$
shows $Jcc\text{-}pieces\ P\ E\ C\ M\ h_1\ (v_1\#vs)\ ls_1\ (pc + size\ (compE_2\ e))\ ics\ frs\ sh_1$
$(I - pcs\ (compxE_2\ e\ pc\ (length\ vs')))\ h_2\ ls_2\ sh_2\ v'\ xa\ e'$

$= (True, (v_1\#vs,ls_1,C,M,pc + size\ (compE_2\ e),ics)\#frs,$
$(xp',h',(v'\#v_1\#vs',ls',C_0,M',pc' - 2,ics')\#frs',sh'),$
$(\exists\, pc_1.\ (pc + size\ (compE_2\ e)) \le pc_1 \wedge pc_1 < pc + size\ (compE_2\ e) + size(compE_2\ e') \wedge$
$\neg\ caught\ (compP_2\ P)\ pc_1\ h_2\ xa\ (compxE_2\ e'\ (pc + size\ (compE_2\ e))\ (size\ (v_1\#vs))) \wedge$
$(\exists\, vs'.\ (compP_2\ P) \vdash (None,h_1,(v_1\#vs,ls_1,C,M,pc + size\ (compE_2\ e),ics)\#frs,sh_1)$
$-jvm\rightarrow\ handle\ (compP_2\ P)\ C\ M\ xa\ h_2\ (vs'@v_1\#vs)\ ls_2\ pc_1\ ics\ frs\ sh_2)))$

⟨*proof*⟩

lemma *Jcc-pieces-SFAss*:
assumes
$Jcc\text{-}pieces\ P\ E\ C\ M\ h_0\ vs\ ls_0\ pc\ ics\ frs\ sh_0\ I\ h'\ ls'\ sh'\ v\ xa\ (C'\cdot_s F\{D\}:=e)$
$= (True, frs_0, (xp',h',(v\#vs',ls',C_0,M',pc',ics')\#frs',sh'), err)$
shows $\exists\, err.\ Jcc\text{-}pieces\ P\ E\ C\ M\ h_0\ vs\ ls_0\ pc\ ics\ frs\ sh_0\ I\ h_1\ ls_1\ sh_1\ v'\ xa\ e$
$= (True, frs_0, (xp',h_1,(v'\#vs',ls_1,C_0,M',pc' - 2,ics')\#frs',sh_1), err)$
⟨*proof*⟩

lemma *Jcc-pieces-Call1*:
assumes
$Jcc\text{-}pieces\ P\ E\ C\ M\ h_0\ vs\ ls_0\ pc\ ics\ frs\ sh_0\ I\ h_3\ ls_3\ sh_3\ v\ xa\ (e\cdot M_0(es))$
$= (True, frs_0, (xp',h',(v\#vs',ls',C',M',pc',ics')\#frs',sh'), err)$
shows $\exists\, err.\ Jcc\text{-}pieces\ P\ E\ C\ M\ h_0\ vs\ ls_0\ pc\ ics\ frs\ sh_0$
$(I - pcs\ (compxEs_2\ es\ (pc + length\ (compE_2\ e))\ (Suc\ (length\ vs'))))\ h_1\ ls_1\ sh_1\ v'\ xa\ e$
$= (True, frs_0,$
$(xp',h_1,(v'\#vs',ls_1,C',M',pc' - size\ (compEs_2\ es) - 1,ics')\#frs',sh_1), err)$
⟨*proof*⟩

lemma *Jcc-pieces-clinit*:
assumes [*simp*]: $P \equiv compP_2\ P_1$
and *cond*: $Jcc\text{-}cond\ P_1\ E\ C\ M\ vs\ pc\ ics\ I\ h\ sh\ (C1\cdot_s clinit([]))$
shows $Jcc\text{-}pieces\ P_1\ E\ C\ M\ h\ vs\ ls\ pc\ ics\ frs\ sh\ I\ h'\ ls'\ sh'\ v\ xa\ (C1\cdot_s clinit([]))$
$= (True, create\text{-}init\text{-}frame\ P\ C1\ \#\ (vs,ls,C,M,pc,ics)\#frs,$
$(None, h', (vs,ls,C,M,pc,ics)\#frs, sh'(C1\mapsto(fst(the(sh'\ C1)),Done))),$
$P \vdash (None,h,create\text{-}init\text{-}frame\ P\ C1\ \#\ (vs,ls,C,M,pc,ics)\#frs,sh)\ -jvm\rightarrow$
$(case\ ics\ of\ Called\ Cs \Rightarrow (None,h',(vs,ls,C,M,pc,Throwing\ Cs\ xa)\#frs,(sh'(C1 \mapsto (fst(the(sh'$
$C1)),Error))))))$
⟨*proof*⟩

lemma *Jcc-pieces-SCall-clinit-body*:
assumes [*simp*]: $P \equiv compP_2\ P_1$ **and** *wf*: $wf\text{-}J_1\text{-}prog\ P_1$
and $Jcc\text{-}pieces\ P_1\ E\ C\ M\ h_0\ vs\ ls_0\ pc\ ics\ frs\ sh_0\ I\ h_3\ ls_2\ sh_3\ v\ xa\ (C1\cdot_s clinit([]))$
$= (True, frs', rhs', err')$
and *method*: $P_1 \vdash C1\ sees\ clinit,Static: []\rightarrow Void = body\ in\ D$
shows $Jcc\text{-}pieces\ P_1\ []\ D\ clinit\ h_2\ []\ (replicate\ (max\text{-}vars\ body)\ undefined)\ 0$
$No\text{-}ics\ (tl\ frs')\ sh_2\ \{..<length\ (compE_2\ body)\}\ h_3\ ls_3\ sh_3\ v\ xa\ body$
$= (True, frs',$
$(None,h_3,([v],ls_3,D,clinit,size(compE_2\ body),\ No\text{-}ics)\#tl\ frs',sh_3),$
$\exists\, pc_1.\ 0 \le pc_1 \wedge pc_1 < size(compE_2\ body) \wedge$
$\neg\ caught\ P\ pc_1\ h_3\ xa\ (compxE_2\ body\ 0\ 0) \wedge$
$(\exists\, vs'.\ P \vdash (None,h_2,frs',sh_2)\ -jvm\rightarrow\ handle\ P\ D\ clinit\ xa\ h_3\ vs'\ ls_3\ pc_1$
$No\text{-}ics\ (tl\ frs')\ sh_3))$
⟨*proof*⟩

lemma *Jcc-pieces-Cons*:
assumes [*simp*]: $P \equiv compP_2\ P_1$

and $P,C,M,pc \rhd compEs_2\ (e\#es)$ **and** $P,C,M \rhd compxEs_2\ (e\#es)\ pc\ (size\ vs)/I,size\ vs$
and $\{pc..<pc+size(compEs_2\ (e\#es))\} \subseteq I$
and $ics = No\text{-}ics$
and $\neg sub\text{-}RIs\ (e\#es)$
shows *Jcc-pieces* $P_1\ E\ C\ M\ h\ vs\ ls\ pc\ ics\ frs\ sh$
$(I - pcs\ (compxEs_2\ es\ (pc + length\ (compE_2\ e))\ (Suc\ (length\ vs))))\ h'\ ls'\ sh'\ v\ xa\ e$
$= (True, (vs, ls, C, M, pc, ics) \# frs,$
$(None, h', (v\#vs, ls', C, M, pc + length\ (compE_2\ e), ics) \# frs, sh'),$
$\exists pc_1 \geq pc.\ pc_1 < pc + length\ (compE_2\ e) \wedge \neg\ caught\ P\ pc_1\ h'\ xa\ (compxE_2\ e\ pc\ (length\ vs))$
$\wedge\ (\exists vs'.\ P \vdash (None, h, (vs, ls, C, M, pc, ics) \# frs, sh)$
$-jvm\rightarrow\ handle\ P\ C\ M\ xa\ h'\ (vs'@vs)\ ls'\ pc_1\ ics\ frs\ sh'))$
⟨*proof*⟩

lemma *Jcc-pieces-InitNone*:
assumes [*simp*]: $P \equiv compP_2\ P_1$
and *Jcc-pieces* $P_1\ E\ C\ M\ h\ vs\ l\ pc\ ics\ frs\ sh\ I\ h'\ l'\ sh'\ v\ xa\ (INIT\ C'\ (C_0\ \#\ Cs,False) \leftarrow e)$
$= (True, frs', (None, h', (vs, l, C, M, pc, Called\ []) \# frs, sh'), err)$
shows
Jcc-pieces $P_1\ E\ C\ M\ h\ vs\ l\ pc\ ics\ frs\ (sh(C_0 \mapsto (sblank\ P\ C_0, Prepared)))$
$I\ h'\ l'\ sh'\ v\ xa\ (INIT\ C'\ (C_0\ \#\ Cs,False) \leftarrow e)$
$= (True, frs', (None, h', (vs, l, C, M, pc, Called\ []) \# frs, sh'),$
$\exists vs'.\ P \vdash (None,h,frs',(sh(C_0 \mapsto (sblank\ P_1\ C_0, Prepared))))$
$-jvm\rightarrow\ handle\ P\ C\ M\ xa\ h'\ (vs'@vs)\ l\ pc\ ics\ frs\ sh')$
⟨*proof*⟩

lemma *Jcc-pieces-InitDP*:
assumes [*simp*]: $P \equiv compP_2\ P_1$
and *Jcc-pieces* $P_1\ E\ C\ M\ h\ vs\ l\ pc\ ics\ frs\ sh\ I\ h'\ l'\ sh'\ v\ xa\ (INIT\ C'\ (C_0\ \#\ Cs,False) \leftarrow e)$
$= (True, frs', (None, h', (vs, l, C, M, pc, Called\ []) \# frs, sh'), err)$
shows
Jcc-pieces $P_1\ E\ C\ M\ h\ vs\ l\ pc\ ics\ frs\ sh\ I\ h'\ l'\ sh'\ v\ xa\ (INIT\ C'\ (Cs,True) \leftarrow e)$
$= (True, (calling\text{-}to\text{-}scalled\ (hd\ frs'))\#(tl\ frs'),$
$(None, h', (vs, l, C, M, pc, Called\ []) \# frs, sh'),$
$\exists vs'.\ P \vdash (None,h,calling\text{-}to\text{-}scalled\ (hd\ frs')\#(tl\ frs'),sh)$
$-jvm\rightarrow\ handle\ P\ C\ M\ xa\ h'\ (vs'@vs)\ l\ pc\ ics\ frs\ sh')$
⟨*proof*⟩

lemma *Jcc-pieces-InitError*:
assumes [*simp*]: $P \equiv compP_2\ P_1$
and *Jcc-pieces* $P_1\ E\ C\ M\ h\ vs\ l\ pc\ ics\ frs\ sh\ I\ h'\ l'\ sh'\ v\ xa\ (INIT\ C'\ (C_0\ \#\ Cs,False) \leftarrow e)$
$= (True, frs', (None, h', (vs, l, C, M, pc, Called\ []) \# frs, sh'), err)$
and *err*: $sh\ C_0 = Some(sfs,Error)$
shows
Jcc-pieces $P_1\ E\ C\ M\ h\ vs\ l\ pc\ ics\ frs\ sh\ I\ h'\ l'\ sh'\ v\ xa\ (RI\ (C_0, THROW\ NoClassDefFoundError);Cs \leftarrow e)$
$= (True, (calling\text{-}to\text{-}throwing\ (hd\ frs')\ (addr\text{-}of\text{-}sys\text{-}xcpt\ NoClassDefFoundError))\#(tl\ frs'),$
$(None, h', (vs, l, C, M, pc, Called\ []) \# frs, sh'),$
$\exists vs'.\ P \vdash (None,h,\ (calling\text{-}to\text{-}throwing\ (hd\ frs')\ (addr\text{-}of\text{-}sys\text{-}xcpt\ NoClassDefFoundError))\#(tl\ frs'),sh)$
$-jvm\rightarrow\ handle\ P\ C\ M\ xa\ h'\ (vs'@vs)\ l\ pc\ ics\ frs\ sh')$
⟨*proof*⟩

lemma *Jcc-pieces-InitObj*:
assumes [*simp*]: $P \equiv compP_2\ P_1$

and *Jcc-pieces* P_1 E C M h vs l pc ics frs sh I h' l' $(sh(C_0 \mapsto (sfs,Processing)))$ v xa $(INIT\ C'\ (C_0\ \#\ Cs,False) \leftarrow e)$
$= (True,\ frs',\ (None,\ h',\ (vs,\ l,\ C,\ M,\ pc,\ Called\ []) \#\ frs,\ sh'),\ err)$
shows
Jcc-pieces P_1 E C M h vs l pc ics frs $(sh(C_0 \mapsto (sfs,Processing)))$ I h' l' sh'' v xa $(INIT\ C'\ (C_0\ \#\ Cs,True) \leftarrow e)$
$= (True,\ calling\text{-}to\text{-}called\ (hd\ frs')\#(tl\ frs'),$
$(None,\ h',\ (vs,\ l,\ C,\ M,\ pc,\ Called\ []) \#\ frs,\ sh''),$
$\exists\, vs'.\ P \vdash (None,h,calling\text{-}to\text{-}called\ (hd\ frs')\#(tl\ frs'),sh')$
$-jvm\rightarrow\ handle\ P\ C\ M\ xa\ h'\ (vs'@vs)\ l\ pc\ ics\ frs\ sh'')$
⟨*proof*⟩

lemma *Jcc-pieces-InitNonObj*:
assumes [*simp*]: $P \equiv compP_2\ P_1$
and *is-class* P_1 D **and** $D \notin set\ (C_0\#Cs)$ **and** $\forall\, C \in set\ (C_0\#Cs).\ P_1 \vdash C \preceq^* D$
and *pcs*: *Jcc-pieces* P_1 E C M h vs l pc ics frs sh I h' l' $(sh(C_0 \mapsto (sfs,Processing)))$ v xa $(INIT\ C'\ (C_0\ \#\ Cs,False) \leftarrow e)$
$= (True,\ frs',\ (None,\ h',\ (vs,\ l,\ C,\ M,\ pc,\ Called\ []) \#\ frs,\ sh'),\ err)$
shows
Jcc-pieces P_1 E C M h vs l pc ics frs $(sh(C_0 \mapsto (sfs,Processing)))$ I h' l' sh'' v xa $(INIT\ C'\ (D\ \#\ C_0\ \#\ Cs,False) \leftarrow e)$
$= (True,\ calling\text{-}to\text{-}calling\ (hd\ frs')\ D\#(tl\ frs'),$
$(None,\ h',\ (vs,\ l,\ C,\ M,\ pc,\ Called\ []) \#\ frs,\ sh''),$
$\exists\, vs'.\ P \vdash (None,h,calling\text{-}to\text{-}calling\ (hd\ frs')\ D\#(tl\ frs'),sh')$
$-jvm\rightarrow\ handle\ P\ C\ M\ xa\ h'\ (vs'@vs)\ l\ pc\ ics\ frs\ sh'')$
⟨*proof*⟩

lemma *Jcc-pieces-InitRInit*:
assumes [*simp*]: $P \equiv compP_2\ P_1$ **and** *wf*: *wf-J*$_1$*-prog* P_1
and *Jcc-pieces* P_1 E C M h vs l pc ics frs sh I h' l' sh' v xa $(INIT\ C'\ (C_0\ \#\ Cs,True) \leftarrow e)$
$= (True,\ frs',\ (None,\ h',\ (vs,\ l,\ C,\ M,\ pc,\ Called\ []) \#\ frs,\ sh'),\ err)$
shows
Jcc-pieces P_1 E C M h vs l pc ics frs sh I h' l' sh' v xa $(RI\ (C_0,C_0{\cdot}_s clinit([]))\ ;\ Cs \leftarrow e)$
$= (True,\ frs',$
$(None,\ h',\ (vs,\ l,\ C,\ M,\ pc,\ Called\ []) \#\ frs,\ sh'),$
$\exists\, vs'.\ P \vdash (None,h,frs',sh)$
$-jvm\rightarrow\ handle\ P\ C\ M\ xa\ h'\ (vs'@vs)\ l\ pc\ ics\ frs\ sh')$
⟨*proof*⟩

lemma *Jcc-pieces-RInit-clinit*:
assumes [*simp*]: $P \equiv compP_2\ P_1$ **and** *wf*: *wf-J*$_1$*-prog* P_1
and *Jcc-pieces* P_1 E C M h vs l pc ics frs sh I h_1 l_1 sh_1 v xa $(RI\ (C_0,C_0{\cdot}_s clinit([]));Cs \leftarrow e)$
$= (True,\ frs',$
$(None,\ h_1,\ (vs,\ l,\ C,\ M,\ pc,\ Called\ []) \#\ frs,\ sh_1),\ err)$
shows
Jcc-pieces P_1 E C M h vs l pc $(Called\ Cs)$ $(tl\ frs')$ sh I h' l' sh' v xa $(C_0{\cdot}_s clinit([]))$
$= (True,\ create\text{-}init\text{-}frame\ P\ C_0\#(vs,l,C,M,pc,Called\ Cs)\#tl\ frs',$
$(None,\ h',\ (vs,l,C,M,pc,Called\ Cs)\#tl\ frs',\ sh'(C_0{\mapsto}(fst(the(sh'\ C_0)),Done))),$
$P \vdash (None,h,create\text{-}init\text{-}frame\ P\ C_0\#(vs,l,C,M,pc,Called\ Cs)\#tl\ frs',sh)$
$-jvm\rightarrow (None,h',(vs,\ l,\ C,\ M,\ pc,\ Throwing\ Cs\ xa)\ \#\ tl\ frs',sh'(C_0 \mapsto (fst(the(sh'\ C_0)),Error))))$
⟨*proof*⟩

lemma *Jcc-pieces-RInit-Init*:
assumes [*simp*]: $P \equiv compP_2\ P_1$ **and** *wf*: *wf-J*$_1$*-prog* P_1

and *proc*: $\forall\, C' \in set\ Cs.\ \exists\, sfs.\ sh''\ C' = \lfloor(sfs,Processing)\rfloor$
and *Jcc-pieces* $P_1\ E\ C\ M\ h\ vs\ l\ pc\ ics\ frs\ sh\ I\ h_1\ l_1\ sh_1\ v\ xa\ (RI\ (C_0,C_0{\cdot}_s clinit([]));Cs \leftarrow e)$
$= (True,\ frs',$
$(None,\ h_1,\ (vs,\ l,\ C,\ M,\ pc,\ Called\ []) \#\ frs,\ sh_1),\ err)$
shows
Jcc-pieces $P_1\ E\ C\ M\ h'\ vs\ l\ pc\ ics\ frs\ sh''\ I\ h_1\ l_1\ sh_1\ v\ xa\ (INIT\ (last\ (C_0\#Cs))\ (Cs,True) \leftarrow e)$
$= (True,\ (vs,\ l,\ C,\ M,\ pc,\ Called\ Cs) \#\ frs,$
$(None,\ h_1,\ (vs,\ l,\ C,\ M,\ pc,\ Called\ []) \#\ frs,\ sh_1),$
$\exists\, vs'.\ P \vdash (None,h',(vs,\ l,\ C,\ M,\ pc,\ Called\ Cs) \#\ frs,sh'')$
$-jvm\rightarrow\ handle\ P\ C\ M\ xa\ h_1\ (vs'@vs)\ l\ pc\ ics\ frs\ sh_1)$
⟨*proof*⟩

lemma *Jcc-pieces-RInit-RInit*:
assumes [*simp*]: $P \equiv compP_2\ P_1$
and *Jcc-pieces* $P_1\ E\ C\ M\ h\ vs\ l\ pc\ ics\ frs\ sh\ I\ h_1\ l_1\ sh_1\ v\ xa\ (RI\ (C_0,e);D\#Cs \leftarrow e')$
$= (True,\ frs',\ rhs,\ err)$
and *hd*: $hd\ frs' = (vs1,l1,C1,M1,pc1,ics1)$
shows
Jcc-pieces $P_1\ E\ C\ M\ h'\ vs\ l\ pc\ ics\ frs\ sh''\ I\ h_1\ l_1\ sh_1\ v\ xa\ (RI\ (D,Throw\ xa)\ ;\ Cs \leftarrow e')$
$= (True,\ (vs1,\ l1,\ C1,\ M1,\ pc1,\ Throwing\ (D\#Cs)\ xa) \#\ tl\ frs',$
$(None,\ h_1,\ (vs,\ l,\ C,\ M,\ pc,\ Called\ []) \#\ frs,\ sh_1),$
$\exists\, vs'.\ P \vdash (None,h',(vs1,\ l1,\ C1,\ M1,\ pc1,\ Throwing\ (D\#Cs)\ xa) \#\ tl\ frs',sh'')$
$-jvm\rightarrow\ handle\ P\ C\ M\ xa\ h_1\ (vs'@vs)\ l\ pc\ ics\ frs\ sh_1)$
⟨*proof*⟩

JVM stepping lemmas

lemma *jvm-Invoke*:
assumes [*simp*]: $P \equiv compP_2\ P_1$
and $P,C,M,pc \triangleright Invoke\ M'\ (length\ Ts)$
and *ha*: $h_2\ a = \lfloor(Ca,\ fs)\rfloor$ **and** *method*: $P_1 \vdash Ca\ sees\ M',\ NonStatic :\ Ts\rightarrow T = body\ in\ D$
and *len*: $length\ pvs = length\ Ts$ **and** ${ls_2}' = Addr\ a \#\ pvs @ replicate\ (max\text{-}vars\ body)\ undefined$
shows $P \vdash (None,\ h_2,\ (rev\ pvs @ Addr\ a \#\ vs,\ ls_2,\ C,\ M,\ pc,\ No\text{-}ics) \#\ frs,\ sh_2) -jvm\rightarrow$
$(None,\ h_2,\ ([],\ {ls_2}',\ D,\ M',\ 0,\ No\text{-}ics) \#\ (rev\ pvs @ Addr\ a \#\ vs,\ ls_2,\ C,\ M,\ pc,\ No\text{-}ics) \#\ frs,\ sh_2)$
⟨*proof*⟩

lemma *jvm-Invokestatic*:
assumes [*simp*]: $P \equiv compP_2\ P_1$
and $P,C,M,pc \triangleright Invokestatic\ C'\ M'\ (length\ Ts)$
and *sh*: $sh_2\ D = Some(sfs,Done)$
and *method*: $P_1 \vdash C'\ sees\ M',\ Static :\ Ts\rightarrow T = body\ in\ D$
and *len*: $length\ pvs = length\ Ts$ **and** ${ls_2}' = pvs @ replicate\ (max\text{-}vars\ body)\ undefined$
shows $P \vdash (None,\ h_2,\ (rev\ pvs @ vs,\ ls_2,\ C,\ M,\ pc,\ No\text{-}ics) \#\ frs,\ sh_2) -jvm\rightarrow$
$(None,\ h_2,\ ([],\ {ls_2}',\ D,\ M',\ 0,\ No\text{-}ics) \#\ (rev\ pvs @ vs,\ ls_2,\ C,\ M,\ pc,\ No\text{-}ics) \#\ frs,\ sh_2)$
⟨*proof*⟩

lemma *jvm-Invokestatic-Called*:
assumes [*simp*]: $P \equiv compP_2\ P_1$
and $P,C,M,pc \triangleright Invokestatic\ C'\ M'\ (length\ Ts)$
and *sh*: $sh_2\ D = Some(sfs,i)$
and *method*: $P_1 \vdash C'\ sees\ M',\ Static :\ Ts\rightarrow T = body\ in\ D$
and *len*: $length\ pvs = length\ Ts$ **and** ${ls_2}' = pvs @ replicate\ (max\text{-}vars\ body)\ undefined$
shows $P \vdash (None,\ h_2,\ (rev\ pvs @ vs,\ ls_2,\ C,\ M,\ pc,\ Called\ []) \#\ frs,\ sh_2) -jvm\rightarrow$
$(None,\ h_2,\ ([],\ {ls_2}',\ D,\ M',\ 0,\ No\text{-}ics) \#\ (rev\ pvs @ vs,\ ls_2,\ C,\ M,\ pc,\ No\text{-}ics) \#\ frs,\ sh_2)$

⟨*proof*⟩

lemma *jvm-Return-Init*:
P,*D*,*clinit*,*0* ▷ *compE*$_2$ *body* @ [*Return*]
⟹ *P* ⊢ (*None*, *h*, (*vs*, *ls*, *D*, *clinit*, *size*(*compE*$_2$ *body*), *No-ics*) # *frs*, *sh*)
−*jvm*→ (*None*, *h*, *frs*, *sh*(*D*↦(*fst*(*the*(*sh D*)),*Done*)))
(**is** *?P* ⟹ *P* ⊢ *?s1* −*jvm*→ *?s2*)
⟨*proof*⟩

lemma *jvm-InitNone*:
⟦ *ics-of f* = *Calling C Cs*;
sh C = *None* ⟧
⟹ *P* ⊢ (*None*,*h*,*f*#*frs*,*sh*) −*jvm*→ (*None*,*h*,*f*#*frs*,*sh*(*C* ↦ (*sblank P C*, *Prepared*)))
(**is** ⟦ *?P*; *?Q* ⟧ ⟹ *P* ⊢ *?s1* −*jvm*→ *?s2*)
⟨*proof*⟩

lemma *jvm-InitDP*:
⟦ *ics-of f* = *Calling C Cs*;
sh C = ⌊(*sfs*,*i*)⌋; *i* = *Done* ∨ *i* = *Processing* ⟧
⟹ *P* ⊢ (*None*,*h*,*f*#*frs*,*sh*) −*jvm*→ (*None*,*h*,(*calling-to-scalled f*)#*frs*,*sh*)
(**is** ⟦ *?P*; *?Q*; *?R* ⟧ ⟹ *P* ⊢ *?s1* −*jvm*→ *?s2*)
⟨*proof*⟩

lemma *jvm-InitError*:
sh C = ⌊(*sfs*,*Error*)⌋
⟹ *P* ⊢ (*None*,*h*,(*vs*,*ls*,*C*$_0$,*M*,*pc*,*Calling C Cs*)#*frs*,*sh*)
−*jvm*→ (*None*,*h*,(*vs*,*ls*,*C*$_0$,*M*,*pc*,*Throwing Cs* (*addr-of-sys-xcpt NoClassDefFoundError*))#*frs*,*sh*)
⟨*proof*⟩

lemma *exec-ErrorThrowing*:
sh C = ⌊(*sfs*,*Error*)⌋
⟹ *exec* (*P*, (*None*,*h*,*calling-to-throwing* (*stk*,*loc*,*D*,*M*,*pc*,*Calling C Cs*) *a*#*frs*,*sh*))
= *Some* (*None*,*h*,*calling-to-sthrowing* (*stk*,*loc*,*D*,*M*,*pc*,*Calling C Cs*) *a* #*frs*,*sh*)
⟨*proof*⟩

lemma *jvm-InitObj*:
⟦ *sh C* = *Some*(*sfs*,*Prepared*);
C = *Object*;
sh′ = *sh*(*C* ↦ (*sfs*,*Processing*)) ⟧
⟹ *P* ⊢ (*None*, *h*, (*vs*,*ls*,*C*$_0$,*M*,*pc*,*Calling C Cs*)#*frs*, *sh*) −*jvm*→
(*None*, *h*, (*vs*,*ls*,*C*$_0$,*M*,*pc*,*Called* (*C*#*Cs*))#*frs*,*sh*′)
(**is** ⟦ *?P*; *?Q*; *?R* ⟧ ⟹ *P* ⊢ *?s1* −*jvm*→ *?s2*)
⟨*proof*⟩

lemma *jvm-InitNonObj*:
⟦ *sh C* = *Some*(*sfs*,*Prepared*);
C ≠ *Object*;
class P C = *Some* (*D*,*r*);
sh′ = *sh*(*C* ↦ (*sfs*,*Processing*)) ⟧
⟹ *P* ⊢ (*None*, *h*, (*vs*,*ls*,*C*$_0$,*M*,*pc*,*Calling C Cs*)#*frs*, *sh*) −*jvm*→
(*None*, *h*, (*vs*,*ls*,*C*$_0$,*M*,*pc*,*Calling D* (*C*#*Cs*))#*frs*, *sh*′)
(**is** ⟦ *?P*; *?Q*; *?R*; *?S* ⟧ ⟹ *P* ⊢ *?s1* −*jvm*→ *?s2*)
⟨*proof*⟩

lemma *jvm-RInit-throw*:
$P \vdash (None,h,(vs,l,C,M,pc,Throwing\ []\ xa)\ \#\ frs,sh)$
$\quad -jvm\rightarrow\ handle\ P\ C\ M\ xa\ h\ vs\ l\ pc\ No\text{-}ics\ frs\ sh$
(**is** $P \vdash ?s1\ -jvm\rightarrow\ ?s2$)
⟨*proof*⟩

lemma *jvm-RInit-throw'*:
$P \vdash (None,h,(vs,l,C,M,pc,Throwing\ [C']\ xa)\ \#\ frs,sh)$
$\quad -jvm\rightarrow\ handle\ P\ C\ M\ xa\ h\ vs\ l\ pc\ No\text{-}ics\ frs\ (sh(C':=Some(fst(the(sh\ C')),\ Error)))$
(**is** $P \vdash ?s1\ -jvm\rightarrow\ ?s2$)
⟨*proof*⟩

lemma *jvm-Called*:
$P \vdash (None,\ h,\ (vs,\ l,\ C,\ M,\ pc,\ Called\ (C_0\ \#\ Cs))\ \#\ frs,\ sh)\ -jvm\rightarrow$
$\quad (None,\ h,\ create\text{-}init\text{-}frame\ P\ C_0\ \#\ (vs,\ l,\ C,\ M,\ pc,\ Called\ Cs)\ \#\ frs,\ sh)$
⟨*proof*⟩

lemma *jvm-Throwing*:
$P \vdash (None,\ h,\ (vs,\ l,\ C,\ M,\ pc,\ Throwing\ (C_0\#Cs)\ xa')\ \#\ frs,\ sh)\ -jvm\rightarrow$
$\quad (None,\ h,\ (vs,\ l,\ C,\ M,\ pc,\ Throwing\ Cs\ xa')\ \#\ frs,\ sh(C_0 \mapsto (fst\ (the\ (sh\ C_0)),\ Error)))$
⟨*proof*⟩

Other lemmas for correctness proof

lemma assumes *wf*:*wf-prog wf-md P*
and *ex*: *class P C = Some a*
shows *create-init-frame-wf-eq*: *create-init-frame* $(compP_2\ P)\ C = (stk,loc,D,M,pc,ics) \Longrightarrow D{=}C$
⟨*proof*⟩

lemma *beforex-try*:
assumes *pcI*: $\{pc..<pc+size(compE_2(try\ e_1\ catch(Ci\ i)\ e_2))\} \subseteq I$
and *bx*: $P,C,M \rhd compxE_2\ (try\ e_1\ catch(Ci\ i)\ e_2)\ pc\ (size\ vs)\ /\ I,size\ vs$
shows $P,C,M \rhd compxE_2\ e_1\ pc\ (size\ vs)\ /\ \{pc..<pc\ +\ length\ (compE_2\ e_1)\},size\ vs$
⟨*proof*⟩

lemma
shows $eval_1\text{-}init\text{-}return$: $P \vdash_1 \langle e,s\rangle \Rightarrow \langle e',s'\rangle$
$\Longrightarrow iconf\ (shp_1\ s)\ e$
$\Longrightarrow (\exists\, Cs\ b.\ e = INIT\ C'\ (Cs,b) \leftarrow unit) \vee (\exists\, C\ e_0\ Cs\ e_i.\ e = RI(C,e_0);Cs@[C'] \leftarrow unit)$
$\quad \vee (\exists\, e_0.\ e = RI(C',e_0);Nil \leftarrow unit)$
$\Longrightarrow (val\text{-}of\ e' = Some\ v \longrightarrow (\exists\, sfs\ i.\ shp_1\ s'\ C' = \lfloor(sfs,i)\rfloor \wedge (i = Done \vee i = Processing)))$
$\wedge\ (throw\text{-}of\ e' = Some\ a \longrightarrow (\exists\, sfs\ i.\ shp_1\ s'\ C' = \lfloor(sfs,Error)\rfloor))$
and $P \vdash_1 \langle es,s\rangle\ [\Rightarrow]\ \langle es',s'\rangle \Longrightarrow True$
⟨*proof*⟩

lemma $init_1\text{-}Val\text{-}PD$: $P \vdash_1 \langle INIT\ C'\ (Cs,b) \leftarrow unit,s\rangle \Rightarrow \langle Val\ v,s'\rangle$
$\Longrightarrow iconf\ (shp_1\ s)\ (INIT\ C'\ (Cs,b) \leftarrow unit)$
$\Longrightarrow \exists\, sfs\ i.\ shp_1\ s'\ C' = \lfloor(sfs,i)\rfloor \wedge (i = Done \vee i = Processing)$
⟨*proof*⟩

lemma $init_1\text{-}throw\text{-}PD$: $P \vdash_1 \langle INIT\ C'\ (Cs,b) \leftarrow unit,s\rangle \Rightarrow \langle throw\ a,s'\rangle$
$\Longrightarrow iconf\ (shp_1\ s)\ (INIT\ C'\ (Cs,b) \leftarrow unit)$
$\Longrightarrow \exists\, sfs\ i.\ shp_1\ s'\ C' = \lfloor(sfs,Error)\rfloor$

⟨*proof*⟩

lemma *rinit*$_1$*-Val-PD*:
assumes *eval*: $P \vdash_1 \langle RI(C,e_0);Cs \leftarrow unit,s\rangle \Rightarrow \langle Val\ v,s'\rangle$
and *iconf*: *iconf* $(shp_1\ s)$ $(RI(C,e_0);Cs \leftarrow unit)$ **and** *last*: $last(C\#Cs) = C'$
shows $\exists\, sfs\ i.\ shp_1\ s'\ C' = \lfloor(sfs,i)\rfloor \wedge (i = Done \vee i = Processing)$
⟨*proof*⟩

lemma *rinit*$_1$*-throw-PD*:
assumes *eval*: $P \vdash_1 \langle RI(C,e_0);Cs \leftarrow unit,s\rangle \Rightarrow \langle throw\ a,s'\rangle$
and *iconf*: *iconf* $(shp_1\ s)$ $(RI(C,e_0);Cs \leftarrow unit)$ **and** *last*: $last(C\#Cs) = C'$
shows $\exists\, sfs.\ shp_1\ s'\ C' = \lfloor(sfs,Error)\rfloor$
⟨*proof*⟩

The proof

lemma fixes P_1 **defines** [*simp*]: $P \equiv compP_2\ P_1$
assumes *wf*: *wf-J*$_1$*-prog* P_1
shows *Jcc*: $P_1 \vdash_1 \langle e,(h_0,ls_0,sh_0)\rangle \Rightarrow \langle ef,(h_1,ls_1,sh_1)\rangle \Longrightarrow$
$(\bigwedge E\ C\ M\ pc\ ics\ v\ xa\ vs\ frs\ I.$
$[\![\ Jcc\text{-}cond\ P_1\ E\ C\ M\ vs\ pc\ ics\ I\ h_0\ sh_0\ e\ ]\!] \Longrightarrow$
$(ef = Val\ v \longrightarrow$
$P \vdash (None,h_0,Jcc\text{-}frames\ P\ C\ M\ vs\ ls_0\ pc\ ics\ frs\ e,sh_0)$
$-jvm\rightarrow\ Jcc\text{-}rhs\ P_1\ E\ C\ M\ vs\ ls_0\ pc\ ics\ frs\ h_1\ ls_1\ sh_1\ v\ e)$
$\wedge$
$(ef = Throw\ xa \longrightarrow Jcc\text{-}err\ P\ C\ M\ h_0\ vs\ ls_0\ pc\ ics\ frs\ sh_0\ I\ h_1\ ls_1\ sh_1\ xa\ e)$
$)$**and** $P_1 \vdash_1 \langle es,(h_0,ls_0,sh_0)\rangle\ [\Rightarrow]\ \langle fs,(h_1,ls_1,sh_1)\rangle \Longrightarrow$
$(\bigwedge C\ M\ pc\ ics\ ws\ xa\ es'\ vs\ frs\ I.$
$[\![\ P,C,M,pc \rhd compEs_2\ es;\ P,C,M \rhd compxEs_2\ es\ pc\ (size\ vs)/I,size\ vs;$
$\{pc..<pc+size(compEs_2\ es)\} \subseteq I;\ ics = No\text{-}ics;$
$\neg sub\text{-}RIs\ es\ ]\!] \Longrightarrow$
$(fs = map\ Val\ ws \longrightarrow$
$P \vdash (None,h_0,(vs,ls_0,C,M,pc,ics)\#frs,sh_0)\ -jvm\rightarrow$
$(None,h_1,(rev\ ws\ @\ vs,ls_1,C,M,pc+size(compEs_2\ es),ics)\#frs,sh_1))$
$\wedge$
$(fs = map\ Val\ ws\ @\ Throw\ xa\ \#\ es' \longrightarrow$
$(\exists\, pc_1.\ pc \le pc_1 \wedge pc_1 < pc + size(compEs_2\ es) \wedge$
$\neg\ caught\ P\ pc_1\ h_1\ xa\ (compxEs_2\ es\ pc\ (size\ vs)) \wedge$
$(\exists\, vs'.\ P \vdash (None,h_0,(vs,ls_0,C,M,pc,ics)\#frs,sh_0)$
$-jvm\rightarrow\ handle\ P\ C\ M\ xa\ h_1\ (vs'@vs)\ ls_1\ pc_1\ ics\ frs\ sh_1))))\langle proof\rangle$

lemma *atLeast0AtMost*[*simp*]: $\{0::nat..n\} = \{..n\}$
⟨*proof*⟩

lemma *atLeast0LessThan*[*simp*]: $\{0::nat..<n\} = \{..<n\}$
⟨*proof*⟩

fun *exception* :: *'a exp* ⇒ *addr option* **where**
exception (*Throw a*) = *Some a*
| *exception e* = *None*

lemma *comp*$_2$*-correct*:
assumes *wf*: *wf-J*$_1$*-prog* P_1
and *method*: $P_1 \vdash C\ sees\ M,b:Ts\rightarrow T = body\ in\ C$

and *eval*: $P_1 \vdash_1 \langle body,(h,ls,sh)\rangle \Rightarrow \langle e',(h',ls',sh')\rangle$
and *nclinit*: $M \neq clinit$
shows $compP_2\ P_1 \vdash (None,h,[([],ls,C,M,0,No\text{-}ics)],sh) -jvm\rightarrow (exception\ e',h',[],sh')\langle proof\rangle$
end

3.8 Combining Stages 1 and 2

theory *Compiler*
imports *Correctness1 Correctness2*
begin

definition *J2JVM* :: *J-prog* ⇒ *jvm-prog*
where
J2JVM ≡ $compP_2 \circ compP_1$

theorem *comp-correct-NonStatic*:
assumes *wf*: *wf-J-prog P*
and *method*: $P \vdash C\ sees\ M,NonStatic:Ts\rightarrow T = (pns,body)\ in\ C$
and *eval*: $P \vdash \langle body,(h,[this\#pns\ [\mapsto]\ vs],sh)\rangle \Rightarrow \langle e',(h',l',sh')\rangle$
and *sizes*: $size\ vs = size\ pns + 1 \quad size\ rest = max\text{-}vars\ body$
shows $J2JVM\ P \vdash (None,h,[([],vs@rest,C,M,0,No\text{-}ics)],sh) -jvm\rightarrow (exception\ e',h',[],sh')\langle proof\rangle$
theorem *comp-correct-Static*:
assumes *wf*: *wf-J-prog P*
and *method*: $P \vdash C\ sees\ M,Static:Ts\rightarrow T = (pns,body)\ in\ C$
and *eval*: $P \vdash \langle body,(h,[pns\ [\mapsto]\ vs],sh)\rangle \Rightarrow \langle e',(h',l',sh')\rangle$
and *sizes*: $size\ vs = size\ pns \quad size\ rest = max\text{-}vars\ body$
and *nclinit*: $M \neq clinit$
shows $J2JVM\ P \vdash (None,h,[([],vs@rest,C,M,0,No\text{-}ics)],sh) -jvm\rightarrow (exception\ e',h',[],sh')\langle proof\rangle$
end

3.9 Preservation of Well-Typedness

theory *TypeComp*
imports *Compiler ../BV/BVSpec*
begin

lemma *max-stack1*: $P,E \vdash_1 e :: T \Longrightarrow 1 \leq max\text{-}stack\ e\langle proof\rangle$
locale *TC0* =
fixes *P* :: J_1-*prog* **and** *mxl* :: *nat*
begin

definition $ty\ E\ e = (THE\ T.\ P,E \vdash_1 e :: T)$

definition $ty_l\ E\ A' = map\ (\lambda i.\ if\ i \in A' \wedge i < size\ E\ then\ OK(E!i)\ else\ Err)\ [0..<mxl]$

definition $ty_i'\ ST\ E\ A = (case\ A\ of\ None \Rightarrow None \mid \lfloor A' \rfloor \Rightarrow Some(ST,\ ty_l\ E\ A'))$

definition $after\ E\ A\ ST\ e = ty_i'\ (ty\ E\ e\ \#\ ST)\ E\ (A \sqcup \mathcal{A}\ e)$

end

lemma (**in** *TC0*) *ty-def2* [*simp*]: $P,E \vdash_1 e :: T \Longrightarrow ty\ E\ e = T\langle proof\rangle$
lemma (**in** *TC0*) [*simp*]: $ty_i'\ ST\ E\ None = None\langle proof\rangle$

lemma (**in** *TC0*) *ty$_l$-app-diff*[*simp*]:
ty$_l$ (*E*@[*T*]) (*A* − {*size E*}) = *ty$_l$ E A*⟨*proof*⟩

lemma (**in** *TC0*) *ty$_i$'-app-diff*[*simp*]:
ty$_i$' ST (*E* @ [*T*]) (*A* ⊖ *size E*) = *ty$_i$' ST E A*⟨*proof*⟩

lemma (**in** *TC0*) *ty$_l$-antimono*:
A ⊆ *A'* ⟹ *P* ⊢ *ty$_l$ E A'* [≤$_\top$] *ty$_l$ E A*⟨*proof*⟩

lemma (**in** *TC0*) *ty$_i$'-antimono*:
A ⊆ *A'* ⟹ *P* ⊢ *ty$_i$' ST E* ⌊*A'*⌋ ≤' *ty$_i$' ST E* ⌊*A*⌋⟨*proof*⟩

lemma (**in** *TC0*) *ty$_l$-env-antimono*:
P ⊢ *ty$_l$* (*E*@[*T*]) *A* [≤$_\top$] *ty$_l$ E A*⟨*proof*⟩

lemma (**in** *TC0*) *ty$_i$'-env-antimono*:
P ⊢ *ty$_i$' ST* (*E*@[*T*]) *A* ≤' *ty$_i$' ST E A*⟨*proof*⟩

lemma (**in** *TC0*) *ty$_i$'-incr*:
P ⊢ *ty$_i$' ST* (*E* @ [*T*]) ⌊*insert* (*size E*) *A*⌋ ≤' *ty$_i$' ST E* ⌊*A*⌋⟨*proof*⟩

lemma (**in** *TC0*) *ty$_l$-incr*:
P ⊢ *ty$_l$* (*E* @ [*T*]) (*insert* (*size E*) *A*) [≤$_\top$] *ty$_l$ E A*⟨*proof*⟩

lemma (**in** *TC0*) *ty$_l$-in-types*:
set E ⊆ *types P* ⟹ *ty$_l$ E A* ∈ *nlists mxl* (*err* (*types P*))⟨*proof*⟩
locale *TC1* = *TC0*
begin

primrec *compT* :: *ty list* ⇒ *nat hyperset* ⇒ *ty list* ⇒ *expr$_1$* ⇒ *ty$_i$' list* **and**
compTs :: *ty list* ⇒ *nat hyperset* ⇒ *ty list* ⇒ *expr$_1$ list* ⇒ *ty$_i$' list* **where**
compT E A ST (*new C*) = []
| *compT E A ST* (*Cast C e*) =
compT E A ST e @ [*after E A ST e*]
| *compT E A ST* (*Val v*) = []
| *compT E A ST* (*e$_1$* «*bop*» *e$_2$*) =
(*let ST$_1$* = *ty E e$_1$*#*ST*; *A$_1$* = *A* ⊔ *A e$_1$ in*
compT E A ST e$_1$ @ [*after E A ST e$_1$*] @
compT E A$_1$ ST$_1$ e$_2$ @ [*after E A$_1$ ST$_1$ e$_2$*])
| *compT E A ST* (*Var i*) = []
| *compT E A ST* (*i* := *e*) = *compT E A ST e* @
[*after E A ST e*, *ty$_i$' ST E* (*A* ⊔ *A e* ⊔ ⌊{*i*}⌋)]
| *compT E A ST* (*e*·*F*{*D*}) =
compT E A ST e @ [*after E A ST e*]
| *compT E A ST* (*C*·$_s$*F*{*D*}) = []
| *compT E A ST* (*e$_1$*·*F*{*D*} := *e$_2$*) =
(*let ST$_1$* = *ty E e$_1$*#*ST*; *A$_1$* = *A* ⊔ *A e$_1$*; *A$_2$* = *A$_1$* ⊔ *A e$_2$ in*
compT E A ST e$_1$ @ [*after E A ST e$_1$*] @
compT E A$_1$ ST$_1$ e$_2$ @ [*after E A$_1$ ST$_1$ e$_2$*] @
[*ty$_i$' ST E A$_2$*])
| *compT E A ST* (*C*·$_s$*F*{*D*} := *e$_2$*) = *compT E A ST e$_2$* @ [*after E A ST e$_2$*] @ [*ty$_i$' ST E* (*A* ⊔ *A e$_2$*)]
| *compT E A ST* {*i*:*T*; *e*} = *compT* (*E*@[*T*]) (*A*⊖*i*) *ST e*
| *compT E A ST* (*e$_1$*;;*e$_2$*) =

$(let\ A_1 = A \sqcup \mathcal{A}\ e_1\ in$
$compT\ E\ A\ ST\ e_1\ @\ [after\ E\ A\ ST\ e_1,\ ty_i'\ ST\ E\ A_1]\ @$
$compT\ E\ A_1\ ST\ e_2)$
$|\ compT\ E\ A\ ST\ (if\ (e)\ e_1\ else\ e_2) =$
$(let\ A_0 = A \sqcup \mathcal{A}\ e;\ \tau = ty_i'\ ST\ E\ A_0\ in$
$compT\ E\ A\ ST\ e\ @\ [after\ E\ A\ ST\ e,\ \tau]\ @$
$compT\ E\ A_0\ ST\ e_1\ @\ [after\ E\ A_0\ ST\ e_1,\ \tau]\ @$
$compT\ E\ A_0\ ST\ e_2)$
$|\ compT\ E\ A\ ST\ (while\ (e)\ c) =$
$(let\ A_0 = A \sqcup \mathcal{A}\ e;\ \ A_1 = A_0 \sqcup \mathcal{A}\ c;\ \tau = ty_i'\ ST\ E\ A_0\ in$
$compT\ E\ A\ ST\ e\ @\ [after\ E\ A\ ST\ e,\ \tau]\ @$
$compT\ E\ A_0\ ST\ c\ @\ [after\ E\ A_0\ ST\ c,\ ty_i'\ ST\ E\ A_1,\ ty_i'\ ST\ E\ A_0])$
$|\ compT\ E\ A\ ST\ (throw\ e) = compT\ E\ A\ ST\ e\ @\ [after\ E\ A\ ST\ e]$
$|\ compT\ E\ A\ ST\ (e{\cdot}M(es)) =$
$compT\ E\ A\ ST\ e\ @\ [after\ E\ A\ ST\ e]\ @$
$compTs\ E\ (A \sqcup \mathcal{A}\ e)\ (ty\quad E\ e\ \#\ ST)\ es$
$|\ compT\ E\ A\ ST\ (C{\cdot}_s M(es)) = compTs\ E\ A\ ST\ es$
$|\ compT\ E\ A\ ST\ (try\ e_1\ catch(C\ i)\ e_2) =$
$compT\ E\ A\ ST\ e_1\ @\ [after\ E\ A\ ST\ e_1]\ @$
$[ty_i'\ (Class\ C\#ST)\ E\ A,\ ty_i'\ ST\ (E@[Class\ C])\ (A \sqcup \lfloor\{i\}\rfloor)]\ @$
$compT\ (E@[Class\ C])\ (A \sqcup \lfloor\{i\}\rfloor)\ ST\ e_2$
$|\ compT\ E\ A\ ST\ (INIT\ C\ (Cs,b) \leftarrow e) = []$
$|\ compT\ E\ A\ ST\ (RI(C,e');Cs \leftarrow e) = []$
$|\ compTs\ E\ A\ ST\ [] = []$
$|\ compTs\ \ E\ A\ ST\ (e\#es) = compT\ E\ A\ ST\ e\ @\ [after\ E\ A\ ST\ e]\ @$
$compTs\ E\ (A \sqcup (\mathcal{A}\ e))\ (ty\ E\ e\ \#\ ST)\ es$

definition $compT_a :: ty\ list \Rightarrow nat\ hyperset \Rightarrow ty\ list \Rightarrow expr_1 \Rightarrow ty_i'\ list$ **where**
$compT_a\ E\ A\ ST\ e = compT\ E\ A\ ST\ e\ @\ [after\ E\ A\ ST\ e]$

end

lemma *compE$_2$-not-Nil*[*simp*]: $P,E \vdash_1 e :: T \Longrightarrow compE_2\ e \neq []\langle proof\rangle$
lemma (**in** *TC1*) *compT-sizes'*:
shows $\bigwedge E\ A\ ST.\ \neg sub\text{-}RI\ e \Longrightarrow size(compT\ E\ A\ ST\ e) = size(compE_2\ e) - 1$
and $\bigwedge E\ A\ ST.\ \neg sub\text{-}RIs\ es \Longrightarrow size(compTs\ E\ A\ ST\ es) = size(compEs_2\ es)\langle proof\rangle$
lemma (**in** *TC1*) *compT-sizes*[*simp*]:
shows $\bigwedge E\ A\ ST.\ P,E \vdash_1 e :: T \Longrightarrow size(compT\ E\ A\ ST\ e) = size(compE_2\ e) - 1$
and $\bigwedge E\ A\ ST.\ P,E \vdash_1 es\ [::]\ Ts \Longrightarrow size(compTs\ E\ A\ ST\ es) = size(compEs_2\ es)\langle proof\rangle$

lemma (**in** *TC1*) [*simp*]: $\bigwedge ST\ E.\ \lfloor\tau\rfloor \notin set\ (compT\ E\ None\ ST\ e)$
and [*simp*]: $\bigwedge ST\ E.\ \lfloor\tau\rfloor \notin set\ (compTs\ E\ None\ ST\ es)\langle proof\rangle$

lemma (**in** *TC0*) *pair-eq-ty$_i$'-conv*:
$(\lfloor(ST,\ LT)\rfloor = ty_i'\ ST_0\ E\ A) =$
$(case\ A\ of\ None \Rightarrow False \mid Some\ A \Rightarrow (ST = ST_0 \wedge LT = ty_l\ E\ A))\langle proof\rangle$

lemma (**in** *TC0*) *pair-conv-ty$_i$'*:
$\lfloor(ST,\ ty_l\ E\ A)\rfloor = ty_i'\ ST\ E\ \lfloor A\rfloor\langle proof\rangle$

lemma (**in** *TC1*) *compT-LT-prefix*:
$\bigwedge E\ A\ ST_0.\ [\![\ \lfloor(ST,LT)\rfloor \in set(compT\ E\ A\ ST_0\ e);\ \mathcal{B}\ e\ (size\ E)\ ]\!]$
$\Longrightarrow P \vdash \lfloor(ST,LT)\rfloor \leq' ty_i'\ ST\ E\ A$
and

$\bigwedge E\ A\ ST_0.\ [\![\ \lfloor(ST,LT)\rfloor \in set(compTs\ E\ A\ ST_0\ es);\ \mathcal{B}s\ es\ (size\ E)\ ]\!]$
$\Longrightarrow P \vdash \lfloor(ST,LT)\rfloor \leq' ty_i'\ ST\ E\ A\langle proof\rangle$

lemma [*iff*]: $OK\ None \in states\ P\ mxs\ mxl\langle proof\rangle$
lemma (**in** *TC0*) *after-in-states*:
assumes *wf*: *wf-prog p P* **and** *wt*: $P,E \vdash_1 e :: T$
and *Etypes*: $set\ E \subseteq types\ P$ **and** *STtypes*: $set\ ST \subseteq types\ P$
and *stack*: $size\ ST + max\text{-}stack\ e \leq mxs$
shows $OK\ (after\ E\ A\ ST\ e) \in states\ P\ mxs\ mxl\langle proof\rangle$

lemma (**in** *TC0*) *OK-ty$_i$'-in-statesI*[*simp*]:
$[\![\ set\ E \subseteq types\ P;\ set\ ST \subseteq types\ P;\ size\ ST \leq mxs\ ]\!]$
$\Longrightarrow OK\ (ty_i'\ ST\ E\ A) \in states\ P\ mxs\ mxl\langle proof\rangle$

lemma *is-class-type-aux*: $is\text{-}class\ P\ C \Longrightarrow is\text{-}type\ P\ (Class\ C)\langle proof\rangle$
theorem (**in** *TC1*) *compT-states*:
assumes *wf*: *wf-prog p P*
shows $\bigwedge E\ T\ A\ ST.$
$[\![\ P,E \vdash_1 e :: T;\ set\ E \subseteq types\ P;\ set\ ST \subseteq types\ P;$
$size\ ST + max\text{-}stack\ e \leq mxs;\ size\ E + max\text{-}vars\ e \leq mxl\ ]\!]$
$\Longrightarrow OK\ `\ set(compT\ E\ A\ ST\ e) \subseteq states\ P\ mxs\ mxl$
and $\bigwedge E\ Ts\ A\ ST.$
$[\![\ P,E \vdash_1 es[::]Ts;\ \ set\ E \subseteq types\ P;\ set\ ST \subseteq types\ P;$
$size\ ST + max\text{-}stacks\ es \leq mxs;\ size\ E + max\text{-}varss\ es \leq mxl\ ]\!]$
$\Longrightarrow OK\ `\ set(compTs\ E\ A\ ST\ es) \subseteq states\ P\ mxs\ mxl\langle proof\rangle$

definition $shift :: nat \Rightarrow ex\text{-}table \Rightarrow ex\text{-}table$
where
$shift\ n\ xt \equiv map\ (\lambda(from,to,C,handler,depth).\ (from+n,to+n,C,handler+n,depth))\ xt$

lemma [*simp*]: $shift\ 0\ xt = xt\langle proof\rangle$
lemma [*simp*]: $shift\ n\ [] = []\langle proof\rangle$
lemma [*simp*]: $shift\ n\ (xt_1\ @\ xt_2) = shift\ n\ xt_1\ @\ shift\ n\ xt_2\langle proof\rangle$
lemma [*simp*]: $shift\ m\ (shift\ n\ xt) = shift\ (m+n)\ xt\langle proof\rangle$
lemma [*simp*]: $pcs\ (shift\ n\ xt) = \{pc+n|pc.\ pc \in pcs\ xt\}\langle proof\rangle$

lemma *shift-compxE$_2$*:
shows $\bigwedge pc\ pc'\ d.\ shift\ pc\ (compxE_2\ e\ pc'\ d) = compxE_2\ e\ (pc' + pc)\ d$
and $\bigwedge pc\ pc'\ d.\ shift\ pc\ (compxEs_2\ es\ pc'\ d) = compxEs_2\ es\ (pc' + pc)\ d\langle proof\rangle$

lemma *compxE$_2$-size-convs*[*simp*]:
shows $n \neq 0 \Longrightarrow compxE_2\ e\ n\ d = shift\ n\ (compxE_2\ e\ 0\ d)$
and $n \neq 0 \Longrightarrow compxEs_2\ es\ n\ d = shift\ n\ (compxEs_2\ es\ 0\ d)\langle proof\rangle$
locale *TC2* = *TC1* +
fixes $T_r :: ty$ **and** $mxs :: pc$
begin

definition
$wt\text{-}instrs :: instr\ list \Rightarrow ex\text{-}table \Rightarrow ty_i'\ list \Rightarrow bool$
$(\langle(\vdash -, - /[::]/ -)\rangle\ [0,0,51]\ 50)$ **where**
$\vdash is,xt\ [::]\ \tau s \longleftrightarrow size\ is < size\ \tau s \wedge pcs\ xt \subseteq \{0..<size\ is\} \wedge$
$(\forall pc<\ size\ is.\ P,T_r,mxs,size\ \tau s,xt \vdash is!pc,pc :: \tau s)$

end

notation *TC2.wt-instrs* (‹(-,-,- ⊢/ -, - /[::]/ -)› [*50*,*50*,*50*,*50*,*50*,*51*] *50*)

lemma (**in** *TC2*) [*simp*]: $\tau s \neq [] \Longrightarrow \vdash [],[]\ [::]\ \tau s\langle proof\rangle$
lemma [*simp*]: *eff i P pc et None* = []⟨*proof*⟩
lemma *wt-instr-appR*:
⟦ *P*,*T*,*m*,*mpc*,*xt* ⊢ *is*!*pc*,*pc* :: τs;
pc < *size is*; *size is* < *size* τs; *mpc* ≤ *size* τs; *mpc* ≤ *mpc*′ ⟧
⟹ *P*,*T*,*m*,*mpc*′,*xt* ⊢ *is*!*pc*,*pc* :: $\tau s @ \tau s'\langle proof\rangle$

lemma *relevant-entries-shift* [*simp*]:
relevant-entries P i (*pc*+*n*) (*shift n xt*) = *shift n* (*relevant-entries P i pc xt*)⟨*proof*⟩

lemma [*simp*]:
xcpt-eff i P (*pc*+*n*) τ (*shift n xt*) =
map (λ(*pc*,τ). (*pc* + *n*, τ)) (*xcpt-eff i P pc* τ *xt*)⟨*proof*⟩

lemma [*simp*]:
app_i (*i*, *P*, *pc*, *m*, *T*, τ) ⟹
eff i P (*pc*+*n*) (*shift n xt*) (*Some* τ) =
map (λ(*pc*,τ). (*pc*+*n*,τ)) (*eff i P pc xt* (*Some* τ))⟨*proof*⟩

lemma [*simp*]:
xcpt-app i P (*pc*+*n*) *mxs* (*shift n xt*) τ = *xcpt-app i P pc mxs xt* $\tau\langle proof\rangle$

lemma *wt-instr-appL*:
assumes *P*,*T*,*m*,*mpc*,*xt* ⊢ *i*,*pc* :: τs **and** *pc* < *size* τs **and** *mpc* ≤ *size* τs
shows *P*,*T*,*m*,*mpc* + *size* $\tau s'$,*shift* (*size* $\tau s'$) *xt* ⊢ *i*,*pc*+*size* $\tau s'$ :: $\tau s' @ \tau s\langle proof\rangle$

lemma *wt-instr-Cons*:
assumes *wti*: *P*,*T*,*m*,*mpc* − *1*,[] ⊢ *i*,*pc* − *1* :: τs
and *pcl*: *0* < *pc* **and** *mpcl*: *0* < *mpc*
and *pcu*: *pc* < *size* τs + *1* **and** *mpcu*: *mpc* ≤ *size* τs + *1*
shows *P*,*T*,*m*,*mpc*,[] ⊢ *i*,*pc* :: $\tau \# \tau s\langle proof\rangle$

lemma *wt-instr-append*:
assumes *wti*: *P*,*T*,*m*,*mpc* − *size* $\tau s'$,[] ⊢ *i*,*pc* − *size* $\tau s'$ :: τs
and *pcl*: *size* $\tau s'$ ≤ *pc* **and** *mpcl*: *size* $\tau s'$ ≤ *mpc*
and *pcu*: *pc* < *size* τs + *size* $\tau s'$ **and** *mpcu*: *mpc* ≤ *size* τs + *size* $\tau s'$
shows *P*,*T*,*m*,*mpc*,[] ⊢ *i*,*pc* :: $\tau s' @ \tau s\langle proof\rangle$

lemma *xcpt-app-pcs*:
pc ∉ *pcs xt* ⟹ *xcpt-app i P pc mxs xt* $\tau\langle proof\rangle$

lemma *xcpt-eff-pcs*:
pc ∉ *pcs xt* ⟹ *xcpt-eff i P pc* τ *xt* = []⟨*proof*⟩

lemma *pcs-shift*:
pc < *n* ⟹ *pc* ∉ *pcs* (*shift n xt*)⟨*proof*⟩

lemma *wt-instr-appRx*:
⟦ *P*,*T*,*m*,*mpc*,*xt* ⊢ *is*!*pc*,*pc* :: τs; *pc* < *size is*; *size is* < *size* τs; *mpc* ≤ *size* τs ⟧
⟹ *P*,*T*,*m*,*mpc*,*xt* @ *shift* (*size is*) *xt*′ ⊢ *is*!*pc*,*pc* :: $\tau s\langle proof\rangle$

lemma *wt-instr-appLx*:
⟦ $P,T,m,mpc,xt \vdash i,pc :: \tau s$; $pc \notin pcs\ xt'$ ⟧
⟹ $P,T,m,mpc,xt'@xt \vdash i,pc :: \tau s\langle proof\rangle$

lemma (**in** *TC2*) *wt-instrs-extR*:
$\vdash is,xt\ [::]\ \tau s \Longrightarrow \vdash is,xt\ [::]\ \tau s\ @\ \tau s'\langle proof\rangle$

lemma (**in** *TC2*) *wt-instrs-ext*:
assumes wt_1: $\vdash is_1,xt_1\ [::]\ \tau s_1@\tau s_2$ **and** wt_2: $\vdash is_2,xt_2\ [::]\ \tau s_2$
and τs-*size*: *size* τs_1 = *size* is_1
shows $\vdash is_1@is_2,\ xt_1\ @\ shift\ (size\ is_1)\ xt_2\ [::]\ \tau s_1@\tau s_2\langle proof\rangle$
corollary (**in** *TC2*) *wt-instrs-ext2*:
⟦ $\vdash is_2,xt_2\ [::]\ \tau s_2$; $\vdash is_1,xt_1\ [::]\ \tau s_1@\tau s_2$; *size* τs_1 = *size* is_1 ⟧
⟹ $\vdash is_1@is_2,\ xt_1\ @\ shift\ (size\ is_1)\ xt_2\ [::]\ \tau s_1@\tau s_2\langle proof\rangle$

corollary (**in** *TC2*) *wt-instrs-ext-prefix* [*trans*]:
⟦ $\vdash is_1,xt_1\ [::]\ \tau s_1@\tau s_2$; $\vdash is_2,xt_2\ [::]\ \tau s_3$;
size τs_1 = *size* is_1; *prefix* $\tau s_3\ \tau s_2$ ⟧
⟹ $\vdash is_1@is_2,\ xt_1\ @\ shift\ (size\ is_1)\ xt_2\ [::]\ \tau s_1@\tau s_2\langle proof\rangle$

corollary (**in** *TC2*) *wt-instrs-app*:
assumes is_1: $\vdash is_1,xt_1\ [::]\ \tau s_1@[\tau]$
assumes is_2: $\vdash is_2,xt_2\ [::]\ \tau\#\tau s_2$
assumes s: *size* τs_1 = *size* is_1
shows $\vdash is_1@is_2,\ xt_1@shift\ (size\ is_1)\ xt_2\ [::]\ \tau s_1@\tau\#\tau s_2\langle proof\rangle$

corollary (**in** *TC2*) *wt-instrs-app-last*[*trans*]:
assumes $\vdash is_2,xt_2\ [::]\ \tau\#\tau s_2 \vdash is_1,xt_1\ [::]\ \tau s_1$
last $\tau s_1 = \tau$ *size* τs_1 = *size* is_1+*1*
shows $\vdash is_1@is_2,\ xt_1@shift\ (size\ is_1)\ xt_2\ [::]\ \tau s_1@\tau s_2\langle proof\rangle$

corollary (**in** *TC2*) *wt-instrs-append-last*[*trans*]:
assumes *wtis*: $\vdash is,xt\ [::]\ \tau s$ **and** *wti*: $P,T_r,mxs,mpc,[] \vdash i,pc :: \tau s$
and *pc*: *pc* = *size is* **and** *mpc*: *mpc* = *size* τs **and** *is-*τs: *size is* + *1* < *size* τs
shows $\vdash is@[i],xt\ [::]\ \tau s\langle proof\rangle$

corollary (**in** *TC2*) *wt-instrs-app2*:
⟦ $\vdash is_2,xt_2\ [::]\ \tau'\#\tau s_2$; $\vdash is_1,xt_1\ [::]\ \tau\#\tau s_1@[\tau']$;
$xt' = xt_1\ @\ shift\ (size\ is_1)\ xt_2$; *size* τs_1+*1* = *size* is_1 ⟧
⟹ $\vdash is_1@is_2,xt'\ [::]\ \tau\#\tau s_1@\tau'\#\tau s_2\langle proof\rangle$

corollary (**in** *TC2*) *wt-instrs-app2-simp*[*trans*,*simp*]:
⟦ $\vdash is_2,xt_2\ [::]\ \tau'\#\tau s_2$; $\vdash is_1,xt_1\ [::]\ \tau\#\tau s_1@[\tau']$; *size* τs_1+*1* = *size* is_1 ⟧
⟹ $\vdash is_1@is_2,\ xt_1@shift\ (size\ is_1)\ xt_2\ [::]\ \tau\#\tau s_1@\tau'\#\tau s_2\langle proof\rangle$

corollary (**in** *TC2*) *wt-instrs-Cons*[*simp*]:
⟦ $\tau s \neq []$; $\vdash [i],[]\ [::]\ [\tau,\tau']$; $\vdash is,xt\ [::]\ \tau'\#\tau s$ ⟧
⟹ $\vdash i\#is,shift\ 1\ xt\ [::]\ \tau\#\tau'\#\tau s\langle proof\rangle\langle proof\rangle\langle proof\rangle\langle proof\rangle\langle proof\rangle\langle proof\rangle\langle proof\rangle\langle proof\rangle\langle proof\rangle\langle proof\rangle\langle proof\rangle\langle proof\rangle\langle proof\rangle\langle pro$
theory *JinjaDCI*
imports
J/Equivalence
J/Annotate
JVM/JVMDefensive

BV/BVExec
BV/LBVJVM
BV/BVNoTypeError
Compiler/TypeComp
begin

end

Bibliography

[1] S. Mansky and E. L. Gunter. Dynamic class initialization semantics: A jinja extension. In *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2019, pages 209–221, New York, NY, USA, 2019. Association for Computing Machinery.