JinjaDCI: a Java semantics with dynamic class initialization

Susannah Mansky

March 17, 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*]:
$((n::nat) + max\ i\ j \le m) = (n + i \le m \wedge n + j \le m)$

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

notation *Some* (‹(⌊-⌋)›)

1.1.1 *distinct-fst*

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

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

lemma *distinct-fst-Cons* [*simp*]:
distinct-fst $((k,x)\#kxs) = ($*distinct-fst* $kxs \wedge (\forall y.\ (k,y) \notin set\ kxs))$
lemma *distinct-fst-appendD*:
distinct-fst$(kxs\ @\ kxs') \Longrightarrow$ *distinct-fst* $kxs \wedge$ *distinct-fst* kxs'
lemma *map-of-SomeI*:
$[\![$ *distinct-fst* $kxs;\ (k,x) \in set\ kxs\ ]\!] \Longrightarrow$ *map-of* $kxs\ k = Some\ x$

1.1.2 Using *list-all2* for relations

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

Convenience lemmas

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

lemma *rel-list-all2-Cons1*:
$list\text{-}all2\ (fun\text{-}of\ S)\ (x\#xs)\ ys =$
$(\exists z\ zs.\ ys = z\#zs \land (x,z) \in S \land list\text{-}all2\ (fun\text{-}of\ S)\ xs\ zs)$

lemma *rel-list-all2-Cons2*:
$list\text{-}all2\ (fun\text{-}of\ S)\ xs\ (y\#ys) =$
$(\exists z\ zs.\ xs = z\#zs \land (z,y) \in S \land list\text{-}all2\ (fun\text{-}of\ S)\ zs\ ys)$

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

lemma *rel-list-all2-antisym*:
$[\![\ (\bigwedge x\ y.\ [\![(x,y) \in S;\ (y,x) \in T]\!] \Longrightarrow x = y);$
$list\text{-}all2\ (fun\text{-}of\ S)\ xs\ ys;\ list\text{-}all2\ (fun\text{-}of\ T)\ ys\ xs\ ]\!] \Longrightarrow xs = ys$

lemma *rel-list-all2-trans*:
$[\![\ \bigwedge a\ b\ c.\ [\![(a,b) \in R;\ (b,c) \in S]\!] \Longrightarrow (a,c) \in T;$
$list\text{-}all2\ (fun\text{-}of\ R)\ as\ bs;\ list\text{-}all2\ (fun\text{-}of\ S)\ bs\ cs]\!]$
$\Longrightarrow list\text{-}all2\ (fun\text{-}of\ T)\ as\ cs$

lemma *rel-list-all2-update-cong*:
$[\![\ i<size\ xs;\ list\text{-}all2\ (fun\text{-}of\ S)\ xs\ ys;\ (x,y) \in S\ ]\!]$
$\Longrightarrow list\text{-}all2\ (fun\text{-}of\ S)\ (xs[i:=x])\ (ys[i:=y])$

lemma *rel-list-all2-nthD*:
$[\![\ list\text{-}all2\ (fun\text{-}of\ S)\ xs\ ys;\ p < size\ xs\ ]\!] \Longrightarrow (xs!p,ys!p) \in S$

lemma *rel-list-all2I*:
$[\![\ length\ a = length\ b;\ \bigwedge n.\ n < length\ a \Longrightarrow (a!n,b!n) \in S\ ]\!] \Longrightarrow list\text{-}all2\ (fun\text{-}of\ S)\ a\ b$

1.1.3 Auxiliary properties of *map-of* function

lemma *map-of-set-pcs-notin*: $C \notin (\lambda t.\ snd\ (fst\ t))\ `\ set\ FDTs \Longrightarrow map\text{-}of\ FDTs\ (F,\ C) = None$
lemma *map-of-insertmap-SomeD'*:
$map\text{-}of\ fs\ F = Some\ y \Longrightarrow map\text{-}of\ (map\ (\lambda(F,\ y).\ (F,\ D,\ y))\ fs)\ F = Some(D,y)$
lemma *map-of-reinsert-neq-None*:
$Ca \neq D \Longrightarrow map\text{-}of\ (map\ (\lambda(F,\ y).\ ((F,\ Ca),\ y))\ fs)\ (F,\ D) = None$
lemma *map-of-remap-insertmap*:
$map\text{-}of\ (map\ ((\lambda((F,\ D),\ b,\ T).\ (F,\ D,\ b,\ T)) \circ (\lambda(F,\ y).\ ((F,\ D),\ y)))\ fs)$
$= map\text{-}of\ (map\ (\lambda(F,\ y).\ (F,\ D,\ y))\ fs)$

lemma *map-of-reinsert-SomeD*:
$map\text{-}of\ (map\ (\lambda(F,\ y).\ ((F,\ D),\ y))\ fs)\ (F,\ D) = Some\ T \Longrightarrow map\text{-}of\ fs\ F = Some\ T$
lemma *map-of-filtered-SomeD*:
$map\text{-}of\ fs\ (F,D) = Some\ (a,\ T) \Longrightarrow Q\ ((F,D),a,T) \Longrightarrow$
$map\text{-}of\ (map\ (\lambda((F,D),\ b,\ T).\ ((F,D),\ P\ T))\ (filter\ Q\ fs))$
$(F,D) = Some\ (P\ T)$

lemma *map-of-remove-filtered-SomeD*:
$map\text{-}of\ fs\ (F,C) = Some\ (a,\ T) \Longrightarrow Q\ ((F,C),a,T) \Longrightarrow$
$map\text{-}of\ (map\ (\lambda((F,D),\ b,\ T).\ (F,\ P\ T))\ [((F,\ D),\ b,\ T) \leftarrow fs\ .\ Q\ ((F,\ D),\ b,\ T) \land D = C])$
$F = Some\ (P\ T)$

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 $map\text{-}of\ (map\ (\lambda((F, D), b, T).\ (F, D, b, T))\ t)\ F = Some\ (C, y)$
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 ≡ *''Object''*

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

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

definition *start-m* :: *string* **where** *start-m* = *''<start>''*
definition *Start* :: *string* **where** *Start* = *''<Start>''*

lemma *start-m-neq-clinit* [*simp*]: *start-m* ≠ *clinit* **by**(*simp add*: *start-m-def clinit-def*)
lemma *Object-neq-Start* [*simp*]: *Object* ≠ *Start* **by**(*simp add*: *Object-def Start-def*)
lemma *Start-neq-Object* [*simp*]: *Start* ≠ *Object* **by**(*simp add*: *Object-def Start-def*)

— field/method static flag

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
$is\text{-}refT\ T\ \equiv\ T = NT \lor (\exists C.\ T = Class\ C)$

lemma [*iff*]: *is-refT NT*
lemma [*iff*]: *is-refT*(*Class C*)
lemma *refTE*:
$[\![is\text{-}refT\ T;\ T = NT \Longrightarrow P;\ \bigwedge C.\ T = Class\ C \Longrightarrow P\ ]\!] \Longrightarrow P$
lemma *not-refTE*:

$[\![\ \neg is\text{-}refT\ T;\ T = Void \lor T = Boolean \lor T = Integer \Longrightarrow P\ ]\!] \Longrightarrow P$
end

1.3 Class Declarations and Programs

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

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

definition $class :: 'm\ prog \Rightarrow cname \rightharpoonup 'm\ class$
where
$class \equiv map\text{-}of$

lemma *class-cons*: $[\![\ C \neq fst\ x\ ]\!] \Longrightarrow class\ (x \# P)\ C = class\ P\ C$
by (*simp add*: *class-def*)

definition $is\text{-}class :: 'm\ prog \Rightarrow cname \Rightarrow bool$
where
$is\text{-}class\ P\ C \equiv class\ P\ C \neq None$

lemma *finite-is-class*: $finite\ \{C.\ is\text{-}class\ P\ C\}$
definition $is\text{-}type :: 'm\ prog \Rightarrow ty \Rightarrow bool$
where
$is\text{-}type\ P\ T \equiv$
$(case\ T\ of\ Void \Rightarrow True \mid Boolean \Rightarrow True \mid Integer \Rightarrow True \mid NT \Rightarrow True$
$\mid Class\ C \Rightarrow is\text{-}class\ P\ C)$

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

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

lemma *class-exists-equiv*:
$(\exists x.\ fst\ x = cn \land x \in set\ P) = (class\ P\ cn \neq None)$
proof(*rule iffI*)
assume $\exists x.\ fst\ x = cn \land x \in set\ P$ **then show** $class\ P\ cn \neq None$
by (*metis class-def image-eqI map-of-eq-None-iff*)
next
assume $class\ P\ cn \neq None$ **then show** $\exists x.\ fst\ x = cn \land x \in set\ P$

by (*metis class-def fst-conv map-of-SomeD option.exhaust*)
qed

lemma *class-exists-equiv2*:
$(\exists x.\ fst\ x = cn \wedge x \in set\ (P1\ @\ P2)) = (class\ P1\ cn \neq None \vee class\ P2\ cn \neq None)$
by (*simp only*: *class-exists-equiv* [**where** $P = P1@P2$], *simp add*: *class-def*)

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 :: $'m\ prog \Rightarrow (cname \times cname)\ set$
and *subcls1'* :: $'m\ prog \Rightarrow [cname, cname] \Rightarrow bool$ (‹- ⊢ - $\prec^1$ -› [71,71,71] 70)
for P :: $'m\ prog$
where
$P \vdash C\ \prec^1\ D \equiv (C,D) \in subcls1\ P$
| *subcls1I*: $[\![class\ P\ C = Some\ (D,rest);\ C \neq Object]\!] \Longrightarrow P \vdash C \prec^1 D$

abbreviation
subcls :: $'m\ prog \Rightarrow [cname, cname] \Rightarrow bool$ (‹- ⊢ - $\preceq^*$ -› [71,71,71] 70)
where $P \vdash C\ \preceq^*\ D \equiv (C,D) \in (subcls1\ P)^*$

lemma *subcls1D*: $P \vdash C \prec^1 D \Longrightarrow C \neq Object \wedge (\exists fs\ ms.\ class\ P\ C = Some\ (D,fs,ms))$
lemma [*iff*]: $\neg\ P \vdash Object \prec^1 C$
lemma [*iff*]: $(P \vdash Object \preceq^* C) = (C = Object)$
lemma *subcls1-def2*:
$subcls1\ P =$
$(SIGMA\ C{:}\{C.\ is\text{-}class\ P\ C\}.\ \{D.\ C{\neq}Object \wedge fst\ (the\ (class\ P\ C)){=}D\})$
lemma *finite-subcls1*: *finite* (*subcls1* P)
primrec *supercls-lst* :: $'m\ prog \Rightarrow cname\ list \Rightarrow bool$ **where**
$supercls\text{-}lst\ P\ (C\#Cs) = ((\forall C' \in set\ Cs.\ P \vdash C' \preceq^* C) \wedge supercls\text{-}lst\ P\ Cs)$ |
$supercls\text{-}lst\ P\ [] = True$

lemma *supercls-lst-app*:
$[\![\ supercls\text{-}lst\ P\ (C\#Cs);\ P \vdash C \preceq^* C'\ ]\!] \Longrightarrow supercls\text{-}lst\ P\ (C'\#C\#Cs)$
by *auto*

1.4.2 The subtype relations

inductive
widen :: $'m\ prog \Rightarrow ty \Rightarrow ty \Rightarrow bool$ (‹- ⊢ - ≤ -› [71,71,71] 70)
for P :: $'m\ prog$
where
widen-refl[*iff*]: $P \vdash T \leq T$
| *widen-subcls*: $P \vdash C \preceq^* D \implies P \vdash Class\ C \leq Class\ D$
| *widen-null*[*iff*]: $P \vdash NT \leq Class\ C$

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

lemma *Class-widen*: $P \vdash Class\ C \leq T \implies \exists D.\ T = Class\ D$
lemma [*iff*]: $(P \vdash T \leq NT) = (T = NT)$
lemma *Class-widen-Class* [*iff*]: $(P \vdash Class\ C \leq Class\ D) = (P \vdash C \preceq^* D)$
lemma *widen-Class*: $(P \vdash T \leq Class\ C) = (T = NT \lor (\exists D.\ T = Class\ D \land P \vdash D \preceq^* C))$

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

1.4.3 Method lookup

inductive
Methods :: [$'m$ *prog*, *cname*, *mname* ⇀ (*staticb* × *ty list* × *ty* × $'m$) × *cname*] ⇒ *bool*
(‹- ⊢ - *sees'-methods* -› [*51*,*51*,*51*] *50*)
for *P* :: $'m$ *prog*
where
sees-methods-Object:
⟦ *class P Object* = *Some*(*D*,*fs*,*ms*); *Mm* = *map-option* (λ*m*. (*m*,*Object*)) ∘ *map-of ms* ⟧
⟹ *P* ⊢ *Object sees-methods Mm*
| *sees-methods-rec*:
⟦ *class P C* = *Some*(*D*,*fs*,*ms*); *C* ≠ *Object*; *P* ⊢ *D sees-methods Mm*;
Mm' = *Mm* ++ (*map-option* (λ*m*. (*m*,*C*)) ∘ *map-of ms*) ⟧
⟹ *P* ⊢ *C sees-methods Mm'*

lemma *sees-methods-fun*:
assumes *1*: *P* ⊢ *C sees-methods Mm*
shows ⋀*Mm'*. *P* ⊢ *C sees-methods Mm'* ⟹ *Mm'* = *Mm*

lemma *visible-methods-exist*:
P ⊢ *C sees-methods Mm* ⟹ *Mm M* = *Some*(*m*,*D*) ⟹
(∃ *D' fs ms*. *class P D* = *Some*(*D'*,*fs*,*ms*) ∧ *map-of ms M* = *Some m*)

lemma *sees-methods-decl-above*:
assumes *Csees*: *P* ⊢ *C sees-methods Mm*
shows *Mm M* = *Some*(*m*,*D*) ⟹ $P \vdash C \preceq^* D$

lemma *sees-methods-idemp*:
assumes *Cmethods*: *P* ⊢ *C sees-methods Mm*
shows ⋀*m D*. *Mm M* = *Some*(*m*,*D*) ⟹
∃ *Mm'*. (*P* ⊢ *D sees-methods Mm'*) ∧ *Mm' M* = *Some*(*m*,*D*)

lemma *sees-methods-decl-mono*:
assumes *sub*: $P \vdash C' \preceq^* C$
shows $P \vdash C$ *sees-methods* $Mm \Longrightarrow$
$\exists Mm'\ Mm_2.\ P \vdash C'$ *sees-methods* $Mm' \wedge Mm' = Mm$ ++ $Mm_2 \wedge$
$(\forall M\ m\ D.\ Mm_2\ M = Some(m,D) \longrightarrow P \vdash D \preceq^* C)$
lemma *sees-methods-is-class-Object*:
$P \vdash D$ *sees-methods* $Mm \Longrightarrow$ *is-class P Object*
by(*induct rule*: *Methods.induct*; *simp add*: *is-class-def*)

lemma *sees-methods-sub-Obj*: $P \vdash C$ *sees-methods* $Mm \Longrightarrow P \vdash C \preceq^*$ *Object*
proof(*induct rule*: *Methods.induct*)
case (*sees-methods-rec C D fs ms Mm Mm'*) **show** *?case*
using *subcls1I*[*OF sees-methods-rec.hyps*(*1*,*2*)] *sees-methods-rec.hyps*(*4*)
by(*rule converse-rtrancl-into-rtrancl*)
qed(*simp*)

definition *Method* :: $'m$ *prog* $\Rightarrow$ *cname* $\Rightarrow$ *mname* $\Rightarrow$ *staticb* $\Rightarrow$ *ty list* $\Rightarrow$ *ty* $\Rightarrow$ $'m$ $\Rightarrow$ *cname* $\Rightarrow$ *bool*
(‹- ⊢ - *sees* -, - : -→- = - *in* -› [*51*,*51*,*51*,*51*,*51*,*51*,*51*,*51*] *50*)
where
$P \vdash C$ *sees* M, b: $Ts \rightarrow T = m$ *in* $D \equiv$
$\exists Mm.\ P \vdash C$ *sees-methods* $Mm \wedge Mm\ M = Some((b,Ts,T,m),D)$

definition *has-method* :: $'m$ *prog* $\Rightarrow$ *cname* $\Rightarrow$ *mname* $\Rightarrow$ *staticb* $\Rightarrow$ *bool*
(‹- ⊢ - *has* -, -› [*51*,*0*,*0*,*51*] *50*)
where
$P \vdash C$ *has* M, $b \equiv \exists Ts\ T\ m\ D.\ P \vdash C$ *sees* M,b:$Ts \rightarrow T = m$ *in* D

lemma *sees-method-fun*:
$[\![P \vdash C$ *sees* M,b:$TS \rightarrow T = m$ *in* D; $P \vdash C$ *sees* M,b':$TS' \rightarrow T' = m'$ *in* $D'\ ]\!]$
$\Longrightarrow b = b' \wedge TS' = TS \wedge T' = T \wedge m' = m \wedge D' = D$

lemma *sees-method-decl-above*:
$P \vdash C$ *sees* M,b:$Ts \rightarrow T = m$ *in* $D \Longrightarrow P \vdash C \preceq^* D$

lemma *visible-method-exists*:
$P \vdash C$ *sees* M,b:$Ts \rightarrow T = m$ *in* $D \Longrightarrow$
$\exists D'$ *fs ms*. *class P D* $= Some(D',fs,ms) \wedge$ *map-of ms M* $= Some(b,Ts,T,m)$

lemma *sees-method-idemp*:
$P \vdash C$ *sees* M,b:$Ts \rightarrow T$=m *in* $D \Longrightarrow P \vdash D$ *sees* M,b:$Ts \rightarrow T$=m *in* D

lemma *sees-method-decl-mono*:
assumes *sub*: $P \vdash C' \preceq^* C$ **and**
C-sees: $P \vdash C$ *sees* M,b:$Ts \rightarrow T$=m *in* D **and**
C'-sees: $P \vdash C'$ *sees* M,b':$Ts' \rightarrow T'$=m' *in* D'
shows $P \vdash D' \preceq^* D$

lemma *sees-methods-is-class*: $P \vdash C$ *sees-methods* $Mm \Longrightarrow$ *is-class P C*
lemma *sees-method-is-class*:
$[\![\ P \vdash C$ *sees* M,b:$Ts \rightarrow T$=m *in* $D\ ]\!] \Longrightarrow$ *is-class P C*
lemma *sees-method-is-class'*:
$[\![\ P \vdash C$ *sees* M,b:$Ts \rightarrow T$=m *in* $D\ ]\!] \Longrightarrow$ *is-class P D*

lemma *sees-method-sub-Obj*: $P \vdash C$ *sees* M,b: $Ts \rightarrow T = m$ *in* $D \Longrightarrow P \vdash C \preceq^* Object$
by(*auto simp*: *Method-def sees-methods-sub-Obj*)

1.4.4 Field lookup

inductive
Fields :: $['m\ prog,\ cname,\ ((vname \times cname) \times staticb \times ty)\ list] \Rightarrow bool$
(‹- ⊢ - has′-fields -› [51,51,51] 50)
for P :: *'m prog*
where
has-fields-rec:
$[\![$ *class* $P\ C = Some(D,fs,ms)$; $C \neq Object$; $P \vdash D$ *has-fields FDTs*;
$FDTs' = map\ (\lambda(F,b,T).\ ((F,C),b,T))\ fs\ @\ FDTs\ ]\!]$
$\Longrightarrow P \vdash C$ *has-fields FDTs′*
| *has-fields-Object*:
$[\![$ *class* $P\ Object = Some(D,fs,ms)$; $FDTs = map\ (\lambda(F,b,T).\ ((F,Object),b,T))\ fs\ ]\!]$
$\Longrightarrow P \vdash Object$ *has-fields FDTs*

lemma *has-fields-is-class*:
$P \vdash C$ *has-fields FDTs* $\Longrightarrow$ *is-class P C*
lemma *has-fields-fun*:
assumes *1*: $P \vdash C$ *has-fields FDTs*
shows $\bigwedge FDTs'.\ P \vdash C$ *has-fields FDTs′* $\Longrightarrow FDTs' = FDTs$

lemma *all-fields-in-has-fields*:
assumes *sub*: $P \vdash C$ *has-fields FDTs*
shows $[\![\ P \vdash C \preceq^* D$; *class* $P\ D = Some(D',fs,ms)$; $(F,b,T) \in set\ fs\ ]\!]$
$\Longrightarrow ((F,D),b,T) \in set\ FDTs$
lemma *has-fields-decl-above*:
assumes *fields*: $P \vdash C$ *has-fields FDTs*
shows $((F,D),b,T) \in set\ FDTs \Longrightarrow P \vdash C \preceq^* D$

lemma *subcls-notin-has-fields*:
assumes *fields*: $P \vdash C$ *has-fields FDTs*
shows $((F,D),b,T) \in set\ FDTs \Longrightarrow (D,C) \notin (subcls1\ P)^+$
lemma *subcls-notin-has-fields2*:
assumes *fields*: $P \vdash C$ *has-fields FDTs*
shows $[\![\ C \neq Object;\ P \vdash C \prec^1 D\ ]\!] \Longrightarrow (D,C) \notin (subcls1\ P)^*$
using *fields* **proof**(*induct arbitrary*: *D*)
case *has-fields-rec*
have $\forall\ C\ C'\ P.\ (C,\ C') \notin subcls1\ P \lor C \neq Object \land (\exists fs\ ms.\ class\ P\ C = \lfloor(C',\ fs,\ ms)\rfloor)$
using *subcls1D* **by** *blast*
then have $(D,\ D) \notin (subcls1\ P)^+$
by (*metis* (*no-types*) *Pair-inject has-fields-rec.hyps*(*1*) *has-fields-rec.hyps*(*4*)
has-fields-rec.prems(*2*) *option.inject tranclD*)
then show *?case*
by (*meson has-fields-rec.prems*(*2*) *rtrancl-into-trancl1*)
qed(*fastforce dest*: *tranclD*)

lemma *has-fields-mono-lem*:
assumes *sub*: $P \vdash D \preceq^* C$
shows $P \vdash C$ *has-fields FDTs*
$\Longrightarrow \exists pre.\ P \vdash D$ *has-fields pre@FDTs* $\land\ dom(map\text{-}of\ pre) \cap dom(map\text{-}of\ FDTs) = \{\}$
lemma *has-fields-declaring-classes*:

shows $P \vdash C\ \textit{has-fields}\ FDTs$
$\Longrightarrow \exists\, pre\ FDTs'.\ FDTs = pre@FDTs'$
$\wedge\ (C \neq Object \longrightarrow (\exists\, D\ fs\ ms.\ class\ P\ C = \lfloor(D,fs,ms)\rfloor \wedge P \vdash D\ \textit{has-fields}\ FDTs'))$
$\wedge\ set(map\ (\lambda t.\ snd(fst\ t))\ pre) \subseteq \{C\}$
$\wedge\ set(map\ (\lambda t.\ snd(fst\ t))\ FDTs') \subseteq \{C'.\ C' \neq C \wedge P \vdash C \preceq^* C'\}$
proof(*induct rule*:*Fields.induct*)
case (*has-fields-rec C D fs ms FDTs FDTs'*)
have *sup1*: $P \vdash C \prec^1 D$ **using** *has-fields-rec.hyps*(*1*,*2*) **by** (*simp add*: *subcls1.subcls1I*)
have $P \vdash C\ \textit{has-fields}\ FDTs'$
using *Fields.has-fields-rec*[*OF has-fields-rec.hyps*(*1*−*3*)] *has-fields-rec* **by** *auto*
then have *nsup*: $(D,\ C) \notin (subcls1\ P)^*$ **using** *subcls-notin-has-fields2 sup1* **by** *auto*
show *?case* **using** *has-fields-rec sup1 nsup*
by(*rule-tac* $x = map\ (\lambda(F,\ y).\ ((F,\ C),\ y))\ fs$ **in** *exI*, *clarsimp*) *auto*
next
case *has-fields-Object* **then show** *?case* **by** *fastforce*
qed

lemma *has-fields-mono-lem2*:
assumes *hf*: $P \vdash C\ \textit{has-fields}\ FDTs$
and *cls*: $class\ P\ C = Some(D,fs,ms)$ **and** *map-of*: $\textit{map-of}\ FDTs\ (F,C) = \lfloor(b,T)\rfloor$
shows $\exists\, FDTs'.\ FDTs = (map\ (\lambda(F,b,T).\ ((F,C),b,T))\ fs)\ @\ FDTs' \wedge \textit{map-of}\ FDTs'\ (F,C) = None$
using *assms*
proof(*cases* $C = Object$)
case *False*
let $?pre = map\ (\lambda(F,b,T).\ ((F,C),b,T))\ fs$
have *sub*: $P \vdash C \preceq^* D$ **using** *cls False* **by** (*simp add*: *r-into-rtrancl subcls1.subcls1I*)
obtain *FDTs'* **where** *fdts'*: $P \vdash D\ \textit{has-fields}\ FDTs'\ FDTs = ?pre\ @\ FDTs'$
using *False assms*(*1*,*2*) *Fields.simps*[*of P C FDTs*] **by** *clarsimp*
then have *int*: $dom\ (\textit{map-of}\ ?pre) \cap dom\ (\textit{map-of}\ FDTs') = \{\}$
using *has-fields-mono-lem*[*OF sub*, *of FDTs'*] *has-fields-fun*[*OF hf*] **by** *fastforce*
have $C \notin (\lambda t.\ snd\ (fst\ t))$ ‘ $set\ FDTs'$
using *has-fields-declaring-classes*[*OF hf*] *cls False*
has-fields-fun[*OF fdts'*(*1*)] *fdts'*(*2*)
by *clarify auto*
then have $\textit{map-of}\ FDTs'\ (F,C) = None$ **by**(*rule map-of-set-pcs-notin*)
then show *?thesis* **using** *fdts' int* **by** *simp*
qed(*auto dest*: *has-fields-Object has-fields-fun*)

lemma *has-fields-is-class-Object*:
$P \vdash D\ \textit{has-fields}\ FDTs \Longrightarrow \textit{is-class}\ P\ Object$
by(*induct rule*: *Fields.induct*; *simp add*: *is-class-def*)

lemma *Object-fields*:
$[\![\ P \vdash Object\ \textit{has-fields}\ FDTs;\ C \neq Object\ ]\!] \Longrightarrow \textit{map-of}\ FDTs\ (F,C) = None$
by(*drule Fields.cases*, *auto simp*: *map-of-reinsert-neq-None*)

definition *has-field* :: $'m\ prog \Rightarrow cname \Rightarrow vname \Rightarrow staticb \Rightarrow ty \Rightarrow cname \Rightarrow bool$
(‹- ⊢ - *has* -,-:- *in* -› [*51*,*51*,*51*,*51*,*51*,*51*] *50*)
where
$P \vdash C\ has\ F,b{:}T\ in\ D\ \equiv$
$\exists\, FDTs.\ P \vdash C\ \textit{has-fields}\ FDTs \wedge \textit{map-of}\ FDTs\ (F,D) = Some\ (b,T)$

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
lemma *has-field-fun*:
$[\![P \vdash C \textit{ has } F,b{:}T \textit{ in } D;\ P \vdash C \textit{ has } F,b'{:}T' \textit{ in } D]\!] \Longrightarrow b = b' \wedge T' = T$

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
lemma *visible-fields-exist*:
assumes *fields*: $P \vdash C$ *has-fields FDTs* **and**
FDTs: *map-of FDTs* $(F,D) = \textit{Some}\ (b,\ T)$
shows $\exists D'\ \textit{fs ms. class}\ P\ D = \textit{Some}(D',\textit{fs},\textit{ms}) \wedge \textit{map-of fs}\ F = \textit{Some}(b,T)$
proof −
have *map-of FDTs* $(F,D) = \textit{Some}\ (b,\ T) \longrightarrow$
$(\exists D'\ \textit{fs ms. class}\ P\ D = \textit{Some}(D',\textit{fs},\textit{ms}) \wedge \textit{map-of fs}\ F = \textit{Some}(b,T))$
using *fields* **proof** *induct*
case (*has-fields-rec* $C'\ D'$ *fs ms FDTs'*)
with *assms map-of-reinsert-SomeD map-of-reinsert-neq-None*[**where** *D*=*D* **and** *F*=*F* **and** *fs*=*fs*]
show *?case* **proof**(*cases* $C' = D$) **qed** *auto*
next
case (*has-fields-Object* D' *fs ms FDTs*)
with *assms map-of-reinsert-SomeD map-of-reinsert-neq-None*[**where** *D*=*D* **and** *F*=*F* **and** *fs*=*fs*]
show *?case* **proof**(*cases Object* = *D*) **qed** *auto*
qed
then show *?thesis* **using** *FDTs* **by** *simp*
qed

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

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

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
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 $map\text{-}of\ (map\ (\lambda((F,\ D),\ b,\ T).\ (F,\ D,\ b,\ T))\ FDTs)\ F = \lfloor(C,\ b,\ T)\rfloor$
proof −
  obtain $D\ fs\ ms$ **where** fs: $class\ P\ C = Some(D,fs,ms)$
    using $visible\text{-}fields\text{-}exist[OF\ assms]$ **by** *clarsimp*
  then obtain $FDTs'$ **where**
    $FDTs = map\ (\lambda(F,\ b,\ T).\ ((F,\ C),\ b,\ T))\ fs\ @\ FDTs' \wedge map\text{-}of\ FDTs'\ (F,\ C) = None$
    using $has\text{-}fields\text{-}mono\text{-}lem2[OF\ hf\ fs\ map]$ **by** *clarsimp*
  then show *?thesis* **using** $map\text{-}of\text{-}Some\text{-}None\text{-}split[OF\ \text{-}\ \text{-}\ map]$ **by** *auto*
qed

lemma *has-field-sees*: $P \vdash C\ has\ F,b{:}T\ in\ C \Longrightarrow P \vdash C\ sees\ F,b{:}T\ in\ C$
by(*auto simp*:*has-field-def sees-field-def has-field-sees-aux*)

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

1.4.5 Functional lookup

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

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

definition *fields* :: $'m\ prog \Rightarrow cname \Rightarrow ((vname \times cname) \times staticb \times ty)\ list$
where
  $fields\ P\ C \equiv\ THE\ FDTs.\ P \vdash C\ has\text{-}fields\ FDTs$

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

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

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

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

lemma *ifields-def2*[*simp*]: $[\![\ P \vdash C\ has\text{-}fields\ FDTs\ ]\!] \Longrightarrow ifields\ P\ C = filter\ (\lambda((F,D),b,T).\ b = NonStatic)\ FDTs$
by (*simp add*: *ifields-def*)

lemma *isfields-def2*[*simp*]: $[\![\ P \vdash C\ has\text{-}fields\ FDTs\ ]\!] \Longrightarrow isfields\ P\ C = filter\ (\lambda((F,D),b,T).\ b = Static \wedge D = C)\ FDTs$
by (*simp add*: *isfields-def*)

lemma *ifields-def3*: $\llbracket$ *P* ⊢ *C sees F*,*b*:*T in D*; *b* = *NonStatic* $\rrbracket$ $\Longrightarrow$ (((*F*,*D*),*b*,*T*) ∈ *set* (*ifields P C*))
lemma *isfields-def3*: $\llbracket$ *P* ⊢ *C sees F*,*b*:*T in D*; *b* = *Static*; *D* = *C* $\rrbracket$ $\Longrightarrow$ (((*F*,*D*),*b*,*T*) ∈ *set* (*isfields P C*))

definition *seeing-class* :: *'m prog* ⇒ *cname* ⇒ *mname* ⇒ *cname option* **where**
seeing-class P C M =
(*if* ∃ *Ts T m D*. *P* ⊢ *C sees M*,*Static*:*Ts*→*T* = *m in D*
then Some (*fst*(*method P C M*))
else None)

lemma *seeing-class-def2*[*simp*]:
P ⊢ *C sees M*,*Static*:*Ts*→*T* = *m in D* $\Longrightarrow$ *seeing-class P C M* = *Some D*
by(*fastforce simp*: *seeing-class-def*)

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*

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
$h \trianglelefteq h' \equiv \forall a\ C\ fs.\ h\ a = Some(C,fs) \longrightarrow (\exists fs'.\ h'\ a = Some(C,fs'))$

primrec $typeof\text{-}h :: heap \Rightarrow val \Rightarrow ty\ option$ $(\langle typeof_{\_}\rangle)$
where
$typeof_h\ Unit = Some\ Void$
$|\ typeof_h\ Null = Some\ NT$
$|\ typeof_h\ (Bool\ b) = Some\ Boolean$
$|\ typeof_h\ (Intg\ i) = Some\ Integer$
$|\ typeof_h\ (Addr\ a) = (case\ h\ a\ of\ None \Rightarrow None\ |\ Some(C,fs) \Rightarrow Some(Class\ C))$

lemma *new-Addr-SomeD*:
$new\text{-}Addr\ h = Some\ a \Longrightarrow h\ a = None$

lemma [*simp*]: $(typeof_h\ v = Some\ Boolean) = (\exists b.\ v = Bool\ b)$

lemma [*simp*]: $(typeof_h\ v = Some\ Integer) = (\exists i.\ v = Intg\ i)$
lemma [*simp*]: $(typeof_h\ v = Some\ NT) = (v = Null)$

lemma [*simp*]: $(typeof_h\ v = Some(Class\ C)) = (\exists a\ fs.\ v = Addr\ a \land h\ a = Some(C,fs))$

lemma [*simp*]: $h\ a = Some(C,fs) \Longrightarrow typeof_{(h(a \mapsto (C,fs')))}\ v = typeof_h\ v$

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

abbreviation
$typeof :: val \Rightarrow ty\ option$ **where**
$typeof\ v == typeof\text{-}h\ Map.empty\ v$

lemma *typeof-lit-typeof*:
$typeof\ v = Some\ T \Longrightarrow typeof_h\ v = Some\ T$

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

1.6.3 Heap extension $\trianglelefteq$

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

1.6.4 Static field information function

datatype $init\text{-}state = Done\ |\ Processing\ |\ Prepared\ |\ Error$
— *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*)
by(*simp only*: *iprog.simps*, *simp*)

lemma *iprog-Error*[*simp*]: (*Error* $\leq_i$ *i*) = (*i* = *Error*)
by(*simp only*: *iprog.simps*, *simp*)

lemma *iprog-Processing*[*simp*]: (*Processing* $\leq_i$ *i*) = (*i* = *Done* ∨ *i* = *Error* ∨ *i* = *Processing*)
by(*simp only*: *iprog.simps*, *simp*)

lemma *iprog-trans*: ⟦ $i \leq_i i'$; $i' \leq_i i''$ ⟧ ⟹ $i \leq_i i''$

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* ⇀ *sfields* × *init-state*

translations
(*type*) *sheap* <= (*type*) *char list* ⇒ (*sfields* × *init-state*) *option*

definition *shext* :: *sheap* ⇒ *sheap* ⇒ *bool* (‹- $\trianglelefteq_s$ -› [*51*,*51*] *50*)
where
  $sh \trianglelefteq_s sh'$ ≡ ∀ *C sfs i*. *sh C* = *Some*(*sfs*,*i*) ⟶ (∃ *sfs' i'*. *sh' C* = *Some*(*sfs'*,*i'*) ∧ $i \leq_i i'$)

lemma *shextI*: ∀ *C sfs i*. *sh C* = *Some*(*sfs*,*i*) ⟶ (∃ *sfs' i'*. *sh' C* = *Some*(*sfs'*,*i'*) ∧ $i \leq_i i'$) ⟹ *sh* $\trianglelefteq_s$ *sh'*
lemma *shext-objD*: ⟦ $sh \trianglelefteq_s sh'$; *sh C* = *Some*(*sfs*,*i*) ⟧ ⟹ ∃ *sfs' i'*. *sh' C* = *Some*(*sfs'*, *i'*) ∧ $i \leq_i i'$
lemma *shext-refl* [*iff*]: $sh \trianglelefteq_s sh$
lemma *shext-new* [*simp*]: *sh C* = *None* ⟹ $sh \trianglelefteq_s sh(C \mapsto x)$
lemma *shext-trans*: ⟦ $sh \trianglelefteq_s sh'$; $sh' \trianglelefteq_s sh''$ ⟧ ⟹ $sh \trianglelefteq_s sh''$
lemma *shext-upd-obj*: ⟦ *sh C* = *Some* (*sfs*,*i*); $i \leq_i i'$ ⟧ ⟹ $sh \trianglelefteq_s sh(C \mapsto (sfs',i'))$
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 ≡ ''*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 $\equiv$ ''*IncompatibleClassChangeError*''

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

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

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

definition *addr-of-sys-xcpt* :: *cname* $\Rightarrow$ *addr*
where
addr-of-sys-xcpt s $\equiv$ *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* $\notin$ *sys-xcpts*
by(*simp add*: *Start-def sys-xcpts-def sys-xcpts-defs*)

lemma *Start-nsys-xcpts1* [*simp*]: *Start* $\neq$ *NullPointer Start* $\neq$ *ClassCast*
Start $\neq$ *OutOfMemory Start* $\neq$ *NoClassDefFoundError*
Start $\neq$ *IncompatibleClassChangeError Start* $\neq$ *NoSuchFieldError*
Start $\neq$ *NoSuchMethodError*
using *Start-nsys-xcpts* **by**(*auto simp*: *sys-xcpts-def*)

lemma *Start-nsys-xcpts2* [*simp*]: *NullPointer* $\neq$ *Start ClassCast* $\neq$ *Start*
OutOfMemory $\neq$ *Start NoClassDefFoundError* $\neq$ *Start*
IncompatibleClassChangeError $\neq$ *Start NoSuchFieldError* $\neq$ *Start*
NoSuchMethodError $\neq$ *Start*
using *Start-nsys-xcpts* **by**(*auto simp*: *sys-xcpts-def dest*: *sym*)

definition *start-heap* :: *'c prog* $\Rightarrow$ *heap*
where
start-heap G $\equiv$ *Map.empty* (*addr-of-sys-xcpt NullPointer* $\mapsto$ *blank G NullPointer*,
addr-of-sys-xcpt ClassCast $\mapsto$ *blank G ClassCast*,
addr-of-sys-xcpt OutOfMemory $\mapsto$ *blank G OutOfMemory*,
addr-of-sys-xcpt NoClassDefFoundError $\mapsto$ *blank G NoClassDefFoundError*,
addr-of-sys-xcpt IncompatibleClassChangeError $\mapsto$ *blank G IncompatibleClass-ChangeError*,
addr-of-sys-xcpt NoSuchFieldError $\mapsto$ *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*
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*

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*)
by(*rule sys-xcpts-cases*[*OF assms*])
(*auto simp add*: *start-heap-def sys-xcpts-def addr-of-sys-xcpt-def sys-xcpts-defs*)

lemma *start-heap-classes*:
start-heap P a = *Some*(*C*,*fs*) $\Longrightarrow$ *C* $\in$ *sys-xcpts*
by(*simp add*: *start-heap-def blank-def split*: *if-split-asm*)

lemma *start-heap-nStart*: *start-heap P a* = *Some obj* $\Longrightarrow$ *fst*(*obj*) $\neq$ *Start*
by(*cases obj*, *auto dest*!: *start-heap-classes simp*: *Start-nsys-xcpts*)

1.7.3 *preallocated*

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

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

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

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

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

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

lemma *typeof-NullPointer* [*simp*]:
preallocated h $\Longrightarrow$ *typeof*$_h$ (*Addr*(*addr-of-sys-xcpt NullPointer*)) = *Some*(*Class NullPointer*)
lemma *typeof-NoClassDefFoundError* [*simp*]:
preallocated h $\Longrightarrow$ *typeof*$_h$ (*Addr*(*addr-of-sys-xcpt NoClassDefFoundError*)) = *Some*(*Class NoClassDefFoundError*)
lemma *typeof-IncompatibleClassChangeError* [*simp*]:
preallocated h $\Longrightarrow$ *typeof*$_h$ (*Addr*(*addr-of-sys-xcpt IncompatibleClassChangeError*)) = *Some*(*Class IncompatibleClassChangeError*)
lemma *typeof-NoSuchFieldError* [*simp*]:
preallocated h $\Longrightarrow$ *typeof*$_h$ (*Addr*(*addr-of-sys-xcpt NoSuchFieldError*)) = *Some*(*Class NoSuchFieldError*)
lemma *typeof-NoSuchMethodError* [*simp*]:
preallocated h $\Longrightarrow$ *typeof*$_h$ (*Addr*(*addr-of-sys-xcpt NoSuchMethodError*)) = *Some*(*Class NoSuchMethodError*)
lemma *preallocated-hext*:
$\llbracket$ *preallocated h*; $h \trianglelefteq h'$ $\rrbracket$ $\Longrightarrow$ *preallocated h'*
lemma *preallocated-start*:
preallocated (*start-heap P*)
by(*auto simp*: *start-heap-sys-xcpts blank-def preallocated-def*)

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

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

lemma *init-rhs-neq* [*simp*]: *e ≠ INIT C (Cs,b) ← e*
proof −
have *size e ≠ size (INIT C (Cs,b) ← e)* **by** *auto*
then show *?thesis* **by** *fastforce*
qed

lemma *init-rhs-neq'* [*simp*]: *INIT C (Cs,b) ← e ≠ e*
proof −
have *size e ≠ size (INIT C (Cs,b) ← e)* **by** *auto*
then show *?thesis* **by** *fastforce*
qed

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

proof –
  have *size e* ≠ *size* (*RI*(*C*,*e'*);*Cs* ← *e*) **by** *auto*
  then show *?thesis* **by** *fastforce*
qed

lemma *ri-rhs-neq'* [*simp*]: *RI*(*C*,*e'*);*Cs* ← *e* ≠ *e*
proof –
  have *size e* ≠ *size* (*RI*(*C*,*e'*);*Cs* ← *e*) **by** *auto*
  then show *?thesis* **by** *fastforce*
qed

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*·$_s$*F*{*D*}) = {}
| *fv*(e_1·*F*{*D*}:=e_2) = *fv* e_1 ∪ *fv* e_2
| *fv*(*C*·$_s$*F*{*D*}:=e_2) = *fv* e_2
| *fv*(*e*·*M*(*es*)) = *fv e* ∪ *fvs es*
| *fv*(*C*·$_s$*M*(*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'*

| $fvs([]) = \{\}$
| $fvs(e\#es) = fv\ e \cup fvs\ es$

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

1.8.3 Accessing expression constructor arguments

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

lemma *val-of-spec*: *val-of* $e = Some\ v \Longrightarrow e = Val\ v$
proof(*cases e*) **qed**(*auto*)

fun *lass-val-of* :: $'a\ exp \Rightarrow ('a \times val)\ option$ **where**
lass-val-of $(V{:=}Val\ v) = Some\ (V,\ v)$ |
lass-val-of - $= None$

lemma *lass-val-of-spec*:
assumes *lass-val-of* $e = \lfloor a \rfloor$
shows $e = (fst\ a{:=}Val\ (snd\ a))$
using *assms* **proof**(*cases e*)
  case $(LAss\ V\ e')$ **then show** *?thesis* **using** *assms* **proof**(*cases* e')**qed**(*auto*)
qed(*auto*)

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

lemma *map-vals-of-spec*: *map-vals-of es* $= Some\ vs \Longrightarrow es = map\ Val\ vs$
proof(*induct es arbitrary*: *vs*) **qed**(*auto simp*: *val-of-spec*)

lemma *map-vals-of-Vals*[*simp*]: *map-vals-of* $(map\ Val\ vs) = \lfloor vs \rfloor$ **by**(*induct vs*, *auto*)

lemma *map-vals-of-throw*[*simp*]:
 map-vals-of $(map\ Val\ vs\ @\ throw\ e\ \#\ es') = None$
 by(*induct vs*, *auto*)

fun *bool-of* :: $'a\ exp \Rightarrow bool\ option$ **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 –
  have *bool-of* $e = Some\ True$ **by** *fact*
  then show *?thesis*
  proof(*cases e*)
    case $(Val\ x3)$ **with** *assms* **show** *?thesis*

    proof(*cases x3*)
      case (*Bool x*) **with** *assms Val* **show** *?thesis*
      proof(*cases x*)**qed**(*auto*)
    qed(*simp-all*)
  qed(*auto*)
qed

lemma *bool-of-specF*:
assumes *bool-of e = Some False* **shows** *e = false*
proof −
  have *bool-of e = Some False* **by** *fact*
  then show *?thesis*
  proof(*cases e*)
    case (*Val x3*) **with** *assms* **show** *?thesis*
    proof(*cases x3*)
      case (*Bool x*) **with** *assms Val* **show** *?thesis*
      proof(*cases x*)**qed**(*auto*)
    qed(*simp-all*)
  qed(*auto*)
qed

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′* $\Longrightarrow$ *e = throw e′*
proof(*cases e*) **qed**(*auto*)

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* = $\lfloor e' \rfloor \Longrightarrow e' \neq e$ **by**(*cases e*, *auto*)
lemma *init-exp-of-neq′*[*simp*]: *init-exp-of e* = $\lfloor e' \rfloor \Longrightarrow e \neq e'$ **by**(*cases e*, *auto*)

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*($C•_sF${*D*}) = *False*
| *sub-RI*(e_1•*F*{*D*}:=e_2) = (*sub-RI* e_1 ∨ *sub-RI* e_2)
| *sub-RI*($C•_sF${*D*}:=e_2) = *sub-RI* e_2
| *sub-RI*(*e*•*M*(*es*)) = (*sub-RI e* ∨ *sub-RIs es*)
| *sub-RI*($C•_sM$(*es*)) = (*M* = *clinit* ∨ *sub-RIs es*)

| $sub\text{-}RI(\{V{:}T;\ e\}) = sub\text{-}RI\ e$
| $sub\text{-}RI(e_1;;e_2) = (sub\text{-}RI\ e_1 \lor sub\text{-}RI\ e_2)$
| $sub\text{-}RI(if\ (b)\ e_1\ else\ e_2) = (sub\text{-}RI\ b \lor sub\text{-}RI\ e_1 \lor sub\text{-}RI\ e_2)$
| $sub\text{-}RI(while\ (b)\ e) = (sub\text{-}RI\ b \lor sub\text{-}RI\ e)$
| $sub\text{-}RI(throw\ e) = sub\text{-}RI\ e$
| $sub\text{-}RI(try\ e_1\ catch(C\ V)\ e_2) = (sub\text{-}RI\ e_1 \lor sub\text{-}RI\ e_2)$
| $sub\text{-}RI(INIT\ C\ (Cs,b) \leftarrow e) = True$
| $sub\text{-}RI(RI\ (C,e);Cs \leftarrow e') = True$
| $sub\text{-}RIs([]) = False$
| $sub\text{-}RIs(e\#es) = (sub\text{-}RI\ e \lor sub\text{-}RIs\ es)$

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

lemma *nsub-RIs-def*[*simp*]:
$\neg sub\text{-}RIs\ es \Longrightarrow \forall\, e \in set\ es.\ \neg sub\text{-}RI\ e$
by(*induct es*, *auto*)

lemma *sub-RI-base*:
$e = INIT\ C\ (Cs,\ b) \leftarrow e' \lor e = RI(C,e_0);Cs \leftarrow e' \Longrightarrow sub\text{-}RI\ e$
by(*cases e*, *auto*)

lemma *nsub-RI-Vals*[*simp*]: $\neg sub\text{-}RIs\ (map\ Val\ vs)$
by(*induct vs*, *auto*)

lemma *lass-val-of-nsub-RI*: $lass\text{-}val\text{-}of\ e = \lfloor a \rfloor \Longrightarrow \neg sub\text{-}RI\ e$
by(*drule lass-val-of-spec*, *simp*)

— is not currently initializing class C' (point past checking flag)
primrec $not\text{-}init :: cname \Rightarrow 'a\ exp \Rightarrow bool$ **and** $not\text{-}inits :: cname \Rightarrow 'a\ exp\ list \Rightarrow bool$ **where**
$not\text{-}init\ C'\ (new\ C) = True$
| $not\text{-}init\ C'\ (Cast\ C\ e) = not\text{-}init\ C'\ e$
| $not\text{-}init\ C'\ (Val\ v) = True$
| $not\text{-}init\ C'\ (e_1\ «bop»\ e_2) = (not\text{-}init\ C'\ e_1 \land not\text{-}init\ C'\ e_2)$
| $not\text{-}init\ C'\ (Var\ V) = True$
| $not\text{-}init\ C'\ (LAss\ V\ e) = not\text{-}init\ C'\ e$
| $not\text{-}init\ C'\ (e{\bullet}F\{D\}) = not\text{-}init\ C'\ e$
| $not\text{-}init\ C'\ (C{\bullet}_sF\{D\}) = True$
| $not\text{-}init\ C'\ (e_1{\bullet}F\{D\}{:=}e_2) = (not\text{-}init\ C'\ e_1 \land not\text{-}init\ C'\ e_2)$
| $not\text{-}init\ C'\ (C{\bullet}_sF\{D\}{:=}e_2) = not\text{-}init\ C'\ e_2$
| $not\text{-}init\ C'\ (e{\bullet}M(es)) = (not\text{-}init\ C'\ e \land not\text{-}inits\ C'\ es)$
| $not\text{-}init\ C'\ (C{\bullet}_sM(es)) = not\text{-}inits\ C'\ es$
| $not\text{-}init\ C'\ (\{V{:}T;\ e\}) = not\text{-}init\ C'\ e$
| $not\text{-}init\ C'\ (e_1;;e_2) = (not\text{-}init\ C'\ e_1 \land not\text{-}init\ C'\ e_2)$
| $not\text{-}init\ C'\ (if\ (b)\ e_1\ else\ e_2) = (not\text{-}init\ C'\ b \land not\text{-}init\ C'\ e_1 \land not\text{-}init\ C'\ e_2)$
| $not\text{-}init\ C'\ (while\ (b)\ e) = (not\text{-}init\ C'\ b \land not\text{-}init\ C'\ e)$
| $not\text{-}init\ C'\ (throw\ e) = not\text{-}init\ C'\ e$
| $not\text{-}init\ C'\ (try\ e_1\ catch(C\ V)\ e_2) = (not\text{-}init\ C'\ e_1 \land not\text{-}init\ C'\ e_2)$
| $not\text{-}init\ C'\ (INIT\ C\ (Cs,b) \leftarrow e) = ((b \longrightarrow Cs = Nil \lor C' \neq hd\ Cs) \land C' \notin set(tl\ Cs) \land not\text{-}init\ C'\ e)$
| $not\text{-}init\ C'\ (RI\ (C,e);Cs \leftarrow e') = (C' \notin set\ (C\#Cs) \land not\text{-}init\ C'\ e \land not\text{-}init\ C'\ e')$
| $not\text{-}inits\ C'\ ([]) = True$
| $not\text{-}inits\ C'\ (e\#es) = (not\text{-}init\ C'\ e \land not\text{-}inits\ C'\ es)$

lemma *not-inits-def'*[*simp*]:
not-inits C es $\Longrightarrow \forall e \in$ *set es. not-init C e*
by(*induct es*, *auto*)

lemma *nsub-RIs-not-inits-aux*: $\forall e \in$ *set es.* $\neg$*sub-RI e* $\longrightarrow$ *not-init C e*
$\Longrightarrow \neg$*sub-RIs es* $\Longrightarrow$ *not-inits C es*
by(*induct es*, *auto*)

lemma *nsub-RI-not-init*: $\neg$*sub-RI e* $\Longrightarrow$ *not-init C e*
proof(*induct e*) **qed**(*auto intro*: *nsub-RIs-not-inits-aux*)

lemma *nsub-RIs-not-inits*: $\neg$*sub-RIs es* $\Longrightarrow$ *not-inits C es*
by(*rule nsub-RIs-not-inits-aux*) (*simp-all add*: *nsub-RI-not-init*)

1.8.5 Subexpressions

primrec *subexp* :: *'a exp* $\Rightarrow$ *'a exp set* **and** *subexps* :: *'a exp list* $\Rightarrow$ *'a exp set* **where**
subexp(*new C*) = {}
| *subexp*(*Cast C e*) = {*e*} $\cup$ *subexp e*
| *subexp*(*Val v*) = {}
| *subexp*(e_1 «*bop*» e_2) = {e_1, e_2} $\cup$ *subexp* e_1 $\cup$ *subexp* e_2
| *subexp*(*Var V*) = {}
| *subexp*(*LAss V e*) = {*e*} $\cup$ *subexp e*
| *subexp*(*e*•*F*{*D*}) = {*e*} $\cup$ *subexp e*
| *subexp*($C{\bullet}_s F\{D\}$) = {}
| *subexp*($e_1{\bullet}F\{D\}{:=}e_2$) = {$e_1$, e_2} $\cup$ *subexp* e_1 $\cup$ *subexp* e_2
| *subexp*($C{\bullet}_s F\{D\}{:=}e_2$) = {$e_2$} $\cup$*subexp* e_2
| *subexp*(*e*•*M*(*es*)) = {*e*} $\cup$ *set es* $\cup$ *subexp e* $\cup$ *subexps es*
| *subexp*($C{\bullet}_s M(es)$) = *set es* $\cup$ *subexps es*
| *subexp*({*V*:*T*; *e*}) = {*e*} $\cup$ *subexp e*
| *subexp*(e_1;;e_2) = {e_1, e_2} $\cup$ *subexp* e_1 $\cup$ *subexp* e_2
| *subexp*(*if* (*b*) e_1 *else* e_2) = {*b*, e_1, e_2} $\cup$ *subexp b* $\cup$ *subexp* e_1 $\cup$ *subexp* e_2
| *subexp*(*while* (*b*) *e*) = {*b*, *e*} $\cup$ *subexp b* $\cup$ *subexp e*
| *subexp*(*throw e*) = {*e*} $\cup$ *subexp e*
| *subexp*(*try* e_1 *catch*(*C V*) e_2) = {e_1, e_2} $\cup$ *subexp* e_1 $\cup$ *subexp* e_2
| *subexp*(*INIT C* (*Cs*,*b*) $\leftarrow$ *e*) = {*e*} $\cup$ *subexp e*
| *subexp*(*RI* (*C*,*e*);*Cs* $\leftarrow$ *e'*) = {*e*, *e'*} $\cup$ *subexp e* $\cup$ *subexp e'*
| *subexps*([]) = {}
| *subexps*(*e#es*) = {*e*} $\cup$ *subexp e* $\cup$ *subexps es*

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

abbreviation *subexp-of* :: *'a exp* $\Rightarrow$ *'a exp* $\Rightarrow$ *bool* **where**
subexp-of e e' $\equiv$ *e* $\in$ *subexp e'*

lemma *subexp-size-le*:
(*e'* $\in$ *subexp e* $\longrightarrow$ *size e'* < *size e*) $\wedge$ (*e'* $\in$ *subexps es* $\longrightarrow$ *size e'* < *size-list size es*)
proof(*induct rule*: *subexp-subexps.induct*)
case *Call*:*11* **then show** *?case* **using** *not-less-eq size-list-estimation* **by** *fastforce*
next
case *SCall*:*12* **then show** *?case* **using** *not-less-eq size-list-estimation* **by** *fastforce*
qed(*auto*)

```
lemma subexps-def2: subexps es = set es ∪ (⋃ e ∈ set es. subexp e) by(induct es, auto)

 — strong induction
lemma shows subexp-induct[consumes 1]:
(⋀e. subexp e = {} ⟹ R e) ⟹ (⋀e. (⋀e′. e′ ∈ subexp e ⟹ R e′) ⟹ R e)
  ⟹ (⋀es. (⋀e′. e′ ∈ subexps es ⟹ R e′) ⟹ Rs es) ⟹ (∀ e′. e′ ∈ subexp e ⟶ R e′) ∧ R e
and subexps-induct[consumes 1]:
 (⋀es. subexps es = {} ⟹ Rs es) ⟹ (⋀e. (⋀e′. e′ ∈ subexp e ⟹ R e′) ⟹ R e)
  ⟹ (⋀es. (⋀e′. e′ ∈ subexps es ⟹ R e′) ⟹ Rs es) ⟹ (∀ e′. e′ ∈ subexps es ⟶ R e′) ∧ Rs es
proof(induct rule: subexp-subexps-induct)
  case (Cast x1 x2)
  then have (∀ e′. subexp-of e′ x2 ⟶ R e′) ∧ R x2 by fast
  then have (∀ e′. subexp-of e′ (Cast x1 x2) ⟶ R e′) by auto
  then show ?case using Cast.prems(2) by fast
next
  case (BinOp x1 x2 x3)
  then have (∀ e′. subexp-of e′ x1 ⟶ R e′) ∧ R x1 and (∀ e′. subexp-of e′ x3 ⟶ R e′) ∧ R x3
   by fast+
  then have (∀ e′. subexp-of e′ (x1 «x2» x3) ⟶ R e′) by auto
  then show ?case using BinOp.prems(2) by fast
next
  case (LAss x1 x2)
  then have (∀ e′. subexp-of e′ x2 ⟶ R e′) ∧ R x2 by fast
  then have (∀ e′. subexp-of e′ (LAss x1 x2) ⟶ R e′) by auto
  then show ?case using LAss.prems(2) by fast
next
  case (FAcc x1 x2 x3)
  then have (∀ e′. subexp-of e′ x1 ⟶ R e′) ∧ R x1 by fast
  then have (∀ e′. subexp-of e′ (x1·x2{x3}) ⟶ R e′) by auto
  then show ?case using FAcc.prems(2) by fast
next
  case (FAss x1 x2 x3 x4)
  then have (∀ e′. subexp-of e′ x1 ⟶ R e′) ∧ R x1 and (∀ e′. subexp-of e′ x4 ⟶ R e′) ∧ R x4
   by fast+
  then have (∀ e′. subexp-of e′ (x1·x2{x3} := x4) ⟶ R e′) by auto
  then show ?case using FAss.prems(2) by fast
next
  case (SFAss x1 x2 x3 x4)
  then have (∀ e′. subexp-of e′ x4 ⟶ R e′) ∧ R x4 by fast
  then have (∀ e′. subexp-of e′ (x1·_s x2{x3} := x4) ⟶ R e′) by auto
  then show ?case using SFAss.prems(2) by fast
next
  case (Call x1 x2 x3)
  then have (∀ e′. subexp-of e′ x1 ⟶ R e′) ∧ R x1 and (∀ e′. e′ ∈ subexps x3 ⟶ R e′) ∧ Rs x3
   by fast+
  then have (∀ e′. subexp-of e′ (x1·x2(x3)) ⟶ R e′) using subexps-def2 by auto
  then show ?case using Call.prems(2) by fast
next
  case (SCall x1 x2 x3)
  then have (∀ e′. e′ ∈ subexps x3 ⟶ R e′) ∧ Rs x3 by fast
  then have (∀ e′. subexp-of e′ (x1·_s x2(x3)) ⟶ R e′) using subexps-def2 by auto
  then show ?case using SCall.prems(2) by fast
next
```

case ($Block\ x1\ x2\ x3$)
then have ($\forall e'.\ subexp\text{-}of\ e'\ x3 \longrightarrow R\ e'$) $\wedge$ $R\ x3$ **by** *fast*
then have ($\forall e'.\ subexp\text{-}of\ e'\ \{x1{:}x2;\ x3\} \longrightarrow R\ e'$) **by** *auto*
then show *?case* **using** *Block.prems*(*2*) **by** *fast*
next
case ($Seq\ x1\ x2$)
then have ($\forall e'.\ subexp\text{-}of\ e'\ x1 \longrightarrow R\ e'$) $\wedge$ $R\ x1$ **and** ($\forall e'.\ subexp\text{-}of\ e'\ x2 \longrightarrow R\ e'$) $\wedge$ $R\ x2$
by *fast+*
then have ($\forall e'.\ subexp\text{-}of\ e'\ (x1;;\ x2) \longrightarrow R\ e'$) **by** *auto*
then show *?case* **using** *Seq.prems*(*2*) **by** *fast*
next
case ($Cond\ x1\ x2\ x3$)
then have ($\forall e'.\ subexp\text{-}of\ e'\ x1 \longrightarrow R\ e'$) $\wedge$ $R\ x1$ **and** ($\forall e'.\ subexp\text{-}of\ e'\ x2 \longrightarrow R\ e'$) $\wedge$ $R\ x2$
and ($\forall e'.\ subexp\text{-}of\ e'\ x3 \longrightarrow R\ e'$) $\wedge$ $R\ x3$ **by** *fast+*
then have ($\forall e'.\ subexp\text{-}of\ e'\ (if\ (x1)\ x2\ else\ x3) \longrightarrow R\ e'$) **by** *auto*
then show *?case* **using** *Cond.prems*(*2*) **by** *fast*
next
case ($While\ x1\ x2$)
then have ($\forall e'.\ subexp\text{-}of\ e'\ x1 \longrightarrow R\ e'$) $\wedge$ $R\ x1$ **and** ($\forall e'.\ subexp\text{-}of\ e'\ x2 \longrightarrow R\ e'$) $\wedge$ $R\ x2$
by *fast+*
then have ($\forall e'.\ subexp\text{-}of\ e'\ (while\ (x1)\ x2) \longrightarrow R\ e'$) **by** *auto*
then show *?case* **using** *While.prems*(*2*) **by** *fast*
next
case ($throw\ x$)
then have ($\forall e'.\ subexp\text{-}of\ e'\ x \longrightarrow R\ e'$) $\wedge$ $R\ x$ **by** *fast*
then have ($\forall e'.\ subexp\text{-}of\ e'\ (throw\ x) \longrightarrow R\ e'$) **by** *auto*
then show *?case* **using** *throw.prems*(*2*) **by** *fast*
next
case ($TryCatch\ x1\ x2\ x3\ x4$)
then have ($\forall e'.\ subexp\text{-}of\ e'\ x1 \longrightarrow R\ e'$) $\wedge$ $R\ x1$ **and** ($\forall e'.\ subexp\text{-}of\ e'\ x4 \longrightarrow R\ e'$) $\wedge$ $R\ x4$
by *fast+*
then have ($\forall e'.\ subexp\text{-}of\ e'\ (try\ x1\ catch(x2\ x3)\ x4) \longrightarrow R\ e'$) **by** *auto*
then show *?case* **using** *TryCatch.prems*(*2*) **by** *fast*
next
case ($INIT\ x1\ x2\ x3\ x4$)
then have ($\forall e'.\ subexp\text{-}of\ e'\ x4 \longrightarrow R\ e'$) $\wedge$ $R\ x4$ **by** *fast*
then have ($\forall e'.\ subexp\text{-}of\ e'\ (INIT\ x1\ (x2,x3) \leftarrow x4) \longrightarrow R\ e'$) **by** *auto*
then show *?case* **using** *INIT.prems*(*2*) **by** *fast*
next
case ($RI\ x1\ x2\ x3\ x4$)
then have ($\forall e'.\ subexp\text{-}of\ e'\ x2 \longrightarrow R\ e'$) $\wedge$ $R\ x2$ **and** ($\forall e'.\ subexp\text{-}of\ e'\ x4 \longrightarrow R\ e'$) $\wedge$ $R\ x4$
by *fast+*
then have ($\forall e'.\ subexp\text{-}of\ e'\ (RI\ (x1,x2)\ ;\ x3 \leftarrow x4) \longrightarrow R\ e'$) **by** *auto*
then show *?case* **using** *RI.prems*(*2*) **by** *fast*
next
case ($Cons\text{-}exp\ x1\ x2$)
then have ($\forall e'.\ subexp\text{-}of\ e'\ x1 \longrightarrow R\ e'$) $\wedge$ $R\ x1$ **and** ($\forall e'.\ e' \in subexps\ x2 \longrightarrow R\ e'$) $\wedge$ $Rs\ x2$
by *fast+*
then have ($\forall e'.\ e' \in subexps\ (x1\ \#\ x2) \longrightarrow R\ e'$) **using** *subexps-def2* **by** *auto*
then show *?case* **using** *Cons-exp.prems*(*3*) **by** *fast*
qed(*auto*)

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

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

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

| *WTCast*:
⟦ *P*,*E* ⊢ *e* :: *Class D*; *is-class P C*; *P* ⊢ *C* ≼* *D* ∨ *P* ⊢ *D* ≼* *C* ⟧
⟹ *P*,*E* ⊢ *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*:
$[\![\ P,E \vdash e_1 :: T_1;\ \ P,E \vdash e_2 :: T_2;\ P \vdash T_1 \leq T_2 \lor P \vdash T_2 \leq T_1\ ]\!]$
$\Longrightarrow P,E \vdash e_1\ «Eq»\ e_2 :: Boolean$

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

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

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

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

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

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

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

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

| *WTCond*:
$\llbracket$ $P,E \vdash e :: Boolean$; $P,E \vdash e_1::T_1$; $P,E \vdash e_2::T_2$;
$P \vdash T_1 \leq T_2 \vee P \vdash T_2 \leq T_1$; $P \vdash T_1 \leq T_2 \longrightarrow T = T_2$; $P \vdash T_2 \leq T_1 \longrightarrow T = T_1$ $\rrbracket$
$\Longrightarrow P,E \vdash if\ (e)\ e_1\ else\ e_2 :: T$

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

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

| *WTTry*:
$\llbracket$ $P,E \vdash e_1 :: T$; $P,E(V \mapsto Class\ C) \vdash e_2 :: T$; $is\text{-}class\ P\ C$ $\rrbracket$
$\Longrightarrow P,E \vdash try\ e_1\ catch(C\ V)\ e_2 :: T$

— well-typed expression lists

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

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

lemma *init-nwt* [*simp*]: $\neg P,E \vdash INIT\ C\ (Cs,b) \leftarrow e :: T$
by(*auto elim*: *WT.cases*)
lemma *ri-nwt* [*simp*]: $\neg P,E \vdash RI(C,e);Cs \leftarrow e' :: T$
by(*auto elim*: *WT.cases*)

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

1.10.2 Some interesting lemmas

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

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

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

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*: $P,E,h,sh \vdash es\ [:]\ Ts \Longrightarrow h \trianglelefteq h' \Longrightarrow P,E,h',sh \vdash es\ [:]\ Ts$

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*: $P,E,h,sh \vdash es\ [:]\ Ts \Longrightarrow sh \trianglelefteq_s sh' \Longrightarrow \neg sub\text{-}RIs\ es \Longrightarrow P,E,h,sh' \vdash es\ [:]\ Ts$
lemma *WTrt-hext-shext-mono*: $P,E,h,sh \vdash e : T$
$\Longrightarrow h \trianglelefteq h' \Longrightarrow sh \trianglelefteq_s sh' \Longrightarrow \neg sub\text{-}RI\ e \Longrightarrow P,E,h',sh' \vdash e : T$
by(*auto intro*: *WTrt-hext-mono WTrt-shext-mono*)

lemma *WTrts-hext-shext-mono*: $P,E,h,sh \vdash es\ [:]\ Ts$
$\Longrightarrow h \trianglelefteq h' \Longrightarrow sh \trianglelefteq_s sh' \Longrightarrow \neg sub\text{-}RIs\ es \Longrightarrow P,E,h',sh' \vdash es\ [:]\ Ts$
by(*auto intro*: *WTrts-hext-mono WTrts-shext-mono*)

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

1.11 Program State

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

type-synonym
locals = *vname* ⇀ *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 ≡ λ(*M*,*b*,*Ts*,*T*,*m*).
(∀ *T*∈*set Ts*. *is-type P T*) ∧ *is-type P T* ∧ *wf-md P C* (*M*,*b*,*Ts*,*T*,*m*)

definition *wf-clinit* :: *'m mdecl list* ⇒ *bool* **where**
wf-clinit ms = (∃ *m*. (*clinit*,*Static*,[], *Void*,*m*)∈*set ms*)

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

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

definition $\mathit{wf\text{-}prog} :: {'m}\ \mathit{wf\text{-}mdecl\text{-}test} \Rightarrow {'m}\ \mathit{prog} \Rightarrow \mathit{bool}$
where
$\mathit{wf\text{-}prog}\ \mathit{wf\text{-}md}\ P \equiv \mathit{wf\text{-}syscls}\ P \wedge (\forall c \in set\ P.\ \mathit{wf\text{-}cdecl}\ \mathit{wf\text{-}md}\ P\ c) \wedge \mathit{distinct\text{-}fst}\ P$

1.13.1 Well-formedness lemmas

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

lemma *class-Object* [*simp*]:
$\mathit{wf\text{-}prog}\ \mathit{wf\text{-}md}\ P \Longrightarrow \exists C\ fs\ ms.\ \mathit{class}\ P\ \mathit{Object} = \mathit{Some}\ (C,fs,ms)$

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

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

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

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

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

lemma *acyclic-subcls1*:
$\mathit{wf\text{-}prog}\ \mathit{wf\text{-}md}\ P \Longrightarrow \mathit{acyclic}\ (\mathit{subcls1}\ P)$

lemma *wf-subcls1*:

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

lemma *single-valued-subcls1*:
$wf\text{-}prog\ wf\text{-}md\ G \Longrightarrow single\text{-}valued\ (subcls1\ G)$

lemma *subcls-induct*:
$[\![\ wf\text{-}prog\ wf\text{-}md\ P;\ \bigwedge C.\ \forall D.\ (C,D) \in (subcls1\ P)^{+} \longrightarrow Q\ D \Longrightarrow Q\ C\ ]\!] \Longrightarrow Q\ C$

lemma *subcls1-induct-aux*:
assumes *is-class P C* **and** *wf*: *wf-prog wf-md P* **and** *QObj*: *Q Object*
shows
$[\![\ \bigwedge C\ D\ fs\ ms.$
$[\![\ C \neq Object;\ is\text{-}class\ P\ C;\ class\ P\ C = Some\ (D,fs,ms)\ \wedge$
$wf\text{-}cdecl\ wf\text{-}md\ P\ (C,D,fs,ms)\ \wedge\ P \vdash C \prec^{1} D\ \wedge\ is\text{-}class\ P\ D\ \wedge\ Q\ D]\!] \Longrightarrow Q\ C\ ]\!]$
$\Longrightarrow Q\ C$

lemma *subcls1-induct* [*consumes 2*, *case-names Object Subcls*]:
$[\![\ wf\text{-}prog\ wf\text{-}md\ P;\ is\text{-}class\ P\ C;\ Q\ Object;$
$\bigwedge C\ D.\ [\![C \neq Object;\ P \vdash C \prec^{1} D;\ is\text{-}class\ P\ D;\ Q\ D]\!] \Longrightarrow Q\ C\ ]\!]$
$\Longrightarrow Q\ C$

lemma *subcls-C-Object*:
assumes *class*: *is-class P C* **and** *wf*: *wf-prog wf-md P*
shows $P \vdash C \preceq^{*} Object$

lemma *is-type-pTs*:
assumes *wf-prog wf-md P* **and** $(C,S,fs,ms) \in set\ P$ **and** $(M,b,Ts,T,m) \in set\ ms$
shows $set\ Ts \subseteq types\ P$
lemma *wf-supercls-distinct-app*:
assumes *wf*:*wf-prog wf-md P*
and *nObj*: $C \neq Object$ **and** *cls*: $class\ P\ C = \lfloor(D,\ fs,\ ms)\rfloor$
and *super*: *supercls-lst P (C#Cs)* **and** *dist*: *distinct (C#Cs)*
shows *distinct (D#C#Cs)*
proof −
have $\neg\ P \vdash D \preceq^{*} C$ **using** *subcls1-wfD[OF subcls1I[OF cls nObj] wf]*
by (*simp add*: *rtrancl-eq-or-trancl*)
then show *?thesis* **using** *assms* **by** *auto*
qed

1.13.2 Well-formedness and method lookup

lemma *sees-wf-mdecl*:
assumes *wf*: *wf-prog wf-md P* **and** *sees*: $P \vdash C\ sees\ M,b{:}Ts{\to}T = m\ in\ D$
shows *wf-mdecl wf-md P D (M,b,Ts,T,m)*
lemma *sees-method-mono* [*rule-format* (*no-asm*)]:
assumes *sub*: $P \vdash C' \preceq^{*} C$ **and** *wf*: *wf-prog wf-md P*
shows $\forall D\ b\ Ts\ T\ m.\ P \vdash C\ sees\ M,b{:}Ts{\to}T = m\ in\ D \longrightarrow$
$(\exists D'\ Ts'\ T'\ m'.\ P \vdash C'\ sees\ M,b{:}Ts'{\to}T' = m'\ in\ D'\ \wedge\ P \vdash Ts\ [\le]\ Ts'\ \wedge\ P \vdash T' \le T)$

lemma *sees-method-mono2*:
$[\![\ P \vdash C' \preceq^{*} C;\ wf\text{-}prog\ wf\text{-}md\ P;$
$P \vdash C\ sees\ M,b{:}Ts{\to}T = m\ in\ D;\ P \vdash C'\ sees\ M,b'{:}Ts'{\to}T' = m'\ in\ D'\ ]\!]$
$\Longrightarrow b = b'\ \wedge\ P \vdash Ts\ [\le]\ Ts'\ \wedge\ P \vdash T' \le T$
lemma *mdecls-visible*:

assumes *wf*: *wf-prog wf-md P* **and** *class*: *is-class P C*
shows $\bigwedge D\ fs\ ms.\ class\ P\ C = Some(D,fs,ms)$
$\quad \Longrightarrow \exists\, Mm.\ P \vdash C\ sees\text{-}methods\ Mm \wedge (\forall\, (M,b,Ts,T,m) \in set\ ms.\ Mm\ M = Some((b,Ts,T,m),C))$
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 $P \vdash C\ sees\ M,b{:}Ts{\rightarrow}T = m\ in\ C$

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 $\exists\, D'\ Ts'\ T'\ m'.$
$\quad P \vdash C'\ sees\ M,b{:}Ts'{\rightarrow}T' = m'\ in\ D' \wedge P \vdash Ts\ [\le]\ Ts' \wedge P \vdash T' \le T \wedge P \vdash C' \preceq^* D'$
$\quad \wedge\ is\text{-}type\ P\ T' \wedge (\forall\, T{\in}set\ Ts'.\ is\text{-}type\ P\ T) \wedge wf\text{-}md\ P\ D'\ (M,b,Ts',T',m')$

lemma *wf-prog-lift*:
  assumes *wf*: *wf-prog* $(\lambda P\ C\ bd.\ A\ P\ C\ bd)\ P$
  and *rule*:
  $\bigwedge wf\text{-}md\ C\ M\ b\ Ts\ C\ T\ m\ bd.$
   $wf\text{-}prog\ wf\text{-}md\ P \Longrightarrow$
   $P \vdash C\ sees\ M,b{:}Ts{\rightarrow}T = m\ in\ C \Longrightarrow$
   $set\ Ts \subseteq\ types\ P \Longrightarrow$
   $bd = (M,b,Ts,T,m) \Longrightarrow$
   $A\ P\ C\ bd \Longrightarrow$
   $B\ P\ C\ bd$
  shows *wf-prog* $(\lambda P\ C\ bd.\ B\ P\ C\ bd)\ P$
lemma *wf-sees-clinit*:
assumes *wf*:*wf-prog wf-md P* **and** *ex*: $class\ P\ C = Some\ a$
shows $\exists\, m.\ P \vdash C\ sees\ clinit,Static{:}[] \rightarrow Void = m\ in\ C$
proof −
  from *ex* **obtain** *D fs ms* **where** $a = (D,fs,ms)$ **by**(*cases a*)
  then have *sP*: $(C,\ D,\ fs,\ ms) \in set\ P$ **using** *ex map-of-SomeD*[*of P C a*] **by**(*simp add*: *class-def*)
  then have *wf-clinit ms* **using** *assms* **by**(*unfold wf-prog-def wf-cdecl-def*, *auto*)
  then obtain *m* **where** *sm*: $(clinit,\ Static,\ [],\ Void,\ m) \in set\ ms$ **by** (*meson wf-clinit-def*)
  then have $P \vdash C\ sees\ clinit,Static{:}[] \rightarrow Void = m\ in\ C$
    using *mdecl-visible*[*OF wf sP sm*] **by** *simp*
  then show *?thesis* **by**(*rule exI*)
qed

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 $b = Static \wedge Ts = [] \wedge T = Void \wedge D = C$
proof −
  obtain m' **where** *sees*: $P \vdash C\ sees\ clinit,Static{:}[] \rightarrow Void = m'\ in\ C$
    using *wf-sees-clinit*[*OF wf ex*] **by** *clarify*
  then show *?thesis* **using** *sees wf* **by** (*meson assms*(*3*) *sees-method-fun*)
qed

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 $M \neq clinit$
proof −
  from *sees-method-is-class*[*OF meth*] **obtain** *a* **where** *cls*: $class\ P\ C = Some\ a$
    by(*clarsimp simp*: *is-class-def*)

with *wf wf-sees-clinit*[*OF wf cls*]
 obtain *m* **where** $P \vdash C$ *sees clinit*,*Static*:[]→*Void*=*m in C* **by** *clarsimp*
with *meth* **show** *?thesis* **by**(*auto dest*: *sees-method-fun*)
qed

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-fields FDTs*

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

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

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(*induct rule*: *rtrancl.induct*)
case (*rtrancl-into-rtrancl a b c*)
with *ns* **show** *?case* **by**(*clarsimp dest*!: *subcls1D wf-cdecl-supD*[*OF class-wf*[*OF - wf*]])
qed(*simp*)

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 −
fix *cd* D' **assume** *cd*: $cd \in$ *set P*
then have *cls*: *is-class P* (*fst cd*) **using** *class-exists-equiv is-class-def* **by** *blast*
with *wf-subcls-nCls*[*OF wf ns*] *ns* **show** $\neg P \vdash$ *fst cd* $\preceq^* C_0$ **by**(*cases fst cd* = D', *auto*)
qed

lemma *wf-nclass-nsub*:
$[\![$ *wf-prog wf-md P*; *is-class P C*; $\neg$*is-class P* C' $]\!] \Longrightarrow \neg P \vdash C \preceq^* C'$
by(*rule notI*, *auto dest*: *wf-subcls-nCls*[**where** C=C' **and** D=C])

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 −
have *Cns*: $C \neq$ *Start* **using** *Start-nsys-xcpts sx* **by** *clarsimp*
show *?thesis* **using** *wf-subcls-nCls*[*OF wf ns - Cns*] **by** *auto*
qed

end

1.14 Weak well-formedness of Jinja programs

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

definition *wwf-J-mdecl* :: *J-prog* ⇒ *cname* ⇒ *J-mb mdecl* ⇒ *bool*
where
wwf-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* ∧ *fv body* ⊆ {*this*} ∪ *set pns*
| *Static* ⇒ *fv body* ⊆ *set pns*)

lemma *wwf-J-mdecl-NonStatic*[*simp*]:
wwf-J-mdecl P C (*M*,*NonStatic*,*Ts*,*T*,*pns*,*body*) =
(*length Ts* = *length pns* ∧ *distinct pns* ∧ ¬*sub-RI body* ∧ *this* ∉ *set pns* ∧ *fv body* ⊆ {*this*} ∪ *set pns*)
lemma *wwf-J-mdecl-Static*[*simp*]:
wwf-J-mdecl P C (*M*,*Static*,*Ts*,*T*,*pns*,*body*) =
(*length Ts* = *length pns* ∧ *distinct pns* ∧ ¬*sub-RI body* ∧ *fv body* ⊆ *set pns*)
abbreviation
wwf-J-prog :: *J-prog* ⇒ *bool* **where**
wwf-J-prog ≡ *wf-prog wwf-J-mdecl*

lemma *sees-wwf-nsub-RI*:
⟦ *wwf-J-prog P*; *P* ⊢ *C sees M*,*b* : *Ts*→*T* = (*pns*, *body*) *in D* ⟧ ⟹ ¬*sub-RI body*
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*:

$[\![\ \nexists sfs.\ sh\ C = Some\ (sfs,\ Done);\ P \vdash \langle INIT\ C\ ([C],False) \leftarrow unit,(h,l,sh)\rangle \Rightarrow \langle Val\ v',(h',l',sh')\rangle;$
$\quad new\text{-}Addr\ h' = None;\ is\text{-}class\ P\ C\ ]\!]$
$\Longrightarrow P \vdash \langle new\ C,(h,l,sh)\rangle \Rightarrow \langle THROW\ OutOfMemory,(h',l',sh')\rangle$

| *NewInitThrow*:
$[\![\ \nexists sfs.\ sh\ C = Some\ (sfs,\ Done);\ P \vdash \langle INIT\ C\ ([C],False) \leftarrow unit,(h,l,sh)\rangle \Rightarrow \langle throw\ a,s'\rangle;$
$\quad is\text{-}class\ P\ C\ ]\!]$
$\Longrightarrow P \vdash \langle new\ C,(h,l,sh)\rangle \Rightarrow \langle throw\ a,s'\rangle$

| *Cast*:
$[\![\ P \vdash \langle e,s_0\rangle \Rightarrow \langle addr\ a,(h,l,sh)\rangle;\ h\ a = Some(D,fs);\ P \vdash D \preceq^*\ C\ ]\!]$
$\Longrightarrow P \vdash \langle Cast\ C\ e,s_0\rangle \Rightarrow \langle addr\ a,(h,l,sh)\rangle$

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

| *CastFail*:
$[\![\ P \vdash \langle e,s_0\rangle\Rightarrow \langle addr\ a,(h,l,sh)\rangle;\ h\ a = Some(D,fs);\ \neg\ P \vdash D \preceq^*\ C\ ]\!]$
$\Longrightarrow P \vdash \langle Cast\ C\ e,s_0\rangle \Rightarrow \langle THROW\ ClassCast,(h,l,sh)\rangle$

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

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

| *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;
$fs(F,D) = Some\ v$ ⟧
$\Longrightarrow P \vdash \langle e{\bullet}F\{D\},s_0\rangle \Rightarrow \langle Val\ v,(h,l,sh)\rangle$

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

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

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

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

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

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

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

| *SFAccNone*:
$[\![\ \neg(\exists\, b\ t.\ P \vdash C\ has\ F,b{:}t\ in\ D)\ ]\!]$
$\Longrightarrow P \vdash \langle C{\bullet}_sF\{D\},s\rangle \Rightarrow \langle THROW\ NoSuchFieldError,s\rangle$

| *SFAccNonStatic*:
$[\![\ P \vdash C\ has\ F,NonStatic{:}t\ in\ D\ ]\!]$
$\Longrightarrow P \vdash \langle C{\bullet}_sF\{D\},s\rangle \Rightarrow \langle THROW\ IncompatibleClassChangeError,s\rangle$

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

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

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

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

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

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

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

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

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

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

| *SFAssNonStatic*:
$\llbracket\ P \vdash \langle e_2,s_0\rangle \Rightarrow \langle Val\ v,(h_2,l_2,sh_2)\rangle;$
$P \vdash C\ has\ F,NonStatic{:}t\ in\ D\ \rrbracket$

$\Longrightarrow P \vdash \langle C\bullet_s F\{D\}{:=}e_2,s_0\rangle \Rightarrow \langle THROW\ IncompatibleClassChangeError,(h_2,l_2,sh_2)\rangle$

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

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

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

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

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

| *Call*:
$[\![P \vdash \langle e,s_0\rangle \Rightarrow \langle addr\ a,s_1\rangle;\ \ P \vdash \langle ps,s_1\rangle\ [\Rightarrow]\ \langle map\ Val\ vs,(h_2,l_2,sh_2)\rangle;$
$h_2\ a = Some(C,fs);\ P \vdash C\ sees\ M,NonStatic{:}Ts{\rightarrow}T = (pns,body)\ in\ D;$
$length\ vs = length\ pns;\ \ l_2{}' = [this{\mapsto}Addr\ a,\ pns[{\mapsto}]vs];$
$P \vdash \langle body,(h_2,l_2{}',sh_2)\rangle \Rightarrow \langle e',(h_3,l_3,sh_3)\rangle]\!]$
$\Longrightarrow P \vdash \langle e\bullet M(ps),s_0\rangle \Rightarrow \langle e',(h_3,l_2,sh_3)\rangle$

| *SCallParamsThrow*:
$[\![P \vdash \langle es,s_0\rangle\ [\Rightarrow]\ \langle map\ Val\ vs\ @\ throw\ ex\ \#\ es',s_2\rangle]\!]$
$\Longrightarrow P \vdash \langle C\bullet_s M(es),s_0\rangle \Rightarrow \langle throw\ ex,s_2\rangle$

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

| *SCallNonStatic*:
$[\![P \vdash \langle ps,s_0\rangle\ [\Rightarrow]\ \langle map\ Val\ vs,s_2\rangle;$
$P \vdash C\ sees\ M,NonStatic{:}Ts{\rightarrow}T = m\ in\ D]\!]$
$\Longrightarrow P \vdash \langle C\bullet_s M(ps),s_0\rangle \Rightarrow \langle THROW\ IncompatibleClassChangeError,s_2\rangle$

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

| *SCallInit*:
$[\![P \vdash \langle ps,s_0\rangle\ [\Rightarrow]\ \langle map\ Val\ vs,(h_1,l_1,sh_1)\rangle;$

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

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

| *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*:
$\llbracket\ P \vdash \langle e_0{,}s_0\rangle \Rightarrow \langle Val\ v{,}s_1\rangle;\ P \vdash \langle e_1{,}s_1\rangle \Rightarrow \langle e_2{,}s_2\rangle\ \rrbracket$
$\Longrightarrow P \vdash \langle e_0{;;}e_1{,}s_0\rangle \Rightarrow \langle e_2{,}s_2\rangle$

| *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*:
$\llbracket\ P \vdash \langle e{,}s_0\rangle \Rightarrow \langle true{,}s_1\rangle;\ P \vdash \langle e_1{,}s_1\rangle \Rightarrow \langle e'{,}s_2\rangle\ \rrbracket$
$\Longrightarrow P \vdash \langle if\ (e)\ e_1\ else\ e_2{,}s_0\rangle \Rightarrow \langle e'{,}s_2\rangle$

| *CondF*:
$\llbracket\ P \vdash \langle e{,}s_0\rangle \Rightarrow \langle false{,}s_1\rangle;\ P \vdash \langle e_2{,}s_1\rangle \Rightarrow \langle e'{,}s_2\rangle\ \rrbracket$
$\Longrightarrow P \vdash \langle if\ (e)\ e_1\ else\ e_2{,}s_0\rangle \Rightarrow \langle e'{,}s_2\rangle$

| *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*:
$\llbracket\ P \vdash \langle e{,}s_0\rangle \Rightarrow \langle true{,}s_1\rangle;\ P \vdash \langle c{,}s_1\rangle \Rightarrow \langle Val\ v_1{,}s_2\rangle;\ P \vdash \langle while\ (e)\ c{,}s_2\rangle \Rightarrow \langle e_3{,}s_3\rangle\ \rrbracket$
$\Longrightarrow P \vdash \langle while\ (e)\ c{,}s_0\rangle \Rightarrow \langle e_3{,}s_3\rangle$

| *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*:
$\llbracket\ P \vdash \langle e{,}s_0\rangle \Rightarrow \langle true{,}s_1\rangle;\ P \vdash \langle c{,}s_1\rangle \Rightarrow \langle throw\ e'{,}s_2\rangle\rrbracket$
$\Longrightarrow P \vdash \langle while\ (e)\ c{,}s_0\rangle \Rightarrow \langle throw\ e'{,}s_2\rangle$

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

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

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

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

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

| $RInit$:
$\llbracket$ $P \vdash \langle e',s\rangle \Rightarrow \langle Val\ v,\ (h',l',sh')\rangle$;
$sh'\ C = Some(sfs,\ i)$; $sh'' = sh'(C \mapsto (sfs,\ Done))$;
$C' = last(C\#Cs)$;
$P \vdash \langle INIT\ C'\ (Cs,True) \leftarrow e,\ (h',l',sh'')\rangle \Rightarrow \langle e_1,s_1\rangle$ $\rrbracket$
$\Longrightarrow P \vdash \langle RI\ (C,e');Cs \leftarrow e,s\rangle \Rightarrow \langle e_1,s_1\rangle$

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

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

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*: $P \vdash \langle es,s\rangle\ [\Rightarrow]\ \langle es',s'\rangle \Longrightarrow finals\ es'$

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

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

lemma *evals-finals-same*:
assumes *finals*: *finals es*
shows $P \vdash \langle es,s\rangle\ [\Rightarrow]\ \langle es',s'\rangle \Longrightarrow es = es' \wedge s = s'$
  using *finals*
proof (*induct es arbitrary*: *es' type*: *list*)
  case *Nil* **then show** *?case* **using** *evals-cases*(*1*) **by** *blast*
next
  case (*Cons e es*)
  have *IH*: $\bigwedge es'.\ P \vdash \langle es,s\rangle\ [\Rightarrow]\ \langle es',s'\rangle \Longrightarrow finals\ es \Longrightarrow es = es' \wedge s = s'$
   and *step*: $P \vdash \langle e \# es,s\rangle\ [\Rightarrow]\ \langle es',s'\rangle$ **and** *finals*: *finals* (*e* # *es*) **by** *fact+*
  then obtain $e'\ es''$ **where** *es'*: $es' = e'\#es''$ **by** (*meson Cons.prems*(*1*) *evals-cases*(*2*))
  have *fe*: *final e* **using** *finals not-finals-ConsI* **by** *auto*
  show *?case*
  proof(*rule evals-cases*(*2*)[*OF step*])
    fix $v\ s_1$ *es1* **assume** *es1*: $es' = Val\ v \#\ es1$
      and *estep*: $P \vdash \langle e,s\rangle \Rightarrow \langle Val\ v,s_1\rangle$ **and** *esstep*: $P \vdash \langle es,s_1\rangle\ [\Rightarrow]\ \langle es1,s'\rangle$
    then have $e = Val\ v$ **using** *eval-final-same fe* **by** *auto*
    then have *finals es* **using** *es' not-finals-ConsI2 finals* **by** *blast*
    then show *?thesis* **using** *es' IH estep esstep es1 eval-final-same fe* **by** *blast*
  next
    fix e' **assume** *es1*: $es' = throw\ e' \#\ es$ **and** *estep*: $P \vdash \langle e,s\rangle \Rightarrow \langle throw\ e',s'\rangle$
    then have $e = throw\ e'$ **using** *eval-final-same fe* **by** *auto*
    then show *?thesis* **using** *es' estep es1 eval-final-same fe* **by** *blast*
  qed
qed

1.15.2 Property preservation

lemma *evals-length*: $P \vdash \langle es,s\rangle\ [\Rightarrow]\ \langle es',s'\rangle \Longrightarrow length\ es = length\ es'$
 by(*induct es arbitrary*:*es' s s'*, *auto elim*: *evals-cases*)

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

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

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*: $P \vdash \langle es,(h_0,l_0,sh_0)\rangle\ [\Rightarrow]\ \langle es',(h_1,l_1,sh_1)\rangle \Longrightarrow dom\ l_0 \subseteq dom\ l_1$
lemma
shows *init-ri-same-loc*: $P \vdash \langle e,(h,l,sh)\rangle \Rightarrow \langle e',(h',l',sh')\rangle$
  $\Longrightarrow (\bigwedge C\ Cs\ b\ M\ a.\ e = INIT\ C\ (Cs,b) \leftarrow unit \vee e = C\bullet_s M([]) \vee e = RI\ (C,Throw\ a)\ ;\ Cs \leftarrow unit$
     $\vee\ e = RI\ (C,C\bullet_s clinit([]))\ ;\ Cs \leftarrow unit$
      $\Longrightarrow l = l')$
  and $P \vdash \langle es,(h,l,sh)\rangle\ [\Rightarrow]\ \langle es',(h',l',sh')\rangle \Longrightarrow True$
proof(*induct rule*: *eval-evals-inducts*)
  case (*RInitInitFail e h l sh a'*)
  then show *?case* **using** *eval-final*[*of - - - throw a'*]
    by(*fastforce dest*: *eval-final-same*[*of - Throw a*])
next
  case *RInitFailFinal* **then show** *?case* **by**(*auto dest*: *eval-final-same*)

qed(*auto dest*: *evals-cases eval-cases*(*17*) *eval-final-same*)

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

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* $\Longrightarrow \exists sfs.\ sh\ C = \lfloor(sfs,\ Processing)\rfloor \Longrightarrow \exists sfs'.\ sh'\ C = \lfloor(sfs',\ Processing)\rfloor$

1.16 Definite assignment

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

1.16.1 Hypersets

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

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

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

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

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

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

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

lemma [*simp*]: $\lfloor\{\}\rfloor \sqcup A = A\ \wedge\ A \sqcup \lfloor\{\}\rfloor = A$
lemma [*simp*]: $\lfloor A\rfloor \sqcup \lfloor B\rfloor = \lfloor A \cup B\rfloor \wedge \lfloor A\rfloor \ominus a = \lfloor A - \{a\}\rfloor$
lemma [*simp*]: $None \sqcup A = None \wedge A \sqcup None = None$
lemma [*simp*]: $a \in\in None \wedge None \ominus a = None$
lemma *hyper-isin-union*: $x \in\in \lfloor A\rfloor \Longrightarrow x \in\in \lfloor A\rfloor \sqcup B$
by(*case-tac B*, *auto simp*: *hyper-isin-def*)

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

1.16.2 Definite assignment

primrec
$\mathcal{A}$ :: $'a\ exp \Rightarrow\ 'a\ hyperset$
and $\mathcal{A}s$ :: $'a\ exp\ list \Rightarrow\ 'a\ hyperset$
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
$\mathcal{D}$ :: $'a\ exp \Rightarrow\ 'a\ hyperset \Rightarrow bool$
and $\mathcal{D}s$ :: $'a\ exp\ list \Rightarrow\ 'a\ hyperset \Rightarrow bool$
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*) = $\lfloor\{\}\rfloor$
lemma *D-append*[*iff*]: $\bigwedge A.\ \mathcal{D}s\ (es\ @\ es')\ A = (\mathcal{D}s\ es\ A \wedge \mathcal{D}s\ es'\ (A \sqcup \mathcal{A}s\ es))$

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

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

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

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

lemma *Ds-Vals*: $\mathcal{D}s$ (*map Val vs*) *A* **by**(*induct vs*, *auto*)

end

1.17 Conformance Relations for Type Soundness Proofs

theory *Conform*
imports *Exceptions*
begin

definition *conf* :: $'m\ prog \Rightarrow heap \Rightarrow val \Rightarrow ty \Rightarrow bool$ (‹-,- ⊢ - :≤ -› [*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* :: $'m\ prog \Rightarrow heap \Rightarrow obj \Rightarrow bool$ (‹-,- ⊢ - √› [*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* :: $'m\ prog \Rightarrow heap \Rightarrow cname \Rightarrow sfields \Rightarrow bool$ (‹-,-,- $\vdash_s$ - √› [*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* :: $'m\ prog \Rightarrow heap \Rightarrow bool$ (‹- ⊢ - √› [*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* $\Rightarrow$ *heap* $\Rightarrow$ *sheap* $\Rightarrow$ *bool* (‹-,- $\vdash_s$ - $\surd$› [*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* $\Rightarrow$ *heap* $\Rightarrow$ (*vname* $\rightharpoonup$ *val*) $\Rightarrow$ (*vname* $\rightharpoonup$ *ty*) $\Rightarrow$ *bool* (‹-,- $\vdash$ - '(:$\leq$') -› [*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* $\Rightarrow$ *heap* $\Rightarrow$ *val list* $\Rightarrow$ *ty list* $\Rightarrow$ *bool*
(‹-,- $\vdash$ - [:$\leq$] -› [*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 :$\leq$

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

1.17.2 Value list conformance [:$\leq$]

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

1.17.3 Object conformance

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

1.17.4 Static object conformance

lemma *soconf-hext*: $P,h,C \vdash_s obj \surd \Longrightarrow h \trianglelefteq h' \Longrightarrow P,h',C \vdash_s obj \surd$

lemma *soconf-sblank*:
$P \vdash C$ *has-fields FDTs* $\Longrightarrow P,h,C \vdash_s$ *sblank P C* $\surd$
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*$(F \mapsto v)$ $\surd$

1.17.5 Heap conformance

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

1.17.6 Class statics conformance

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

1.17.7 Local variable conformance

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

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* $]\!] \Longrightarrow fv(blocks(Vs,Ts,vs,e)) = fv\ e - set\ Vs$

lemma *sub-RI-blocks-body*[*iff*]: *length vs* = *length pns* $\Longrightarrow$ *length Ts* = *length pns*
$\Longrightarrow$ *sub-RI* (*blocks* (*pns*, *Ts*, *vs*, *body*)) $\longleftrightarrow$ *sub-RI body*
proof(*induct pns arbitrary*: *Ts vs*)
case *Nil* **then show** *?case* **by** *simp*
next
case *Cons* **then show** *?case* **by**(*cases vs*; *cases Ts*) *auto*
qed

definition *assigned* :: $'a \Rightarrow 'a\ exp \Rightarrow bool$
where
$assigned\ V\ e\ \equiv\ \exists\, v\ e'.\ e = (V := Val\ v;;\ e')$

— 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* :: $'m\ prog \Rightarrow cname \Rightarrow 'a\ exp \Rightarrow bool$ **where**
$icheck\ P\ C'\ (new\ C) = (C' = C)\ |$
$icheck\ P\ D'\ (C{\cdot}_sF\{D\}) = ((D' = D) \wedge (\exists\, T.\ P \vdash C\ has\ F{,}Static{:}T\ in\ D))\ |$
$icheck\ P\ D'\ (C{\cdot}_sF\{D\}{:=}(Val\ v)) = ((D' = D) \wedge (\exists\, T.\ P \vdash C\ has\ F{,}Static{:}T\ in\ D))\ |$
$icheck\ P\ D\ (C{\cdot}_sM(es)) = ((\exists\, vs.\ es = map\ Val\ vs) \wedge (\exists\, Ts\ T\ m.\ P \vdash C\ sees\ M{,}Static{:}Ts{\rightarrow}T = m\ in\ D))\ |$
$icheck\ P\ \text{-}\ \text{-} = False$

lemma *nicheck-SFAss-nonVal*: $val\text{-}of\ e_2 = None \Longrightarrow \neg icheck\ P\ C'\ (C{\cdot}_sF\{D\} := (e_2{::}'a\ exp))$
by(*rule notI*, *cases* e_2, *auto*)

inductive-set
red :: $J\text{-}prog \Rightarrow ((expr \times state \times bool) \times (expr \times state \times bool))\ set$
and *reds* :: $J\text{-}prog \Rightarrow ((expr\ list \times state \times bool) \times (expr\ list \times state \times bool))\ set$
and *red'* :: $J\text{-}prog \Rightarrow expr \Rightarrow state \Rightarrow bool \Rightarrow expr \Rightarrow state \Rightarrow bool \Rightarrow bool$
(‹- ⊢ ((1⟨-,/-,/-⟩) →/ (1⟨-,/-,/-⟩))› [51,0,0,0,0,0,0] 81)
and *reds'* :: $J\text{-}prog \Rightarrow expr\ list \Rightarrow state \Rightarrow bool \Rightarrow expr\ list \Rightarrow state \Rightarrow bool \Rightarrow bool$
(‹- ⊢ ((1⟨-,/-,/-⟩) [→]/ (1⟨-,/-,/-⟩))› [51,0,0,0,0,0,0] 81)
for P :: *J-prog*
where

$P \vdash \langle e,s,b\rangle \rightarrow \langle e',s',b'\rangle \equiv ((e,s,b),\ e',s',b') \in red\ P$
$|\ P \vdash \langle es,s,b\rangle\ [\rightarrow]\ \langle es',s',b'\rangle \equiv ((es,s,b),\ es',s',b') \in reds\ P$

| *RedNew*:
$[\![\ new\text{-}Addr\ h = Some\ a;\ P \vdash C\ has\text{-}fields\ FDTs;\ h' = h(a{\mapsto}blank\ P\ C)\ ]\!]$
$\Longrightarrow P \vdash \langle new\ C,\ (h,l,sh),\ True\rangle \rightarrow \langle addr\ a,\ (h',l,sh),\ False\rangle$

| *RedNewFail*:
$[\![\ new\text{-}Addr\ h = None;\ is\text{-}class\ P\ C\ ]\!] \Longrightarrow$
$P \vdash \langle new\ C,\ (h,l,sh),\ True\rangle \rightarrow \langle THROW\ OutOfMemory,\ (h,l,sh),\ False\rangle$

| *NewInitDoneRed*:
$sh\ C = Some\ (sfs,\ Done) \Longrightarrow$
$P \vdash \langle new\ C,\ (h,l,sh),\ False\rangle \rightarrow \langle new\ C,\ (h,l,sh),\ True\rangle$

| *NewInitRed*:
$[\![\ \nexists\, sfs.\ sh\ C = Some\ (sfs,\ Done);\ is\text{-}class\ P\ C\ ]\!]$
$\Longrightarrow P \vdash \langle new\ C{,}(h,l,sh){,}False\rangle \rightarrow \langle INIT\ C\ ([C]{,}False) \leftarrow new\ C{,}(h,l,sh){,}False\rangle$

| *CastRed*:
$P \vdash \langle e,s,b\rangle \rightarrow \langle e',s',b'\rangle \Longrightarrow$
$P \vdash \langle Cast\ C\ e,\ s,\ b\rangle \rightarrow \langle Cast\ C\ e',\ s',\ b'\rangle$

| *RedCastNull*:
$P \vdash \langle Cast\ C\ null,\ s,\ b\rangle \rightarrow \langle null,s,b\rangle$

| *RedCast*:
$[\![\ h\ a = Some(D,fs);\ P \vdash D \preceq^* C\ ]\!]$
$\Longrightarrow P \vdash \langle Cast\ C\ (addr\ a),\ (h,l,sh),\ b\rangle \rightarrow \langle addr\ a,\ (h,l,sh),\ b\rangle$

| *RedCastFail*:
$[\![\ h\ a = Some(D,fs);\ \neg\ P \vdash D \preceq^* C\ ]\!]$
$\Longrightarrow P \vdash \langle Cast\ C\ (addr\ a),\ (h,l,sh),\ b\rangle \rightarrow \langle THROW\ ClassCast,\ (h,l,sh),\ b\rangle$

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

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

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

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

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

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

| *FAccRed*:
$P \vdash \langle e,s,b\rangle \rightarrow \langle e',s',b'\rangle \Longrightarrow$
$P \vdash \langle e\text{•}F\{D\},\ s,\ b\rangle \rightarrow \langle e'\text{•}F\{D\},\ s',\ b'\rangle$

| *RedFAcc*:
$[\![\ h\ a = Some(C,fs);\ fs(F,D) = Some\ v;$
$\quad P \vdash C\ has\ F,NonStatic{:}t\ in\ D\ ]\!]$
$\Longrightarrow P \vdash \langle (addr\ a)\text{•}F\{D\},\ (h,l,sh),\ b\rangle \rightarrow \langle Val\ v,(h,l,sh),b\rangle$

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

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

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

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

$sfs\ F = Some\ v\ \rrbracket$
$\Longrightarrow P \vdash \langle C\bullet_s F\{D\},(h,l,sh),True\rangle \to \langle Val\ v,(h,l,sh),False\rangle$

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

| *SCallInitDoneRed*:
  $[\![\ P \vdash C\ sees\ M, Static{:}Ts{\rightarrow}T = (pns,body)\ in\ D;$

$sh\ D = Some(sfs,Done) \lor (M = clinit \land sh\ D = Some(sfs,Processing))$ ⟧
$\Longrightarrow P \vdash \langle C\bullet_s M(map\ Val\ vs),(h,l,sh),\ False\rangle \to \langle C\bullet_s M(map\ Val\ vs),(h,l,sh),\ True\rangle$

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

| *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*:
$P \vdash \langle e,s,b\rangle \rightarrow \langle e',s',b'\rangle \Longrightarrow$
$P \vdash \langle \mathit{throw}\ e,\ s,\ b\rangle \rightarrow \langle \mathit{throw}\ e',\ s',\ b'\rangle$

| *RedThrowNull*:
$P \vdash \langle \mathit{throw}\ \mathit{null},\ s,\ b\rangle \rightarrow \langle \mathit{THROW}\ \mathit{NullPointer},\ s,\ b\rangle$

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

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

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

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

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

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

— Initialization procedure

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

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

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

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

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

| *InitObjectRed*:

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

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

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

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

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

— Exception propagation

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

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* ⊢ ⟨*e*,*s*,*b*⟩ →∗ ⟨*e*′,*s*′,*b*′⟩ ≡ ((*e*,*s*,*b*), *e*′,*s*′,*b*′) ∈ (*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* ⊢ ⟨*es*,*s*,*b*⟩ [→]∗ ⟨*es*′,*s*′,*b*′⟩ ≡ ((*es*,*s*,*b*), *es*′,*s*′,*b*′) ∈ (*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* ⊢ ⟨*e*,(*h*,*l*,*sh*),*b*⟩ →∗ ⟨*e*′,(*h*′,*l*′,*sh*′),*b*′⟩
and ⋀*e h l sh b*. *R e h l sh b e h l sh b*
and ⋀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* ⊢ ⟨e_0,(h_0,l_0,sh_0),b_0⟩ → ⟨e_1,(h_1,l_1,sh_1),b_1⟩; *R* e_1 h_1 l_1 sh_1 b_1 *e*′ *h*′ *l*′ *sh*′ *b*′ ⟧
⟹ *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*′

1.18.2 Some easy lemmas

lemma [*iff*]: ¬ *P* ⊢ ⟨[],*s*,*b*⟩ [→] ⟨*es*′,*s*′,*b*′⟩
lemma [*iff*]: ¬ *P* ⊢ ⟨*Val v*,*s*,*b*⟩ → ⟨*e*′,*s*′,*b*′⟩
lemma *val-no-step*: *val-of e* = ⌊*v*⌋ ⟹ ¬ *P* ⊢ ⟨*e*,*s*,*b*⟩ → ⟨*e*′,*s*′,*b*′⟩
lemma [*iff*]: ¬ *P* ⊢ ⟨*Throw a*,*s*,*b*⟩ → ⟨*e*′,*s*′,*b*′⟩

lemma *map-Vals-no-step* [*iff*]: ¬ *P* ⊢ ⟨*map Val vs*,*s*,*b*⟩ [→] ⟨*es*′,*s*′,*b*′⟩
lemma *vals-no-step*: *map-vals-of es* = ⌊*vs*⌋ ⟹ ¬ *P* ⊢ ⟨*es*,*s*,*b*⟩ [→] ⟨*es*′,*s*′,*b*′⟩
lemma *vals-throw-no-step* [*iff*]: ¬ *P* ⊢ ⟨*map Val vs* @ *Throw a* # *es*,*s*,*b*⟩ [→] ⟨*es*′,*s*′,*b*′⟩
lemma *lass-val-of-red*:
⟦ *lass-val-of e* = ⌊*a*⌋; *P* ⊢ ⟨*e*,(*h*, *l*, *sh*),*b*⟩ → ⟨*e*′,(*h*′, *l*′, *sh*′),*b*′⟩ ⟧
⟹ *e*′ = *unit* ∧ *h*′ = *h* ∧ *l*′ = *l*(*fst a*↦*snd a*) ∧ *sh*′ = *sh* ∧ *b* = *b*′

lemma *final-no-step* [*iff*]: *final e* ⟹ ¬ *P* ⊢ ⟨*e*,*s*,*b*⟩ → ⟨*e*′,*s*′,*b*′⟩
lemma *finals-no-step* [*iff*]: *finals es* ⟹ ¬ *P* ⊢ ⟨*es*,*s*,*b*⟩ [→] ⟨*es*′,*s*′,*b*′⟩
lemma *reds-final-same*:
P ⊢ ⟨*e*,*s*,*b*⟩ →∗ ⟨*e*′,*s*′,*b*′⟩ ⟹ *final e* ⟹ *e* = *e*′ ∧ *s* = *s*′ ∧ *b* = *b*′
proof(*induct rule*:*converse-rtrancl-induct3*)
  case *refl* **show** *?case* **by** *simp*
next
  case (*step e0 s0 b0 e1 s1 b1*) **show** *?case*
  proof(*rule finalE*[*OF step.prems*(*1*)])
    fix *v* **assume** *e0* = *Val v* **then show** *?thesis* **using** *step* **by** *simp*
  next
    fix *a* **assume** *e0* = *Throw a* **then show** *?thesis* **using** *step* **by** *simp*
  qed
qed

lemma *reds-throw*:
P ⊢ ⟨*e*,*s*,*b*⟩ →∗ ⟨*e*′,*s*′,*b*′⟩ ⟹ (⋀e_t. *throw-of e* = ⌊e_t⌋ ⟹ ∃ e_t′. *throw-of e*′ = ⌊e_t′⌋)
proof(*induct rule*:*converse-rtrancl-induct3*)
  case *refl* **then show** *?case* **by** *simp*

next
  case (*step e0 s0 b0 e1 s1 b1*)
  then show *?case* **by**(*auto elim*: *red.cases*)
qed

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

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

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

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*])
 by (*metis append-butlast-last-id last-ConsR list.simps*(*3*))

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 ($C{\bullet}_s F\{D\}$) = *Some D* |
init-class P ($C{\bullet}_s F\{D\}{:=}e_2$) = *Some D* |
init-class P ($C{\bullet}_s M(es)$) = *seeing-class P C M* |
init-class - - = *None*

lemma *icheck-init-class*: *icheck P C e* $\implies$ *init-class P e* = $\lfloor C\rfloor$
proof(*induct e*)
  case (*SFAss x1 x2 x3 e'*)
  then show *?case* **by**(*case-tac e'*) *auto*
qed *auto*

— exp to take next small step (in particular, subexp that may contain initialization)
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* ⇒ (*case val-of* e_2 *of Some v* ⇒ e_1 «*bop*» e_2 | - ⇒ *ss-exp* e_2)
| - ⇒ *ss-exp* e_1)
| *ss-exp* (*Var V*) = *Var V*
| *ss-exp* (*LAss V e*) = (*case val-of e of Some v* ⇒ *LAss V e* | - ⇒ *ss-exp e*)
| *ss-exp* $(e{\bullet}F\{D\})$ = (*case val-of e of Some v* ⇒ $e{\bullet}F\{D\}$ | - ⇒ *ss-exp e*)
| *ss-exp* $(C{\bullet}_sF\{D\})$ = $C{\bullet}_sF\{D\}$
| *ss-exp* $(e_1{\bullet}F\{D\}{:=}e_2)$ = (*case val-of* e_1 *of Some v* ⇒ (*case val-of* e_2 *of Some v* ⇒ $e_1{\bullet}F\{D\}{:=}e_2$ | - ⇒ *ss-exp* e_2)
| - ⇒ *ss-exp* e_1)
| *ss-exp* $(C{\bullet}_sF\{D\}{:=}e_2)$ = (*case val-of* e_2 *of Some v* ⇒ $C{\bullet}_sF\{D\}{:=}e_2$ | - ⇒ *ss-exp* e_2)
| *ss-exp* $(e{\bullet}M(es))$ = (*case val-of e of Some v* ⇒ (*case map-vals-of es of Some t* ⇒ $e{\bullet}M(es)$ | - ⇒ *the*(*ss-exps es*))
| - ⇒ *ss-exp e*)
| *ss-exp* $(C{\bullet}_sM(es))$ = (*case map-vals-of es of Some t* ⇒ $C{\bullet}_sM(es)$ | - ⇒ *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* ⇒ *ss-exp* e_2
| *None* ⇒ (*case lass-val-of* e_1 *of Some p* ⇒ *ss-exp* e_2
| *None* ⇒ *ss-exp* e_1))
| *ss-exp* (*if* (*b*) e_1 *else* e_2) = (*case bool-of b of Some True* ⇒ *if* (*b*) e_1 *else* e_2
| *Some False* ⇒ *if* (*b*) e_1 *else* e_2
| - ⇒ *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*
using *assms*
proof(*cases e*)
 case (*SFAss C F D e*) **then show** *?thesis* **using** *assms*
 proof(*cases e*)**qed**(*auto*)
qed(*auto*)

lemma *ss-exps-Vals-None*[*simp*]:
ss-exps (*map Val vs*) = *None*
by(*induct vs*) (*auto*)

lemma *ss-exps-Vals-NoneI*:
ss-exps es = *None* ⟹ ∃ *vs*. *es* = *map Val vs*
using *val-of-spec* **by**(*induct es*) (*auto*)

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

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

$\Longrightarrow e' = throw\ e$
by(*induct vs*) (*auto*)

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'*: $\bigwedge e'.\ ss\text{-}exp\ e = e' \Longrightarrow icheck\ P\ C\ e' \Longrightarrow curr\text{-}init\ P\ e = \lfloor C \rfloor$
and *icheck-curr-inits'*: $\bigwedge e.\ ss\text{-}exps\ es = \lfloor e \rfloor \Longrightarrow icheck\ P\ C\ e \Longrightarrow curr\text{-}inits\ P\ es = \lfloor C \rfloor$
proof(*induct rule*: *ss-exp-ss-exps-induct*)
qed(*simp-all add*: *icheck-init-class*)

lemma *icheck-curr-init*: $icheck\ P\ C\ e' \Longrightarrow ss\text{-}exp\ e = e' \Longrightarrow curr\text{-}init\ P\ e = \lfloor C \rfloor$
by(*rule icheck-curr-init'*)

lemma *icheck-curr-inits*: $icheck\ P\ C\ e \Longrightarrow ss\text{-}exps\ es = \lfloor e \rfloor \Longrightarrow curr\text{-}inits\ P\ es = \lfloor C \rfloor$
by(*rule icheck-curr-inits'*)

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* ($e{\cdot}F\{D\}$) = *iconf sh e*
| *iconf sh* ($C{\cdot}_sF\{D\}$) = *True*
| *iconf sh* ($e_1{\cdot}F\{D\}{:=}e_2$) = (*case val-of* e_1 *of Some v* ⇒ *iconf sh* e_2 | - ⇒ *iconf sh* e_1 ∧ ¬*sub-RI* e_2)
| *iconf sh* ($C{\cdot}_sF\{D\}{:=}e_2$) = *iconf sh* e_2
| *iconf sh* ($e{\cdot}M(es)$) = (*case val-of e of Some v* ⇒ *iconfs sh es* | - ⇒ *iconf sh e* ∧ ¬*sub-RIs es*)
| *iconf sh* ($C{\cdot}_sM(es)$) = *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*
by(*induct vs*,*auto*)

lemma *nsub-RI-iconf-aux*:
$(\neg sub\text{-}RI\ (e{::}'a\ exp) \longrightarrow (\forall e'.\ e' \in subexp\ e \longrightarrow \neg sub\text{-}RI\ e' \longrightarrow iconf\ sh\ e') \longrightarrow iconf\ sh\ e)$
$\wedge\ (\neg sub\text{-}RIs\ (es{::}'a\ exp\ list) \longrightarrow (\forall e'.\ e' \in subexps\ es \longrightarrow \neg sub\text{-}RI\ e' \longrightarrow iconf\ sh\ e') \longrightarrow iconfs\ sh\ es)$
proof(*induct rule*: *sub-RI-sub-RIs.induct*) **qed**(*auto*)

lemma *nsub-RI-iconf-aux'*:
$(\bigwedge e'.\ subexp\text{-}of\ e'\ e \Longrightarrow \neg sub\text{-}RI\ e' \longrightarrow iconf\ sh\ e') \Longrightarrow (\neg sub\text{-}RI\ e \Longrightarrow iconf\ sh\ e)$
by(*simp add*: *nsub-RI-iconf-aux*)

lemma *nsub-RI-iconf*: $\neg sub\text{-}RI\ e \Longrightarrow iconf\ sh\ e$
and *nsub-RIs-iconfs*: $\neg sub\text{-}RIs\ es \Longrightarrow iconfs\ sh\ es$
proof −
let $?R = \lambda e.\ \neg sub\text{-}RI\ e \longrightarrow iconf\ sh\ e$
let $?Rs = \lambda es.\ \neg sub\text{-}RIs\ es \longrightarrow iconfs\ sh\ es$
have $(\forall e'.\ subexp\text{-}of\ e'\ e \longrightarrow ?R\ e') \wedge ?R\ e$
by(*rule subexp-induct*[**where** *?Rs* = *?Rs*]; *clarsimp simp*: *nsub-RI-iconf-aux*)
moreover have $(\forall e'.\ e' \in subexps\ es \longrightarrow ?R\ e') \wedge ?Rs\ es$
by(*rule subexps-induct*; *clarsimp simp*: *nsub-RI-iconf-aux*)
ultimately show $\neg sub\text{-}RI\ e \Longrightarrow iconf\ sh\ e$
and $\neg sub\text{-}RIs\ es \Longrightarrow iconfs\ sh\ es$ **by** *simp+*
qed

lemma *lass-val-of-iconf*: $lass\text{-}val\text{-}of\ e = \lfloor a \rfloor \Longrightarrow iconf\ sh\ e$
by(*drule lass-val-of-nsub-RI*, *erule nsub-RI-iconf*)

lemma *icheck-iconf*:
assumes *icheck P C e* **shows** *iconf sh e*
using *assms*
proof(*cases e*)
case (*SFAss C F D e*) **then show** *?thesis* **using** *assms*
proof(*cases e*)**qed**(*auto*)
next
case (*SCall C M es*) **then show** *?thesis* **using** *assms*
by (*auto simp*: *nsub-RIs-iconfs*)
next
qed(*auto*)

1.19.2 Indicator boolean conformance

definition *bconf* :: $'m\ prog \Rightarrow sheap \Rightarrow 'a\ exp \Rightarrow bool \Rightarrow bool$ (‹-,- $\vdash_b$ '(-,-') $\surd$› [51,51,0,0] 50)
where
$P,sh \vdash_b (e,b) \surd\ \equiv b \longrightarrow (\exists C.\ icheck\ P\ C\ (ss\text{-}exp\ e) \wedge initPD\ sh\ C)$

definition *bconfs* :: $'m\ prog \Rightarrow sheap \Rightarrow 'a\ exp\ list \Rightarrow bool \Rightarrow bool$ (‹-,- $\vdash_b$ '(-,-') $\surd$› [51,51,0,0] 50)
where
$P,sh \vdash_b (es,b) \surd\ \equiv b \longrightarrow (\exists C.\ (icheck\ P\ C\ (the(ss\text{-}exps\ es))$
$\wedge\ (curr\text{-}inits\ P\ es = Some\ C) \wedge initPD\ sh\ C))$

— bconf helper lemmas

lemma *bconf-nonVal*[*simp*]:
$P,sh \vdash_b (e,True) \surd \Longrightarrow val\text{-}of\ e = None$

by(*cases e*) (*auto simp*: *bconf-def*)

lemma *bconfs-nonVals*[*simp*]:
$P,sh \vdash_b (es,True) \surd \Longrightarrow$ *map-vals-of es* $=$ *None*
by(*induct es*) (*auto simp*: *bconfs-def*)

lemma *bconf-Cast*[*iff*]:
$P,sh \vdash_b (Cast\ C\ e,b) \surd \longleftrightarrow P,sh \vdash_b (e,b) \surd$
by(*cases b*) (*auto simp*: *bconf-def dest*: *val-of-spec*)

lemma *bconf-BinOp*[*iff*]:
$P,sh \vdash_b (e1 \ «bop» \ e2,b) \surd$
$\longleftrightarrow$ (*case val-of e1 of Some v* $\Rightarrow P,sh \vdash_b (e2,b) \surd$ | - $\Rightarrow P,sh \vdash_b (e1,b) \surd$)
by(*cases b*) (*auto simp*: *bconf-def dest*: *val-of-spec*)

lemma *bconf-LAss*[*iff*]:
$P,sh \vdash_b (LAss\ V\ e,b) \surd \longleftrightarrow P,sh \vdash_b (e,b) \surd$
by(*cases b*) (*auto simp*: *bconf-def dest*: *val-of-spec*)

lemma *bconf-FAcc*[*iff*]:
$P,sh \vdash_b (e{\cdot}F\{D\},b) \surd \longleftrightarrow P,sh \vdash_b (e,b) \surd$
by(*cases b*) (*auto simp*: *bconf-def dest*: *val-of-spec*)

lemma *bconf-FAss*[*iff*]:
$P,sh \vdash_b (FAss\ e1\ F\ D\ e2,b) \surd$
$\longleftrightarrow$ (*case val-of e1 of Some v* $\Rightarrow P,sh \vdash_b (e2,b) \surd$ | - $\Rightarrow P,sh \vdash_b (e1,b) \surd$)
by(*cases b*) (*auto simp*: *bconf-def dest*: *val-of-spec*)

lemma *bconf-SFAss*[*iff*]:
val-of e2 $=$ *None* $\Longrightarrow P,sh \vdash_b (SFAss\ C\ F\ D\ e2,b) \surd \longleftrightarrow P,sh \vdash_b (e2,b) \surd$
by(*cases b*) (*auto simp*: *bconf-def*)

lemma *bconfs-Vals*[*iff*]:
$P,sh \vdash_b (map\ Val\ vs,\ b) \surd \longleftrightarrow \neg\ b$
by(*unfold bconfs-def*) *simp*

lemma *bconf-Call*[*iff*]:
$P,sh \vdash_b (e{\cdot}M(es),b) \surd$
$\longleftrightarrow$ (*case val-of e of Some v* $\Rightarrow P,sh \vdash_b (es,b) \surd$ | - $\Rightarrow P,sh \vdash_b (e,b) \surd$)
proof(*cases b*)
  case *True*
  then show *?thesis*
  proof(*cases ss-exps es*)
    case *None*
    then obtain *vs* **where** *es* $=$ *map Val vs* **using** *ss-exps-Vals-NoneI* **by** *auto*
    then have *mv*: *map-vals-of es* $= \lfloor vs \rfloor$ **by** *simp*
    then show *?thesis* **by**(*auto simp*: *bconf-def*) (*simp add*: *bconfs-def*)
  next
    case (*Some a*)
    then show *?thesis* **by**(*auto simp*: *bconf-def*) (*auto simp*: *bconfs-def icheck-init-class*)
  qed
qed(*simp add*: *bconf-def bconfs-def*)

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(*cases b*)
  case *True*
  then show *?thesis*
  proof(*cases ss-exps es*)
    case *None*
      then have $\exists$ *vs. es* = *map Val vs* **using** *ss-exps-Vals-NoneI* **by** *auto*
      then show *?thesis* **using** *mvn finals-def* **by** *clarsimp*
    next
    case (*Some a*)
      then show *?thesis* **by**(*auto simp*: *bconf-def*) (*auto simp*: *bconfs-def icheck-init-class*)
    qed
qed(*simp add*: *bconf-def bconfs-def*)

lemma *bconf-Cons*[*iff*]:
 $P,sh \vdash_b (e\#es,b) \surd$
  $\longleftrightarrow$ (*case val-of e of Some v* $\Rightarrow P,sh \vdash_b (es,b) \surd$ | - $\Rightarrow P,sh \vdash_b (e,b) \surd$)
proof(*cases b*)
  case *True*
  then show *?thesis*
  proof(*cases ss-exps es*)
    case *None*
      then have $\exists$ *vs. es* = *map Val vs* **using** *ss-exps-Vals-NoneI* **by** *auto*
      then show *?thesis* **using** *None* **by**(*auto simp*: *bconf-def bconfs-def icheck-init-class*)
    next
    case (*Some a*)
      then show *?thesis* **by**(*auto simp*: *bconf-def bconfs-def icheck-init-class*)
    qed
qed(*simp add*: *bconf-def bconfs-def*)

lemma *bconf-InitBlock*[*iff*]:
 $P,sh \vdash_b (\{V{:}T;\ V{:=}Val\ v;;\ e_2\},b) \surd \longleftrightarrow P,sh \vdash_b (e_2,b) \surd$
 by(*cases b*) (*auto simp*: *bconf-def assigned-def*)

lemma *bconf-Block*[*iff*]:
 $P,sh \vdash_b (\{V{:}T;\ e\},b) \surd \longleftrightarrow P,sh \vdash_b (e,b) \surd$
 by(*cases b*) (*auto simp*: *bconf-def*)

lemma *bconf-Seq*[*iff*]:
 $P,sh \vdash_b (e1;;e2,b) \surd$
  $\longleftrightarrow$ (*case val-of e1 of Some v* $\Rightarrow P,sh \vdash_b (e2,b) \surd$
        | - $\Rightarrow$ (*case lass-val-of e1 of Some p* $\Rightarrow P,sh \vdash_b (e2,b) \surd$
            | *None* $\Rightarrow P,sh \vdash_b (e1,b) \surd$))
 by(*cases b*) (*auto simp*: *bconf-def dest*: *val-of-spec lass-val-of-spec*)

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(*cases bool-of b*)
  case *None*
  then show *?thesis* **by**(*auto simp*: *bconf-def*)
next
  case (*Some a*)
  then show *?thesis* **by**(*case-tac a*) (*auto simp*: *bconf-def dest*: *bool-of-specT bool-of-specF*)

qed

lemma *bconf-While*[*iff*]:
$P,sh \vdash_b (while\ (b)\ e,b') \surd \longleftrightarrow \neg b'$
by(*cases b*) (*auto simp*: *bconf-def*)

lemma *bconf-Throw*[*iff*]:
$P,sh \vdash_b (throw\ e,b) \surd \longleftrightarrow P,sh \vdash_b (e,b) \surd$
by(*cases b*) (*auto simp*: *bconf-def dest*: *val-of-spec*)

lemma *bconf-Try*[*iff*]:
$P,sh \vdash_b (try\ e_1\ catch(C\ V)\ e_2,b) \surd \longleftrightarrow P,sh \vdash_b (e_1,b) \surd$
by(*cases b*) (*auto simp*: *bconf-def dest*: *val-of-spec*)

lemma *bconf-INIT*[*iff*]:
$P,sh \vdash_b (INIT\ C\ (Cs,b') \leftarrow e,b) \surd \longleftrightarrow \neg b$
by(*cases b*) (*auto simp*: *bconf-def*)

lemma *bconf-RI*[*iff*]:
$P,sh \vdash_b (RI(C,e);Cs \leftarrow e',b) \surd \longleftrightarrow P,sh \vdash_b (e,b) \surd$
by(*cases b*) (*auto simp*: *bconf-def dest*: *val-of-spec*)

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

end

1.20 Progress of Small Step Semantics

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

lemma *final-addrE*:
$[\![\ P,E,h,sh \vdash e : Class\ C;\ final\ e;$
$\bigwedge a.\ e = addr\ a \Longrightarrow R;$
$\bigwedge a.\ e = Throw\ a \Longrightarrow R\ ]\!] \Longrightarrow R$

lemma *finalRefE*:
$[\![\ P,E,h,sh \vdash e : T;\ is\text{-}refT\ T;\ final\ e;$
$e = null \Longrightarrow R;$
$\bigwedge a\ C.\ [\![\ e = addr\ a;\ T = Class\ C\ ]\!] \Longrightarrow R;$
$\bigwedge a.\ e = Throw\ a \Longrightarrow R\ ]\!] \Longrightarrow R$

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

$$
\begin{array}{l}
\;\; P,E,h,sh \vdash e :' T \equiv WTrt'\ P\ h\ sh\ E\ e\ T \\
\mid P,E,h,sh \vdash es\ [:'] \ Ts \equiv WTrts'\ P\ h\ sh\ E\ es\ Ts \\
\\
\mid is\text{-}class\ P\ C \implies P,E,h,sh \vdash new\ C :' Class\ C \\
\mid [\![\ P,E,h,sh \vdash e :' T;\ is\text{-}refT\ T;\ is\text{-}class\ P\ C\ ]\!] \\
\;\; \implies P,E,h,sh \vdash Cast\ C\ e :' Class\ C \\
\mid typeof_h\ v = Some\ T \implies P,E,h,sh \vdash Val\ v :' T \\
\mid E\ v = Some\ T \implies P,E,h,sh \vdash Var\ v :' T \\
\mid [\![\ P,E,h,sh \vdash e_1 :' T_1;\ \ P,E,h,sh \vdash e_2 :' T_2\ ]\!] \\
\;\; \implies P,E,h,sh \vdash e_1 \ll Eq \gg e_2 :' Boolean \\
\mid [\![\ P,E,h,sh \vdash e_1 :' Integer;\ \ P,E,h,sh \vdash e_2 :' Integer\ ]\!] \\
\;\; \implies P,E,h,sh \vdash e_1 \ll Add \gg e_2 :' Integer \\
\mid [\![\ P,E,h,sh \vdash Var\ V :' T;\ \ P,E,h,sh \vdash e :' T';\ \ P \vdash T' \leq T\ ]\!] \\
\;\; \implies P,E,h,sh \vdash V{:=}e :' Void \\
\mid [\![\ P,E,h,sh \vdash e :' Class\ C;\ P \vdash C\ has\ F,NonStatic{:}T\ in\ D\ ]\!] \implies P,E,h,sh \vdash e{\bullet}F\{D\} :' T \\
\mid P,E,h,sh \vdash e :' NT \implies P,E,h,sh \vdash e{\bullet}F\{D\} :' T \\
\mid [\![\ P \vdash C\ has\ F,Static{:}T\ in\ D\ ]\!] \implies P,E,h,sh \vdash C{\bullet_s}F\{D\} :' T \\
\mid [\![\ 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\ ]\!] \\
\;\; \implies P,E,h,sh \vdash e_1{\bullet}F\{D\}{:=}e_2 :' Void \\
\mid [\![\ P,E,h,sh \vdash e_1{:'}NT;\ P,E,h,sh \vdash e_2 :' T_2\ ]\!] \implies P,E,h,sh \vdash e_1{\bullet}F\{D\}{:=}e_2 :' Void \\
\mid [\![\ P \vdash C\ has\ F,Static{:}T\ in\ D; \\
\;\;\;\; P,E,h,sh \vdash e_2 :' T_2;\ \ P \vdash T_2 \leq T\ ]\!] \\
\;\; \implies P,E,h,sh \vdash C{\bullet_s}F\{D\}{:=}e_2 :' Void \\
\mid [\![\ P,E,h,sh \vdash e :' Class\ C;\ P \vdash C\ sees\ M,NonStatic{:}Ts \to T = (pns,body)\ in\ D; \\
\;\;\;\; P,E,h,sh \vdash es\ [:']\ Ts';\ P \vdash Ts'\ [\leq]\ Ts\ ]\!] \\
\;\; \implies P,E,h,sh \vdash e{\bullet}M(es) :' T \\
\mid [\![\ P,E,h,sh \vdash e :' NT;\ P,E,h,sh \vdash es\ [:']\ Ts\ ]\!] \implies P,E,h,sh \vdash e{\bullet}M(es) :' T \\
\mid [\![\ P \vdash C\ sees\ M,Static{:}Ts \to T = (pns,body)\ in\ D; \\
\;\;\;\; P,E,h,sh \vdash es\ [:']\ Ts';\ P \vdash Ts'\ [\leq]\ Ts; \\
\;\;\;\; M = clinit \longrightarrow sh\ D = \lfloor(sfs,Processing)\rfloor \wedge es = map\ Val\ vs\ ]\!] \\
\;\; \implies P,E,h,sh \vdash C{\bullet_s}M(es) :' T \\
\mid P,E,h,sh \vdash []\ [:']\ [] \\
\mid [\![\ P,E,h,sh \vdash e :' T;\ \ P,E,h,sh \vdash es\ [:']\ Ts\ ]\!] \implies P,E,h,sh \vdash e\#es\ [:']\ T\#Ts \\
\mid [\![\ typeof_h\ v = Some\ T_1;\ P \vdash T_1 \leq T;\ P,E(V \mapsto T),h,sh \vdash e_2 :' T_2\ ]\!] \\
\;\; \implies P,E,h,sh \vdash \{V{:}T := Val\ v;\ e_2\} :' T_2 \\
\mid [\![\ P,E(V \mapsto T),h,sh \vdash e :' T';\ \neg\ assigned\ V\ e\ ]\!] \implies P,E,h,sh \vdash \{V{:}T;\ e\} :' T' \\
\mid [\![\ P,E,h,sh \vdash e_1{:'}\ T_1;\ \ P,E,h,sh \vdash e_2{:'}T_2\ ]\!] \implies P,E,h,sh \vdash e_1;;e_2 :' T_2 \\
\mid [\![\ 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\ ]\!] \\
\;\; \implies P,E,h,sh \vdash if\ (e)\ e_1\ else\ e_2 :' T \\
\mid [\![\ P,E,h,sh \vdash e :' Boolean;\ \ P,E,h,sh \vdash c{:'}\ T\ ]\!] \\
\;\; \implies P,E,h,sh \vdash while(e)\ c :' Void \\
\mid [\![\ P,E,h,sh \vdash e :' T_r;\ is\text{-}refT\ T_r\ ]\!] \implies P,E,h,sh \vdash throw\ e :' T \\
\mid [\![\ 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\ ]\!] \\
\;\; \implies P,E,h,sh \vdash try\ e_1\ catch(C\ V)\ e_2 :' T_2 \\
\mid [\![\ 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\ ]\!] \implies P,E,h,sh \vdash INIT\ C\ (Cs,\ b) \leftarrow e :' T \\
\mid [\![\ 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;
\end{array}
$$

$\forall C' \in set\ Cs.\ \exists sfs.\ sh\ C' = \lfloor(sfs,Processing)\rfloor$;
$\exists sfs.\ sh\ C = \lfloor(sfs,\ Processing)\rfloor \vee (sh\ C = \lfloor(sfs,\ Error)\rfloor \wedge e = THROW\ NoClassDefFoundError)$;
$distinct\ (C\#Cs)$; $supercls\text{-}lst\ P\ (C\#Cs)$ $\rrbracket$
$\Longrightarrow P,E,h,sh \vdash RI(C,\ e);Cs \leftarrow e' :' T'$

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

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

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

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

corollary *wts'-iff-wts*: $(P,E,h,sh \vdash es\ [:']\ Ts) = (P,E,h,sh \vdash es\ [:]\ Ts)$
theorem assumes *wf*: *wwf-J-prog P* **and** *hconf*: $P \vdash h \surd$ **and** *shconf*: $P,h \vdash_s sh \surd$
shows *progress*: $P,E,h,sh \vdash e : T \Longrightarrow$
$(\bigwedge l.\ \llbracket \mathcal{D}\ e\ \lfloor dom\ l\rfloor;\ P,sh \vdash_b (e,b) \surd;\ \neg\ final\ e\ \rrbracket \Longrightarrow \exists e'\ s'\ b'.\ P \vdash \langle e,(h,l,sh),b\rangle \rightarrow \langle e',s',b'\rangle)$
and $P,E,h,sh \vdash es\ [:]\ Ts \Longrightarrow$
$(\bigwedge l.\ \llbracket \mathcal{D}s\ es\ \lfloor dom\ l\rfloor;\ P,sh \vdash_b (es,b) \surd;\ \neg\ finals\ es\ \rrbracket \Longrightarrow \exists es'\ s'\ b'.\ P \vdash \langle es,(h,l,sh),b\rangle\ [\rightarrow]\ \langle es',s',b'\rangle)$
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 $\equiv$ $\lambda(M,b,Ts,T,(pns,body))$.
$length\ Ts = length\ pns\ \wedge$
$distinct\ pns\ \wedge$
$\neg sub\text{-}RI\ body\ \wedge$
$(case\ b\ of$
$NonStatic \Rightarrow this \notin set\ pns\ \wedge$
$(\exists T'.\ P,[this{\mapsto}Class\ C,pns[\mapsto]Ts] \vdash body :: T' \wedge P \vdash T' \leq T)\ \wedge$
$\mathcal{D}\ body\ \lfloor\{this\} \cup set\ pns\rfloor$
$|\ Static \Rightarrow (\exists T'.\ P,[pns[\mapsto]Ts] \vdash body :: T' \wedge P \vdash T' \leq T)\ \wedge$
$\mathcal{D}\ body\ \lfloor set\ pns\rfloor)$

lemma *wf-J-mdecl-NonStatic*[*simp*]:
$wf\text{-}J\text{-}mdecl\ P\ C\ (M,NonStatic,Ts,T,pns,body) \equiv$
$(length\ Ts = length\ pns\ \wedge$
$distinct\ pns\ \wedge$
$\neg sub\text{-}RI\ body\ \wedge$
$this \notin set\ pns\ \wedge$
$(\exists T'.\ P,[this{\mapsto}Class\ C,pns[\mapsto]Ts] \vdash body :: T' \wedge P \vdash T' \leq T)\ \wedge$
$\mathcal{D}\ body\ \lfloor\{this\} \cup set\ pns\rfloor)$
lemma *wf-J-mdecl-Static*[*simp*]:

$wf\text{-}J\text{-}mdecl\ P\ C\ (M,Static,Ts,T,pns,body) \equiv$
$(length\ Ts = length\ pns\ \wedge$
$distinct\ pns\ \wedge$
$\neg sub\text{-}RI\ body\ \wedge$
$(\exists\, T'.\ P,[pns[\mapsto]\,Ts] \vdash body :: T' \wedge P \vdash T' \leq T)\ \wedge$
$\mathcal{D}\ body\ \lfloor set\ pns \rfloor)$

abbreviation
$wf\text{-}J\text{-}prog :: J\text{-}prog \Rightarrow bool$ **where**
$wf\text{-}J\text{-}prog == wf\text{-}prog\ wf\text{-}J\text{-}mdecl$

lemma *wf-J-prog-wf-J-mdecl*:
$[\![\ wf\text{-}J\text{-}prog\ P;\ (C,\ D,\ fds,\ mths) \in set\ P;\ jmdcl \in set\ mths\ ]\!]$
$\Longrightarrow wf\text{-}J\text{-}mdecl\ P\ C\ jmdcl$
lemma *wf-mdecl-wwf-mdecl*: $wf\text{-}J\text{-}mdecl\ P\ C\ Md \Longrightarrow wwf\text{-}J\text{-}mdecl\ P\ C\ Md$
lemma *wf-prog-wwf-prog*: $wf\text{-}J\text{-}prog\ P \Longrightarrow wwf\text{-}J\text{-}prog\ P$

end

1.22 Type Safety Proof

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

lemma *red-shext-incr*: $P \vdash \langle e,(h,l,sh),b\rangle \rightarrow \langle e',(h',l',sh'),b'\rangle$
$\Longrightarrow (\bigwedge E\ T.\ P,E,h,sh \vdash e : T \Longrightarrow sh \trianglelefteq_s sh')$
and *reds-shext-incr*: $P \vdash \langle es,(h,l,sh),b\rangle\ [\rightarrow]\ \langle es',(h',l',sh'),b'\rangle$
$\Longrightarrow (\bigwedge E\ Ts.\ P,E,h,sh \vdash es\ [:]\ Ts \Longrightarrow sh \trianglelefteq_s sh')$
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 $P,E,h,sh \vdash C{\bullet_s}clinit([]) : Void$
proof –
from *ex* **obtain** *D fs ms* **where** $a = (D,fs,ms)$ **by**(*cases a*)
then have *sP*: $(C,\ D,\ fs,\ ms) \in set\ P$ **using** *ex map-of-SomeD*[*of P C a*] **by**(*simp add*: *class-def*)
then have *wf-clinit ms* **using** *assms* **by**(*unfold wf-prog-def wf-cdecl-def*, *auto*)
then obtain *pns body* **where** *sm*: $(clinit,\ Static,\ [],\ Void,\ pns,\ body) \in set\ ms$
by(*unfold wf-clinit-def*) *auto*
then have $P \vdash C\ sees\ clinit,Static:[] \rightarrow Void = (pns,body)\ in\ C$
using *mdecl-visible*[*OF wf sP sm*] **by** *simp*
then show *?thesis* **using** *WTrtSCall proc* **by** *simp*
qed

1.22.1 Basic preservation lemmas

First some easy preservation lemmas.

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

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

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

lemma *Seq-bconf-preserve-aux*:
assumes $P \vdash \langle e,(h,\ l,\ sh),b\rangle \rightarrow \langle e',(h',\ l',\ sh'),b'\rangle$ **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 $P,sh' \vdash_b (e';;e_2,b')\ \surd$
proof(*cases val-of e*)
  case *None* **show** *?thesis*
  proof(*cases lass-val-of e*)
    case *lNone*: *None* **show** *?thesis*
    proof(*cases lass-val-of e′*)
      case *lNone′*: *None*
      then show *?thesis* **using** *None assms lNone*
        by(*auto dest*: *val-of-spec simp*: *bconf-def option.distinct*(*1*))
    next
      case (*Some a*)
      then show *?thesis* **using** *None assms lNone* **by**(*auto dest*: *lass-val-of-spec simp*: *bconf-def*)
    qed
  next
    case (*Some a*)
    then show *?thesis* **using** *None assms* **by**(*auto dest*: *lass-val-of-spec*)
  qed
next
  case (*Some a*)
  then show *?thesis* **using** *assms* **by**(*auto dest*: *val-of-spec*)
qed

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

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

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

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

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 $P \vdash \langle es,(h,l,sh),b\rangle\ [\rightarrow]\ \langle es',(h',l',sh'),b'\rangle \Longrightarrow \mathcal{D}s\ es\ \lfloor dom\ l\rfloor \Longrightarrow \mathcal{D}s\ es'\ \lfloor dom\ l'\rfloor$

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

definition *sconf* :: *J-prog* $\Rightarrow$ *env* $\Rightarrow$ *state* $\Rightarrow$ *bool* (‹-,- ⊢ - $\surd$› [*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*:
$[\![\ P \vdash \langle e,s,b\rangle \rightarrow \langle e',s',b'\rangle;\ P,E,hp\ s,shp\ s \vdash e : T;\ P,E \vdash s\ \surd\ ]\!] \Longrightarrow P,E \vdash s'\ \surd$
lemma *reds-preserves-sconf*:
$[\![\ P \vdash \langle es,s,b\rangle\ [\rightarrow]\ \langle es',s',b'\rangle;\ P,E,hp\ s,shp\ s \vdash es\ [:]\ Ts;\ P,E \vdash s\ \surd\ ]\!] \Longrightarrow P,E \vdash s'\ \surd$

1.22.2 Subject reduction

lemma *wt-blocks*:
$\bigwedge E.\ [\![\ length\ Vs = length\ Ts;\ length\ vs = length\ Ts\ ]\!] \Longrightarrow$
$(P,E,h,sh \vdash blocks(Vs,Ts,vs,e) : T) =$
$(P,E(Vs[\mapsto]Ts),h,sh \vdash e{:}T \wedge (\exists\ Ts'.\ map\ (typeof_h)\ vs = map\ Some\ Ts' \wedge P \vdash Ts'\ [\leq]\ Ts))$
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*: $P \vdash \langle es,(h,l,sh),b\rangle\ [\rightarrow]\ \langle es',(h',l',sh'),b'\rangle \Longrightarrow$
$(\bigwedge E\ Ts.\ [\![\ P,E \vdash (h,l,sh)\ \surd;\ iconfs\ sh\ es;\ P,E,h,sh \vdash es\ [:]\ Ts\ ]\!]$
$\Longrightarrow \exists\ Ts'.\ P,E,h',sh' \vdash es'\ [:]\ Ts' \wedge P \vdash Ts'\ [\leq]\ Ts)$

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

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 $\bigwedge T.\ [\![\ P,E,hp\ s,shp\ s \vdash e : T;\ iconf\ (shp\ s)\ e;\ P,E \vdash s\ \surd\ ]\!] \Longrightarrow P,E \vdash s'\ \surd$
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 $iconf\ (shp\ s)\ e \Longrightarrow iconf\ (shp\ s')\ e'$
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 $iconfs\ (shp\ s)\ es \Longrightarrow iconfs\ (shp\ s')\ es'$

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 $iconf\ (shp\ s)\ e \Longrightarrow P,(shp\ s) \vdash_b (e,b) \surd \Longrightarrow P,(shp\ s') \vdash_b (e'::expr,b') \surd$
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 $iconfs\ (shp\ s)\ es \Longrightarrow P,(shp\ s) \vdash_b (es,b) \surd \Longrightarrow P,(shp\ s') \vdash_b (es'::expr\ list,b') \surd$
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 $\mathcal{D}\ e\ \lfloor dom(lcl\ s)\rfloor \Longrightarrow \mathcal{D}\ e'\ \lfloor dom(lcl\ s')\rfloor$
using *reds*
proof (*induct rule*:*converse-rtrancl-induct3*)
 case *refl* **thus** *?case* **.**
next
 case (*step e s b e′ s′ b′*) **thus** *?case*
  by(*cases s*,*cases s′*)(*auto dest*:*red-preserves-defass*[*OF wf*])
qed

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 $!!T.\ [\![\ P,E \vdash s\surd;\ iconf\ (shp\ s)\ e;\ P,E,hp\ s,shp\ s \vdash e:T\ ]\!]$
 $\Longrightarrow \exists T'.\ P \vdash T' \le T \wedge P,E,hp\ s',shp\ s' \vdash e':T'$

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* (‹-,-,- ⊢ - : - √› [*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 $P \vdash \langle e,s,b\rangle \to \langle e',s',b'\rangle \Longrightarrow P,E,s \vdash e : T \surd$
 $\Longrightarrow \exists T'.\ P,E,s' \vdash e' : T' \surd \wedge P \vdash T' \le T$

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 $\bigwedge T.\ P,E,s \vdash e:T \surd \Longrightarrow \exists T'.\ P,E,s' \vdash e':T' \surd \wedge P \vdash T' \le T$

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

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 $\leftarrow$) 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* :: $'a\ exp \Rightarrow\ 'a\ exp \Rightarrow\ 'a\ exp$ **where**
init-switch $(INIT\ C\ (Cs,b) \leftarrow e_i)\ e = (INIT\ C\ (Cs,b) \leftarrow e)$ |
init-switch $(RI(C,e');Cs \leftarrow e_i)\ e = (RI(C,e');Cs \leftarrow e)$ |
init-switch $e'\ e = e'$

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

lemma *init-red-init*:
$[\![$ *init-exp-of* $e_0 = \lfloor e \rfloor;\ P \vdash \langle e_0,s_0,b_0\rangle \rightarrow \langle e_1,s_1,b_1\rangle$ $]\!]$
$\Longrightarrow$ (*init-exp-of* $e_1 = \lfloor e \rfloor \wedge$ (*INIT-class* $e_0 = \lfloor C \rfloor \longrightarrow$ *INIT-class* $e_1 = \lfloor C \rfloor$))
$\vee$ $(e_1 = e \wedge b_1 = icheck\ P\ (the(INIT\text{-}class\ e_0))\ e) \vee (\exists\, a.\ e_1 = throw\ a)$
by(*erule red.cases, auto*)

lemma *init-exp-switch*[*simp*]:
init-exp-of $e_0 = \lfloor e \rfloor \Longrightarrow$ *init-exp-of* (*init-switch* $e_0\ e'$) $= \lfloor e' \rfloor$
by(*cases* e_0, *simp-all*)

lemma *init-red-sync*:
$[\![\ P \vdash \langle e_0,s_0,b_0\rangle \rightarrow \langle e_1,s_1,b_1\rangle$; *init-exp-of* $e_0 = \lfloor e \rfloor;\ e_1 \neq e\ ]\!]$
$\Longrightarrow (\bigwedge e'.\ P \vdash \langle init\text{-}switch\ e_0\ e',s_0,b_0\rangle \rightarrow \langle init\text{-}switch\ e_1\ e',s_1,b_1\rangle)$
proof(*induct rule*: *red.cases*) **qed**(*simp-all add*: *red-reds.intros*)

lemma *init-red-sync-end*:
$[\![\ P \vdash \langle e_0,s_0,b_0\rangle \rightarrow \langle e_1,s_1,b_1\rangle$; *init-exp-of* $e_0 = \lfloor e \rfloor;\ e_1 = e$; *throw-of* e = *None* $]\!]$
$\Longrightarrow (\bigwedge e'.\ \neg sub\text{-}RI\ e' \Longrightarrow P \vdash \langle init\text{-}switch\ e_0\ e',s_0,b_0\rangle \rightarrow \langle e',s_1,icheck\ P\ (the(INIT\text{-}class\ e_0))\ e'\rangle)$
proof(*induct rule*: *red.cases*) **qed**(*simp-all add*: *red-reds.intros*)

lemma *init-reds-sync-unit'*:
$[\![\ P \vdash \langle e_0,s_0,b_0\rangle \rightarrow\!*\ \langle Val\ v',s_1,b_1\rangle$; *init-exp-of* $e_0 = \lfloor unit \rfloor$; *INIT-class* $e_0 = \lfloor C \rfloor\ ]\!]$
$\Longrightarrow (\bigwedge e'.\ \neg sub\text{-}RI\ e' \Longrightarrow P \vdash \langle init\text{-}switch\ e_0\ e',s_0,b_0\rangle \rightarrow\!*\ \langle e',s_1,icheck\ P\ (the(INIT\text{-}class\ e_0))\ e'\rangle)$
proof(*induct rule*:*converse-rtrancl-induct3*)
case *refl* **then show** *?case* **by** *simp*
next
  case (*step e0 s0 b0 e1 s1 b1*)
  have (*init-exp-of e1* $= \lfloor unit \rfloor \wedge$ (*INIT-class e0* $= \lfloor C \rfloor \longrightarrow$ *INIT-class e1* $= \lfloor C \rfloor$))
        $\vee$ (*e1* = *unit* $\wedge$ *b1* = *icheck P* (*the*(*INIT-class e0*)) *unit*) $\vee$ ($\exists\, a.$ *e1* = *throw a*)
    using *init-red-init*[*OF step.prems*(*1*) *step.hyps*(*1*)] **by** *simp*
  then show *?case*

proof(*rule disjE*)
assume *assm*: *init-exp-of e1* = $\lfloor$*unit*$\rfloor$ $\wedge$ (*INIT-class e0* = $\lfloor C \rfloor$ $\longrightarrow$ *INIT-class e1* = $\lfloor C \rfloor$)
then have *red*: $P \vdash \langle$*init-switch e0 e'*,*s0*,*b0*$\rangle \to \langle$*init-switch e1 e'*,*s1*,*b1*$\rangle$
using *init-red-sync*[*OF step.hyps*(*1*) *step.prems*(*1*)] **by** *simp*
have *reds*: $P \vdash \langle$*init-switch e1 e'*,*s1*,*b1*$\rangle \to* \langle e',s_1,$*icheck P* (*the* (*INIT-class e0*)) *e'*$\rangle$
using *step.hyps*(*3*)[*OF - - step.prems*(*3*)] *assm step.prems*(*2*) **by** *simp*
show *?thesis* **by**(*rule converse-rtrancl-into-rtrancl*[*OF red reds*])
next
assume (*e1* = *unit* $\wedge$ *b1* = *icheck P* (*the*(*INIT-class e0*)) *unit*) $\vee$ ($\exists$ *a*. *e1* = *throw a*)
then show *?thesis*
proof(*rule disjE*)
assume *assm*: *e1* = *unit* $\wedge$ *b1* = *icheck P* (*the*(*INIT-class e0*)) *unit* **then have** *e1*: *e1* = *unit* **by** *simp*
have *sb*: *s1* = s_1 *b1* = b_1 **using** *reds-final-same*[*OF step.hyps*(*2*)] *assm*
by(*simp-all add*: *final-def*)
then have *step'*: $P \vdash \langle$*init-switch e0 e'*,*s0*,*b0*$\rangle \to \langle e',s_1,$*icheck P* (*the* (*INIT-class e0*)) *e'*$\rangle$
using *init-red-sync-end*[*OF step.hyps*(*1*) *step.prems*(*1*) *e1* - *step.prems*(*3*)] **by** *auto*
then have $P \vdash \langle$*init-switch e0 e'*,*s0*,*b0*$\rangle \to* \langle e',s_1,$*icheck P* (*the* (*INIT-class e0*)) *e'*$\rangle$
using *r-into-rtrancl* **by** *auto*
then show *?thesis* **using** *step assm sb* **by** *simp*
next
assume $\exists$ *a*. *e1* = *throw a* **then obtain** *a* **where** *e1* = *throw a* **by** *clarsimp*
then have *tof*: *throw-of e1* = $\lfloor a \rfloor$ **by** *simp*
then show *?thesis* **using** *reds-throw*[*OF step.hyps*(*2*) *tof*] **by** *simp*
qed
qed
qed

lemma *init-reds-sync-unit-throw'*:
$[\![$ $P \vdash \langle e_0,s_0,b_0\rangle \to* \langle$*throw a*,$s_1,b_1\rangle$; *init-exp-of* e_0 = $\lfloor$*unit*$\rfloor$ $]\!]$
$\Longrightarrow$ ($\bigwedge e'$. $P \vdash \langle$*init-switch* e_0 e',$s_0,b_0\rangle \to* \langle$*throw a*,$s_1,b_1\rangle$)
proof(*induct rule*:*converse-rtrancl-induct3*)
case *refl* **then show** *?case* **by** *simp*
next
case (*step e0 s0 b0 e1 s1 b1*)
have *init-exp-of e1* = $\lfloor$*unit*$\rfloor$ $\wedge$ ($\forall$ *C*. *INIT-class e0* = $\lfloor C \rfloor$ $\longrightarrow$ *INIT-class e1* = $\lfloor C \rfloor$) $\vee$
e1 = *unit* $\wedge$ *b1* = *icheck P* (*the* (*INIT-class e0*)) *unit* $\vee$ ($\exists$ *a*. *e1* = *throw a*)
using *init-red-init*[*OF step.prems*(*1*) *step.hyps*(*1*)] **by** *auto*
then show *?case*
proof(*rule disjE*)
assume *assm*: *init-exp-of e1* = $\lfloor$*unit*$\rfloor$ $\wedge$ ($\forall$ *C*. *INIT-class e0* = $\lfloor C \rfloor$ $\longrightarrow$ *INIT-class e1* = $\lfloor C \rfloor$)
then have $P \vdash \langle$*init-switch e0 e'*,*s0*,*b0*$\rangle \to \langle$*init-switch e1 e'*,*s1*,*b1*$\rangle$
using *step init-red-sync*[*OF step.hyps*(*1*) *step.prems*] **by** *simp*
then show *?thesis* **using** *step assm* **by** (*meson converse-rtrancl-into-rtrancl*)
next
assume *e1* = *unit* $\wedge$ *b1* = *icheck P* (*the* (*INIT-class e0*)) *unit* $\vee$ ($\exists$ *a*. *e1* = *throw a*)
then show *?thesis*
proof(*rule disjE*)
assume *e1* = *unit* $\wedge$ *b1* = *icheck P* (*the* (*INIT-class e0*)) *unit*
then show *?thesis* **using** *step* **using** *final-def reds-final-same* **by** *blast*
next
assume *assm*: $\exists$ *a*. *e1* = *throw a*
then have $P \vdash \langle$*init-switch e0 e'*,*s0*,*b0*$\rangle \to \langle$*e1*,*s1*,*b1*$\rangle$
using *init-red-sync*[*OF step.hyps*(*1*) *step.prems*] **by** *clarsimp*

then show *?thesis* **using** *step* **by** *simp*
qed
qed
qed

lemma *init-reds-sync-unit*:
assumes $P \vdash \langle e_0, s_0, b_0\rangle \to* \langle Val\ v', s_1, b_1\rangle$ **and** $init\text{-}exp\text{-}of\ e_0 = \lfloor unit\rfloor$ **and** $INIT\text{-}class\ e_0 = \lfloor C\rfloor$
and $\neg sub\text{-}RI\ e'$
shows $P \vdash \langle init\text{-}switch\ e_0\ e', s_0, b_0\rangle \to* \langle e', s_1, icheck\ P\ (the(INIT\text{-}class\ e_0))\ e'\rangle$
using *init-reds-sync-unit'*[*OF assms*] **by** *clarsimp*

lemma *init-reds-sync-unit-throw*:
assumes $P \vdash \langle e_0, s_0, b_0\rangle \to* \langle throw\ a, s_1, b_1\rangle$ **and** $init\text{-}exp\text{-}of\ e_0 = \lfloor unit\rfloor$
shows $P \vdash \langle init\text{-}switch\ e_0\ e', s_0, b_0\rangle \to* \langle throw\ a, s_1, b_1\rangle$
using *init-reds-sync-unit-throw'*[*OF assms*] **by** *clarsimp*

— init reds lemmas

lemma *InitSeqReds*:
assumes $P \vdash \langle INIT\ C\ ([C], b) \leftarrow unit, s_0, b_0\rangle \to* \langle Val\ v', s_1, b_1\rangle$
and $P \vdash \langle e, s_1, icheck\ P\ C\ e\rangle \to* \langle e_2, s_2, b_2\rangle$ **and** $\neg sub\text{-}RI\ e$
shows $P \vdash \langle INIT\ C\ ([C], b) \leftarrow e, s_0, b_0\rangle \to* \langle e_2, s_2, b_2\rangle$
using *assms*
proof −
have $P \vdash \langle init\text{-}switch\ (INIT\ C\ ([C], b) \leftarrow unit)\ e, s_0, b_0\rangle \to* \langle e, s_1, icheck\ P\ C\ e\rangle$
using *init-reds-sync-unit*[*OF assms*(*1*) - - *assms*(*3*)] **by** *simp*
then show *?thesis* **using** *assms*(*2*) **by** *simp*
qed

lemma *InitSeqThrowReds*:
assumes $P \vdash \langle INIT\ C\ ([C], b) \leftarrow unit, s_0, b_0\rangle \to* \langle throw\ a, s_1, b_1\rangle$
shows $P \vdash \langle INIT\ C\ ([C], b) \leftarrow e, s_0, b_0\rangle \to* \langle throw\ a, s_1, b_1\rangle$
using *assms*
proof −
have $P \vdash \langle init\text{-}switch\ (INIT\ C\ ([C], b) \leftarrow unit)\ e, s_0, b_0\rangle \to* \langle throw\ a, s_1, b_1\rangle$
using *init-reds-sync-unit-throw*[*OF assms*(*1*)] **by** *simp*
then show *?thesis* **by** *simp*
qed

lemma *InitNoneReds*:
$[\![\ sh\ C = None;$
$P \vdash \langle INIT\ C'\ (C\ \#\ Cs, False) \leftarrow e, (h,\ l,\ sh(C \mapsto (sblank\ P\ C,\ Prepared))), b\rangle \to* \langle e', s', b'\rangle\ ]\!]$
$\Longrightarrow P \vdash \langle INIT\ C'\ (C\#Cs, False) \leftarrow e, (h, l, sh), b\rangle \to* \langle e', s', b'\rangle$
lemma *InitDoneReds*:
$[\![\ sh\ C = Some(sfs, Done);\ P \vdash \langle INIT\ C'\ (Cs, True) \leftarrow e, (h, l, sh), b\rangle \to* \langle e', s', b'\rangle\ ]\!]$
$\Longrightarrow P \vdash \langle INIT\ C'\ (C\#Cs, False) \leftarrow e, (h, l, sh), b\rangle \to* \langle e', s', b'\rangle$
lemma *InitProcessingReds*:
$[\![\ sh\ C = Some(sfs, Processing);\ P \vdash \langle INIT\ C'\ (Cs, True) \leftarrow e, (h, l, sh), b\rangle \to* \langle e', s', b'\rangle\ ]\!]$
$\Longrightarrow P \vdash \langle INIT\ C'\ (C\#Cs, False) \leftarrow e, (h, l, sh), b\rangle \to* \langle e', s', b'\rangle$
lemma *InitErrorReds*:
$[\![\ sh\ C = Some(sfs, Error);\ P \vdash \langle RI\ (C, THROW\ NoClassDefFoundError); Cs \leftarrow e, (h, l, sh), b\rangle \to* \langle e', s', b'\rangle\ ]\!]$
$\Longrightarrow P \vdash \langle INIT\ C'\ (C\#Cs, False) \leftarrow e, (h, l, sh), b\rangle \to* \langle e', s', b'\rangle$
lemma *InitObjectReds*:

$\llbracket$ *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'*$\rangle$ $\rrbracket$
$\Longrightarrow$ *P* $\vdash$ $\langle$*INIT C'* (*C*#*Cs*,*False*) $\leftarrow$ *e*,(*h*,*l*,*sh*),*b*$\rangle$ $\rightarrow$* $\langle$*e'*,*s'*,*b'*$\rangle$
lemma *InitNonObjectReds*:
$\llbracket$ *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'*$\rangle$ $\rrbracket$
$\Longrightarrow$ *P* $\vdash$ $\langle$*INIT C'* (*C*#*Cs*,*False*) $\leftarrow$ *e*,(*h*,*l*,*sh*),*b*$\rangle$ $\rightarrow$* $\langle$*e'*,*s'*,*b'*$\rangle$
lemma *RedsInitRInit*:
 P $\vdash$ $\langle$*RI* (*C*,*C*$\bullet_s$*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'*$\rangle$
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'*$\rangle$
lemma *RedsRInit*:
$\llbracket$ *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'*$\rangle$ $\rrbracket$
 $\Longrightarrow$ *P* $\vdash$ $\langle$*RI* (*C*, e_0);*Cs* $\leftarrow$ *e*,s_0,$b_0\rangle$ $\rightarrow$* $\langle$*e'*,*s'*,*b'*$\rangle$

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

New

lemma *NewInitDoneReds*:
 $\llbracket$ *sh C* = *Some* (*sfs*, *Done*); *new-Addr h* = *Some a*;
 P $\vdash$ *C has-fields FDTs*; *h'* = *h*(*a*$\mapsto$*blank P C*) $\rrbracket$
 $\Longrightarrow$ *P* $\vdash$ $\langle$*new C*,(*h*,*l*,*sh*),*False*$\rangle$ $\rightarrow$* $\langle$*addr a*,(*h'*,*l*,*sh*),*False*$\rangle$
lemma *NewInitDoneReds2*:
 $\llbracket$ *sh C* = *Some* (*sfs*, *Done*); *new-Addr h* = *None*; *is-class P C* $\rrbracket$
 $\Longrightarrow$ *P* $\vdash$ $\langle$*new C*,(*h*,*l*,*sh*),*False*$\rangle$ $\rightarrow$* $\langle$*THROW OutOfMemory*, (*h*,*l*,*sh*), *False*$\rangle$
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*$\rangle$ $\rightarrow$* $\langle$*Val v'*,(*h'*,*l'*,*sh'*),*b'*$\rangle$
 and *Addr*: *new-Addr h'* = *Some a* **and** *has*: *P* $\vdash$ *C has-fields FDTs*
 and *h''*: *h''* = *h'*(*a*$\mapsto$*blank P C*) **and** *cls-C*: *is-class P C*
shows *P* $\vdash$ $\langle$*new C*,*s*,*False*$\rangle$ $\rightarrow$* $\langle$*addr a*,(*h''*,*l'*,*sh'*),*False*$\rangle$
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*$\rangle$ $\rightarrow$* $\langle$*Val v'*,(*h'*,*l'*,*sh'*),*b'*$\rangle$
 and *Addr*: *new-Addr h'* = *None* **and** *cls-C*: *is-class P C*
shows *P* $\vdash$ $\langle$*new C*,*s*,*False*$\rangle$ $\rightarrow$* $\langle$*THROW OutOfMemory*,(*h'*,*l'*,*sh'*),*False*$\rangle$
lemma *NewInitThrowReds*:

assumes *nDone*: $\nexists$ *sfs. shp s C = Some (sfs, Done)*
and *cls-C*: *is-class P C*
and *INIT-steps*: $P \vdash \langle INIT\ C\ ([C],False) \leftarrow unit,s,False\rangle \rightarrow* \langle throw\ a,s',b'\rangle$
shows $P \vdash \langle new\ C,s,False\rangle \rightarrow* \langle throw\ a,s',b'\rangle$

Cast

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

LAss

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

BinOp

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

FAcc

lemma *FAccReds*:
$P \vdash \langle e,s,b\rangle \rightarrow* \langle e',s',b'\rangle \Longrightarrow P \vdash \langle e{\cdot}F\{D\},\ s,b\rangle \rightarrow* \langle e'{\cdot}F\{D\},\ s',b'\rangle$
lemma *FAccRedsVal*:
$[\![\ P \vdash \langle e,s,b\rangle \rightarrow* \langle addr\ a,s',b'\rangle;\ hp\ s'\ a = Some(C,fs);\ fs(F,D) = Some\ v;$
$P \vdash C\ has\ F,NonStatic{:}t\ in\ D\ ]\!]$
$\Longrightarrow P \vdash \langle e{\cdot}F\{D\},s,b\rangle \rightarrow* \langle Val\ v,s',b'\rangle$

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

SFAcc

lemma *SFAccReds*:
$[\![\ P \vdash C\ has\ F,Static{:}t\ in\ D;$
$shp\ s\ D = Some(sfs,Done);\ sfs\ F = Some\ v\ ]\!]$
$\Longrightarrow P \vdash \langle C{\cdot}_sF\{D\},s,True\rangle \rightarrow* \langle Val\ v,s,False\rangle$
lemma *SFAccRedsNone*:
$\neg(\exists\, b\ t.\ P \vdash C\ has\ F,b{:}t\ in\ D)$
$\Longrightarrow P \vdash \langle C{\cdot}_sF\{D\},s,b\rangle \rightarrow* \langle THROW\ NoSuchFieldError,s,False\rangle$
lemma *SFAccRedsNonStatic*:
$P \vdash C\ has\ F,NonStatic{:}t\ in\ D$
$\Longrightarrow P \vdash \langle C{\cdot}_sF\{D\},s,b\rangle \rightarrow* \langle THROW\ IncompatibleClassChangeError,s,False\rangle$
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 $P \vdash \langle C{\cdot}_sF\{D\},\ s,\ b\rangle \rightarrow* \langle Val\ v,\ s,\ False\rangle$
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*: $P \vdash \langle INIT\ D\ ([D],False) \leftarrow unit,s,False\rangle \rightarrow* \langle Val\ v',s',b'\rangle$
and *shp'*: $shp\ s'\ D = Some\ (sfs,i)$ **and** *sfs*: $sfs\ F = Some\ v$
shows $P \vdash \langle C{\cdot}_sF\{D\},s,False\rangle \rightarrow* \langle Val\ v,s',False\rangle$
lemma *SFAccInitThrowReds*:
$[\![\ P \vdash C\ has\ F,Static{:}t\ in\ D;$
$\nexists sfs.\ shp\ s\ D = Some\ (sfs,Done);$
$P \vdash \langle INIT\ D\ ([D],False) \leftarrow unit,s,False\rangle \rightarrow* \langle throw\ a,s',b'\rangle\ ]\!]$
$\Longrightarrow P \vdash \langle C{\cdot}_sF\{D\},s,False\rangle \rightarrow* \langle throw\ a,s',b'\rangle$

FAss

lemma *FAssReds1*:
$P \vdash \langle e,s,b\rangle \rightarrow* \langle e',s',b'\rangle \Longrightarrow P \vdash \langle e{\cdot}F\{D\}{:=}e_2,\ s,b\rangle \rightarrow* \langle e'{\cdot}F\{D\}{:=}e_2,\ s',b'\rangle$
lemma *FAssReds2*:
$P \vdash \langle e,s,b\rangle \rightarrow* \langle e',s',b'\rangle \Longrightarrow P \vdash \langle Val\ v{\cdot}F\{D\}{:=}e,\ s,b\rangle \rightarrow* \langle Val\ v{\cdot}F\{D\}{:=}e',\ s',b'\rangle$
lemma *FAssRedsVal*:
assumes e_1*-steps*: $P \vdash \langle e_1,s_0,b_0\rangle \rightarrow* \langle addr\ a,s_1,b_1\rangle$
and e_2*-steps*: $P \vdash \langle e_2,s_1,b_1\rangle \rightarrow* \langle Val\ v,(h_2,l_2,sh_2),b_2\rangle$
and *cF*: $P \vdash C\ has\ F,NonStatic{:}t\ in\ D$ **and** *hC*: $Some(C,fs) = h_2\ a$
shows $P \vdash \langle e_1{\cdot}F\{D\}{:=}e_2,\ s_0,\ b_0\rangle \rightarrow* \langle unit,\ (h_2(a{\mapsto}(C,fs((F,D){\mapsto}v))),l_2,sh_2),b_2\rangle$

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

SFAss

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

shows $P \vdash \langle C\bullet_s F\{D\}{:=}e_2{,}s_0{,}b_0\rangle \to* \langle throw\ a{,}s'{,}b'\rangle$

;;

lemma *SeqReds*:
$P \vdash \langle e{,}s{,}b\rangle \to* \langle e'{,}s'{,}b'\rangle \Longrightarrow P \vdash \langle e;;e_2,\ s{,}b\rangle \to* \langle e';;e_2,\ s'{,}b'\rangle$
lemma *SeqRedsThrow*:
$P \vdash \langle e{,}s{,}b\rangle \to* \langle throw\ e'{,}s'{,}b'\rangle \Longrightarrow P \vdash \langle e;;e_2,\ s{,}b\rangle \to* \langle throw\ e',\ s'{,}b'\rangle$
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 $P \vdash \langle e_1;;e_2,\ s_0{,}b_0\rangle \to* \langle {e_2}'{,}s_2{,}b_2\rangle$

If

lemma *CondReds*:
$P \vdash \langle e{,}s{,}b\rangle \to* \langle e'{,}s'{,}b'\rangle \Longrightarrow P \vdash \langle if\ (e)\ e_1\ else\ e_2{,}s{,}b\rangle \to* \langle if\ (e')\ e_1\ else\ e_2{,}s'{,}b'\rangle$
lemma *CondRedsThrow*:
$P \vdash \langle e{,}s{,}b\rangle \to* \langle throw\ a{,}s'{,}b'\rangle \Longrightarrow P \vdash \langle if\ (e)\ e_1\ else\ e_2,\ s{,}b\rangle \to* \langle throw\ a{,}s'{,}b'\rangle$
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 $P \vdash \langle if\ (e)\ e_1\ else\ e_2,\ s_0{,}b_0\rangle \to* \langle e'{,}s_2{,}b_2\rangle$
lemma *CondReds2F*:
assumes *e-steps*: $P \vdash \langle e{,}s_0{,}b_0\rangle \to* \langle false{,}s_1{,}b_1\rangle$
and e_2*-steps*: $P \vdash \langle e_2,\ s_1{,}b_1\rangle \to* \langle e'{,}s_2{,}b_2\rangle$
shows $P \vdash \langle if\ (e)\ e_1\ else\ e_2,\ s_0{,}b_0\rangle \to* \langle e'{,}s_2{,}b_2\rangle$

While

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

Throw

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

InitBlock

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

Block

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

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

try-catch

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

List

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

Call

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

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

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

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

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

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

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

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'': $l'' = override\text{-}on\ l'\ l\ (set\ Vs)$
shows $P \vdash \langle blocks(Vs,Ts,vs,e),\ (h,l,sh),b\rangle \to{*}\ \langle e',\ (h',l'',sh'),b'\rangle$

An now the actual method call reduction lemmas.

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

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

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 $P \vdash \langle e{\cdot}M(es),\ s_0,b_0\rangle \to{*}\ \langle ef,(h_3,l_2,sh_3),b_3\rangle$

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 $P \vdash \langle e{\cdot}M(es),s_0,b_0\rangle \to{*}\ \langle throw\ a,s_2,b_2\rangle$

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

lemma *CallRedsNull*:

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

1.23.2 SCall

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

The main Theorem

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

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

lemma *blocksEval*:
$\bigwedge Ts\ vs\ l\ l'.\ [\![size\ ps = size\ Ts;\ size\ ps = size\ vs;\ P \vdash \langle blocks(ps,Ts,vs,e),(h,l,sh)\rangle \Rightarrow \langle e',(h',l',sh')\rangle\ ]\!]$
$\Longrightarrow \exists\ l''.\ P \vdash \langle e,(h,l(ps[\mapsto]vs),sh)\rangle \Rightarrow \langle e',(h',l'',sh')\rangle$
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 $P \vdash \langle es,(h,l,sh)\rangle\ [\Rightarrow]\ \langle es',(h',l',sh')\rangle \Longrightarrow (\bigwedge W.\ fvs\ es \subseteq W \Longrightarrow P \vdash \langle es,(h,l|`W,sh)\rangle\ [\Rightarrow]\ \langle es',(h',l'|`W,sh')\rangle)$

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 $P \vdash \langle es,(h,l,sh)\rangle\ [\Rightarrow]\ \langle es',(h',l',sh')\rangle \Longrightarrow (\bigwedge V.\ V \notin fvs\ es \Longrightarrow l'\ V = l\ V)$

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

lemma *list-eval-Throw*:
assumes *eval-e*: $P \vdash \langle throw\ x,s\rangle \Rightarrow \langle e',s'\rangle$

shows $P \vdash \langle map\ Val\ vs\ @\ throw\ x\ \#\ es',s\rangle\ [\Rightarrow]\ \langle map\ Val\ vs\ @\ e'\ \#\ es',s'\rangle$
— separate evaluation of first subexp of a sequence
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 $P \vdash \langle e\ ;;\ e_0,s\rangle \Rightarrow \langle e',s'\rangle$
proof(*rule eval-cases*(*14*)[*OF seq*]) — Seq
fix $v'\ s_1$ **assume** e'': $P \vdash \langle e'',s''\rangle \Rightarrow \langle Val\ v',s_1\rangle$ **and** *estep*: $P \vdash \langle e_0,s_1\rangle \Rightarrow \langle e',s'\rangle$
have $P \vdash \langle e,s\rangle \Rightarrow \langle Val\ v',s_1\rangle$ **using** e'' *IH* **by** *simp*
then show *?thesis* **using** *estep Seq* **by** *simp*
next
fix e_t **assume** e'': $P \vdash \langle e'',s''\rangle \Rightarrow \langle throw\ e_t,s'\rangle$ **and** e': $e' = throw\ e_t$
have $P \vdash \langle e,s\rangle \Rightarrow \langle throw\ e_t,s'\rangle$ **using** e'' *IH* **by** *simp*
then show *?thesis* **using** *eval-evals.SeqThrow* e' **by** *simp*
qed

— separate evaluation of *RI* subexp, val case
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 $P \vdash \langle RI\ (C,e)\ ;\ Cs \leftarrow e_0,s\rangle \Rightarrow \langle Val\ v',s_1\rangle$
proof(*rule eval-cases*(*20*)[*OF ri*]) — RI
fix $v''\ h'\ l'\ sh'\ sfs\ i$
assume $e''step$: $P \vdash \langle e'',s''\rangle \Rightarrow \langle Val\ v'',(h',\ l',\ sh')\rangle$
and *shC*: $sh'\ C = \lfloor(sfs,\ i)\rfloor$
and *init*: $P \vdash \langle INIT\ (if\ Cs = []\ then\ C\ else\ last\ Cs)\ (Cs,True) \leftarrow e_0,(h',\ l',\ sh'(C \mapsto (sfs,\ Done)))\rangle$
$\Rightarrow$
$\langle Val\ v',s_1\rangle$
have $P \vdash \langle e,s\rangle \Rightarrow \langle Val\ v'',(h',\ l',\ sh')\rangle$ **using** *IH*[*OF* $e''step$] **by** *simp*
then show *?thesis* **using** *RInit init shC* **by** *auto*
next
fix $a\ h'\ l'\ sh'\ sfs\ i\ D\ Cs'$
assume $e''step$: $P \vdash \langle e'',s''\rangle \Rightarrow \langle throw\ a,(h',\ l',\ sh')\rangle$
and *riD*: $P \vdash \langle RI\ (D,throw\ a)\ ;\ Cs' \leftarrow e_0,(h',\ l',\ sh'(C \mapsto (sfs,\ Error)))\rangle \Rightarrow \langle Val\ v',s_1\rangle$
have $P \vdash \langle e,s\rangle \Rightarrow \langle throw\ a,(h',\ l',\ sh')\rangle$ **using** *IH*[*OF* $e''step$] **by** *simp*
then show *?thesis* **using** *riD rinit-throwE* **by** *blast*
qed(*simp*)

— separate evaluation of *RI* subexp, throw case
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 $P \vdash \langle RI\ (C,e)\ ;\ Cs \leftarrow e_0,s\rangle \Rightarrow \langle throw\ e_t,s'\rangle$
proof(*rule eval-cases*(*20*)[*OF ri*]) — RI
fix $v\ h'\ l'\ sh'\ sfs\ i$
assume $e''step$: $P \vdash \langle e'',s''\rangle \Rightarrow \langle Val\ v,(h',\ l',\ sh')\rangle$
and *shC*: $sh'\ C = \lfloor(sfs,\ i)\rfloor$
and *init*: $P \vdash \langle INIT\ (if\ Cs = []\ then\ C\ else\ last\ Cs)\ (Cs,True) \leftarrow e_0,(h',\ l',\ sh'(C \mapsto (sfs,\ Done)))\rangle$
$\Rightarrow$
$\langle throw\ e_t,s'\rangle$
have $P \vdash \langle e,s\rangle \Rightarrow \langle Val\ v,(h',\ l',\ sh')\rangle$ **using** *IH*[*OF* $e''step$] **by** *simp*
then show *?thesis* **using** *RInit init shC* **by** *auto*
next
fix $a\ h'\ l'\ sh'\ sfs\ i\ D\ Cs'$

assume $e''step$: $P \vdash \langle e'',s''\rangle \Rightarrow \langle throw\ a,(h',\ l',\ sh')\rangle$
and shC: $sh'\ C = \lfloor(sfs,\ i)\rfloor$
and riD: $P \vdash \langle RI\ (D,throw\ a)\ ;\ Cs' \leftarrow e_0,(h',\ l',\ sh'(C \mapsto (sfs,\ Error)))\rangle \Rightarrow \langle throw\ e_t,s'\rangle$
and $cons$: $Cs = D\ \#\ Cs'$
have $estep'$: $P \vdash \langle e,s\rangle \Rightarrow \langle throw\ a,(h',\ l',\ sh')\rangle$ **using** $IH[OF\ e''step]$ **by** $simp$
then show *?thesis* **using** *RInitInitFail cons riD shC* **by** *simp*
next
fix $a\ h'\ l'\ sh'\ sfs\ i$
assume $throw\ e_t = throw\ a$
and $s' = (h',\ l',\ sh'(C \mapsto (sfs,\ Error)))$
and $P \vdash \langle e'',s''\rangle \Rightarrow \langle throw\ a,(h',\ l',\ sh')\rangle$
and $sh'\ C = \lfloor(sfs,\ i)\rfloor$
and $Cs = []$
then show *?thesis* **using** *RInitFailFinal IH* **by** *auto*
qed

— separate evaluation of *RI* subexp
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 $\bigwedge e'\ s'.\ P \vdash \langle RI\ (C,e'')\ ;\ Cs \leftarrow e_0,s''\rangle \Rightarrow \langle e',s'\rangle$
$\Longrightarrow P \vdash \langle RI\ (C,e)\ ;\ Cs \leftarrow e_0,s\rangle \Rightarrow \langle e',s'\rangle$
proof −
fix $e'\ s'$ **assume** ri'': $P \vdash \langle RI\ (C,e'')\ ;\ Cs \leftarrow e_0,s''\rangle \Rightarrow \langle e',s'\rangle$
then have $final\ e'$ **using** *eval-final* **by** *simp*
then show $P \vdash \langle RI\ (C,e)\ ;\ Cs \leftarrow e_0,s\rangle \Rightarrow \langle e',s'\rangle$
proof(*rule finalE*)
fix v **assume** $e' = Val\ v$ **then show** *?thesis* **using** $rinit\text{-}Val\text{-}ext[OF\ \text{-}\ IH]\ ri''$ **by** *simp*
next
fix a **assume** $e' = throw\ a$ **then show** *?thesis* **using** $rinit\text{-}throw\text{-}ext[OF\ \text{-}\ IH]\ ri''$ **by** *simp*
qed
qed

— *INIT* and *RI* return either *Val* with *Done* or *Processing* flag or *Throw* with *Error* flag
lemma
shows *eval-init-return*: $P \vdash \langle e,s\rangle \Rightarrow \langle e',s'\rangle$
$\Longrightarrow iconf\ (shp\ s)\ e$
$\Longrightarrow (\exists\ Cs\ b.\ e = INIT\ C'\ (Cs,b) \leftarrow unit) \vee (\exists\ C\ e_0\ Cs\ e_i.\ e = RI(C,e_0);Cs@[C'] \leftarrow unit)$
$\vee (\exists\ e_0.\ e = RI(C',e_0);Nil \leftarrow unit)$
$\Longrightarrow (val\text{-}of\ e' = Some\ v \longrightarrow (\exists\ sfs\ i.\ shp\ s'\ C' = \lfloor(sfs,i)\rfloor \wedge (i = Done \vee i = Processing)))$
$\wedge (throw\text{-}of\ e' = Some\ a \longrightarrow (\exists\ sfs\ i.\ shp\ s'\ C' = \lfloor(sfs,Error)\rfloor))$
and $P \vdash \langle es,s\rangle\ [\Rightarrow]\ \langle es',s'\rangle \Longrightarrow True$
proof(*induct rule*: *eval-evals.inducts*)
case $(InitFinal\ e\ s\ e'\ s'\ C\ b)$ **then show** *?case*
by(*fastforce simp*: *initPD-def dest*: *eval-final-same*)
next
case $(InitDone\ sh\ C\ sfs\ C'\ Cs\ e\ h\ l\ e'\ s')$
then have $final\ e'$ **using** *eval-final* **by** *simp*
then show *?case*
proof(*rule finalE*)
fix v **assume** e': $e' = Val\ v$ **then show** *?thesis* **using** *InitDone initPD-def*
proof(*cases Cs*) **qed**(*auto*)
next
fix a **assume** e': $e' = throw\ a$ **then show** *?thesis* **using** *InitDone initPD-def*
proof(*cases Cs*) **qed**(*auto*)

qed
next
case (*InitProcessing sh C sfs* C' *Cs e h l* e' s')
then have *final* e' **using** *eval-final* **by** *simp*
then show *?case*
proof(*rule finalE*)
fix *v* **assume** e': $e' =$ *Val v* **then show** *?thesis* **using** *InitProcessing initPD-def*
proof(*cases Cs*) **qed**(*auto*)
next
fix *a* **assume** e': $e' =$ *throw a* **then show** *?thesis* **using** *InitProcessing initPD-def*
proof(*cases Cs*) **qed**(*auto*)
qed
next
case (*InitError sh C sfs Cs e h l* e' s' C') **show** *?case*
proof(*cases Cs*)
case *Nil* **then show** *?thesis* **using** *InitError* **by** *simp*
next
case (*Cons C2 list*)
then have *final* e' **using** *InitError eval-final* **by** *simp*
then show *?thesis*
proof(*rule finalE*)
fix *v* **assume** e': $e' =$ *Val v* **then show** *?thesis*
using *InitError*
proof −
obtain *ccss* :: *char list list* **and** *bb* :: *bool* **where**
INIT C' (*C* # *Cs*,*False*) ← *e* = *INIT* C' (*ccss*,*bb*) ← *unit*
using *InitError.prems*(*2*) **by** *blast*
then show *?thesis* **using** *InitError.hyps*(*2*) e' **by**(*auto dest*!: *rinit-throwE*)
qed
next
fix *a* **assume** e': $e' =$ *throw a*
then show *?thesis* **using** *Cons InitError cons-to-append*[*of list*] **by** *clarsimp*
qed
qed
next
case (*InitRInit C Cs h l sh* e' s' C') **show** *?case*
proof(*cases Cs*)
case *Nil* **then show** *?thesis* **using** *InitRInit* **by** *simp*
next
case (*Cons* C' *list*) **then show** *?thesis*
using *InitRInit Cons cons-to-append*[*of list*] **by** *clarsimp*
qed
next
case (*RInit e s v* h' l' sh' *C sfs i* sh'' C' *Cs* e' e_1 s_1)
then have *final*: *final* e_1 **using** *eval-final* **by** *simp*
then show *?case*
proof(*cases Cs*)
case *Nil* **show** *?thesis* **using** *final*
proof(*rule finalE*)
fix *v* **assume** e': $e_1 =$ *Val v* **show** *?thesis*
using *RInit Nil* **by**(*auto simp*: *fun-upd-same initPD-def*)
next
fix *a* **assume** e': $e_1 =$ *throw a* **show** *?thesis*
using *RInit Nil* **by**(*auto simp*: *fun-upd-same initPD-def*)

qed
next
case (*Cons a list*) **show** *?thesis*
proof(*rule finalE*[*OF final*])
fix *v* **assume** e': e_1 = *Val v* **then show** *?thesis*
using *RInit Cons* **by** *clarsimp* (*metis last.simps last-appendR list.distinct*(*1*))
next
fix *a* **assume** e': e_1 = *throw a* **then show** *?thesis*
using *RInit Cons* **by** *clarsimp* (*metis last.simps last-appendR list.distinct*(*1*))
qed
qed
next
case (*RInitInitFail e s a h' l' sh' C sfs i sh'' D Cs e'* e_1 s_1)
then have *final*: *final* e_1 **using** *eval-final* **by** *simp*
then show *?case*
proof(*rule finalE*)
fix *v* **assume** e': e_1 = *Val v* **then show** *?thesis*
using *RInitInitFail* **by** *clarsimp* (*meson exp.distinct*(*101*) *rinit-throwE*)
next
fix a' **assume** e': e_1 = *Throw* a'
then have *iconf* ($sh'(C \mapsto (sfs,\ Error)))$ *a*
using *RInitInitFail.hyps*(*1*) *eval-final* **by** *fastforce*
then show *?thesis* **using** *RInitInitFail* e'
by *clarsimp* (*meson Cons-eq-append-conv list.inject*)
qed
qed(*auto simp*: *fun-upd-same*)

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*) ($INIT\ C'\ (Cs,b) \leftarrow unit$)
$\Longrightarrow \exists\, sfs\ i.\ shp\ s'\ C' = \lfloor(sfs,i)\rfloor \wedge (i = Done \vee i = Processing)$
by(*drule-tac v* = *v* **in** *eval-init-return*, *simp*+)

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*) ($INIT\ C'\ (Cs,b) \leftarrow unit$)
$\Longrightarrow \exists\, sfs\ i.\ shp\ s'\ C' = \lfloor(sfs,Error)\rfloor$
by(*drule-tac a* = *a* **in** *eval-init-return*, *simp*+)

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*) ($RI(C,e_0);Cs \leftarrow unit) \Longrightarrow last(C\#Cs) = C'$
$\Longrightarrow \exists\, sfs\ i.\ shp\ s'\ C' = \lfloor(sfs,i)\rfloor \wedge (i = Done \vee i = Processing)$
by(*auto dest*!: *eval-init-return* [**where** C'=C']
append-butlast-last-id[*THEN sym*])

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*) ($RI(C,e_0);Cs \leftarrow unit) \Longrightarrow last(C\#Cs) = C'$
$\Longrightarrow \exists\, sfs\ i.\ shp\ s'\ C' = \lfloor(sfs,Error)\rfloor$
by(*auto dest*!: *eval-init-return* [**where** C'=C']
append-butlast-last-id[*THEN sym*])

— combining the above to get evaluation of *INIT* in a sequence

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(*induct rule*: *eval-evals.inducts*)
case *InitFinal* **then show** *?case* **by**(*auto simp*: *Seq*[*OF eval-evals.InitFinal*[*OF Val*[**where** *v*=*Unit*]]])
next
case (*InitNone sh*) **then show** *?case*
using *seq-ext*[*OF eval-evals.InitNone*[**where** *sh*=*sh* **and** *e*=*unit*, *OF InitNone.hyps*(*1*)]] **by** *fastforce*
next
case (*InitDone sh*) **then show** *?case*
using *seq-ext*[*OF eval-evals.InitDone*[**where** *sh*=*sh* **and** *e*=*unit*, *OF InitDone.hyps*(*1*)]] **by** *fastforce*
next
case (*InitProcessing sh*) **then show** *?case*
using *seq-ext*[*OF eval-evals.InitProcessing*[**where** *sh*=*sh* **and** *e*=*unit*, *OF InitProcessing.hyps*(*1*)]]
by *fastforce*
next
case (*InitError sh*) **then show** *?case*
using *seq-ext*[*OF eval-evals.InitError*[**where** *sh*=*sh* **and** *e*=*unit*, *OF InitError.hyps*(*1*)]] **by** *fast-force*
next
case (*InitObject sh*) **then show** *?case*
using *seq-ext*[*OF eval-evals.InitObject*[**where** *sh*=*sh* **and** *e*=*unit*, *OF InitObject.hyps*(*1*)]]
by *fastforce*
next
case (*InitNonObject sh*) **then show** *?case*
using *seq-ext*[*OF eval-evals.InitNonObject*[**where** *sh*=*sh* **and** *e*=*unit*, *OF InitNonObject.hyps*(*1*)]]
by *fastforce*
next
case (*InitRInit C Cs e h l sh* e' s' C') **then show** *?case*
using *seq-ext*[*OF eval-evals.InitRInit*[**where** *e*=*unit*]] **by** *fastforce*
next
case *RInit* **then show** *?case*
using *seq-ext*[*OF eval-evals.RInit*[**where** *e*=*unit*, *OF RInit.hyps*(*1*)]] **by** *fastforce*
next
case *RInitInitFail* **then show** *?case*
using *seq-ext*[*OF eval-evals.RInitInitFail*[**where** *e*=*unit*, *OF RInitInitFail.hyps*(*1*)]] **by** *fastforce*
next
case *RInitFailFinal*
then show *?case* **using** *eval-evals.RInitFailFinal eval-evals.SeqThrow* **by** *auto*
qed(*auto*)

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*)$\rangle \Rightarrow \langle e'$,$s'\rangle$
by(*auto dest*: *eval-init-seq'*)

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 (*induct rule*: *red-reds.inducts*)
case (*RedNew h a C FDTs h' l sh*)

then have e':e' = *addr a* **and** s':s' = (h(a ↦ *blank P C*), *l*, *sh*)
  using *eval-cases*(*3*) **by** *fastforce*+
obtain *sfs i* **where** *shC*: *sh C* = ⌊(*sfs*, *i*)⌋ **and** *i* = *Done* ∨ *i* = *Processing*
 using *RedNew* **by** (*clarsimp simp*: *bconf-def initPD-def*)
then show *?case*
proof(*cases i*)
  case *Done* **then show** *?thesis* **using** *RedNew shC* e' s' *New* **by** *simp*
next
  case *Processing*
  then have *shC'*: ∄ *sfs*. *sh C* = *Some*(*sfs*,*Done*) **and** *shP*: *sh C* = *Some*(*sfs*,*Processing*)
    using *shC* **by** *simp*+
  then have *init*: *P* ⊢ ⟨*INIT C* ([*C*],*False*) ← *unit*,(*h*,*l*,*sh*)⟩ ⇒ ⟨*unit*,(*h*,*l*,*sh*)⟩
   by(*simp add*: *InitFinal InitProcessing Val*)
  have *P* ⊢ ⟨*new C*,(*h*, *l*, *sh*)⟩ ⇒ ⟨*addr a*,(*h*(*a* ↦ *blank P C*),*l*,*sh*)⟩
    using *RedNew shC'* **by**(*auto intro*: *NewInit*[*OF - init*])
  then show *?thesis* **using** e' s' **by** *simp*
qed(*auto*)
next
case (*RedNewFail h C l sh*)
then have e':e' = *THROW OutOfMemory* **and** s':s' = (*h*, *l*, *sh*)
  using *eval-final-same final-def* **by** *fastforce*+
obtain *sfs i* **where** *shC*: *sh C* = ⌊(*sfs*, *i*)⌋ **and** *i* = *Done* ∨ *i* = *Processing*
 using *RedNewFail* **by** (*clarsimp simp*: *bconf-def initPD-def*)
then show *?case*
proof(*cases i*)
  case *Done* **then show** *?thesis* **using** *RedNewFail shC* e' s' *NewFail* **by** *simp*
next
  case *Processing*
  then have *shC'*: ∄ *sfs*. *sh C* = *Some*(*sfs*,*Done*) **and** *shP*: *sh C* = *Some*(*sfs*,*Processing*)
    using *shC* **by** *simp*+
  then have *init*: *P* ⊢ ⟨*INIT C* ([*C*],*False*) ← *unit*,(*h*,*l*,*sh*)⟩ ⇒ ⟨*unit*,(*h*,*l*,*sh*)⟩
   by(*simp add*: *InitFinal InitProcessing Val*)
  have *P* ⊢ ⟨*new C*,(*h*, *l*, *sh*)⟩ ⇒ ⟨*THROW OutOfMemory*,(*h*,*l*,*sh*)⟩
    using *RedNewFail shC'* **by**(*auto intro*: *NewInitOOM*[*OF - init*])
  then show *?thesis* **using** e' s' **by** *simp*
qed(*auto*)
next
case (*NewInitRed sh C h l*)
then have *seq*: *P* ⊢ ⟨(*INIT C* ([*C*],*False*) ← *unit*);; *new C*,(*h*, *l*, *sh*)⟩ ⇒ ⟨e',s'⟩
  using *eval-init-seq* **by** *simp*
then show *?case*
proof(*rule eval-cases*(*14*)) — Seq
  fix *v* s_1 **assume** *init*: *P* ⊢ ⟨*INIT C* ([*C*],*False*) ← *unit*,(*h*, *l*, *sh*)⟩ ⇒ ⟨*Val v*,s_1⟩
   and *new*: *P* ⊢ ⟨*new C*,s_1⟩ ⇒ ⟨e',s'⟩
  obtain h_1 l_1 sh_1 **where** s_1: s_1 = (h_1,l_1,sh_1) **by**(*cases* s_1)
  then obtain *sfs i* **where** *shC*: sh_1 *C* = ⌊(*sfs*, *i*)⌋ **and** *iDP*: *i* = *Done* ∨ *i* = *Processing*
    using *init-Val-PD*[*OF init*] **by** *auto*
  show *?thesis*
  proof(*rule eval-cases*(*1*)[*OF new*]) — New
    fix *sha ha a FDTs la*
    assume s_1a: s_1 = (*ha*, *la*, *sha*) **and** e': e' = *addr a*
      and s': s' = (*ha*(*a* ↦ *blank P C*), *la*, *sha*)
      and *addr*: *new-Addr ha* = ⌊*a*⌋ **and** *fields*: *P* ⊢ *C has-fields FDTs*
    then show *?thesis* **using** *NewInit*[*OF - - addr fields*] *NewInitRed.hyps init* **by** *simp*

next
fix *sha ha la*
assume $s_1 = (ha,\ la,\ sha)$ **and** $e' = THROW\ OutOfMemory$
and $s' = (ha,\ la,\ sha)$ **and** *new-Addr ha = None*
then show *?thesis* **using** *NewInitOOM NewInitRed.hyps init* **by** *simp*
next
fix *sha ha la* v' h' l' sh' *a FDTs*
assume s_1a: $s_1 = (ha,\ la,\ sha)$ **and** e': $e' = addr\ a$
and s': $s' = (h'(a \mapsto blank\ P\ C),\ l',\ sh')$
and *shaC*: $\forall\, sfs.\ sha\ C \neq \lfloor(sfs,\ Done)\rfloor$
and $init'$: $P \vdash \langle INIT\ C\ ([C],False) \leftarrow unit,(ha,\ la,\ sha)\rangle \Rightarrow \langle Val\ v',(h',\ l',\ sh')\rangle$
and *addr*: $new\text{-}Addr\ h' = \lfloor a\rfloor$ **and** *fields*: $P \vdash C$ *has-fields FDTs*
then have i: $i = Processing$ **using** *iDP shC* s_1 **by** *simp*
then have $(h',\ l',\ sh') = (ha,\ la,\ sha)$ **using** $init'$ *init-ProcessingE* s_1 s_1a *shC* **by** *blast*
then show *?thesis* **using** *NewInit NewInitRed.hyps* s_1a *addr fields init shaC* e' s' **by** *auto*
next
fix *sha ha la* v' h' l' sh'
assume s_1a: $s_1 = (ha,\ la,\ sha)$ **and** e': $e' = THROW\ OutOfMemory$
and s': $s' = (h',\ l',\ sh')$ **and** $\forall\, sfs.\ sha\ C \neq \lfloor(sfs,\ Done)\rfloor$
and $init'$: $P \vdash \langle INIT\ C\ ([C],False) \leftarrow unit,(ha,\ la,\ sha)\rangle \Rightarrow \langle Val\ v',(h',\ l',\ sh')\rangle$
and *addr*: $new\text{-}Addr\ h' = None$
then have i: $i = Processing$ **using** *iDP shC* s_1 **by** *simp*
then have $(h',\ l',\ sh') = (ha,\ la,\ sha)$ **using** $init'$ *init-ProcessingE* s_1 s_1a *shC* **by** *blast*
then show *?thesis*
using *NewInitOOM NewInitRed.hyps* e' *addr* s' s_1a *init* **by** *auto*
next
fix *sha ha la a*
assume s_1a: $s_1 = (ha,\ la,\ sha)$
and $\forall\, sfs.\ sha\ C \neq \lfloor(sfs,\ Done)\rfloor$
and $init'$: $P \vdash \langle INIT\ C\ ([C],False) \leftarrow unit,(ha,\ la,\ sha)\rangle \Rightarrow \langle throw\ a,s'\rangle$
then have i: $i = Processing$ **using** *iDP shC* s_1 **by** *simp*
then show *?thesis* **using** $init'$ *init-ProcessingE* s_1 s_1a *shC* **by** *blast*
qed
next
fix e **assume** e': $e' = throw\ e$
and *init*: $P \vdash \langle INIT\ C\ ([C],False) \leftarrow unit,(h,\ l,\ sh)\rangle \Rightarrow \langle throw\ e,s'\rangle$
obtain h' l' sh' **where** s': $s' = (h',l',sh')$ **by**(*cases* s')
then obtain *sfs i* **where** *shC*: $sh'\ C = \lfloor(sfs,\ i)\rfloor$ **and** *iDP*: $i = Error$
using *init-throw-PD*[*OF init*] **by** *auto*
then show *?thesis* **by** (*simp add*: *NewInitRed.hyps NewInitThrow* e' *init*)
qed
next
case *CastRed* **then show** *?case*
by(*fastforce elim*!: *eval-cases intro*: *eval-evals.intros intro*!: *CastFail*)
next
case *RedCastNull*
then show *?case*
by *simp* (*iprover elim*: *eval-cases intro*: *eval-evals.intros*)
next
case *RedCast*
then show *?case*
by (*auto elim*: *eval-cases intro*: *eval-evals.intros*)
next
case *RedCastFail*

then show *?case*
by (*auto elim*!: *eval-cases intro*: *eval-evals.intros*)
next
case *BinOpRed1* **then show** *?case*
by(*fastforce elim*!: *eval-cases intro*: *eval-evals.intros simp*: *val-no-step*)
next
case *BinOpRed2*
thus *?case*
by (*fastforce elim*!: *eval-cases intro*: *eval-evals.intros eval-finalId*)
next
case *RedBinOp*
thus *?case*
by *simp* (*iprover elim*: *eval-cases intro*: *eval-evals.intros*)
next
case *RedVar*
thus *?case*
by (*fastforce elim*: *eval-cases intro*: *eval-evals.intros*)
next
case *LAssRed* **thus** *?case*
by(*fastforce elim*: *eval-cases intro*: *eval-evals.intros*)
next
case *RedLAss*
thus *?case*
by (*fastforce elim*: *eval-cases intro*: *eval-evals.intros*)
next
case *FAccRed* **thus** *?case*
by(*fastforce elim*: *eval-cases intro*: *eval-evals.intros*)
next
case *RedFAcc* **then show** *?case*
by (*fastforce elim*: *eval-cases intro*: *eval-evals.intros*)
next
case *RedFAccNull* **then show** *?case*
by (*fastforce elim*!: *eval-cases intro*: *eval-evals.intros*)
next
case *RedFAccNone* **thus** *?case*
by(*fastforce elim*: *eval-cases intro*: *eval-evals.intros*)
next
case *RedFAccStatic* **thus** *?case*
by(*fastforce elim*: *eval-cases intro*: *eval-evals.intros*)
next
case (*RedSFAcc C F t D sh sfs i v h l*)
then have e':$e' = Val\ v$ **and** s':$s' = (h,\ l,\ sh)$
using *eval-cases*(*3*) **by** *fastforce+*
have $i = Done \lor i = Processing$ **using** *RedSFAcc* **by** (*clarsimp simp*: *bconf-def initPD-def*)
then show *?case*
proof(*cases i*)
case *Done* **then show** *?thesis* **using** *RedSFAcc* e' s' *SFAcc* **by** *simp*
next
case *Processing*
then have shC': $\nexists sfs.\ sh\ D = Some(sfs,Done)$ **and** *shP*: $sh\ D = Some(sfs,Processing)$
using *RedSFAcc* **by** *simp+*
then have *init*: $P \vdash \langle INIT\ D\ ([D],False) \leftarrow unit,(h,l,sh)\rangle \Rightarrow \langle unit,(h,l,sh)\rangle$
by(*simp add*: *InitFinal InitProcessing Val*)
have $P \vdash \langle C{\cdot_s}F\{D\},(h,\ l,\ sh)\rangle \Rightarrow \langle Val\ v,(h,l,sh)\rangle$

by(*rule SFAccInit*[*OF RedSFAcc.hyps*(*1*) *shC′ init shP RedSFAcc.hyps*(*3*)])
then show *?thesis* **using** e' s' **by** *simp*
qed(*auto*)
next
case (*SFAccInitRed C F t D sh h l*)
then have *seq*: $P \vdash \langle(INIT\ D\ ([D],False) \leftarrow unit);;\ C\bullet_s F\{D\},(h,\ l,\ sh)\rangle \Rightarrow \langle e',s'\rangle$
using *eval-init-seq* **by** *simp*
then show *?case*
proof(*rule eval-cases*(*14*)) — Seq
fix v s_1 **assume** *init*: $P \vdash \langle INIT\ D\ ([D],False) \leftarrow unit,(h,\ l,\ sh)\rangle \Rightarrow \langle Val\ v,s_1\rangle$
and *acc*: $P \vdash \langle C\bullet_s F\{D\},s_1\rangle \Rightarrow \langle e',s'\rangle$
obtain h_1 l_1 sh_1 **where** s_1: $s_1 = (h_1,l_1,sh_1)$ **by**(*cases* s_1)
then obtain *sfs i* **where** *shD*: $sh_1\ D = \lfloor(sfs,\ i)\rfloor$ **and** *iDP*: $i = Done \vee i = Processing$
using *init-Val-PD*[*OF init*] **by** *auto*
show *?thesis*
proof(*rule eval-cases*(*8*)[*OF acc*]) — SFAcc
fix *t sha sfs v ha la*
assume $s_1 = (ha,\ la,\ sha)$ **and** $e' = Val\ v$
and $s' = (ha,\ la,\ sha)$ **and** $P \vdash C\ has\ F,Static{:}t\ in\ D$
and $sha\ D = \lfloor(sfs,\ Done)\rfloor$ **and** $sfs\ F = \lfloor v\rfloor$
then show *?thesis* **using** *SFAccInit SFAccInitRed.hyps*(*2*) *init* **by** *auto*
next
fix t sha ha la v' h' l' sh' sfs i' v
assume s_1a: $s_1 = (ha,\ la,\ sha)$ **and** e': $e' = Val\ v$
and s': $s' = (h',\ l',\ sh')$ **and** *field*: $P \vdash C\ has\ F,Static{:}t\ in\ D$
and $\forall sfs.\ sha\ D \neq \lfloor(sfs,\ Done)\rfloor$
and *init′*: $P \vdash \langle INIT\ D\ ([D],False) \leftarrow unit,(ha,\ la,\ sha)\rangle \Rightarrow \langle Val\ v',(h',\ l',\ sh')\rangle$
and *shD′*: $sh'\ D = \lfloor(sfs,\ i')\rfloor$ **and** *sfsF*: $sfs\ F = \lfloor v\rfloor$
then have i: $i = Processing$ **using** *iDP shD* s_1 **by** *simp*
then have $(h',\ l',\ sh') = (ha,\ la,\ sha)$ **using** *init′ init-ProcessingE* s_1 s_1a *shD* **by** *blast*
then show *?thesis* **using** *SFAccInit SFAccInitRed.hyps*(*2*) e' s' *field init* s_1a *sfsF shD′* **by** *auto*
next
fix *t sha ha la a*
assume s_1a: $s_1 = (ha,\ la,\ sha)$ **and** $e' = throw\ a$
and $P \vdash C\ has\ F,Static{:}t\ in\ D$ **and** $\forall sfs.\ sha\ D \neq \lfloor(sfs,\ Done)\rfloor$
and *init′*: $P \vdash \langle INIT\ D\ ([D],False) \leftarrow unit,(ha,\ la,\ sha)\rangle \Rightarrow \langle throw\ a,s'\rangle$
then have i: $i = Processing$ **using** *iDP shD* s_1 **by** *simp*
then show *?thesis* **using** *init′ init-ProcessingE* s_1 s_1a *shD* **by** *blast*
next
assume $\forall b\ t.\ \neg\ P \vdash C\ has\ F,b{:}t\ in\ D$
then show *?thesis* **using** *SFAccInitRed.hyps*(*1*) **by** *blast*
next
fix t **assume** *field*: $P \vdash C\ has\ F,NonStatic{:}t\ in\ D$
then show *?thesis* **using** *has-field-fun*[*OF SFAccInitRed.hyps*(*1*) *field*] **by** *simp*
qed
next
fix e **assume** e': $e' = throw\ e$
and *init*: $P \vdash \langle INIT\ D\ ([D],False) \leftarrow unit,(h,\ l,\ sh)\rangle \Rightarrow \langle throw\ e,s'\rangle$
obtain h' l' sh' **where** s': $s' = (h',l',sh')$ **by**(*cases* s')
then obtain *sfs i* **where** *shC*: $sh'\ D = \lfloor(sfs,\ i)\rfloor$ **and** *iDP*: $i = Error$
using *init-throw-PD*[*OF init*] **by** *auto*
then show *?thesis*
using *SFAccInitRed.hyps*(*1*) *SFAccInitRed.hyps*(*2*) *SFAccInitThrow* e' *init* **by** *auto*
qed

```
next
  case RedSFAccNone thus ?case
    by(fastforce elim: eval-cases intro: eval-evals.intros)
next
  case RedSFAccNonStatic thus ?case
    by(fastforce elim: eval-cases intro: eval-evals.intros)
next
  case (FAssRed1 e s b e1 s1 b1 F D e2)
  obtain h' l' sh' where s'=(h',l',sh') by(cases s')
  with FAssRed1 show ?case
    by(fastforce elim!: eval-cases(9)[where e1=e1] intro: eval-evals.intros simp: val-no-step
               intro!: FAss FAssNull FAssNone FAssStatic FAssThrow2)
next
  case FAssRed2
  obtain h' l' sh' where s'=(h',l',sh') by(cases s')
  with FAssRed2 show ?case
    by(auto elim!: eval-cases intro: eval-evals.intros
          intro!: FAss FAssNull FAssNone FAssStatic FAssThrow2 Val)
next
  case RedFAss
  thus ?case
    by (fastforce elim!: eval-cases intro: eval-evals.intros)
next
  case RedFAssNull
  thus ?case
    by (fastforce elim!: eval-cases intro: eval-evals.intros)
next
  case RedFAssNone
  then show ?case
    by(auto elim!: eval-cases intro: eval-evals.intros eval-finalId)
next
  case RedFAssStatic
  then show ?case
    by(auto elim!: eval-cases intro: eval-evals.intros eval-finalId)
next
  case (SFAssRed e s b e'' s'' b'' C F D)
  obtain h l sh where [simp]: s = (h,l,sh) by(cases s)
  obtain h' l' sh' where [simp]: s'=(h',l',sh') by(cases s')
  have val-of e = None using val-no-step SFAssRed.hyps(1) by(meson option.exhaust)
  then have bconf: P,sh ⊢b (e,b) √ using SFAssRed by simp
  show ?case using SFAssRed.prems(2) SFAssRed bconf
  proof cases
    case 2 with SFAssRed bconf show ?thesis by(auto intro!: SFAssInit)
  next
    case 3 with SFAssRed bconf show ?thesis by(auto intro!: SFAssInitThrow)
  qed(auto intro: eval-evals.intros intro!: SFAss SFAssInit SFAssNone SFAssNonStatic)
next
  case (RedSFAss C F t D sh sfs i sfs' v sh' h l)
  let ?sfs' = sfs(F ↦ v)
  have e':e' = unit and s':s' = (h, l, sh(D ↦ (?sfs', i)))
    using RedSFAss eval-cases(3) by fastforce+
  have i = Done ∨ i = Processing using RedSFAss by(clarsimp simp: bconf-def initPD-def)
  then show ?case
  proof(cases i)
```

case *Done* **then show** *?thesis* **using** *RedSFAss* e' s' *SFAss Val* **by** *auto*
next
case *Processing*
then have shC': $\nexists sfs.\ sh\ D = Some(sfs, Done)$ **and** *shP*: $sh\ D = Some(sfs, Processing)$
using *RedSFAss* **by** *simp+*
then have *init*: $P \vdash \langle INIT\ D\ ([D], False) \leftarrow unit, (h, l, sh)\rangle \Rightarrow \langle unit, (h, l, sh)\rangle$
by(*simp add*: *InitFinal InitProcessing Val*)
have $P \vdash \langle C{\bullet_s}F\{D\} := Val\ v, (h,\ l,\ sh)\rangle \Rightarrow \langle unit, (h, l, sh(D \mapsto (?sfs',\ i)))\rangle$
using *Processing* **by**(*auto intro*: *SFAssInit*[*OF Val RedSFAss.hyps*(*1*) shC' *init shP*])
then show *?thesis* **using** e' s' **by** *simp*
qed(*auto*)
next
case (*SFAssInitRed C F t D sh v h l*)
then have *seq*: $P \vdash \langle (INIT\ D\ ([D], False) \leftarrow unit);;\ C{\bullet_s}F\{D\} := Val\ v, (h,\ l,\ sh)\rangle \Rightarrow \langle e', s'\rangle$
using *eval-init-seq* **by** *simp*
then show *?case*
proof(*rule eval-cases*(*14*)) — Seq
fix v' s_1 **assume** *init*: $P \vdash \langle INIT\ D\ ([D], False) \leftarrow unit, (h,\ l,\ sh)\rangle \Rightarrow \langle Val\ v', s_1\rangle$
and *acc*: $P \vdash \langle C{\bullet_s}F\{D\} := Val\ v, s_1\rangle \Rightarrow \langle e', s'\rangle$
obtain h_1 l_1 sh_1 **where** s_1: $s_1 = (h_1, l_1, sh_1)$ **by**(*cases* s_1)
then obtain *sfs i* **where** *shD*: $sh_1\ D = \lfloor(sfs,\ i)\rfloor$ **and** *iDP*: $i = Done \vee i = Processing$
using *init-Val-PD*[*OF init*] **by** *auto*
show *?thesis*
proof(*rule eval-cases*(*10*)[*OF acc*]) — SFAss
fix *va* h_1 l_1 sh_1 *t sfs*
assume e': $e' = unit$ **and** s': $s' = (h_1,\ l_1,\ sh_1(D \mapsto (sfs(F \mapsto va),\ Done)))$
and *val*: $P \vdash \langle Val\ v, s_1\rangle \Rightarrow \langle Val\ va, (h_1,\ l_1,\ sh_1)\rangle$
and *field*: $P \vdash C\ has\ F, Static{:}t\ in\ D$ **and** shD': $sh_1\ D = \lfloor(sfs,\ Done)\rfloor$
have $v = va$ **and** $s_1 = (h_1,\ l_1,\ sh_1)$ **using** *eval-final-same*[*OF val*] **by** *auto*
then show *?thesis* **using** *SFAssInit field SFAssInitRed.hyps*(*2*) shD' e' s' *init val*
by (*metis eval-final eval-finalId*)
next
fix *va* h_1 l_1 sh_1 t v' h' l' sh' *sfs* i'
assume e': $e' = unit$ **and** s': $s' = (h',\ l',\ sh'(D \mapsto (sfs(F \mapsto va),\ i')))$
and *val*: $P \vdash \langle Val\ v, s_1\rangle \Rightarrow \langle Val\ va, (h_1,\ l_1,\ sh_1)\rangle$
and *field*: $P \vdash C\ has\ F, Static{:}t\ in\ D$ **and** *nDone*: $\forall sfs.\ sh_1\ D \neq \lfloor(sfs,\ Done)\rfloor$
and $init'$: $P \vdash \langle INIT\ D\ ([D], False) \leftarrow unit, (h_1,\ l_1,\ sh_1)\rangle \Rightarrow \langle Val\ v', (h',\ l',\ sh')\rangle$
and shD': $sh'\ D = \lfloor(sfs,\ i')\rfloor$
have v: $v = va$ **and** s_1a: $s_1 = (h_1,\ l_1,\ sh_1)$ **using** *eval-final-same*[*OF val*] **by** *auto*
then have i: $i = Processing$ **using** *iDP shD* s_1 *nDone* **by** *simp*
then have $(h_1,\ l_1,\ sh_1) = (h',\ l',\ sh')$ **using** $init'$ *init-ProcessingE* s_1 s_1a *shD* **by** *blast*
then show *?thesis* **using** *SFAssInit SFAssInitRed.hyps*(*2*) e' s' *field init* v s_1a shD' *val*
by (*metis eval-final eval-finalId*)
next
fix *va* h_1 l_1 sh_1 *t a*
assume $e' = throw\ a$ **and** *val*: $P \vdash \langle Val\ v, s_1\rangle \Rightarrow \langle Val\ va, (h_1,\ l_1,\ sh_1)\rangle$
and $P \vdash C\ has\ F, Static{:}t\ in\ D$ **and** *nDone*: $\forall sfs.\ sh_1\ D \neq \lfloor(sfs,\ Done)\rfloor$
and $init'$: $P \vdash \langle INIT\ D\ ([D], False) \leftarrow unit, (h_1,\ l_1,\ sh_1)\rangle \Rightarrow \langle throw\ a, s'\rangle$
have v: $v = va$ **and** s_1a: $s_1 = (h_1,\ l_1,\ sh_1)$ **using** *eval-final-same*[*OF val*] **by** *auto*
then have i: $i = Processing$ **using** *iDP shD* s_1 *nDone* **by** *simp*
then have $(h_1,\ l_1,\ sh_1) = s'$ **using** $init'$ *init-ProcessingE* s_1 s_1a *shD* **by** *blast*
then show *?thesis* **using** $init'$ *init-ProcessingE* s_1 s_1a *shD i* **by** *blast*
next
fix e'' **assume** *val*:$P \vdash \langle Val\ v, s_1\rangle \Rightarrow \langle throw\ e'', s'\rangle$

then show ?thesis **using** eval-final-same[OF val] **by** simp
next
assume ∀ b t. ¬ P ⊢ C has F,b:t in D
then show ?thesis **using** SFAssInitRed.hyps(1) **by** blast
next
fix t **assume** field: P ⊢ C has F,NonStatic:t in D
then show ?thesis **using** has-field-fun[OF SFAssInitRed.hyps(1) field] **by** simp
qed
next
fix e **assume** e′: e′ = throw e
and init: P ⊢ ⟨INIT D ([D],False) ← unit,(h, l, sh)⟩ ⇒ ⟨throw e,s′⟩
obtain h′ l′ sh′ **where** s′: s′ = (h′,l′,sh′) **by**(cases s′)
then obtain sfs i **where** shC: sh′ D = ⌊(sfs, i)⌋ **and** iDP: i = Error
using init-throw-PD[OF init] **by** auto
then show ?thesis **using** SFAssInitRed.hyps(1) SFAssInitRed.hyps(2) SFAssInitThrow Val
by (metis e′ init)
qed
next
case (RedSFAssNone C F D v s b) **then show** ?case
by(cases s) (auto elim!: eval-cases intro: eval-evals.intros eval-finalId)
next
case (RedSFAssNonStatic C F t D v s b) **then show** ?case
by(cases s) (auto elim!: eval-cases intro: eval-evals.intros eval-finalId)
next
case CallObj
note val-no-step[simp]
from CallObj.prems(2) CallObj **show** ?case
proof cases
case 2 **with** CallObj **show** ?thesis **by**(fastforce intro!: CallParamsThrow)
next
case 3 **with** CallObj **show** ?thesis **by**(fastforce intro!: CallNull)
next
case 4 **with** CallObj **show** ?thesis **by**(fastforce intro!: CallNone)
next
case 5 **with** CallObj **show** ?thesis **by**(fastforce intro!: CallStatic)
qed(fastforce intro!: CallObjThrow Call)+
next
case (CallParams es s b es′′ s′′ b′′ v M s′)
then obtain h′ l′ sh′ **where** s′ = (h′,l′,sh′) **by**(cases s′)
with CallParams **show** ?case
by(auto elim!: eval-cases intro!: CallNone eval-finalId CallStatic Val)
(auto intro!: CallParamsThrow CallNull Call Val)
next
case (RedCall h a C fs M Ts T pns body D vs l sh b)
have P ⊢ ⟨addr a,(h,l,sh)⟩ ⇒ ⟨addr a,(h,l,sh)⟩ **by** (rule eval-evals.intros)
moreover
have finals: finals(map Val vs) **by** simp
with finals **have** P ⊢ ⟨map Val vs,(h,l,sh)⟩ [⇒] ⟨map Val vs,(h,l,sh)⟩
by (iprover intro: eval-finalsId)
moreover have h a = Some (C, fs) **using** RedCall **by** simp
moreover have method: P ⊢ C sees M,NonStatic: Ts→T = (pns, body) in D **by** fact
moreover have same-len$_1$: length Ts = length pns
and this-distinct: this ∉ set pns **and** fv: fv (body) ⊆ {this} ∪ set pns
using method wf **by** (fastforce dest!:sees-wf-mdecl simp:wf-mdecl-def)+

have *same-len*: *length vs* = *length pns* **by** *fact*
moreover
obtain $l_2{}'$ **where** $l_2{}'$: $l_2{}' = [this{\mapsto}Addr\ a,pns[{\mapsto}]vs]$ **by** *simp*
moreover
obtain h_3 l_3 sh_3 **where** s': $s' = (h_3,l_3,sh_3)$ **by** (*cases* s')
have *eval-blocks*:
$P \vdash \langle(blocks\ (this\ \#\ pns,\ Class\ D\ \#\ Ts,\ Addr\ a\ \#\ vs,\ body)),(h,l,sh)\rangle \Rightarrow \langle e',s'\rangle$ **by** *fact*
hence *id*: $l_3 = l$ **using** *fv* s' $same\text{-}len_1$ *same-len*
by(*fastforce elim*: *eval-closed-lcl-unchanged*)
from *eval-blocks* **obtain** $l_3{}'$ **where** $P \vdash \langle body,(h,l_2{}',sh)\rangle \Rightarrow \langle e',(h_3,l_3{}',sh_3)\rangle$
proof −
from $same\text{-}len_1$ **have** $length(this\#pns) = length(Class\ D\#Ts)$ **by** *simp*
moreover from $same\text{-}len_1$ *same-len*
have $same\text{-}len_2$: $length\ (this\#pns) = length\ (Addr\ a\#vs)$ **by** *simp*
moreover from *eval-blocks*
have $P \vdash \langle blocks\ (this\#pns,Class\ D\#Ts,Addr\ a\#vs,body),(h,l,sh)\rangle$
$\Rightarrow\langle e',(h_3,l_3,sh_3)\rangle$ **using** s' $same\text{-}len_1$ $same\text{-}len_2$ **by** *simp*
ultimately obtain l''
where $P \vdash \langle body,(h,l(this\ \#\ pns[{\mapsto}]Addr\ a\ \#\ vs),sh)\rangle{\Rightarrow}\langle e',(h_3,\ l'',sh_3)\rangle$
by (*blast dest*:*blocksEval*)
from *eval-restrict-lcl*[*OF wf this fv*] *this-distinct* $same\text{-}len_1$ *same-len*
have $P \vdash \langle body,(h,[this\ \#\ pns[{\mapsto}]Addr\ a\ \#\ vs],sh)\rangle \Rightarrow$
$\langle e',(h_3,\ l''|`(set(this\#pns)),sh_3)\rangle$ **using** *wf method*
by(*simp add*:*subset-insert-iff insert-Diff-if*)
thus *?thesis* **by**(*fastforce intro*!:*that simp add*: $l_2{}'$)
qed
ultimately
have $P \vdash \langle(addr\ a){\cdot}M(map\ Val\ vs),(h,l,sh)\rangle \Rightarrow \langle e',(h_3,l,sh_3)\rangle$ **by** (*rule Call*)
with s' *id* **show** *?case* **by** *simp*
next
case *RedCallNull*
thus *?case*
by (*fastforce elim*: *eval-cases intro*: *eval-evals.intros eval-finalsId*)
next
case (*RedCallNone h a C fs M vs l sh b*)
then have *tes*: *THROW NoSuchMethodError* = $e' \wedge (h,l,sh) = s'$
using *eval-final-same* **by** *simp*
have $P \vdash \langle addr\ a,(h,l,sh)\rangle \Rightarrow \langle addr\ a,(h,l,sh)\rangle$ **and** $P \vdash \langle map\ Val\ vs,(h,l,sh)\rangle\ [\Rightarrow]\ \langle map\ Val\ vs,(h,l,sh)\rangle$
using *eval-finalId eval-finalsId* **by** *auto*
then show *?case* **using** *RedCallNone CallNone tes* **by** *auto*
next
case (*RedCallStatic h a C fs M Ts T m D vs l sh b*)
then have *tes*: *THROW IncompatibleClassChangeError* = $e' \wedge (h,l,sh) = s'$
using *eval-final-same* **by** *simp*
have $P \vdash \langle addr\ a,(h,l,sh)\rangle \Rightarrow \langle addr\ a,(h,l,sh)\rangle$ **and** $P \vdash \langle map\ Val\ vs,(h,l,sh)\rangle\ [\Rightarrow]\ \langle map\ Val\ vs,(h,l,sh)\rangle$
using *eval-finalId eval-finalsId* **by** *auto*
then show *?case* **using** *RedCallStatic CallStatic tes* **by** *fastforce*
next
case (*SCallParams es s b es'' s'' b' C M s'*)
obtain h' l' sh' **where** s'[*simp*]: $s' = (h',l',sh')$ **by**(*cases* s')
obtain *h l sh* **where** *s*[*simp*]: $s = (h,l,sh)$ **by**(*cases s*)
have *es*: *map-vals-of es* = *None* **using** *vals-no-step SCallParams.hyps*(*1*) **by** (*meson not-Some-eq*)

have *bconf*: $P,sh \vdash_b (es,b) \surd$ **using** *s SCallParams.prems*(*1*) **by** (*simp add*: *bconf-SCall*[*OF es*])
from *SCallParams.prems*(*2*) *SCallParams bconf* **show** *?case*
proof *cases*
  case *2* **with** *SCallParams bconf* **show** *?thesis* **by**(*auto intro*!: *SCallNone*)
next
  case *3* **with** *SCallParams bconf* **show** *?thesis* **by**(*auto intro*!: *SCallNonStatic*)
next
  case *4* **with** *SCallParams bconf* **show** *?thesis* **by**(*auto intro*!: *SCallInitThrow*)
next
  case *5* **with** *SCallParams bconf* **show** *?thesis* **by**(*auto intro*!: *SCallInit*)
qed(*auto intro*!: *SCallParamsThrow SCall*)
next
case (*RedSCall C M Ts T pns body D vs s*)
then obtain *h l sh* **where** *s*: $s = (h,l,sh)$ **by**(*cases s*)
then obtain *sfs i* **where** *shC*: $sh\ D = \lfloor(sfs,\ i)\rfloor$ **and** $i = Done \vee i = Processing$
 using *RedSCall* **by**(*auto simp*: *bconf-def initPD-def dest*: *sees-method-fun*)
have *finals*: *finals*(*map Val vs*) **by** *simp*
with *finals* **have** *mVs*: $P \vdash \langle map\ Val\ vs,(h,l,sh)\rangle\ [\Rightarrow]\ \langle map\ Val\ vs,(h,l,sh)\rangle$
  by (*iprover intro*: *eval-finalsId*)
obtain *sfs i* **where** *shC*: $sh\ D = \lfloor(sfs,\ i)\rfloor$
 using *RedSCall s* **by**(*auto simp*: *bconf-def initPD-def dest*: *sees-method-fun*)
then have *iDP*: $i = Done \vee i = Processing$ **using** *RedSCall s*
  by (*auto simp*: *bconf-def initPD-def dest*: *sees-method-fun*[*OF RedSCall.hyps*(*1*)])
have *method*: $P \vdash C$ *sees* M,*Static*: $Ts \rightarrow T = (pns,\ body)$ *in* D **by** *fact*
have *same-len*$_1$: *length* $Ts =$ *length pns* **and** *fv*: *fv* (*body*) $\subseteq$ *set pns*
  using *method wf* **by** (*fastforce dest*!:*sees-wf-mdecl simp*:*wf-mdecl-def*)+
have *same-len*: *length* $vs =$ *length pns* **by** *fact*
obtain $l_2{}'$ **where** $l_2{}'$: $l_2{}' = [pns[\mapsto]vs]$ **by** *simp*
obtain $h_3\ l_3\ sh_3$ **where** s': $s' = (h_3,l_3,sh_3)$ **by** (*cases* s')
have *eval-blocks*:
  $P \vdash \langle(blocks\ (pns,\ Ts,\ vs,\ body)),(h,l,sh)\rangle \Rightarrow \langle e',s'\rangle$ **using** *RedSCall.prems*(*2*) *s* **by** *simp*
hence *id*: $l_3 = l$ **using** *fv* s' *same-len*$_1$ *same-len*
  by(*fastforce elim*: *eval-closed-lcl-unchanged*)
from *eval-blocks* **obtain** $l_3{}'$ **where** *body*: $P \vdash \langle body,(h,l_2{}',sh)\rangle \Rightarrow \langle e',(h_3,l_3{}',sh_3)\rangle$
proof −
  from *eval-blocks*
  have $P \vdash \langle blocks\ (pns,Ts,vs,body),(h,l,sh)\rangle$
      $\Rightarrow\langle e',(h_3,l_3,sh_3)\rangle$ **using** s' *same-len same-len*$_1$ **by** *simp*
  then obtain l''
    where $P \vdash \langle body,(h,l(pns[\mapsto]vs),sh)\rangle \Rightarrow \langle e',(h_3,\ l'',sh_3)\rangle$
    by (*blast dest*:*blocksEval*[*OF same-len*$_1$[*THEN sym*] *same-len*[*THEN sym*]])
  from *eval-restrict-lcl*[*OF wf this fv*] *same-len*$_1$ *same-len*
  have $P \vdash \langle body,(h,[pns[\mapsto]vs],sh)\rangle \Rightarrow \langle e',(h_3,\ l''|`(set(pns)),sh_3)\rangle$ **using** *wf method*
    by(*simp add*:*subset-insert-iff insert-Diff-if*)
  thus *?thesis* **by**(*fastforce intro*!:*that simp add*: $l_2{}'$)
qed
show *?case* **using** *iDP*
proof(*cases i*)
  case *Done*
  then have shC': $sh\ D = \lfloor(sfs,\ Done)\rfloor \vee M = clinit \wedge sh\ D = \lfloor(sfs,\ Processing)\rfloor$
    using *shC* **by** *simp*
  have $P \vdash \langle C\bullet_s M(map\ Val\ vs),(h,l,sh)\rangle \Rightarrow \langle e',(h_3,l,sh_3)\rangle$
   by (*rule SCall*[*OF mVs method* shC' *same-len* $l_2{}'$ *body*])
  with *s* s' *id* **show** *?thesis* **by** *simp*

next
case *Processing*
then have shC': $\nexists sfs.\ sh\ D = Some(sfs,Done)$ **and** shP: $sh\ D = Some(sfs,Processing)$
using shC **by** $simp+$
then have $init$: $P \vdash \langle INIT\ D\ ([D],False) \leftarrow unit,(h,l,sh)\rangle \Rightarrow \langle unit,(h,l,sh)\rangle$
by(*simp add*: *InitFinal InitProcessing Val*)
have $P \vdash \langle C\bullet_s M(map\ Val\ vs),(h,l,sh)\rangle \Rightarrow \langle e',(h_3,l,sh_3)\rangle$
proof(*cases* $M = clinit$)
case *False* **show** *?thesis* **by**(*rule SCallInit*[*OF mVs method shC' False init same-len* l_2' *body*])
next
case *True*
then have shC': $sh\ D = \lfloor(sfs,\ Done)\rfloor \vee M = clinit \wedge sh\ D = \lfloor(sfs,\ Processing)\rfloor$
using *shC Processing* **by** *simp*
have $P \vdash \langle C\bullet_s M(map\ Val\ vs),(h,l,sh)\rangle \Rightarrow \langle e',(h_3,l,sh_3)\rangle$
by (*rule SCall*[*OF mVs method shC' same-len* l_2' *body*])
with $s\ s'\ id$ **show** *?thesis* **by** *simp*
qed
with $s\ s'\ id$ **show** *?thesis* **by** *simp*
qed(*auto*)
next
case (*SCallInitRed C M Ts T pns body D sh vs h l*)
then have *seq*: $P \vdash \langle(INIT\ D\ ([D],False) \leftarrow unit);;\ C\bullet_s M(map\ Val\ vs),(h,\ l,\ sh)\rangle \Rightarrow \langle e',s'\rangle$
using *eval-init-seq* **by** *simp*
then show *?case*
proof(*rule eval-cases*(*14*)) — Seq
fix $v'\ s_1$ **assume** *init*: $P \vdash \langle INIT\ D\ ([D],False) \leftarrow unit,(h,\ l,\ sh)\rangle \Rightarrow \langle Val\ v',s_1\rangle$
and *call*: $P \vdash \langle C\bullet_s M(map\ Val\ vs),s_1\rangle \Rightarrow \langle e',s'\rangle$
obtain $h_1\ l_1\ sh_1$ **where** s_1: $s_1 = (h_1,l_1,sh_1)$ **by**(*cases* s_1)
then obtain *sfs i* **where** *shD*: $sh_1\ D = \lfloor(sfs,\ i)\rfloor$ **and** *iDP*: $i = Done \vee i = Processing$
using *init-Val-PD*[*OF init*] **by** *auto*
show *?thesis*
proof(*rule eval-cases*(*12*)[*OF call*]) — SCall
fix *vsa ex es'* **assume** $P \vdash \langle map\ Val\ vs,s_1\rangle\ [\Rightarrow]\ \langle map\ Val\ vsa\ @\ throw\ ex\ \#\ es',s'\rangle$
then show *?thesis* **using** *evals-finals-same* **by** (*meson finals-def map-Val-nthrow-neq*)
next
assume $\forall b\ Ts\ T\ a\ ba\ x.\ \neg\ P \vdash C\ sees\ M,\ b:\ Ts \rightarrow T = (a,\ ba)\ in\ x$
then show *?thesis* **using** *SCallInitRed.hyps*(*1*) **by** *auto*
next
fix *Ts T m D* **assume** $P \vdash C\ sees\ M,\ NonStatic:\ Ts \rightarrow T = m\ in\ D$
then show *?thesis* **using** *sees-method-fun*[*OF SCallInitRed.hyps*(*1*)] **by** *blast*
next
fix *vsa h1 l1 sh1 Ts T pns body D' a*
assume $e' = throw\ a$ **and** *vals*: $P \vdash \langle map\ Val\ vs,s_1\rangle\ [\Rightarrow]\ \langle map\ Val\ vsa,(h1,\ l1,\ sh1)\rangle$
and *method*: $P \vdash C\ sees\ M,\ Static:\ Ts \rightarrow T = (pns,\ body)\ in\ D'$
and *nDone*: $\forall sfs.\ sh1\ D' \neq \lfloor(sfs,\ Done)\rfloor$
and *init'*: $P \vdash \langle INIT\ D'\ ([D'],False) \leftarrow unit,(h1,\ l1,\ sh1)\rangle \Rightarrow \langle throw\ a,s'\rangle$
have *vs*: $vs = vsa$ **and** s_1a: $s_1 = (h1,\ l1,\ sh1)$
using *evals-finals-same*[*OF - vals*] *map-Val-eq* **by** *auto*
have *D*: $D = D'$ **using** *sees-method-fun*[*OF SCallInitRed.hyps*(*1*) *method*] **by** *simp*
then have *i*: $i = Processing$ **using** *iDP shD* s_1 s_1a *nDone* **by** *simp*
then show *?thesis* **using** *D init' init-ProcessingE* s_1 s_1a *shD* **by** *blast*
next
fix *vsa h1 l1 sh1 Ts T pns' body' D' v'* $h_2\ l_2\ sh_2\ h_3\ l_3\ sh_3$
assume s': $s' = (h_3,\ l_2,\ sh_3)$

and *vals*: $P \vdash \langle map\ Val\ vs,s_1\rangle\ [\Rightarrow]\ \langle map\ Val\ vsa,(h1,\ l1,\ sh1)\rangle$
and *method*: $P \vdash C\ sees\ M,\ Static :\ Ts \rightarrow T = (pns',\ body')\ in\ D'$
and *nDone*: $\forall sfs.\ sh1\ D' \neq \lfloor(sfs,\ Done)\rfloor$
and *init'*: $P \vdash \langle INIT\ D'\ ([D'],False) \leftarrow unit,(h1,\ l1,\ sh1)\rangle \Rightarrow \langle Val\ v',(h_2,\ l_2,\ sh_2)\rangle$
and *len*: $length\ vsa = length\ pns'$
and *bstep*: $P \vdash \langle body',(h_2,\ [pns'\ [\mapsto]\ vsa],\ sh_2)\rangle \Rightarrow \langle e',(h_3,\ l_3,\ sh_3)\rangle$
have *vs*: $vs = vsa$ **and** *s_1a*: $s_1 = (h1,\ l1,\ sh1)$
using *evals-finals-same*[*OF - vals*] *map-Val-eq* **by** *auto*
have *D*: $D = D'$ **and** *pns*: $pns = pns'$ **and** *body*: $body = body'$
using *sees-method-fun*[*OF SCallInitRed.hyps*(*1*) *method*] **by** *auto*
then have *i*: $i = Processing$ **using** *iDP shD* s_1 s_1a *nDone* **by** *simp*
then have *s2*: $(h_2,\ l_2,\ sh_2) = s_1$ **using** *D init' init-ProcessingE* s_1 s_1a *shD* **by** *blast*
then show *?thesis*
using *eval-finalId SCallInit*[*OF eval-finalsId*[*of map Val vs P* (*h*,*l*,*sh*)]
SCallInitRed.hyps(*1*)] *init init' len bstep nDone D pns body s'* s_1 s_1a *shD vals vs*
SCallInitRed.hyps(*2*−*3*) *s2* **by** *auto*
next
fix *vsa* h_2 l_2 sh_2 *Ts T pns' body' D' sfs* h_3 l_3 sh_3
assume *s'*: $s' = (h_3,\ l_2,\ sh_3)$ **and** *vals*: $P \vdash \langle map\ Val\ vs,s_1\rangle\ [\Rightarrow]\ \langle map\ Val\ vsa,(h_2,\ l_2,\ sh_2)\rangle$
and *method*: $P \vdash C\ sees\ M,\ Static :\ Ts \rightarrow T = (pns',\ body')\ in\ D'$
and $sh_2\ D' = \lfloor(sfs,\ Done)\rfloor \vee M = clinit \wedge sh_2\ D' = \lfloor(sfs,\ Processing)\rfloor$
and *len*: $length\ vsa = length\ pns'$
and *bstep*: $P \vdash \langle body',(h_2,\ [pns'\ [\mapsto]\ vsa],\ sh_2)\rangle \Rightarrow \langle e',(h_3,\ l_3,\ sh_3)\rangle$
have *vs*: $vs = vsa$ **and** *s_1a*: $s_1 = (h_2,\ l_2,\ sh_2)$
using *evals-finals-same*[*OF - vals*] *map-Val-eq* **by** *auto*
have *D*: $D = D'$ **and** *pns*: $pns = pns'$ **and** *body*: $body = body'$
using *sees-method-fun*[*OF SCallInitRed.hyps*(*1*) *method*] **by** *auto*
then show *?thesis* **using** *SCallInit*[*OF eval-finalsId*[*of map Val vs P* (*h*,*l*,*sh*)]
SCallInitRed.hyps(*1*)] *bstep SCallInitRed.hyps*(*2*−*3*) *len s'* s_1a *vals vs init* **by** *auto*
qed
next
fix *e* **assume** *e'*: $e' = throw\ e$
and *init*: $P \vdash \langle INIT\ D\ ([D],False) \leftarrow unit,(h,\ l,\ sh)\rangle \Rightarrow \langle throw\ e,s'\rangle$
obtain *h' l' sh'* **where** *s'*: $s' = (h',l',sh')$ **by**(*cases s'*)
then obtain *sfs i* **where** *shC*: $sh'\ D = \lfloor(sfs,\ i)\rfloor$ **and** *iDP*: $i = Error$
using *init-throw-PD*[*OF init*] **by** *auto*
then show *?thesis* **using** *SCallInitRed.hyps*(*2*−*3*) *init e'*
SCallInitThrow[*OF eval-finalsId*[*of map Val vs -*] *SCallInitRed.hyps*(*1*)]
by *auto*
qed
next
case (*RedSCallNone C M vs s b*)
then have *tes*: $THROW\ NoSuchMethodError = e' \wedge s = s'$
using *eval-final-same* **by** *simp*
have $P \vdash \langle map\ Val\ vs,s\rangle\ [\Rightarrow]\ \langle map\ Val\ vs,s\rangle$ **using** *eval-finalsId* **by** *simp*
then show *?case* **using** *RedSCallNone eval-evals.SCallNone tes* **by** *auto*
next
case (*RedSCallNonStatic C M Ts T m D vs s b*)
then have *tes*: $THROW\ IncompatibleClassChangeError = e' \wedge s = s'$
using *eval-final-same* **by** *simp*
have $P \vdash \langle map\ Val\ vs,s\rangle\ [\Rightarrow]\ \langle map\ Val\ vs,s\rangle$ **using** *eval-finalsId* **by** *simp*
then show *?case* **using** *RedSCallNonStatic eval-evals.SCallNonStatic tes* **by** *auto*
next
case *InitBlockRed*

thus *?case*
  by (*fastforce elim*!: *eval-cases intro*: *eval-evals.intros eval-finalId*
        simp: *assigned-def map-upd-triv fun-upd-same*)
next
  case (*RedInitBlock V T v u s b*)
  then have $P \vdash \langle Val\ u,s\rangle \Rightarrow \langle e',s'\rangle$ **by** *simp*
  then obtain s': $s'=s$ **and** e': $e'=Val\ u$ **by** *cases simp*
  obtain *h l sh* **where** *s*: $s=(h,l,sh)$ **by** (*cases s*)
  have $P \vdash \langle\{V{:}T := Val\ v;\ Val\ u\},(h,l,sh)\rangle \Rightarrow \langle Val\ u,(h,\ (l(V\mapsto v))(V{:=}l\ V),\ sh)\rangle$
    by (*fastforce intro*!: *eval-evals.intros*)
  then have $P \vdash \langle\{V{:}T :=\ Val\ v;\ Val\ u\},s\rangle \Rightarrow \langle e',s'\rangle$ **using** $s\ s'\ e'$ **by** *simp*
  then show *?case* **by** *simp*
next
  case *BlockRedNone*
  thus *?case*
    by (*fastforce elim*!: *eval-cases intro*: *eval-evals.intros*
        simp add: *fun-upd-same fun-upd-idem*)
next
  case *BlockRedSome*
  thus *?case*
    by (*fastforce elim*!: *eval-cases intro*: *eval-evals.intros*
        simp add: *fun-upd-same fun-upd-idem*)
next
 case (*RedBlock V T v s b*)
 then have $P \vdash \langle Val\ v,s\rangle \Rightarrow \langle e',s'\rangle$ **by** *simp*
 then obtain s': $s'=s$ **and** e': $e'=Val\ v$
    by *cases simp*
  obtain *h l sh* **where** *s*: $s=(h,l,sh)$ **by** (*cases s*)
 have $P \vdash \langle Val\ v,(h,l(V{:=}None),sh)\rangle \Rightarrow \langle Val\ v,(h,l(V{:=}None),sh)\rangle$
   by (*rule eval-evals.intros*)
 hence $P \vdash \langle\{V{:}T;Val\ v\},(h,l,sh)\rangle \Rightarrow \langle Val\ v,(h,(l(V{:=}None))(V{:=}l\ V),sh)\rangle$
   by (*rule eval-evals.Block*)
 then have $P \vdash \langle\{V{:}T;\ Val\ v\},s\rangle \Rightarrow \langle e',s'\rangle$ **using** $s\ s'\ e'$ **by** *simp*
 then show *?case* **by** *simp*
next
  case (*SeqRed e s b e1 s1 b1* e_2) **show** *?case*
  proof(*cases val-of e*)
    case *None* **show** *?thesis*
    proof(*cases lass-val-of e*)
      case *lNone*:*None*
      then have *bconf*: $P,shp\ s \vdash_b (e,b)\ \surd$ **using** *SeqRed.prems*(*1*) *None* **by** *simp*
      then show *?thesis* **using** *SeqRed* **using** *seq-ext* **by** *fastforce*
    next
      case (*Some p*)
      obtain $V'\ v'$ **where** *p*: $p = (V',v')$ **and** *e*: $e = V'{:=}Val\ v'$
        using *lass-val-of-spec*[*OF Some*] **by**(*cases p*, *auto*)
      obtain $h\ l\ sh\ h'\ l'\ sh'$ **where** *s*: $s = (h,l,sh)$ **and** *s1*: $s1 = (h',l',sh')$ **by**(*cases s*, *cases s1*)
      then have *red*: $P \vdash \langle e,(h,l,sh),b\rangle \to \langle e1,(h',l',sh'),b1\rangle$ **using** *SeqRed.hyps*(*1*) **by** *simp*
      then have s_1': $e1 = unit \wedge h' = h \wedge l' = l(V' \mapsto v') \wedge sh' = sh$
        using *lass-val-of-red*[*OF Some red*] *p e* **by** *simp*
      then have *eval*: $P \vdash \langle e,s\rangle \Rightarrow \langle e1,s1\rangle$ **using** *e s s1 LAss Val* **by** *auto*
      then show *?thesis*
        by (*metis SeqRed.prems*(*2*) *eval-final eval-final-same seq-ext*)
    qed

next
case (*Some a*) **then show** *?thesis* **using** *SeqRed.hyps*(*1*) *val-no-step* **by** *blast*
qed
next
case *RedSeq*
thus *?case*
by (*fastforce elim*: *eval-cases intro*: *eval-evals.intros*)
next
case *CondRed*
thus *?case*
by (*fastforce elim*: *eval-cases intro*: *eval-evals.intros simp*: *val-no-step*)
next
case *RedCondT*
thus *?case*
by (*fastforce elim*: *eval-cases intro*: *eval-evals.intros*)
next
case *RedCondF*
thus *?case*
by (*fastforce elim*: *eval-cases intro*: *eval-evals.intros*)
next
case *RedWhile*
thus *?case*
by (*auto simp add*: *unfold-while intro*:*eval-evals.intros elim*:*eval-cases*)
next
case *ThrowRed* **then show** *?case* **by**(*fastforce elim*: *eval-cases simp*: *eval-evals.intros*)
next
case *RedThrowNull*
thus *?case*
by (*fastforce elim*: *eval-cases intro*: *eval-evals.intros*)
next
case *TryRed* **thus** *?case*
by(*fastforce elim*: *eval-cases intro*: *eval-evals.intros*)
next
case *RedTryCatch*
thus *?case*
by (*fastforce elim*: *eval-cases intro*: *eval-evals.intros*)
next
case (*RedTryFail s a D fs C V* e_2 *b*)
thus *?case*
by (*cases s*)(*auto elim*!: *eval-cases intro*: *eval-evals.intros*)
next
case *ListRed1*
thus *?case*
by (*fastforce elim*: *evals-cases intro*: *eval-evals.intros simp*: *val-no-step*)
next
case *ListRed2*
thus *?case*
by (*fastforce elim*!: *evals-cases eval-cases*
intro: *eval-evals.intros eval-finalId*)
next
case (*RedInit e1 C b s1 b'*) **then show** *?case* **using** *InitFinal* **by** *simp*
next
case (*InitNoneRed sh C C' Cs e h l b*)
show *?case* **using** *InitNone InitNoneRed.hyps InitNoneRed.prems*(*2*) **by** *auto*

next
  case (*RedInitDone sh C sfs C′ Cs e h l b*)
  show *?case* **using** *InitDone RedInitDone.hyps RedInitDone.prems*(*2*) **by** *auto*
next
  case (*RedInitProcessing sh C sfs C′ Cs e h l b*)
  show *?case* **using** *InitProcessing RedInitProcessing.hyps RedInitProcessing.prems*(*2*) **by** *auto*
next
  case (*RedInitError sh C sfs C′ Cs e h l b*)
  show *?case* **using** *InitError RedInitError.hyps RedInitError.prems*(*2*) **by** *auto*
next
  case (*InitObjectRed sh C sfs sh′ C′ Cs e h l b*) **show** *?case* **using** *InitObject InitObjectRed* **by** *auto*
next
  case (*InitNonObjectSuperRed sh C sfs D r sh′ C′ Cs e h l b*)
  show *?case* **using** *InitNonObject InitNonObjectSuperRed* **by** *auto*
next
  case (*RedInitRInit C′ C Cs e h l sh b*)
  show *?case* **using** *InitRInit RedInitRInit* **by** *auto*
next
  case (*RInitRed e s b e″ s″ b″ C Cs* e_0)
  then have *IH*: $\bigwedge e'\ s'.\ P \vdash \langle e'',s'' \rangle \Rightarrow \langle e',s' \rangle \Longrightarrow P \vdash \langle e,s \rangle \Rightarrow \langle e',s' \rangle$ **by** *simp*
  show *?case* **using** *RInitRed rinit-ext*[*OF IH*] **by** *simp*
next
  case (*RedRInit sh C sfs i sh′ C′ Cs v e h l b s′ e′*)
  then have *init*: $P \vdash \langle (INIT\ C'\ (Cs,True) \leftarrow e),\ (h,\ l,\ sh(C \mapsto (sfs,\ Done))) \rangle \Rightarrow \langle e',s' \rangle$
    using *RedRInit* **by** *simp*
  then show *?case* **using** *RInit RedRInit.hyps*(*1*) *RedRInit.hyps*(*3*) *Val* **by** *fastforce*
next
  case *BinOpThrow2*
  thus *?case*
    by (*fastforce elim*: *eval-cases intro*: *eval-evals.intros*)
next
  case *FAssThrow2*
  thus *?case*
    by (*fastforce elim*: *eval-cases intro*: *eval-evals.intros*)
next
  case *SFAssThrow*
  then show *?case*
    by (*fastforce elim*: *eval-cases intro*: *eval-evals.intros*)
next
  case (*CallThrowParams es vs e es′ v M s b*)
  have *val*: $P \vdash \langle Val\ v,s \rangle \Rightarrow \langle Val\ v,s \rangle$ **by** (*rule eval-evals.intros*)
  have *eval-e*: $P \vdash \langle throw\ (e),s \rangle \Rightarrow \langle e',s' \rangle$ **using** *CallThrowParams* **by** *simp*
  then obtain *xa* **where** *e′*: $e' = Throw\ xa$ **by** (*cases*) (*auto dest*!: *eval-final*)
  with *list-eval-Throw* [*OF eval-e*]
  have *vals*: $P \vdash \langle es,s \rangle\ [\Rightarrow]\ \langle map\ Val\ vs\ @\ Throw\ xa\ \#\ es',s' \rangle$
    using *CallThrowParams.hyps*(*1*) *eval-e list-eval-Throw* **by** *blast*
  then have $P \vdash \langle Val\ v{\cdot}M(es),s \rangle \Rightarrow \langle Throw\ xa,s' \rangle$
   using *eval-evals.CallParamsThrow*[*OF val vals*] **by** *simp*
  thus *?case* **using** *e′* **by** *simp*
next
  case (*SCallThrowParams es vs e es′ C M s b*)
  have *eval-e*: $P \vdash \langle throw\ (e),s \rangle \Rightarrow \langle e',s' \rangle$ **using** *SCallThrowParams* **by** *simp*
  then obtain *xa* **where** *e′*: $e' = Throw\ xa$ **by** (*cases*) (*auto dest*!: *eval-final*)
  then have $P \vdash \langle es,s \rangle\ [\Rightarrow]\ \langle map\ Val\ vs\ @\ Throw\ xa\ \#\ es',s' \rangle$

using *SCallThrowParams.hyps*(*1*) *eval-e list-eval-Throw* **by** *blast*
then have $P \vdash \langle C{\bullet}_s M(es),s\rangle \Rightarrow \langle Throw\ xa,s'\rangle$
by (*rule eval-evals.SCallParamsThrow*)
thus *?case* **using** e' **by** *simp*
next
case (*BlockThrow V T a s b*)
then have $P \vdash \langle Throw\ a,\ s\rangle \Rightarrow \langle e',s'\rangle$ **by** *simp*
then obtain s': $s' = s$ **and** e': $e' = Throw\ a$
by *cases* (*auto elim*!:*eval-cases*)
obtain *h l sh* **where** *s*: $s{=}(h,l,sh)$ **by** (*cases s*)
have $P \vdash \langle Throw\ a,\ (h,l(V{:=}None),sh)\rangle \Rightarrow \langle Throw\ a,\ (h,l(V{:=}None),sh)\rangle$
by (*simp add*:*eval-evals.intros eval-finalId*)
hence $P\vdash\langle\{V{:}T;Throw\ a\},(h,l,sh)\rangle \Rightarrow \langle Throw\ a,\ (h,(l(V{:=}None))(V{:=}l\ V),sh)\rangle$
by (*rule eval-evals.Block*)
then have $P \vdash \langle\{V{:}T;\ Throw\ a\},s\rangle \Rightarrow \langle e',s'\rangle$ **using** $s\ s'\ e'$ **by** *simp*
then show *?case* **by** *simp*
next
case (*InitBlockThrow V T v a s b*)
then have $P \vdash \langle Throw\ a,s\rangle \Rightarrow \langle e',s'\rangle$ **by** *simp*
then obtain s': $s' = s$ **and** e': $e' = Throw\ a$
by *cases* (*auto elim*!:*eval-cases*)
obtain *h l sh* **where** *s*: $s = (h,l,sh)$ **by** (*cases s*)
have $P \vdash \langle\{V{:}T := Val\ v;\ Throw\ a\},(h,l,sh)\rangle \Rightarrow \langle Throw\ a,\ (h,\ (l(V{\mapsto}v))(V{:=}l\ V),sh)\rangle$
by(*fastforce intro*:*eval-evals.intros*)
then have $P \vdash \langle\{V{:}T := Val\ v;\ Throw\ a\},s\rangle \Rightarrow \langle e',s'\rangle$ **using** $s\ s'\ e'$ **by** *simp*
then show *?case* **by** *simp*
next
case (*RInitInitThrow sh C sfs i sh' a D Cs e h l b*)
have *IH*: $\bigwedge e'\ s'.\ P \vdash \langle RI\ (D,Throw\ a)\ ;\ Cs \leftarrow e,(h,\ l,\ sh(C \mapsto (sfs,\ Error)))\rangle \Rightarrow \langle e',s'\rangle \Longrightarrow$
$P \vdash \langle RI\ (C,Throw\ a)\ ;\ D\ \#\ Cs \leftarrow e,(h,\ l,\ sh)\rangle \Rightarrow \langle e',s'\rangle$
using *RInitInitFail*[*OF eval-finalId*] *RInitInitThrow* **by** *simp*
then show *?case* **using** *RInitInitThrow.hyps*(*2*) *RInitInitThrow.prems*(*2*) **by** *auto*
next
case (*RInitThrow sh C sfs i sh' a e h l b*)
then have e': $e' = Throw\ a$ **and** s': $s' = (h,l,sh')$
using *eval-final-same final-def* **by** *fastforce+*
show *?case* **using** *RInitFailFinal RInitThrow.hyps*(*1*) *RInitThrow.hyps*(*2*) e' *eval-finalId* s' **by** *auto*
qed(*auto elim*: *eval-cases simp*: *eval-evals.intros*)

Its extension to $\to*$:

lemma *extend-eval*:
assumes *wf*: *wwf-J-prog P*
shows $[\![\ P \vdash \langle e,s,b\rangle \to* \langle e'',s'',b''\rangle;\ P \vdash \langle e'',s''\rangle \Rightarrow \langle e',s'\rangle;$
$iconf\ (shp\ s)\ e;\ P,shp\ s \vdash_b (e{::}expr,b)\ \surd\ ]\!]$
$\Longrightarrow P \vdash \langle e,s\rangle \Rightarrow \langle e',s'\rangle$

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

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

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

1.23.4 Equivalence

And now, the crowning achievement:

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

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** $P,E \vdash s\sqrt{}$
shows $P,E \vdash s'\sqrt{}$

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** $P,E \vdash s\sqrt{}$ **and** *iconf (shp s) e* **and** $P,E \vdash e::T$
shows $\exists\, T'.\ P \vdash T' \le T \wedge P,E,hp\ s',shp\ s' \vdash e':T'$

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: $P,E \vdash new\ C \leadsto new\ C$
| *AnnoCast*: $P,E \vdash e \leadsto e' \Longrightarrow P,E \vdash Cast\ C\ e \leadsto Cast\ C\ e'$
| *AnnoVal*: $P,E \vdash Val\ v \leadsto Val\ v$
| *AnnoVarVar*: $E\ V = \lfloor T\rfloor \Longrightarrow P,E \vdash Var\ V \leadsto Var\ V$
| *AnnoVarField*: $[\![\ E\ V = None;\ E\ this = \lfloor Class\ C\rfloor;\ P \vdash C\ sees\ V,NonStatic{:}T\ in\ D\ ]\!]$
$\Longrightarrow P,E \vdash Var\ V \leadsto Var\ this\cdot V\{D\}$

$|$ *AnnoBinOp*:
$[\![\ P,E \vdash e1 \leadsto e1';\ \ P,E \vdash e2 \leadsto e2'\ ]\!]$
$\Longrightarrow P,E \vdash e1\ «bop»\ e2 \leadsto e1'\ «bop»\ e2'$
$|$ *AnnoLAssVar*:
$[\![\ E\ V = \lfloor T \rfloor;\ P,E \vdash e \leadsto e'\ ]\!] \Longrightarrow P,E \vdash V{:=}e \leadsto V{:=}e'$
$|$ *AnnoLAssField*:
$[\![\ E\ V = None;\ E\ this = \lfloor Class\ C \rfloor;\ P \vdash C\ sees\ V,NonStatic{:}T\ in\ D;\ P,E \vdash e \leadsto e'\ ]\!]$
$\Longrightarrow P,E \vdash V{:=}e \leadsto Var\ this\boldsymbol{\cdot}V\{D\} := e'$
$|$ *AnnoFAcc*:
$[\![\ P,E \vdash e \leadsto e';\ \ P,E \vdash e' :: Class\ C;\ \ P \vdash C\ sees\ F,NonStatic{:}T\ in\ D\ ]\!]$
$\Longrightarrow P,E \vdash e\boldsymbol{\cdot}F\{[]\} \leadsto e'\boldsymbol{\cdot}F\{D\}$
$|$ *AnnoSFAcc*:
$[\![\ P \vdash C\ sees\ F,Static{:}T\ in\ D\ ]\!]$
$\Longrightarrow P,E \vdash C\boldsymbol{\cdot}_s F\{[]\} \leadsto C\boldsymbol{\cdot}_s F\{D\}$
$|$ *AnnoFAss*: $[\![\ P,E \vdash e1 \leadsto e1';\ \ P,E \vdash e2 \leadsto e2';$
$P,E \vdash e1' :: Class\ C;\ P \vdash C\ sees\ F,NonStatic{:}T\ in\ D\ ]\!]$
$\Longrightarrow P,E \vdash e1\boldsymbol{\cdot}F\{[]\} := e2 \leadsto e1'\boldsymbol{\cdot}F\{D\} := e2'$
$|$ *AnnoSFAss*: $[\![\ P,E \vdash e2 \leadsto e2';\ P \vdash C\ sees\ F,Static{:}T\ in\ D\ ]\!]$
$\Longrightarrow P,E \vdash C\boldsymbol{\cdot}_s F\{[]\} := e2 \leadsto C\boldsymbol{\cdot}_s F\{D\} := e2'$
$|$ *AnnoCall*:
$[\![\ P,E \vdash e \leadsto e';\ \ P,E \vdash es\ [\leadsto]\ es'\ ]\!]$
$\Longrightarrow P,E \vdash Call\ e\ M\ es \leadsto Call\ e'\ M\ es'$
$|$ *AnnoSCall*:
$[\![\ P,E \vdash es\ [\leadsto]\ es'\ ]\!]$
$\Longrightarrow P,E \vdash SCall\ C\ M\ es \leadsto SCall\ C\ M\ es'$
$|$ *AnnoBlock*:
$P,E(V \mapsto T) \vdash e \leadsto e' \implies P,E \vdash \{V{:}T;\ e\} \leadsto \{V{:}T;\ e'\}$
$|$ *AnnoComp*: $[\![\ P,E \vdash e1 \leadsto e1';\ \ P,E \vdash e2 \leadsto e2'\ ]\!]$
$\Longrightarrow\ P,E \vdash e1;;e2 \leadsto e1';;e2'$
$|$ *AnnoCond*: $[\![\ P,E \vdash e \leadsto e';\ P,E \vdash e1 \leadsto e1';\ \ P,E \vdash e2 \leadsto e2'\ ]\!]$
$\Longrightarrow P,E \vdash if\ (e)\ e1\ else\ e2 \leadsto if\ (e')\ e1'\ else\ e2'$
$|$ *AnnoLoop*: $[\![\ P,E \vdash e \leadsto e';\ \ P,E \vdash c \leadsto c'\ ]\!]$
$\Longrightarrow P,E \vdash while\ (e)\ c \leadsto while\ (e')\ c'$
$|$ *AnnoThrow*: $P,E \vdash e \leadsto e' \implies P,E \vdash throw\ e \leadsto throw\ e'$
$|$ *AnnoTry*: $[\![\ P,E \vdash e1 \leadsto e1';\ \ P,E(V \mapsto Class\ C) \vdash e2 \leadsto e2'\ ]\!]$
$\Longrightarrow P,E \vdash try\ e1\ catch(C\ V)\ e2 \leadsto try\ e1'\ catch(C\ V)\ e2'$

$|$ *AnnoNil*: $P,E \vdash []\ [\leadsto]\ []$
$|$ *AnnoCons*: $[\![\ P,E \vdash e \leadsto e';\ \ P,E \vdash es\ [\leadsto]\ es'\ ]\!]$
$\Longrightarrow\ P,E \vdash e\#es\ [\leadsto]\ e'\#es'$

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* $\equiv (\lambda x.\ None)(Start \mapsto (Map.empty,Done))$

definition *start-sheap-preloaded* :: *'m prog* $\Rightarrow$ *sheap*
where
  start-sheap-preloaded $P \equiv fold\ (\lambda(C,cl)\ f.\ f(C := Some\ (sblank\ P\ C,\ Prepared)))\ P\ (\lambda x.\ None)$

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 $curr\text{-}stk$ :: $frame \Rightarrow val\ list$ **where**
$curr\text{-}stk\ (stk,\ loc,\ C,\ M,\ pc,\ ics) = stk$

fun $curr\text{-}class$ :: $frame \Rightarrow cname$ **where**
$curr\text{-}class\ (stk,\ loc,\ C,\ M,\ pc,\ ics) = C$

fun $curr\text{-}method$ :: $frame \Rightarrow mname$ **where**
$curr\text{-}method\ (stk,\ loc,\ C,\ M,\ pc,\ ics) = M$

fun $curr\text{-}pc$ :: $frame \Rightarrow nat$ **where**
$curr\text{-}pc\ (stk,\ loc,\ C,\ M,\ pc,\ ics) = pc$

fun $init\text{-}status$ :: $frame \Rightarrow init\text{-}call\text{-}status$ **where**
$init\text{-}status\ (stk,\ loc,\ C,\ M,\ pc,\ ics) = ics$

fun $ics\text{-}of$ :: $frame \Rightarrow init\text{-}call\text{-}status$ **where**
$ics\text{-}of\ fr = snd(snd(snd(snd(snd\ fr))))$

2.1.2 Runtime State

type-synonym
$jvm\text{-}state = addr\ option \times heap \times frame\ list \times sheap$
— exception flag, heap, frames, static heap

translations
$(type)\ jvm\text{-}state$ <= $(type)\ nat\ option \times heap \times frame\ list \times sheap$

fun $frames\text{-}of$ :: $jvm\text{-}state \Rightarrow frame\ list$ **where**
$frames\text{-}of\ (xp,\ h,\ frs,\ sh) = frs$

abbreviation $sheap$ :: $jvm\text{-}state \Rightarrow sheap$ **where**
$sheap\ js \equiv snd\ (snd\ (snd\ js))$

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*
  $s \le pc \land pc < e \land P \vdash C \preceq^* C'$

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(*induct P a h frs sh rule*: *find-handler.induct*)
case *1* **then show** *?case* **by** *clarsimp*
next
case (*2 P a h fr frs sh*) **then show** *?case*
by(*cases fr*, *auto split*: *bool.splits list.splits init-call-status.splits*)
qed

lemma *find-handler-heap*[*simp*]:
find-handler P a h frs sh = (*xp'*,*h'*,*frs'*,*sh'*) ⟹ *h'* = *h*
by(*auto dest*: *find-handler-cases*)

lemma *find-handler-sheap*[*simp*]:
find-handler P a h frs sh = (*xp'*,*h'*,*frs'*,*sh'*) ⟹ *sh'* = *sh*
by(*auto dest*: *find-handler-cases*)

lemma *find-handler-frames*[*simp*]:
find-handler P a h frs sh = (*xp'*,*h'*,*frs'*,*sh'*) ⟹ *length frs'* ≤ *length frs*
proof(*induct frs*)
case *Nil* **then show** *?case* **by** *simp*
next
case (*Cons a frs*) **then show** *?case*
by(*auto simp*: *split-beta split*: *bool.splits list.splits init-call-status.splits*)
qed

lemma *find-handler-None*:
find-handler P a h frs sh = (*None*, *h*, *frs'*, *sh'*) ⟹ *frs'* ≠ []
by (*drule find-handler-cases*, *clarsimp*)

lemma *find-handler-Some*:
find-handler P a h frs sh = (*Some x*, *h*, *frs'*, *sh'*) ⟹ *frs'* = []
by (*drule find-handler-cases*, *clarsimp*)

lemma *find-handler-Some-same-error-same-heap*[*simp*]:
find-handler P a h frs sh = (*Some x*, *h′*, *frs′*, *sh′*) $\Longrightarrow$ *x* = *a* ∧ *h* = *h′* ∧ *sh* = *sh′*
by(*auto dest*: *find-handler-cases*)

lemma *find-handler-prealloc-pres*:
assumes *preallocated h*
and *fh*: *find-handler P a h frs sh* = (*xp′*,*h′*,*frs′*,*sh′*)
shows *preallocated h′*
using *assms find-handler-heap*[*OF fh*] **by** *simp*

lemma *find-handler-frs-tl-neq*:
ics-of f ≠ *No-ics*
$\Longrightarrow$ (*xp*, *h*, *f*#*frs*, *sh*) ≠ *find-handler P xa h′* (*f′* # *frs*) *sh′*
proof(*induct frs arbitrary*: *f f′*)
case *Nil* **then show** *?case* **by**(*auto simp*: *split-beta split*: *bool.splits*)
next
case (*Cons a frs*)
obtain *xp1 h1 frs1 sh1* **where** *fh*: *find-handler P xa h′* (*a* # *frs*) *sh′* = (*xp1*,*h1*,*frs1*,*sh1*)
by(*cases find-handler P xa h′* (*a* # *frs*) *sh′*)
then have *length frs1* ≤ *length* (*a*#*frs*)
by(*rule find-handler-frames*[**where** *P*=*P* **and** *a*=*xa* **and** *h*=*h′* **and** *frs*=*a*#*frs* **and** *sh*=*sh′*])
then have *neq*: *f*#*a*#*frs* ≠ *frs1* **by**(*clarsimp dest*: *impossible-Cons*)
then show *?case*
proof(*cases find-handler P xa h′* (*f′* # *a* # *frs*) *sh′* = *find-handler P xa h′* (*a* # *frs*) *sh′*)
case *True* **then show** *?thesis* **using** *neq fh* **by** *simp*
next
case *False* **then show** *?thesis* **using** *Cons.prems*
by(*fastforce simp*: *split-beta split*: *bool.splits init-call-status.splits list.splits*)
qed
qed

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 C0 M0 pc ics frs sh =
  (None, h, (tl stk, loc[n:=hd stk], C0, M0, Suc pc, ics)#frs, sh)

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

| exec-instr-New:
exec-instr (New C) P h stk loc C0 M0 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, C0, M0, pc, No-ics)#frs, sh)
             | Some a ⇒ (None, h(a↦blank P C), (Addr a#stk, loc, C0, M0, Suc pc, No-ics)#frs, sh)
          )
      | (-, Some(obj, Done)) ⇒
          (case new-Addr h of
                None ⇒ (⌊addr-of-sys-xcpt OutOfMemory⌋, h, (stk, loc, C0, M0, pc, ics)#frs, sh)
              | Some a ⇒ (None, h(a↦blank P C), (Addr a#stk, loc, C0, M0, Suc pc, ics)#frs, sh)
          )
      | - ⇒ (None, h, (stk, loc, C0, M0, 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* C_0 M_0 *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*, C_0, M_0, *pc*+*1*, *ics*)#*frs*, *sh*)
| *Some x* ⇒ (*xp'*, *h*, (*stk*, *loc*, C_0, M_0, *pc*, *ics*)#*frs*, *sh*)
)

| *exec-instr-Putstatic*:
exec-instr (*Putstatic C F D*) *P h stk loc* C_0 M_0 *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 M_0 = clinit of True ⇒ sh(C_0:=Some(fst(the(sh C_0)), Done))
| - ⇒ sh
)
in (None, h, [], sh′)
| (stk′,loc′,C′,m′,pc′,ics′)#frs′
⇒ let (D,b,Ts,T,(mxs,mxl_0,ins,xt)) = method P C_0 M_0;
offset = case b of NonStatic ⇒ 1 | Static ⇒ 0;
(sh″, stk″, pc″) = (case M_0 = clinit of True ⇒ (sh(C_0:=Some(fst(the(sh C_0)), Done)), stk′, pc′)
| - ⇒ (sh, (hd stk_0)#(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 C_0 M_0 pc ics frs sh = (None, h, (tl stk, loc, C_0, M_0, Suc pc, ics)#frs, sh)

| exec-instr-IAdd:
exec-instr IAdd P h stk loc C_0 M_0 pc ics frs sh =
(None, h, (Intg (the-Intg (hd (tl stk)) + the-Intg (hd stk))#(tl (tl stk)), loc, C_0, M_0, Suc pc, ics)#frs, sh)

| exec-instr-IfFalse:
exec-instr (IfFalse i) P h stk loc C_0 M_0 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, C_0, M_0, pc′, ics)#frs, sh))

| exec-instr-CmpEq:

exec-instr CmpEq P h stk loc C_0 M_0 *pc ics frs sh* =
  (*None*, *h*, (*Bool* (*hd* (*tl stk*) = *hd stk*) # *tl* (*tl stk*), *loc*, C_0, M_0, *Suc pc*, *ics*)#*frs*, *sh*)

| *exec-instr-Goto*:
exec-instr (*Goto i*) *P h stk loc* C_0 M_0 *pc ics frs sh* =
  (*None*, *h*, (*stk*, *loc*, C_0, M_0, *nat*(*int pc*+*i*), *ics*)#*frs*, *sh*)

| *exec-instr-Throw*:
exec-instr Throw P h stk loc C_0 M_0 *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*, C_0, M_0, *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*}
using *assms*
proof(*cases i*)
  case (*New C1*) **then show** *?thesis* **using** *assms*
  proof(*cases sh C1*)
    case (*Some a*)
    then obtain *sfs i* **where** *sfsi*: *a* = (*sfs*,*i*) **by**(*cases a*)
    then show *?thesis* **using** *Some New assms*
    proof(*cases i*) **qed**(*cases ics'*, *auto*)+
  qed(*cases ics'*, *auto*)
next
  case (*Getfield F1 C1*)
  obtain *D' b t* **where** *field*: *field P C1 F1* = (*D'*,*b*,*t*) **by**(*cases field P C1 F1*)
  obtain *D fs* **where** *addr*: *the* (*h* (*the-Addr* (*hd stk*))) = (*D*,*fs*) **by**(*cases the* (*h* (*the-Addr* (*hd stk*))))
  show *?thesis* **using** *addr field Getfield assms*
  proof(*cases hd stk* = *Null*)
    case *nNull*:*False* **then show** *?thesis* **using** *addr field Getfield assms*
    proof(*cases* ∄ *t b*. *P* ⊢ (*cname-of h* (*the-Addr* (*hd stk*))) *has F1*,*b*:*t in C1*)
      case *exists*:*False* **show** *?thesis*
      proof(*cases fst*(*snd*(*field P C1 F1*)))
        case *Static*
        then show *?thesis* **using** *exists nNull addr field Getfield assms*
         by(*auto simp*: *sys-xcpts-def split-beta split*: *staticb.splits*)
      next
        case *NonStatic*
        then show *?thesis* **using** *exists nNull addr field Getfield assms*
         by(*auto simp*: *sys-xcpts-def split-beta split*: *staticb.splits*)
      qed
    qed(*auto*)
  qed(*auto*)
next
  case (*Getstatic C1 F1 D1*)
  obtain *D' b t* **where** *field*: *field P D1 F1* = (*D'*,*b*,*t*) **by**(*cases field P D1 F1*)
  show *?thesis* **using** *field Getstatic assms*
  proof(*cases* ∄ *t b*. *P* ⊢ *C1 has F1*,*b*:*t in D1*)
    case *exists*:*False* **then show** *?thesis* **using** *field Getstatic assms*

```
    proof(cases fst(snd(field P D1 F1)))
      case Static
      then obtain sfs i where the(sh D') = (sfs, i) by(cases the(sh D'))
      then show ?thesis using exists field Static Getstatic assms by(cases ics'; cases i, auto)
    qed(auto)
  qed(auto)
next
  case (Putfield F1 C1)
  let ?r = hd(tl stk)
  obtain D' b t where field: field P C1 F1 = (D',b,t) by(cases field P C1 F1)
  obtain D fs where addr: the (h (the-Addr ?r)) = (D,fs)
    by(cases the (h (the-Addr ?r)))
  show ?thesis using addr field Putfield assms
  proof(cases ?r = Null)
    case nNull:False then show ?thesis using addr field Putfield assms
    proof(cases ∄t b. P ⊢ (cname-of h (the-Addr ?r)) has F1,b:t in C1)
      case exists:False show ?thesis
      proof(cases b)
        case Static
        then show ?thesis using exists nNull addr field Putfield assms
         by(auto simp: sys-xcpts-def split-beta split: staticb.splits)
      next
        case NonStatic
        then show ?thesis using exists nNull addr field Putfield assms
         by(auto simp: sys-xcpts-def split-beta split: staticb.splits)
      qed
    qed(auto)
  qed(auto)
next
  case (Putstatic C1 F1 D1)
  obtain D' b t where field: field P D1 F1 = (D',b,t) by(cases field P D1 F1)
  show ?thesis using field Putstatic assms
  proof(cases ∄t b. P ⊢ C1 has F1,b:t in D1)
    case exists:False then show ?thesis using field Putstatic assms
    proof(cases b)
      case Static
      then obtain sfs i where the(sh D') = (sfs, i) by(cases the(sh D'))
      then show ?thesis using exists field Static Putstatic assms by(cases ics'; cases i, auto)
    qed(auto)
  qed(auto)
next
  case (Checkcast C1) then show ?thesis using assms by(cases cast-ok P C1 h (hd stk), auto)
next
  case (Invoke M n)
  let ?r = stk!n
  let ?C = cname-of h (the-Addr ?r)
  show ?thesis using Invoke assms
  proof(cases ?r = Null)
    case nNull:False then show ?thesis using Invoke assms
    proof(cases ¬(∃Ts T m D b. P ⊢ ?C sees M,b:Ts → T = m in D))
      case exists:False then show ?thesis using nNull Invoke assms
      proof(cases fst(snd(method P ?C M)))
        case Static
        then show ?thesis using exists nNull Invoke assms
```

```
        by(auto simp: sys-xcpts-def split-beta split: staticb.splits)
      next
        case NonStatic
        then show ?thesis using exists nNull Invoke assms
          by(auto simp: sys-xcpts-def split-beta split: staticb.splits)
      qed
    qed(auto)
  qed(auto)
next
  case (Invokestatic C1 M n)
  show ?thesis using Invokestatic assms
  proof(cases ¬(∃ Ts T m D b. P ⊢ C1 sees M,b:Ts → T = m in D))
    case exists:False then show ?thesis using Invokestatic assms
    proof(cases fst(snd(method P C1 M)))
      case Static
      then obtain sfs i where the(sh (fst(method P C1 M))) = (sfs, i)
        by(cases the(sh (fst(method P C1 M))))
      then show ?thesis using exists Static Invokestatic assms
        by(auto split: init-call-status.splits init-state.splits)
    qed(auto)
  qed(auto)
next
  case Return then show ?thesis using assms
  proof(cases frs)
    case (Cons f frs') then show ?thesis using Return assms
      by(cases f, cases method P C M, cases M=clinit, auto)
  qed(auto)
next
  case (IfFalse x17) then show ?thesis using assms
  proof(cases hd stk)
    case (Bool b) then show ?thesis using IfFalse assms by(cases b, auto)
  qed(auto)
qed(auto)

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'
using assms
proof(cases i)
  case (New C1)
  then obtain sfs i where sfsi: the(sh C1) = (sfs,i) by(cases the(sh C1))
  then show ?thesis using preallocated-new[of h] New assms
    by(cases blank P C1, auto dest: new-Addr-SomeD split: init-call-status.splits init-state.splits)
next
  case (Getfield F1 C1) then show ?thesis using assms
    by(cases the (h (the-Addr (hd stk))), cases field P C1 F1, auto)
next
  case (Getstatic C1 F1 D1)
  then obtain sfs i where sfsi: the(sh (fst (field P D1 F1))) = (sfs, i)
   by(cases the(sh (fst (field P D1 F1))))
  then show ?thesis using Getstatic assms
   by(cases field P D1 F1, auto split: init-call-status.splits init-state.splits)
next
```

case (*Putfield F1 C1*) **then show** *?thesis* **using** *preallocated-new preallocated-upd-obj assms*
by(*cases the* (*h* (*the-Addr* (*hd* (*tl stk*)))), *cases field P C1 F1*, *auto*, *metis option.collapse*)
next
case (*Putstatic C1 F1 D1*)
then obtain *sfs i* **where** *sfsi*: *the*(*sh* (*fst* (*field P D1 F1*))) = (*sfs*, *i*)
by(*cases the*(*sh* (*fst* (*field P D1 F1*))))
then show *?thesis* **using** *Putstatic assms*
by(*cases field P D1 F1*, *auto split*: *init-call-status.splits init-state.splits*)
next
case (*Checkcast C1*)
then show *?thesis* **using** *assms*
proof(*cases hd stk* = *Null*)
case *False* **then show** *?thesis*
using *Checkcast assms*
by(*cases* $P \vdash$ *cname-of h* (*the-Addr* (*hd stk*)) $\preceq^*$ *C1*, *auto simp*: *cast-ok-def*)
qed(*simp add*: *cast-ok-def*)
next
case (*Invoke M n*)
then show *?thesis* **using** *assms* **by**(*cases method P* (*cname-of h* (*the-Addr* (*stk* ! *n*))) *M*, *auto*)
next
case (*Invokestatic C1 M n*)
show *?thesis*
proof(*cases sh* (*fst* (*method P C1 M*)))
case *None* **then show** *?thesis* **using** *Invokestatic assms*
by(*cases method P C1 M*, *auto split*: *init-call-status.splits*)
next
case (*Some a*)
then obtain *sfs i* **where** *sfsi*: *a* = (*sfs*, *i*) **by**(*cases a*)
then show *?thesis* **using** *Some Invokestatic assms*
proof(*cases i*) **qed**(*cases method P C1 M*, *auto split*: *init-call-status.splits*)+
qed
next
case *Return*
then show *?thesis* **using** *assms* **by**(*cases method P* C_0 M_0, *auto simp*: *split-beta split*: *list.splits*)
next
case (*IfFalse x17*) **then show** *?thesis* **using** *assms* **by**(*auto split*: *val.splits bool.splits*)
next
case *Throw* **then show** *?thesis* **using** *assms* **by**(*auto split*: *val.splits*)
qed(*auto*)

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*(*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*

$(\langle\text{-} \vdash\text{/ - } -jvm\rightarrow_1\text{/ -}\rangle\ [61,61,61]\ 60)$
for P :: *jvm-prog*
where
$P \vdash \sigma\ -jvm\rightarrow_1\ \sigma' \equiv (\sigma,\sigma') \in exec\text{-}1\ P$
| *exec-1I*: $exec\ (P,\sigma) = Some\ \sigma' \Longrightarrow P \vdash \sigma\ -jvm\rightarrow_1\ \sigma'$

— reflexive transitive closure:
definition *exec-all* :: $jvm\text{-}prog \Rightarrow jvm\text{-}state \Rightarrow jvm\text{-}state \Rightarrow bool$
$(\langle(\text{-} \vdash\text{/ - } -jvm\rightarrow\text{/ -})\rangle\ [61,61,61]60)$ **where**
exec-all-def1: $P \vdash \sigma\ -jvm\rightarrow\ \sigma' \longleftrightarrow (\sigma,\sigma') \in (exec\text{-}1\ P)^*$

notation (*ASCII*)
exec-all $(\langle\text{-}\ |-\text{/ - } -jvm->\text{/ -}\rangle\ [61,61,61]60)$

lemma *exec-1-eq*:
$exec\text{-}1\ P = \{(\sigma,\sigma').\ exec\ (P,\sigma) = Some\ \sigma'\}$
lemma *exec-1-iff*:
$P \vdash \sigma\ -jvm\rightarrow_1\ \sigma' = (exec\ (P,\sigma) = Some\ \sigma')$
lemma *exec-all-def*:
$P \vdash \sigma\ -jvm\rightarrow\ \sigma' = ((\sigma,\sigma') \in \{(\sigma,\sigma').\ exec\ (P,\sigma) = Some\ \sigma'\}^*)$
lemma *jvm-refl*[*iff*]: $P \vdash \sigma\ -jvm\rightarrow\ \sigma$
lemma *jvm-trans*[*trans*]:
$[\![\ P \vdash \sigma\ -jvm\rightarrow\ \sigma';\ P \vdash \sigma'\ -jvm\rightarrow\ \sigma''\ ]\!] \Longrightarrow P \vdash \sigma\ -jvm\rightarrow\ \sigma''$
lemma *jvm-one-step1*[*trans*]:
$[\![\ P \vdash \sigma\ -jvm\rightarrow_1\ \sigma';\ P \vdash \sigma'\ -jvm\rightarrow\ \sigma''\ ]\!] \Longrightarrow P \vdash \sigma\ -jvm\rightarrow\ \sigma''$
lemma *jvm-one-step2*[*trans*]:
$[\![\ P \vdash \sigma\ -jvm\rightarrow\ \sigma';\ P \vdash \sigma'\ -jvm\rightarrow_1\ \sigma''\ ]\!] \Longrightarrow P \vdash \sigma\ -jvm\rightarrow\ \sigma''$
lemma *exec-all-conf*:
$[\![\ P \vdash \sigma\ -jvm\rightarrow\ \sigma';\ P \vdash \sigma\ -jvm\rightarrow\ \sigma''\ ]\!]$
$\Longrightarrow P \vdash \sigma'\ -jvm\rightarrow\ \sigma'' \vee P \vdash \sigma''\ -jvm\rightarrow\ \sigma'$
lemma *exec-1-exec-all-conf*:
$[\![\ exec\ (P,\ \sigma) = Some\ \sigma';\ P \vdash \sigma\ -jvm\rightarrow\ \sigma'';\ \sigma \neq \sigma''\ ]\!]$
$\Longrightarrow P \vdash \sigma'\ -jvm\rightarrow\ \sigma''$
by(*auto elim*: *converse-rtranclE simp*: *exec-1-eq exec-all-def*)

lemma *exec-all-finalD*: $P \vdash (x,\ h,\ [],\ sh)\ -jvm\rightarrow\ \sigma \Longrightarrow \sigma = (x,\ h,\ [],\ sh)$
lemma *exec-all-deterministic*:
$[\![\ P \vdash \sigma\ -jvm\rightarrow\ (x,h,[],sh);\ P \vdash \sigma\ -jvm\rightarrow\ \sigma'\ ]\!] \Longrightarrow P \vdash \sigma'\ -jvm\rightarrow\ (x,h,[],sh)$

2.5.1 Preservation of preallocated

lemma *exec-Calling-prealloc-pres*:
assumes *preallocated h*
and $exec\text{-}Calling\ C\ Cs\ P\ h\ stk\ loc\ C_0\ M_0\ pc\ frs\ sh = (xp',h',frs',sh')$
shows *preallocated* h'
using *assms*
proof(*cases sh C*)
case (*Some a*)
then obtain *sfs i* **where** *sfsi*:$a = (sfs,\ i)$ **by**(*cases a*)
then show *?thesis* **using** *Some assms*
proof(*cases i*)
case *Prepared*
then show *?thesis* **using** *sfsi Some assms* **by**(*cases method P C clinit*, *auto split*: *if-split-asm*)

```
  next
    case Error
    then show ?thesis using sfsi Some assms by(cases method P C clinit, auto)
  qed(auto)
qed(auto)

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(cases ics)
  case No-ics
  then show ?thesis using exec-instr-prealloc-pres assms by auto
next
  case Calling
  then show ?thesis using exec-Calling-prealloc-pres assms by auto
next
  case (Called Cs)
  then show ?thesis using exec-instr-prealloc-pres assms by(cases Cs, auto)
next
  case (Throwing Cs a)
  then show ?thesis using assms by(cases Cs, auto)
qed

lemma exec-prealloc-pres:
assumes pre: preallocated h
  and exec (P, xp, h, frs, sh) = Some(xp′,h′,frs′,sh′)
shows preallocated h′
using assms
proof(cases ∃ x. xp = ⌊x⌋ ∨ frs = [])
  case False
  then obtain f1 frs1 where frs: frs = f1#frs1 by(cases frs, simp+)
  then obtain stk1 loc1 C1 M1 pc1 ics1 where f1: f1 = (stk1,loc1,C1,M1,pc1,ics1) by(cases f1)
  let ?i = instrs-of P C1 M1 ! pc1
  obtain xp2 h2 frs2 sh2
     where exec-step: exec-step P h stk1 loc1 C1 M1 pc1 ics1 frs1 sh = (xp2,h2,frs2,sh2)
    by(cases exec-step P h stk1 loc1 C1 M1 pc1 ics1 frs1 sh)
  then show ?thesis using exec-step-prealloc-pres[OF pre exec-step] f1 frs False assms
  proof(cases xp2)
    case (Some a)
    show ?thesis
      using find-handler-prealloc-pres[OF pre, where a=a]
            exec-step-prealloc-pres[OF pre]
            exec-step f1 frs Some False assms
       by(auto split: bool.splits init-call-status.splits list.splits)
  qed(auto split: init-call-status.splits)
qed(auto)
```

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* M_0 *pc frs sh* =
    (*n* < *length loc*)

| *check-instr-Store*:

$check\text{-}instr\ (Store\ n)\ P\ h\ stk\ loc\ C_0\ M_0\ pc\ frs\ sh =$
$(0 < length\ stk \wedge n < length\ loc)$

| $check\text{-}instr\text{-}Push$:
$check\text{-}instr\ (Push\ v)\ P\ h\ stk\ loc\ C_0\ M_0\ pc\ frs\ sh =$
$(\neg is\text{-}Addr\ v)$

| $check\text{-}instr\text{-}New$:
$check\text{-}instr\ (New\ C)\ P\ h\ stk\ loc\ C_0\ M_0\ pc\ frs\ sh =$
$is\text{-}class\ P\ C$

| $check\text{-}instr\text{-}Getfield$:
$check\text{-}instr\ (Getfield\ F\ C)\ P\ h\ stk\ loc\ C_0\ M_0\ pc\ frs\ sh =$
$(0 < length\ stk \wedge (\exists C'\ T.\ P \vdash C\ sees\ F,NonStatic{:}T\ in\ C') \wedge$
$(let\ (C',\ b,\ T) = field\ P\ C\ F;\ ref = hd\ stk\ in$
$C' = C \wedge is\text{-}Ref\ ref \wedge (ref \neq Null \longrightarrow$
$h\ (the\text{-}Addr\ ref) \neq None \wedge$
$(let\ (D,vs) = the\ (h\ (the\text{-}Addr\ ref))\ in$
$P \vdash D \preceq^* C \wedge vs\ (F,C) \neq None \wedge P,h \vdash the\ (vs\ (F,C)) :\leq T))))$

| $check\text{-}instr\text{-}Getstatic$:
$check\text{-}instr\ (Getstatic\ C\ F\ D)\ P\ h\ stk\ loc\ C_0\ M_0\ pc\ frs\ sh =$
$((\exists T.\ P \vdash C\ sees\ F,Static{:}T\ in\ D) \wedge$
$(let\ (C',\ b,\ T) = field\ P\ C\ F\ in$
$C' = D \wedge (sh\ D \neq None \longrightarrow$
$(let\ (sfs,i) = the\ (sh\ D)\ in$
$sfs\ F \neq None \wedge P,h \vdash the\ (sfs\ F) :\leq T))))$

| $check\text{-}instr\text{-}Putfield$:
$check\text{-}instr\ (Putfield\ F\ C)\ P\ h\ stk\ loc\ C_0\ M_0\ pc\ frs\ sh =$
$(1 < length\ stk \wedge (\exists C'\ T.\ P \vdash C\ sees\ F,NonStatic{:}T\ in\ C') \wedge$
$(let\ (C',\ b,\ T) = field\ P\ C\ F;\ v = hd\ stk;\ ref = hd\ (tl\ stk)\ in$
$C' = C \wedge is\text{-}Ref\ ref \wedge (ref \neq Null \longrightarrow$
$h\ (the\text{-}Addr\ ref) \neq None \wedge$
$(let\ D = fst\ (the\ (h\ (the\text{-}Addr\ ref)))\ in$
$P \vdash D \preceq^* C \wedge P,h \vdash v :\leq T))))$

| $check\text{-}instr\text{-}Putstatic$:
$check\text{-}instr\ (Putstatic\ C\ F\ D)\ P\ h\ stk\ loc\ C_0\ M_0\ pc\ frs\ sh =$
$(0 < length\ stk \wedge (\exists T.\ P \vdash C\ sees\ F,Static{:}T\ in\ D) \wedge$
$(let\ (C',\ b,\ T) = field\ P\ C\ F;\ v = hd\ stk\ in$
$C' = D \wedge P,h \vdash v :\leq T))$

| $check\text{-}instr\text{-}Checkcast$:
$check\text{-}instr\ (Checkcast\ C)\ P\ h\ stk\ loc\ C_0\ M_0\ pc\ frs\ sh =$
$(0 < length\ stk \wedge is\text{-}class\ P\ C \wedge is\text{-}Ref\ (hd\ stk))$

| $check\text{-}instr\text{-}Invoke$:
$check\text{-}instr\ (Invoke\ M\ n)\ P\ h\ stk\ loc\ C_0\ M_0\ pc\ frs\ sh =$
$(n < length\ stk \wedge is\text{-}Ref\ (stk!n) \wedge$
$(stk!n \neq Null \longrightarrow$
$(let\ a = the\text{-}Addr\ (stk!n);$
$C = cname\text{-}of\ h\ a;$
$Ts = fst\ (snd\ (snd\ (method\ P\ C\ M)))$

$in\ h\ a \neq None \wedge P \vdash C\ has\ M{,}NonStatic \wedge$
$P{,}h \vdash rev\ (take\ n\ stk)\ [:\leq]\ Ts)))$

$|\ check\text{-}instr\text{-}Invokestatic$:
$check\text{-}instr\ (Invokestatic\ C\ M\ n)\ P\ h\ stk\ loc\ C_0\ M_0\ pc\ frs\ sh =$
$(n \leq length\ stk \wedge$
$(let\ Ts = fst\ (snd\ (snd\ (method\ P\ C\ M)))$
$in\ P \vdash C\ has\ M{,}Static \wedge$
$P{,}h \vdash rev\ (take\ n\ stk)\ [:\leq]\ Ts))$

$|\ check\text{-}instr\text{-}Return$:
$check\text{-}instr\ Return\ P\ h\ stk\ loc\ C_0\ M_0\ pc\ frs\ sh =$
$(case\ (M_0 = clinit)\ of\ False \Rightarrow (0 < length\ stk \wedge ((0 < length\ frs) \longrightarrow$
$(\exists\, b.\ P \vdash C_0\ has\ M_0{,}b) \wedge$
$(let\ v = hd\ stk;$
$T = fst\ (snd\ (snd\ (snd\ (method\ P\ C_0\ M_0))))$
$in\ P{,}h \vdash v :\leq T)))$
$|\ True \Rightarrow P \vdash C_0\ has\ M_0{,}Static)$

$|\ check\text{-}instr\text{-}Pop$:
$check\text{-}instr\ Pop\ P\ h\ stk\ loc\ C_0\ M_0\ pc\ frs\ sh =$
$(0 < length\ stk)$

$|\ check\text{-}instr\text{-}IAdd$:
$check\text{-}instr\ IAdd\ P\ h\ stk\ loc\ C_0\ M_0\ pc\ frs\ sh =$
$(1 < length\ stk \wedge is\text{-}Intg\ (hd\ stk) \wedge is\text{-}Intg\ (hd\ (tl\ stk)))$

$|\ check\text{-}instr\text{-}IfFalse$:
$check\text{-}instr\ (IfFalse\ b)\ P\ h\ stk\ loc\ C_0\ M_0\ pc\ frs\ sh =$
$(0 < length\ stk \wedge is\text{-}Bool\ (hd\ stk) \wedge 0 \leq int\ pc{+}b)$

$|\ check\text{-}instr\text{-}CmpEq$:
$check\text{-}instr\ CmpEq\ P\ h\ stk\ loc\ C_0\ M_0\ pc\ frs\ sh =$
$(1 < length\ stk)$

$|\ check\text{-}instr\text{-}Goto$:
$check\text{-}instr\ (Goto\ b)\ P\ h\ stk\ loc\ C_0\ M_0\ pc\ frs\ sh =$
$(0 \leq int\ pc{+}b)$

$|\ check\text{-}instr\text{-}Throw$:
$check\text{-}instr\ Throw\ P\ h\ stk\ loc\ C_0\ M_0\ pc\ frs\ sh =$
$(0 < length\ stk \wedge is\text{-}Ref\ (hd\ stk))$

definition $check :: jvm\text{-}prog \Rightarrow jvm\text{-}state \Rightarrow bool$ **where**
$check\ P\ \sigma = (let\ (xcpt,\ h,\ frs,\ sh) = \sigma\ in$
$(case\ frs\ of\ [] \Rightarrow True\ |\ (stk{,}loc{,}C{,}M{,}pc{,}ics)\#frs' \Rightarrow$
$\exists\, b.\ P \vdash C\ has\ M,\ b\ \wedge$
$(let\ (C'{,}b{,}Ts{,}T{,}mxs{,}mxl_0{,}ins{,}xt) = method\ P\ C\ M;\ i = ins!pc\ in$
$pc < size\ ins \wedge size\ stk \leq mxs \wedge$
$check\text{-}instr\ i\ P\ h\ stk\ loc\ C\ M\ pc\ frs'\ sh)))$

definition $exec\text{-}d :: jvm\text{-}prog \Rightarrow jvm\text{-}state \Rightarrow jvm\text{-}state\ option\ type\text{-}error$ **where**
$exec\text{-}d\ P\ \sigma = (if\ check\ P\ \sigma\ then\ Normal\ (exec\ (P,\ \sigma))\ else\ TypeError)$

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
  $P \vdash \sigma$ −*jvmd*→$_1$ $\sigma' \equiv (\sigma,\sigma') \in$ *exec-1-d P*
| *exec-1-d-ErrorI*: *exec-d P* σ = *TypeError* $\Longrightarrow$ $P \vdash$ *Normal* σ −*jvmd*→$_1$ *TypeError*
| *exec-1-d-NormalI*: *exec-d P* σ = *Normal* (*Some* σ') $\Longrightarrow$ $P \vdash$ *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: $P \vdash \sigma$ −*jvmd*→ $\sigma' \longleftrightarrow (\sigma,\sigma') \in$ (*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')}
by (*auto elim*!: *exec-1-d.cases intro*!: *exec-1-d.intros*)

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

lemma *if-neq* [*dest*!]:
  (*if P then A else B*) ≠ *B* ⟹ *P*
  by (*cases P*, *auto*)

lemma *exec-d-no-errorI* [*intro*]:
  check P σ ⟹ *exec-d P* σ ≠ *TypeError*
  by (*unfold exec-d-def*) *simp*

theorem *no-type-error-commutes*:
  exec-d P σ ≠ *TypeError* ⟹ *exec-d P* σ = *Normal* (*exec* (*P*, σ))
  by (*unfold exec-d-def*, *auto*)

lemma *defensive-imp-aggressive*:
  P ⊢ (*Normal* σ) −*jvmd*→ (*Normal* σ') ⟹ *P* ⊢ σ −*jvm*→ σ'
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* $P\ C \equiv$ *fst* (*the* (*class* $P\ C$))

lemma *superI*:
  $(C,D) \in$ *subcls1* $P \Longrightarrow$ *super* $P\ C = D$
  by (*unfold super-def*) (*auto dest*: *subcls1D*)

primrec *the-Class* :: *ty* $\Rightarrow$ *cname*
where
  the-Class (*Class* C) = C

definition *sup* :: $'c$ *prog* $\Rightarrow$ *ty* $\Rightarrow$ *ty* $\Rightarrow$ *ty err*
where
  sup $P\ T_1\ T_2 \equiv$
  if is-refT $T_1 \wedge$ *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 $P = (\lambda T_1\ T_2.$
  if is-refT $T_1 \wedge$ *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*))
  by (*simp add*: *sup-def fun-eq-iff*)

abbreviation
  subtype :: $'c$ *prog* $\Rightarrow$ *ty* $\Rightarrow$ *ty* $\Rightarrow$ *bool*
  where *subtype* $P \equiv$ *widen* P

definition *esl* :: $'c$ *prog* $\Rightarrow$ *ty esl*
where
  esl $P \equiv$ (*types* P, *subtype* P, *sup* P)

lemma *is-class-is-subcls*:
  wf-prog $m\ P \Longrightarrow$ *is-class* $P\ C = P \vdash C \preceq^* Object$

lemma *subcls-antisym*:
  $[\![$*wf-prog* $m\ P$; $P \vdash C \preceq^* D$; $P \vdash D \preceq^* C]\!] \Longrightarrow C = D$

lemma *widen-antisym*:
  $[\![$ *wf-prog* $m\ P$; $P \vdash T \leq U$; $P \vdash U \leq T\ ]\!] \Longrightarrow T = U$
lemma *order-widen* [*intro*,*simp*]:
  wf-prog $m\ P \Longrightarrow$ *order* (*subtype* P) (*types* P)

lemma *NT-widen*:
$P \vdash NT \leq T = (T = NT \vee (\exists C.\ T = Class\ C))$

lemma *Class-widen2*: $P \vdash Class\ C \leq T = (\exists D.\ T = Class\ D \wedge P \vdash C \preceq^* D)$
lemma *wf-converse-subcls1-impl-acc-subtype*:
$wf\ ((subcls1\ P)\hat{}{-1}) \Longrightarrow acc\ (subtype\ P)$
lemma *wf-subtype-acc* [*intro*, *simp*]:
$wf\text{-}prog\ wf\text{-}mb\ P \Longrightarrow acc\ (subtype\ P)$
lemma *exec-lub-refl* [*simp*]: $exec\text{-}lub\ r\ f\ T\ T = T$
lemma *closed-err-types*:
$wf\text{-}prog\ wf\text{-}mb\ P \Longrightarrow closed\ (err\ (types\ P))\ (lift2\ (sup\ P))$

lemma *sup-subtype-greater*:
$[\![\ wf\text{-}prog\ wf\text{-}mb\ P;\ is\text{-}type\ P\ t1;\ is\text{-}type\ P\ t2;\ sup\ P\ t1\ t2 = OK\ s\ ]\!]$
$\Longrightarrow subtype\ P\ t1\ s \wedge subtype\ P\ t2\ s$
lemma *sup-subtype-smallest*:
$[\![\ wf\text{-}prog\ wf\text{-}mb\ P;\ is\text{-}type\ P\ a;\ is\text{-}type\ P\ b;\ is\text{-}type\ P\ c;$
$subtype\ P\ a\ c;\ subtype\ P\ b\ c;\ sup\ P\ a\ b = OK\ d\ ]\!]$
$\Longrightarrow subtype\ P\ d\ c$
lemma *sup-exists*:
$[\![\ subtype\ P\ a\ c;\ subtype\ P\ b\ c\ ]\!] \Longrightarrow \exists T.\ sup\ P\ a\ b = OK\ T$
lemma *err-semilat-JType-esl*:
$wf\text{-}prog\ wf\text{-}mb\ P \Longrightarrow err\text{-}semilat\ (esl\ P)$

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 $stk\text{-}esl :: {}'c\ prog \Rightarrow nat \Rightarrow ty_s\ esl$
where
$stk\text{-}esl\ P\ mxs \equiv upto\text{-}esl\ mxs\ (SemiType.esl\ P)$

definition $loc\text{-}sl :: {}'c\ prog \Rightarrow nat \Rightarrow ty_l\ sl$
where
$loc\text{-}sl\ P\ mxl \equiv Listn.sl\ mxl\ (Err.sl\ (SemiType.esl\ P))$

definition $sl :: {}'c\ prog \Rightarrow nat \Rightarrow nat \Rightarrow ty_i{}'\ err\ sl$
where
$sl\ P\ mxs\ mxl \equiv$
$Err.sl(Opt.esl(Product.esl\ (stk\text{-}esl\ P\ mxs)\ (Err.esl(loc\text{-}sl\ P\ mxl))))$

definition $states :: {}'c\ prog \Rightarrow nat \Rightarrow nat \Rightarrow ty_i{}'\ err\ set$

where $states\ P\ mxs\ mxl \equiv fst(sl\ P\ mxs\ mxl)$

definition $le :: 'c\ prog \Rightarrow nat \Rightarrow nat \Rightarrow ty_i'\ err\ ord$
where
$le\ P\ mxs\ mxl \equiv fst(snd(sl\ P\ mxs\ mxl))$

definition $sup :: 'c\ prog \Rightarrow nat \Rightarrow nat \Rightarrow ty_i'\ err\ binop$
where
$sup\ P\ mxs\ mxl \equiv snd(snd(sl\ P\ mxs\ mxl))$

definition $sup\text{-}ty\text{-}opt :: ['c\ prog, ty\ err, ty\ err] \Rightarrow bool$
$(\langle\text{-} \vdash \text{-} \leq_\top \text{-}\rangle\ [71,71,71]\ 70)$
where
$sup\text{-}ty\text{-}opt\ P \equiv Err.le\ (subtype\ P)$

definition $sup\text{-}state :: ['c\ prog, ty_i, ty_i] \Rightarrow bool$
$(\langle\text{-} \vdash \text{-} \leq_i \text{-}\rangle\ [71,71,71]\ 70)$
where
$sup\text{-}state\ P \equiv Product.le\ (Listn.le\ (subtype\ P))\ (Listn.le\ (sup\text{-}ty\text{-}opt\ P))$

definition $sup\text{-}state\text{-}opt :: ['c\ prog, ty_i', ty_i'] \Rightarrow bool$
$(\langle\text{-} \vdash \text{-} \leq'' \text{-}\rangle\ [71,71,71]\ 70)$
where
$sup\text{-}state\text{-}opt\ P \equiv Opt.le\ (sup\text{-}state\ P)$

abbreviation
$sup\text{-}loc :: ['c\ prog, ty_l, ty_l] \Rightarrow bool\ \ (\langle\text{-} \vdash \text{-}\ [\leq_\top]\ \text{-}\rangle\ [71,71,71]\ 70)$
where $P \vdash LT\ [\leq_\top]\ LT' \equiv list\text{-}all2\ (sup\text{-}ty\text{-}opt\ P)\ LT\ LT'$

notation $(ASCII)$
$sup\text{-}ty\text{-}opt\ \ (\langle\text{-}\ |\text{-}\ \text{-} <=T \text{-}\rangle\ [71,71,71]\ 70)$ **and**
$sup\text{-}state\ \ (\langle\text{-}\ |\text{-}\ \text{-} <=i \text{-}\rangle\ [71,71,71]\ 70)$ **and**
$sup\text{-}state\text{-}opt\ \ (\langle\text{-}\ |\text{-}\ \text{-} <=' \text{-}\rangle\ [71,71,71]\ 70)$ **and**
$sup\text{-}loc\ \ (\langle\text{-}\ |\text{-}\ \text{-}\ [<=T]\ \text{-}\rangle\ [71,71,71]\ 70)$

2.8.1 Unfolding

lemma *JVM-states-unfold*:
$states\ P\ mxs\ mxl \equiv err(opt((Union\ \{nlists\ n\ (types\ P)\ |n.\ n <= mxs\}) \times nlists\ mxl\ (err(types\ P))))$
lemma *JVM-le-unfold*:
$le\ P\ m\ n \equiv$
$Err.le(Opt.le(Product.le(Listn.le(subtype\ P))(Listn.le(Err.le(subtype\ P)))))$
lemma *sl-def2*:
$JVM\text{-}SemiType.sl\ P\ mxs\ mxl \equiv$
$(states\ P\ mxs\ mxl,\ JVM\text{-}SemiType.le\ P\ mxs\ mxl,\ JVM\text{-}SemiType.sup\ P\ mxs\ mxl)$

lemma *JVM-le-conv*:
$le\ P\ m\ n\ (OK\ t1)\ (OK\ t2) = P \vdash t1 \leq' t2$
lemma *JVM-le-Err-conv*:
$le\ P\ m\ n = Err.le\ (sup\text{-}state\text{-}opt\ P)$
lemma *err-le-unfold* [*iff*]:
$Err.le\ r\ (OK\ a)\ (OK\ b) = r\ a\ b$

2.8.2 Semilattice

lemma *order-sup-state-opt'* [*intro*, *simp*]:
wf-prog wf-mb P $\Longrightarrow$
order (*sup-state-opt P*) (*opt* (($\bigcup$ {*nlists n* (*types P*) |*n*. $n \leq mxs$}) $\times$ *nlists* (*Suc* (*length Ts* + mxl_0)) (*err* (*types P*))))

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* $\Rightarrow$ ty_i $\Rightarrow$ *pc* $\Rightarrow$ *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*:: $ty \Rightarrow cname$ **where**
  the-class (*Class C*) = *C*

fun eff_i :: $instr \times {'m}\ prog \times ty_i \Rightarrow ty_i$ **where**
  eff_i-*Load*:
    $\mathit{eff}_i\ (Load\ n,\ P,\ (ST,\ LT)) = (ok\text{-}val\ (LT\ !\ n)\ \#\ ST,\ LT)$
| eff_i-*Store*:
    $\mathit{eff}_i\ (Store\ n,\ P,\ (T\#ST,\ LT)) = (ST,\ LT[n{:=}\ OK\ T])$
| eff_i-*Push*:
    $\mathit{eff}_i\ (Push\ v,\ P,\ (ST,\ LT)) = (the\ (typeof\ v)\ \#\ ST,\ LT)$
| eff_i-*Getfield*:
    $\mathit{eff}_i\ (Getfield\ F\ C,\ P,\ (T\#ST,\ LT)) = (snd\ (snd\ (field\ P\ C\ F))\ \#\ ST,\ LT)$
| eff_i-*Getstatic*:
    $\mathit{eff}_i\ (Getstatic\ C\ F\ D,\ P,\ (ST,\ LT)) = (snd\ (snd\ (field\ P\ C\ F))\ \#\ ST,\ LT)$
| eff_i-*Putfield*:
   $\mathit{eff}_i\ (Putfield\ F\ C,\ P,\ (T_1\#T_2\#ST,\ LT)) = (ST{,}LT)$
| eff_i-*Putstatic*:
   $\mathit{eff}_i\ (Putstatic\ C\ F\ D,\ P,\ (T\#ST,\ LT)) = (ST{,}LT)$
| eff_i-*New*:
   $\mathit{eff}_i\ (New\ C,\ P,\ (ST{,}LT)) = (Class\ C\ \#\ ST,\ LT)$
| eff_i-*Checkcast*:
   $\mathit{eff}_i\ (Checkcast\ C,\ P,\ (T\#ST{,}LT)) = (Class\ C\ \#\ ST{,}LT)$
| eff_i-*Pop*:
   $\mathit{eff}_i\ (Pop,\ P,\ (T\#ST{,}LT)) = (ST{,}LT)$
| eff_i-*IAdd*:
   $\mathit{eff}_i\ (IAdd,\ P{,}(T_1\#T_2\#ST{,}LT)) = (Integer\#ST{,}LT)$
| eff_i-*CmpEq*:
   $\mathit{eff}_i\ (CmpEq,\ P,\ (T_1\#T_2\#ST{,}LT)) = (Boolean\#ST{,}LT)$
| eff_i-*IfFalse*:
   $\mathit{eff}_i\ (IfFalse\ b,\ P,\ (T_1\#ST{,}LT)) = (ST{,}LT)$
| eff_i-*Invoke*:
   $\mathit{eff}_i\ (Invoke\ M\ n,\ P,\ (ST{,}LT)) =$
    $(let\ C = the\text{-}class\ (ST!n);\ (D{,}b{,}Ts{,}T_r{,}m) = method\ P\ C\ M$
     $in\ (T_r\ \#\ drop\ (n{+}1)\ ST,\ LT))$
| eff_i-*Invokestatic*:
   $\mathit{eff}_i\ (Invokestatic\ C\ M\ n,\ P,\ (ST{,}LT)) =$
    $(let\ (D{,}b{,}Ts{,}T_r{,}m) = method\ P\ C\ M$
     $in\ (T_r\ \#\ drop\ n\ ST,\ LT))$
| eff_i-*Goto*:
   $\mathit{eff}_i\ (Goto\ n,\ P,\ s) = s$

fun *is-relevant-class* :: $instr \Rightarrow {'m}\ prog \Rightarrow cname \Rightarrow bool$ **where**
  rel-Getfield:
    is-relevant-class (*Getfield F D*)
     $= (\lambda P\ C.\ P \vdash NullPointer \preceq^* C \lor P \vdash NoSuchFieldError \preceq^* C$
          $\lor P \vdash IncompatibleClassChangeError \preceq^* C)$
| *rel-Getstatic*:
    is-relevant-class (*Getstatic C F D*)
     $= (\lambda P\ C.\ True)$
| *rel-Putfield*:
    is-relevant-class (*Putfield F D*)
     $= (\lambda P\ C.\ P \vdash NullPointer \preceq^* C \lor P \vdash NoSuchFieldError \preceq^* C$
          $\lor P \vdash IncompatibleClassChangeError \preceq^* C)$

| *rel-Putstatic*:
  is-relevant-class (*Putstatic C F D*)
   = (λ*P C*. *True*)
| *rel-Checkcast*:
  is-relevant-class (*Checkcast D*) = (λ*P C*. *P* $\vdash$ *ClassCast* $\preceq^*$ *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* $\Rightarrow$ *instr* $\Rightarrow$ *pc* $\Rightarrow$ *ex-entry* $\Rightarrow$ *bool* **where**
is-relevant-entry P i pc e $\longleftrightarrow$ (*let* (*f*,*t*,*C*,*h*,*d*) = *e in is-relevant-class i P C* $\wedge$ *pc* $\in$ {*f*..<*t*})

definition *relevant-entries* :: *'m prog* $\Rightarrow$ *instr* $\Rightarrow$ *pc* $\Rightarrow$ *ex-table* $\Rightarrow$ *ex-table* **where**
relevant-entries P i pc = *filter* (*is-relevant-entry P i pc*)

definition *xcpt-eff* :: *instr* $\Rightarrow$ *'m prog* $\Rightarrow$ *pc* $\Rightarrow$ ty_i
  $\Rightarrow$ *ex-table* $\Rightarrow$ (*pc* $\times$ 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* $\Rightarrow$ *'m prog* $\Rightarrow$ *nat* $\Rightarrow$ ty_i $\Rightarrow$ (*pc* $\times$ 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* $\Rightarrow$ *'m prog* $\Rightarrow$ *pc* $\Rightarrow$ *ex-table* $\Rightarrow$ ty_i' $\Rightarrow$ (*pc* $\times$ ty_i') *list* **where**
eff i P pc et t = (*case t of*
  None $\Rightarrow$ []
| *Some* τ $\Rightarrow$ (*norm-eff i P pc* τ) @ (*xcpt-eff i P pc* τ *et*))

lemma *eff-None*:
eff i P pc xt None = []
by (*simp add*: *eff-def*)

lemma *eff-Some*:
eff i P pc xt (*Some* τ) = *norm-eff i P pc* τ @ *xcpt-eff i P pc* τ *xt*
by (*simp add*: *eff-def*)

Conditions under which eff is applicable:

fun app_i :: *instr* $\times$ *'m prog* $\times$ *pc* $\times$ *nat* $\times$ *ty* $\times$ ty_i $\Rightarrow$ *bool* **where**
app_i*-Load*:
  app_i (*Load n*, *P*, *pc*, *mxs*, T_r, (*ST*,*LT*)) =
  (*n* < *length LT* $\wedge$ *LT* ! *n* $\neq$ *Err* $\wedge$ *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*)) =

$(\textit{length } ST < mxs \wedge \textit{typeof } v \neq \textit{None})$
$|$ app_i-*Getfield*:
app_i $(\textit{Getfield } F\ C,\ P,\ pc,\ mxs,\ T_r,\ (T\#ST,\ LT)) =$
$(\exists\, T_f.\ P \vdash C \textit{ sees } F,\textit{NonStatic}{:}T_f \textit{ in } C \wedge P \vdash T \leq \textit{Class } C)$
$|$ app_i-*Getstatic*:
app_i $(\textit{Getstatic } C\ F\ D,\ P,\ pc,\ mxs,\ T_r,\ (ST,\ LT)) =$
$(\textit{length } ST < mxs \wedge (\exists\, T_f.\ P \vdash C \textit{ sees } F,\textit{Static}{:}T_f \textit{ in } D))$
$|$ app_i-*Putfield*:
app_i $(\textit{Putfield } F\ C,\ P,\ pc,\ mxs,\ T_r,\ (T_1\#T_2\#ST,\ LT)) =$
$(\exists\, T_f.\ P \vdash C \textit{ sees } F,\textit{NonStatic}{:}T_f \textit{ in } C \wedge P \vdash T_2 \leq (\textit{Class } C) \wedge P \vdash T_1 \leq T_f)$
$|$ app_i-*Putstatic*:
app_i $(\textit{Putstatic } C\ F\ D,\ P,\ pc,\ mxs,\ T_r,\ (T\#ST,\ LT)) =$
$(\exists\, T_f.\ P \vdash C \textit{ sees } F,\textit{Static}{:}T_f \textit{ in } D \wedge P \vdash T \leq T_f)$
$|$ app_i-*New*:
app_i $(\textit{New } C,\ P,\ pc,\ mxs,\ T_r,\ (ST,LT)) =$
$(\textit{is-class } P\ C \wedge \textit{length } ST < mxs)$
$|$ app_i-*Checkcast*:
app_i $(\textit{Checkcast } C,\ P,\ pc,\ mxs,\ T_r,\ (T\#ST,LT)) =$
$(\textit{is-class } P\ C \wedge \textit{is-refT } T)$
$|$ app_i-*Pop*:
app_i $(\textit{Pop},\ P,\ pc,\ mxs,\ T_r,\ (T\#ST,LT)) =$
True
$|$ app_i-*IAdd*:
app_i $(\textit{IAdd},\ P,\ pc,\ mxs,\ T_r,\ (T_1\#T_2\#ST,LT)) = (T_1 = T_2 \wedge T_1 = \textit{Integer})$
$|$ app_i-*CmpEq*:
app_i $(\textit{CmpEq},\ P,\ pc,\ mxs,\ T_r,\ (T_1\#T_2\#ST,LT)) =$
$(T_1 = T_2 \vee \textit{is-refT } T_1 \wedge \textit{is-refT } T_2)$
$|$ app_i-*IfFalse*:
app_i $(\textit{IfFalse } b,\ P,\ pc,\ mxs,\ T_r,\ (\textit{Boolean}\#ST,LT)) =$
$(0 \leq \textit{int } pc + b)$
$|$ app_i-*Goto*:
app_i $(\textit{Goto } b,\ P,\ pc,\ mxs,\ T_r,\ s) =$
$(0 \leq \textit{int } pc + b)$
$|$ app_i-*Return*:
app_i $(\textit{Return},\ P,\ pc,\ mxs,\ T_r,\ (T\#ST,LT)) =$
$(P \vdash T \leq T_r)$
$|$ app_i-*Throw*:
app_i $(\textit{Throw},\ P,\ pc,\ mxs,\ T_r,\ (T\#ST,LT)) =$
$\textit{is-refT } T$
$|$ app_i-*Invoke*:
app_i $(\textit{Invoke } M\ n,\ P,\ pc,\ mxs,\ T_r,\ (ST,LT)) =$
$(n < \textit{length } ST \wedge$
$(ST!n \neq NT \longrightarrow$
$(\exists\, C\ D\ Ts\ T\ m.\ ST!n = \textit{Class } C \wedge P \vdash C \textit{ sees } M,\textit{NonStatic}{:}Ts \rightarrow T = m \textit{ in } D \wedge$
$P \vdash \textit{rev } (\textit{take } n\ ST)\ [\leq]\ Ts)))$
$|$ app_i-*Invokestatic*:
app_i $(\textit{Invokestatic } C\ M\ n,\ P,\ pc,\ mxs,\ T_r,\ (ST,LT)) =$
$(\textit{length } ST - n < mxs \wedge n \leq \textit{length } ST \wedge M \neq \textit{clinit} \wedge$
$(\exists\, D\ Ts\ T\ m.\ P \vdash C \textit{ sees } M,\textit{Static}{:}Ts \rightarrow T = m \textit{ in } D \wedge$
$P \vdash \textit{rev } (\textit{take } n\ ST)\ [\leq]\ Ts))$

$|$ app_i-*default*:
app_i $(i,P,\ pc,mxs,T_r,s) = \textit{False}$

definition *xcpt-app* :: *instr* ⇒ *'m prog* ⇒ *pc* ⇒ *nat* ⇒ *ex-table* ⇒ ty_i ⇒ *bool* **where**
xcpt-app i P pc mxs xt τ ⟷ (∀ (*f*,*t*,*C*,*h*,*d*) ∈ *set* (*relevant-entries P i pc xt*). *is-class P C* ∧ *d* ≤ *size* (*fst* τ) ∧ *d* < *mxs*)

definition *app* :: *instr* ⇒ *'m prog* ⇒ *nat* ⇒ *ty* ⇒ *nat* ⇒ *nat* ⇒ *ex-table* ⇒ ty_i' ⇒ *bool* **where**
app i P mxs T_r *pc mpc xt t* = (*case t of None* ⇒ *True* | *Some* τ ⇒
app_i (*i*,*P*,*pc*,*mxs*,T_r,τ) ∧ *xcpt-app i P pc mxs xt* τ ∧
(∀ (*pc'*,τ') ∈ *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) ∧ *xcpt-app i P pc mxs xt* τ ∧
(∀ (*pc'*,*s'*) ∈ *set* (*eff i P pc xt* (*Some* τ)). *pc'* < *mpc*))
by (*simp add*: *app-def*)

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* τ ≡ *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* τ ≡ *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* τ ≡ *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* τ ≡ *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 ⋀*LT*. *P* ([],*LT*)
assumes ⋀*l ST LT*. *P* (*l*#*ST*,*LT*)
shows *P s*
by (*cases s*, *cases fst s*) (*auto intro*!: *assms*)

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 $P\ s$
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 $P\ s$

simp rules for *app*

lemma *appNone*[*simp*]: $app\ i\ P\ mxs\ T_r\ pc\ mpc\ et\ None = True$
  by (*simp add*: *app-def*)

lemma *appLoad*[*simp*]:
$app_i\ (Load\ idx,\ P,\ T_r,\ mxs,\ pc,\ s) = (\exists ST\ LT.\ s = (ST,LT) \wedge idx < length\ LT \wedge LT!idx \neq Err \wedge length\ ST < mxs)$
  by (*cases s*, *simp*)

lemma *appStore*[*simp*]:
$app_i\ (Store\ idx,P,pc,mxs,T_r,s) = (\exists ts\ ST\ LT.\ s = (ts\#ST,LT) \wedge idx < length\ LT)$
  by (*rule length-cases2*, *auto*)

lemma *appPush*[*simp*]:
$app_i\ (Push\ v,P,pc,mxs,T_r,s) =$
 $(\exists ST\ LT.\ s = (ST,LT) \wedge length\ ST < mxs \wedge typeof\ v \neq None)$
  by (*cases s*, *simp*)

lemma *appGetField*[*simp*]:
$app_i\ (Getfield\ F\ C,P,pc,mxs,T_r,s) =$
 $(\exists\ oT\ vT\ ST\ LT.\ s = (oT\#ST,\ LT) \wedge$
  $P \vdash C\ sees\ F,NonStatic{:}vT\ in\ C \wedge P \vdash oT \leq (Class\ C))$
  by (*rule length-cases2* [*of - s*]) *auto*

lemma *appGetStatic*[*simp*]:
$app_i\ (Getstatic\ C\ F\ D,P,pc,mxs,T_r,s) =$
 $(\exists\ vT\ ST\ LT.\ s = (ST,\ LT) \wedge length\ ST < mxs \wedge P \vdash C\ sees\ F,Static{:}vT\ in\ D)$
  by (*rule length-cases2* [*of - s*]) *auto*

lemma *appPutField*[*simp*]:
$app_i\ (Putfield\ F\ C,P,pc,mxs,T_r,s) =$
 $(\exists\ vT\ vT'\ oT\ ST\ LT.\ s = (vT\#oT\#ST,\ LT) \wedge$
  $P \vdash C\ sees\ F,NonStatic{:}vT'\ in\ C \wedge P \vdash oT \leq (Class\ C) \wedge P \vdash vT \leq vT')$
  by (*rule length-cases4* [*of - s*], *auto*)

lemma *appPutstatic*[*simp*]:
$app_i\ (Putstatic\ C\ F\ D,P,pc,mxs,T_r,s) =$
 $(\exists\ vT\ vT'\ ST\ LT.\ s = (vT\#ST,\ LT) \wedge$
  $P \vdash C\ sees\ F,Static{:}vT'\ in\ D \wedge P \vdash vT \leq vT')$
  by (*rule length-cases4* [*of - s*], *auto*)

lemma *appNew*[*simp*]:
app_i (*New C*,*P*,*pc*,*mxs*,T_r,*s*) =
($\exists$ *ST LT*. *s*=(*ST*,*LT*) $\wedge$ *is-class P C* $\wedge$ *length ST* < *mxs*)
by (*cases s*, *simp*)

lemma *appCheckcast*[*simp*]:
app_i (*Checkcast C*,*P*,*pc*,*mxs*,T_r,*s*) =
($\exists$ *T ST LT*. *s* = (*T*#*ST*,*LT*) $\wedge$ *is-class P C* $\wedge$ *is-refT T*)
by (*cases s*, *cases fst s*, *simp add*: *app-def*) (*cases hd* (*fst s*), *auto*)

lemma app_i*Pop*[*simp*]:
app_i (*Pop*,*P*,*pc*,*mxs*,T_r,*s*) = ($\exists$ *ts ST LT*. *s* = (*ts*#*ST*,*LT*))
by (*rule length-cases2*, *auto*)

lemma *appIAdd*[*simp*]:
app_i (*IAdd*,*P*,*pc*,*mxs*,T_r,*s*) = ($\exists$ *ST LT*. *s* = (*Integer*#*Integer*#*ST*,*LT*))

lemma *appIfFalse* [*simp*]:
app_i (*IfFalse b*,*P*,*pc*,*mxs*,T_r,*s*) =
($\exists$ *ST LT*. *s* = (*Boolean*#*ST*,*LT*) $\wedge$ *0* $\leq$ *int pc* + *b*)
lemma *appCmpEq*[*simp*]:
app_i (*CmpEq*,*P*,*pc*,*mxs*,T_r,*s*) =
($\exists$ T_1 T_2 *ST LT*. *s* = (T_1#T_2#*ST*,*LT*) $\wedge$ ($\neg$*is-refT* T_1 $\wedge$ T_2 = T_1 $\vee$ *is-refT* T_1 $\wedge$ *is-refT* T_2))
by (*rule length-cases4*, *auto*)

lemma *appReturn*[*simp*]:
app_i (*Return*,*P*,*pc*,*mxs*,T_r,*s*) = ($\exists$ *T ST LT*. *s* = (*T*#*ST*,*LT*) $\wedge$ *P* $\vdash$ *T* $\leq$ T_r)
by (*rule length-cases2*, *auto*)

lemma *appThrow*[*simp*]:
app_i (*Throw*,*P*,*pc*,*mxs*,T_r,*s*) = ($\exists$ *T ST LT*. *s*=(*T*#*ST*,*LT*) $\wedge$ *is-refT T*)
by (*rule length-cases2*, *auto*)

lemma *effNone*:
(*pc'*, *s'*) $\in$ *set* (*eff i P pc et None*) $\Longrightarrow$ *s'* = *None*
by (*auto simp add*: *eff-def xcpt-eff-def norm-eff-def*)

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'*
by (*unfold relevant-entries-def*) *simp*

lemma *xcpt-app-append* [*iff*]:
xcpt-app i P pc mxs (*xt*@*xt'*) τ = (*xcpt-app i P pc mxs xt* τ $\wedge$ *xcpt-app i P pc mxs xt'* τ)
by (*unfold xcpt-app-def*) *fastforce*

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'*
by (*unfold xcpt-eff-def*, *cases* τ) *simp*

lemma *app-append* [*simp*]:
app i P pc T mxs mpc (*xt*@*xt'*) τ = (*app i P pc T mxs mpc xt* τ $\wedge$ *app i P pc T mxs mpc xt'* τ)
by (*unfold app-def eff-def*) *auto*

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*: $P \vdash \tau \leq_i \tau'$
shows $app_i\ (i,P,mxs,mpc,rT,\tau') \Longrightarrow app_i\ (i,P,mxs,mpc,rT,\tau)$
lemma *succs-mono*:
assumes *wf*: *wf-prog p P* **and** app_i: $app_i\ (i,P,mxs,mpc,rT,\tau')$
shows $P \vdash \tau \leq_i \tau' \Longrightarrow set\ (succs\ i\ \tau\ pc) \subseteq set\ (succs\ i\ \tau'\ pc)$

lemma *app-mono*:
assumes *wf*: *wf-prog p P*
assumes *less'*: $P \vdash \tau \leq' \tau'$
shows $app\ i\ P\ m\ rT\ pc\ mpc\ xt\ \tau' \Longrightarrow app\ i\ P\ m\ rT\ pc\ mpc\ xt\ \tau$

lemma eff_i*-mono*:
assumes *wf*: *wf-prog p P*
assumes *less*: $P \vdash \tau \leq_i \tau'$
assumes app_i: $app\ i\ P\ m\ rT\ pc\ mpc\ xt\ (Some\ \tau')$
assumes *succs*: $succs\ i\ \tau\ pc \neq []\ \ succs\ i\ \tau'\ pc \neq []$
shows $P \vdash eff_i\ (i,P,\tau) \leq_i eff_i\ (i,P,\tau')$
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:
$check\text{-}types :: {'m}\ prog \Rightarrow nat \Rightarrow nat \Rightarrow ty_i{'}\ err\ list \Rightarrow bool$
where
$check\text{-}types\ P\ mxs\ mxl\ \tau s \equiv set\ \tau s \subseteq states\ P\ mxs\ mxl$

— An instruction is welltyped if it is applicable and its effect
— is compatible with the type at all successor instructions:
definition
$wt\text{-}instr :: [{'m}\ prog,ty,nat,pc,ex\text{-}table,instr,pc,ty_m] \Rightarrow bool$
(‹-,-,-,-,- ⊢ -,- :: -› [60,0,0,0,0,0,0,61] 60)
where
$P,T,mxs,mpc,xt \vdash i,pc :: \tau s \equiv$
$app\ i\ P\ mxs\ T\ pc\ mpc\ xt\ (\tau s!pc) \wedge$
$(\forall (pc',\tau') \in set\ (eff\ i\ P\ pc\ xt\ (\tau s!pc)).\ P \vdash \tau' \leq' \tau s!pc')$

— The type at $pc=0$ conforms to the method calling convention:
definition *wt-start* :: $['m\ prog,cname,staticb,ty\ list,nat,ty_m] \Rightarrow bool$
where
$wt\text{-}start\ P\ C\ b\ Ts\ mxl_0\ \tau s \equiv$
$case\ b\ of\ NonStatic \Rightarrow P \vdash Some\ ([],OK\ (Class\ C)\#map\ OK\ Ts@replicate\ mxl_0\ Err) \leq'\ \tau s!0$
$\quad |\ Static \Rightarrow\ P \vdash Some\ ([],map\ OK\ Ts@replicate\ mxl_0\ Err) \leq'\ \tau s!0$

— 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* :: $['m\ prog,cname,staticb,ty\ list,ty,nat,nat,instr\ list,$
$\quad ex\text{-}table,ty_m] \Rightarrow bool$
where
$wt\text{-}method\ P\ C\ b\ Ts\ T_r\ mxs\ mxl_0\ is\ xt\ \tau s \equiv (b = Static \lor b = NonStatic) \land$
$0 < size\ is \land size\ \tau s = size\ is \land$
$check\text{-}types\ P\ mxs\ ((case\ b\ of\ Static \Rightarrow 0\ |\ NonStatic \Rightarrow 1)+size\ Ts+mxl_0)\ (map\ OK\ \tau s) \land$
$wt\text{-}start\ P\ C\ b\ Ts\ mxl_0\ \tau s \land$
$(\forall pc < size\ is.\ P,T_r,mxs,size\ is,xt \vdash is!pc,pc :: \tau s)$

— A program is welltyped if it is wellformed and all methods are welltyped
definition *wf-jvm-prog-phi* :: $ty_P \Rightarrow jvm\text{-}prog \Rightarrow bool$ (‹*wf′-jvm′-prog*$_{\_}$›)
where
$wf\text{-}jvm\text{-}prog_\Phi \equiv$
$\quad wf\text{-}prog\ (\lambda P\ C\ (M,b,Ts,T_r,(mxs,mxl_0,is,xt)).$
$\quad\quad wt\text{-}method\ P\ C\ b\ Ts\ T_r\ mxs\ mxl_0\ is\ xt\ (\Phi\ C\ M))$

definition *wf-jvm-prog* :: $jvm\text{-}prog \Rightarrow bool$
where
$wf\text{-}jvm\text{-}prog\ P \equiv \exists \Phi.\ wf\text{-}jvm\text{-}prog_\Phi\ P$

lemma *wt-jvm-progD*:
$wf\text{-}jvm\text{-}prog_\Phi\ P \Longrightarrow \exists wt.\ wf\text{-}prog\ wt\ P$
lemma *wt-jvm-prog-impl-wt-instr*:
assumes *wf*: $wf\text{-}jvm\text{-}prog_\Phi\ P$ **and**
sees: $P \vdash C\ sees\ M,b{:}Ts \rightarrow T = (mxs,mxl_0,ins,xt)\ in\ C$ **and**
pc: $pc < size\ ins$
shows $P,T,mxs,size\ ins,xt \vdash ins!pc,pc :: \Phi\ C\ M$
lemma *wt-jvm-prog-impl-wt-start*:
assumes *wf*: $wf\text{-}jvm\text{-}prog_\Phi\ P$ **and**
sees: $P \vdash C\ sees\ M,b{:}Ts \rightarrow T = (mxs,mxl_0,ins,xt)\ in\ C$
shows $0 < size\ ins \land wt\text{-}start\ P\ C\ b\ Ts\ mxl_0\ (\Phi\ C\ M)$
lemma *wf-jvm-prog-nclinit*:
assumes *wtp*: $wf\text{-}jvm\text{-}prog_\Phi\ P$
and *meth*: $P \vdash C\ sees\ M,\ b :\ Ts \rightarrow T = (mxs,\ mxl_0,\ ins,\ xt)\ in\ D$
and *wt*: $P,T,mxs,size\ ins,xt \vdash ins!pc,pc :: \Phi\ C\ M$
and *pc*: $pc < length\ ins$ **and** Φ: $\Phi\ C\ M\ !\ pc = Some(ST,LT)$
and *ins*: $ins\ !\ pc = Invokestatic\ C_0\ M_0\ n$
shows $M_0 \neq clinit$
using *assms* **by**(*simp add*: *wf-jvm-prog-phi-def wt-instr-def app-def*)

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* ≡ *size is*
  assumes *staticb*: *b* = *Static* ∨ *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* ⊆ *types P*

  fixes *first* :: ty_i*′* **and** *start*
  defines [*simp*]:
  first ≡ *Some* ([],(*case b of Static* ⇒ [] | *NonStatic* ⇒ [*OK* (*Class C*)]) @ *map OK Ts* @ *replicate* mxl_0 *Err*)
  defines [*simp*]:
  start ≡ (*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* ∨ *b* = *NonStatic*
    and *wf-prog p P*
    and *is-class P C*
    and *set Ts* ⊆ *types P*
   shows *start-context P b Ts p C*
  using *start-context.intro*[*OF JVM-sl.intro*] *start-context-axioms-def assms* **by** *auto*

2.12.1 Connecting JVM and Framework

lemma (**in** *start-context*) *semi*: *semilat* $(A,\ r,\ f)$
apply (*insert semilat-JVM*[*OF wf*])
apply (*unfold A-def r-def f-def JVM-SemiType.le-def JVM-SemiType.sup-def states-def*)
apply *auto*
done

lemma (**in** *JVM-sl*) *step-def-exec*: $step \equiv exec\ P\ mxs\ T_r\ xt\ is$
by (*simp add*: *exec-def*)

lemma *special-ex-swap-lemma* [*iff*]:
$(?\ X.\ (?\ n.\ X = A\ n\ \&\ P\ n)\ \&\ Q\ X) = (?\ n.\ Q(A\ n)\ \&\ P\ n)$
by *blast*

lemma *ex-in-list* [*iff*]:
$(\exists\, n.\ ST \in nlists\ n\ A \wedge n \le mxs) = (set\ ST \subseteq A \wedge size\ ST \le mxs)$
by (*unfold nlists-def*) *auto*

lemma *singleton-nlists*:
$(\exists\, n.\ [Class\ C] \in nlists\ n\ (types\ P) \wedge n \le mxs) = (is\text{-}class\ P\ C \wedge 0 < mxs)$
by *auto*

lemma *set-drop-subset*:
$set\ xs \subseteq A \Longrightarrow set\ (drop\ n\ xs) \subseteq A$
by (*auto dest*: *in-set-dropD*)

lemma *Suc-minus-minus-le*:
$n < mxs \Longrightarrow Suc\ (n - (n - b)) \le mxs$
by *arith*

lemma *in-nlistsE*:
$[\![\ xs \in nlists\ n\ A;\ [\![size\ xs = n;\ set\ xs \subseteq A]\!] \Longrightarrow P\ ]\!] \Longrightarrow P$
by (*unfold nlists-def*) *blast*

declare *is-relevant-entry-def* [*simp*]
declare *set-drop-subset* [*simp*]

theorem (**in** *start-context*) *exec-pres-type*:
pres-type step (*size is*) *A*
declare *is-relevant-entry-def* [*simp del*]
declare *set-drop-subset* [*simp del*]

lemma *lesubstep-type-simple*:
$xs\ [\sqsubseteq_{Product.le\ (=)\ r}]\ ys \Longrightarrow set\ xs\ \{\sqsubseteq_r\}\ set\ ys$
declare *is-relevant-entry-def* [*simp del*]

lemma *conjI2*: $[\![\ A;\ A \Longrightarrow B\ ]\!] \Longrightarrow A \wedge B$ **by** *blast*

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*)
lemma (**in** *JVM-sl*) *bounded-step*: *bounded step* (*size is*)
theorem (**in** *JVM-sl*) *step-mono*:
  wf-prog wf-mb P $\Longrightarrow$ *mono r step* (*size is*) *A*

lemma (**in** *start-context*) *first-in-A* [*iff*]: *OK first* $\in$ *A*
  using *Ts C* **by** (*cases b*; *force intro*!: *nlists-appendI simp add*: *JVM-states-unfold*)

lemma (**in** *JVM-sl*) *wt-method-def2*:
  wt-method P C′ b Ts T_r *mxs* mxl_0 *is xt* τs =
  (*is* $\neq$ [] $\wedge$
   (*b* = *Static* $\vee$ *b* = *NonStatic*) $\wedge$
   size τs = *size is* $\wedge$
   OK ‘ *set* $\tau s \subseteq$ *states P mxs mxl* $\wedge$
   wt-start P C′ b Ts mxl_0 τs $\wedge$
   wt-app-eff (*sup-state-opt P*) *app eff* τs)

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-jvm-prog*$_k$ :: *jvm-prog* ⇒ *bool*
where
  wf-jvm-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*]:
$x\#xs\ [\sqsubset_r]\ y\#ys = (x \sqsubset_r y \wedge xs\ [\sqsubseteq_r]\ ys \vee x = y \wedge xs\ [\sqsubset_r]\ ys)$
unfolding *lesssub-def*
by (*metis Cons-le-Cons JVM-le-Err-conv lesub-def list.inject r-def sup-state-opt-err*)

lemma *acc-le-listI3* [*intro*!]:
$acc\ r \Longrightarrow acc\ (Listn.le\ r)$
unfolding *acc-def*
apply (*subgoal-tac*
$wf(UN\ n.\ \{(ys,xs).\ size\ xs = n \wedge size\ ys = n \wedge xs <\text{-}(Listn.le\ r)\ ys\})$)
apply (*erule wf-subset*)
apply (*blast intro*: *lesssub-lengthD*)
apply (*rule wf-UN*)
prefer *2*
apply *force*
apply (*rename-tac n*)
apply (*induct-tac n*)
apply (*simp add*: *lesssub-def cong*: *conj-cong*)
apply (*rename-tac k*)
apply (*simp add*: *wf-eq-minimal del*: *r-def f-def step-def A-def*)
apply (*simp* (*no-asm*) *add*: *length-Suc-conv cong*: *conj-cong del*: *r-def f-def step-def A-def*)
apply *clarify*
apply (*rename-tac M m*)
apply (*case-tac* $\exists x\ xs.\ size\ xs = k \wedge x\#xs \in M$)
prefer *2*
apply *blast*
apply (*erule-tac* $x = \{a.\ \exists xs.\ size\ xs = k \wedge a\#xs{:}M\}$ **in** *allE*)
apply (*erule impE*)
apply *blast*
apply (*thin-tac* $\exists x\ xs.\ P\ x\ xs$ **for** P)
apply *clarify*
apply (*rename-tac maxA xs*)
apply (*erule-tac* $x = \{ys.\ size\ ys = size\ xs \wedge maxA\#ys \in M\}$ **in** *allE*)
apply (*erule impE*)
apply *blast*
apply *clarify*
using *Cons-less-Conss3* **by** *blast*

lemma *wf-jvm*: $wf\ \{(ss', ss).\ ss\ [\sqsubset_r]\ ss'\}$
using *acc-le-listI3 acc-JVM* [*OF wf*] **by** (*simp add*: *acc-def r-def*)

lemma *iter-properties-bv*[*rule-format*]:
shows $[\![\ \forall p{\in}w0.\ p < n;\ ss0 \in nlists\ n\ A;\ \forall p{<}n.\ p \notin w0 \longrightarrow stable\ r\ step\ ss0\ p\ ]\!] \Longrightarrow$
$iter\ f\ step\ ss0\ w0 = (ss',w') \longrightarrow$
$ss' \in nlists\ n\ A \wedge stables\ r\ step\ ss' \wedge ss0\ [\sqsubseteq_r]\ ss' \wedge$
$(\forall ts{\in}nlists\ n\ A.\ ss0\ [\sqsubseteq_r]\ ts \wedge stables\ r\ step\ ts \longrightarrow ss'\ [\sqsubseteq_r]\ ts)$

lemma *kildall-properties-bv*:
shows $[\![\ ss0 \in nlists\ n\ A\ ]\!] \Longrightarrow$
$kildall\ r\ f\ step\ ss0 \in nlists\ n\ A \wedge$
$stables\ r\ step\ (kildall\ r\ f\ step\ ss0) \wedge$

$ss0\ [\sqsubseteq_r]\ kildall\ r\ f\ step\ ss0\ \wedge$
$(\forall\ ts{\in}nlists\ n\ A.\ ss0\ [\sqsubseteq_r]\ ts \wedge stables\ r\ step\ ts \longrightarrow$
$\quad kildall\ r\ f\ step\ ss0\ [\sqsubseteq_r]\ ts)$

2.14 LBV for the JVM

theory *LBVJVM*
imports *Jinja.Abstract-BV TF-JVM*
begin

type-synonym $prog\text{-}cert = cname \Rightarrow mname \Rightarrow ty_i{}'\ err\ list$

definition $check\text{-}cert :: jvm\text{-}prog \Rightarrow nat \Rightarrow nat \Rightarrow nat \Rightarrow ty_i{}'\ err\ list \Rightarrow bool$
where
$check\text{-}cert\ P\ mxs\ mxl\ n\ cert \equiv check\text{-}types\ P\ mxs\ mxl\ cert \wedge size\ cert = n{+}1 \wedge$
$\quad (\forall\ i{<}n.\ cert!i \neq Err) \wedge cert!n = OK\ None$

definition $lbvjvm :: jvm\text{-}prog \Rightarrow nat \Rightarrow nat \Rightarrow ty \Rightarrow ex\text{-}table \Rightarrow$
$\quad ty_i{}'\ err\ list \Rightarrow instr\ list \Rightarrow ty_i{}'\ err \Rightarrow ty_i{}'\ err$
where
$lbvjvm\ P\ mxs\ maxr\ T_r\ et\ cert\ bs \equiv$
$wtl\text{-}inst\text{-}list\ bs\ cert\ (JVM\text{-}SemiType.sup\ P\ mxs\ maxr)\ (JVM\text{-}SemiType.le\ P\ mxs\ maxr)\ Err\ (OK\ None)\ (exec\ P\ mxs\ T_r\ et\ bs)\ 0$

definition $wt\text{-}lbv :: jvm\text{-}prog \Rightarrow cname \Rightarrow staticb \Rightarrow ty\ list \Rightarrow ty \Rightarrow nat \Rightarrow nat \Rightarrow$
$\quad ex\text{-}table \Rightarrow ty_i{}'\ err\ list \Rightarrow instr\ list \Rightarrow bool$
where
$wt\text{-}lbv\ P\ C\ b\ Ts\ T_r\ mxs\ mxl_0\ et\ cert\ ins \equiv (b = Static \vee b = NonStatic) \wedge$
$check\text{-}cert\ P\ mxs\ ((case\ b\ of\ Static \Rightarrow 0 \mid NonStatic \Rightarrow 1){+}size\ Ts{+}mxl_0)\ (size\ ins)\ cert \wedge$
$0 < size\ ins \wedge$
$(let\ start = Some\ ([],(case\ b\ of\ Static \Rightarrow [] \mid NonStatic \Rightarrow [OK\ (Class\ C)])$
$\quad @((map\ OK\ Ts))@(replicate\ mxl_0\ Err));$
$\quad result = lbvjvm\ P\ mxs\ ((case\ b\ of\ Static \Rightarrow 0 \mid NonStatic \Rightarrow 1){+}size\ Ts{+}mxl_0)\ T_r\ et\ cert\ ins\ (OK\ start)$
$\quad in\ result \neq Err)$

definition $wt\text{-}jvm\text{-}prog\text{-}lbv :: jvm\text{-}prog \Rightarrow prog\text{-}cert \Rightarrow bool$
where
$wt\text{-}jvm\text{-}prog\text{-}lbv\ P\ cert \equiv$
$wf\text{-}prog\ (\lambda P\ C\ (mn,b,Ts,T_r,(mxs,mxl_0,ins,et)).\ wt\text{-}lbv\ P\ C\ b\ Ts\ T_r\ mxs\ mxl_0\ et\ (cert\ C\ mn)\ ins)\ P$

definition $mk\text{-}cert :: jvm\text{-}prog \Rightarrow nat \Rightarrow ty \Rightarrow ex\text{-}table \Rightarrow instr\ list$
$\quad \Rightarrow ty_m \Rightarrow ty_i{}'\ err\ list$
where
$mk\text{-}cert\ P\ mxs\ T_r\ et\ bs\ phi \equiv make\text{-}cert\ (exec\ P\ mxs\ T_r\ et\ bs)\ (map\ OK\ phi)\ (OK\ None)$

definition $prg\text{-}cert :: jvm\text{-}prog \Rightarrow ty_P \Rightarrow prog\text{-}cert$
where
$prg\text{-}cert\ P\ phi\ C\ mn \equiv let\ (C,b,Ts,T_r,(mxs,mxl_0,ins,et)) = method\ P\ C\ mn$
$\quad in\ mk\text{-}cert\ P\ mxs\ T_r\ et\ ins\ (phi\ C\ mn)$

lemma *check-certD* [*intro?*]:
$check\text{-}cert\ P\ mxs\ mxl\ n\ cert \Longrightarrow cert\text{-}ok\ cert\ n\ Err\ (OK\ None)\ (states\ P\ mxs\ mxl)$

by (*unfold cert-ok-def check-cert-def check-types-def*) *auto*

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-step r Err step* $\tau s \wedge$ *OK first* $\sqsubseteq_r$ $\tau s!0$

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* τs

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*

theorem *jvm-lbv-correct*:
wt-jvm-prog-lbv P Cert $\Longrightarrow$ *wf-jvm-prog P*
theorem *jvm-lbv-complete*:
assumes *wt*: *wf-jvm-prog*$_\Phi$ *P*
shows *wt-jvm-prog-lbv P* (*prg-cert P* Φ)
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* ⊢ *v* :≤⊤ *E* $\equiv$ *case E of Err* $\Rightarrow$ *True* | *OK T* $\Rightarrow$ *P*,*h* ⊢ *v* :≤ *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* ⊢ *vs* [:≤⊤] *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* ⊢ *C has F*,*Static*:*t in D*)) |
Called-context P C_0 (*Putstatic C F D*) = ((C_0=*D*) $\wedge$ ($\exists$ *t*. *P* ⊢ *C has F*,*Static*:*t in D*)) |
Called-context P C_0 (*Invokestatic C M n*)

= (∃ *Ts T m D.* (C_0=*D*) ∧ *P* ⊢ *C sees M*,*Static*:*Ts* → *T* = *m in D*) |
Called-context P - - = *False*

abbreviation *Called-set* :: *instr set* **where**
Called-set ≡ {*i.* ∃ *C. i* = *New C*} ∪ {*i.* ∃ *C M n. i* = *Invokestatic C M n*}
∪ {*i.* ∃ *C F D. i* = *Getstatic C F D*} ∪ {*i.* ∃ *C F D. i* = *Putstatic C F D*}

lemma *Called-context-Called-set*:
Called-context P D i ⟹ *i* ∈ *Called-set* **by**(*cases i*, *auto*)

fun *valid-ics* :: *jvm-prog* ⇒ *heap* ⇒ *sheap* ⇒ *cname* × *mname* × *pc* × *init-call-status* ⇒ *bool*
(‹-,-,- $\vdash_i$ -› [*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* [:$\leq_\top$] *LT* ∧ *pc* < *size is* ∧ *P*,*h*,*sh* $\vdash_i$ (*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* [:$\leq_\top$] *LT* ∧ *pc* < *size is* ∧ *P*,*h*,*sh* $\vdash_i$ (*C*,*M*,*pc*,*ics*)
by (*simp add*: *conf-f-def*)

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 $\preceq^*$ *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*$\vdash_s$ *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* :$\leq_\top$ *Err*
  by (*simp add*: *confT-def*)

lemma *confT-OK* [*iff*]: *P*,*h* ⊢ *x* :$\leq_\top$ *OK T* = (*P*,*h* ⊢ *x* :≤ *T*)
  by (*simp add*: *confT-def*)

lemma *confT-cases*:
  P,*h* ⊢ *x* :$\leq_\top$ *X* = (*X* = *Err* ∨ (∃ *T*. *X* = *OK T* ∧ *P*,*h* ⊢ *x* :≤ *T*))
  by (*cases X*) *auto*

lemma *confT-hext* [*intro?*, *trans*]:
  ⟦ *P*,*h* ⊢ *x* :$\leq_\top$ *T*; *h* ⊴ *h*′ ⟧ ⟹ *P*,*h*′ ⊢ *x* :$\leq_\top$ *T*
  by (*cases T*) (*blast intro*: *conf-hext*)+

lemma *confT-widen* [*intro?*, *trans*]:
  ⟦ *P*,*h* ⊢ *x* :$\leq_\top$ *T*; *P* ⊢ *T* $\leq_\top$ *T*′ ⟧ ⟹ *P*,*h* ⊢ *x* :$\leq_\top$ *T*′
  by (*cases T*′, *auto intro*: *conf-widen*)

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 $P,h \vdash (loc!n) :\leq T'$
lemma *confTs-hext* [*intro?*]:
$P,h \vdash loc\ [:\leq_\top]\ LT \Longrightarrow h \trianglelefteq h' \Longrightarrow P,h' \vdash loc\ [:\leq_\top]\ LT$
by (*fast elim*: *list-all2-mono confT-hext*)

lemma *confTs-widen* [*intro?*, *trans*]:
$P,h \vdash loc\ [:\leq_\top]\ LT \Longrightarrow P \vdash LT\ [\leq_\top]\ LT' \Longrightarrow P,h \vdash loc\ [:\leq_\top]\ LT'$
by (*rule list-all2-trans*, *rule confT-widen*)

lemma *confTs-map* [*iff*]:
$\bigwedge vs.\ (P,h \vdash vs\ [:\leq_\top]\ map\ OK\ Ts) = (P,h \vdash vs\ [:\leq]\ Ts)$
by (*induct Ts*) (*auto simp*: *list-all2-Cons2*)

lemma *reg-widen-Err* [*iff*]:
$\bigwedge LT.\ (P \vdash replicate\ n\ Err\ [\leq_\top]\ LT) = (LT = replicate\ n\ Err)$
by (*induct n*) (*auto simp*: *list-all2-Cons1*)

lemma *confTs-Err* [*iff*]:
$P,h \vdash replicate\ n\ v\ [:\leq_\top]\ replicate\ n\ Err$
by (*induct n*) *auto*

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 $P,h,sh(C' \mapsto (sfs, i')) \vdash_i (C, M, pc, ics)$
using *assms* **by**(*cases ics*; *clarsimp simp*: *fun-upd-apply*) *fastforce*

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*)
using *assms* **by**(*auto simp*: *conf-f-def2*)

lemma *conf-f-shupd*:
assumes *conf-f P h sh* (*ST*,*LT*) *ins f*
and *i* = *Processing*
$\vee$ (*distinct* (*C*#*ics-classes* (*ics-of f*)) $\wedge$ (*curr-method f* = *clinit* $\longrightarrow$ *C* $\neq$ *curr-class f*))
shows *conf-f P h* (*sh*(*C* $\mapsto$ (*sfs*, *i*))) (*ST*,*LT*) *ins f*
using *assms*
by(*cases f*, *cases ics-of f*; *clarsimp simp*: *conf-f-def2 fun-upd-apply*) *fastforce+*

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*

using *assms*
by(*cases f*, *cases ics-of f*; *clarsimp simp*: *conf-f-def2 fun-upd-apply*) *fastforce*+

2.15.6 correct-frames

lemmas [*simp del*] = *fun-upd-apply*

lemma *conf-fs-hext*:
$\bigwedge C\ M\ n\ T_r.$
$[\![$ *conf-fs* $P\ h\ sh\ \Phi\ C\ M\ n\ T_r$ *frs*; $h \trianglelefteq h'$ $]\!] \Longrightarrow$ *conf-fs* $P\ h'\ sh\ \Phi\ C\ M\ n\ T_r$ *frs*

lemma *conf-fs-shupd*:
assumes *conf-fs* $P\ h\ sh\ \Phi\ C_0\ M\ n\ T$ *frs*
and *dist*: *distinct* (C#*clinit-classes frs*)
shows *conf-fs* $P\ h\ (sh(C \mapsto (sfs,\ i)))\ \Phi\ C_0\ M\ n\ T$ *frs*
using *assms* **proof**(*induct frs arbitrary*: $C_0\ C\ M\ n\ T$)
case (*Cons* f' *frs'*)
then obtain *stk' loc'* $C'\ M'$ *pc' ics'* **where** f': $f' = (stk',loc',C',M',pc',ics')$ **by**(*cases* f')
with *assms Cons* **obtain** *ST LT b Ts T1 mxs* mxl_0 *ins xt* **where**
ty: $\Phi\ C'\ M'$! pc' = *Some* (*ST*,*LT*) **and**
meth: $P \vdash C'$ *sees* M',*b*:*Ts* $\rightarrow$ *T1* = (*mxs*,mxl_0,*ins*,*xt*) *in* C' **and**
conf: *conf-f P h sh* (*ST*, *LT*) *ins* f' **and**
confs: *conf-fs* $P\ h\ sh\ \Phi\ C'\ M'$ (*size Ts*) *T1 frs'* **by** *clarsimp*

from f' *Cons.prems*(*2*) **have**
distinct (C#*ics-classes* (*ics-of* f')) $\wedge$ (*curr-method* f' = *clinit* $\longrightarrow C \neq$ *curr-class* f')
by *fastforce*
with *conf-f-shupd*[**where** C=C, *OF conf*] **have**
conf': *conf-f* $P\ h\ (sh(C \mapsto (sfs,\ i)))$ (*ST*, *LT*) *ins* f' **by** *simp*

from *Cons.prems*(*2*) **have** *dist'*: *distinct* (C # *clinit-classes frs'*)
by(*auto simp*: *distinct-length-2-or-more*)
from *Cons.hyps*[*OF confs dist'*] **have**
confs': *conf-fs* $P\ h\ (sh(C \mapsto (sfs,\ i)))\ \Phi\ C'\ M'$ (*length Ts*) *T1 frs'* **by** *simp*

from *conf' confs' ty meth* f' *Cons.prems* **show** *?case* **by**(*fastforce dest*: *sees-method-fun*)
qed(*simp*)

lemma *conf-fs-shupd'*:
assumes *conf-fs* $P\ h\ sh\ \Phi\ C_0\ M\ n\ T$ *frs*
and *shC*: $sh\ C$ = *Some*(*sfs*,*i*)
shows *conf-fs* $P\ h\ (sh(C \mapsto (sfs',\ i)))\ \Phi\ C_0\ M\ n\ T$ *frs*
using *assms* **proof**(*induct frs arbitrary*: $C_0\ C\ M\ n\ T$ *sfs i sfs'*)
case (*Cons* f' *frs'*)
then obtain *stk' loc'* $C'\ M'$ *pc' ics'* **where** f': $f' = (stk',loc',C',M',pc',ics')$ **by**(*cases* f')
with *assms Cons* **obtain** *ST LT b Ts T1 mxs* mxl_0 *ins xt* **where**
ty: $\Phi\ C'\ M'$! pc' = *Some* (*ST*,*LT*) **and**
meth: $P \vdash C'$ *sees* M',*b*:*Ts* $\rightarrow$ *T1* = (*mxs*,mxl_0,*ins*,*xt*) *in* C' **and**
conf: *conf-f P h sh* (*ST*, *LT*) *ins* f' **and**
confs: *conf-fs* $P\ h\ sh\ \Phi\ C'\ M'$ (*size Ts*) *T1 frs'* **and**
shC': $sh\ C$ = *Some*(*sfs*,*i*) **by** *clarsimp*

have *conf'*: *conf-f* $P\ h\ (sh(C \mapsto (sfs',\ i)))$ (*ST*, *LT*) *ins* f' **by**(*rule conf-f-shupd'*[*OF conf shC'*])

from *Cons.hyps*[*OF confs shC'*] **have**
confs': *conf-fs P h* (*sh*(*C* $\mapsto$ (*sfs'*, *i*))) Φ *C' M'* (*length Ts*) *T1 frs'* **by** *simp*

from *conf' confs' ty meth f' Cons.prems* **show** *?case* **by**(*fastforce dest*: *sees-method-fun*)
qed(*simp*)

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 −
from *assms* **have** *dist*: *distinct-clinit* (*f#frs*)
by(*cases curr-method f* = *clinit*, *auto simp*: *conf-clinit-def*)
then have *dist'*: *distinct-clinit frs* **by**(*simp add*: *distinct-clinit-def*)

with *assms* **show** *?thesis* **by**(*cases frs*; *fastforce simp*: *conf-clinit-def*)
qed

lemma *conf-clinit-Cons-Cons*:
conf-clinit P sh (*f'#f#frs*) $\Longrightarrow$ *conf-clinit P sh* (*f'#frs*)
by(*auto simp*: *conf-clinit-def distinct-clinit-def*)

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*)
using *assms* **by**(*cases M* = *clinit*, *simp-all add*: *conf-clinit-def distinct-clinit-def*)

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*)
using *assms* **by**(*cases M* = *clinit*, *simp-all add*: *conf-clinit-def distinct-clinit-def*)

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*)
by(*simp add*: *conf-clinit-def distinct-clinit-def*)

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*)
by(*simp add*: *conf-clinit-def distinct-clinit-def*)

lemma *conf-clinit-Called*:
$[\![$ *conf-clinit P sh* ((*stk*, *loc*, *C*, *M*, *pc*, *Called* (*C'#Cs*)) # *frs*);
P $\vdash$ *C' sees clinit*,*Static*: [] $\rightarrow$ *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*)
by(*simp add*: *conf-clinit-def distinct-clinit-def*)

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 −
from *nclinit*

have *clinit-classes* ((*stk*, *loc*, *C*, *M*, *pc*, *No-ics*) # *frs*) = *clinit-classes frs* **by** *simp*
with *assms* **show** *?thesis* **by**(*simp add*: *conf-clinit-def distinct-clinit-def*)
qed

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*)
using *assms conf-clinit-Cons-nclinit conf-clinit-diff'* **by** *auto*

lemma *conf-clinit-nProc-dist*:
assumes *conf-clinit P sh frs*
and ∀ *sfs*. *sh C* ≠ *Some*(*sfs*,*Processing*)
shows *distinct* (*C* # *clinit-classes frs*)
using *assms* **by**(*auto simp*: *conf-clinit-def distinct-clinit-def*)

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*
using *assms* **by**(*simp add*: *conf-clinit-def fun-upd-apply*)

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*
using *assms* **by**(*fastforce simp*: *conf-clinit-def fun-upd-apply*)

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*)
using *assms* **by**(*clarsimp simp*: *conf-clinit-def fun-upd-apply distinct-clinit-def*)

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*)
using *assms* **by**(*clarsimp simp*: *conf-clinit-def fun-upd-apply distinct-clinit-def*)

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 −
from *cr* **have** *dist*: *conf-clinit P sh* (*f*#*frs*)
by(*simp add*: *correct-state-def*)
then have *conf-clinit P sh frs* **by**(*rule conf-clinit-Cons*)

with *cr* **show** *?thesis* **by**(*cases frs*; *fastforce simp*: *correct-state-def*)

qed

lemma *correct-state-shupd*:
assumes *cs*: P,Φ |− (xp,h,frs,sh) [*ok*] **and** *shC*: $sh\ C = Some(sfs,i)$
and *dist*: *distinct* ($C\#clinit\text{-}classes\ frs$)
shows P,Φ |− ($xp,h,frs,sh(C \mapsto (sfs,\ i'))$) [*ok*]
using *assms*
proof(*cases xp*)
 case *None* **with** *assms* **show** *?thesis*
 proof(*cases frs*)
  case (*Cons f′ frs′*)
  let $?sh = sh(C \mapsto (sfs,\ i'))$

  obtain *stk′ loc′ C′ M′ pc′ ics′* **where** f': $f' = (stk',loc',C',M',pc',ics')$ **by**(*cases f′*)
  with *cs Cons None* **obtain** *b Ts T mxs* mxl_0 *ins xt ST LT* **where**
   meth: $P \vdash C'$ *sees* $M',b{:}Ts{\to}T = (mxs,mxl_0,ins,xt)$ *in* C'
  and *ty*: $\Phi\ C'\ M'\ !\ pc' = Some\ (ST,LT)$ **and** *conf*: *conf-f P h sh* (ST,LT) *ins f′*
  and *confs*: *conf-fs P h sh* Φ *C′ M′* (*size Ts*) *T frs′*
  and *confc*: *conf-clinit P sh frs*
  and *h-ok*: $P\vdash h\surd$ **and** *sh-ok*: $P,h \vdash_s sh\ \surd$
  by(*auto simp*: *correct-state-def*)

  from *Cons dist* **have** *dist′*: *distinct* ($C\#clinit\text{-}classes\ frs'$)
  by(*auto simp*: *distinct-length-2-or-more*)

  from *shconf-upd-obj*[*OF sh-ok shconfD*[*OF sh-ok shC*]] **have** *sh-ok′*: $P,h \vdash_s\ ?sh\ \surd$
   by *simp*

  from *conf f′ valid-ics-shupd Cons dist* **have** *conf′*: *conf-f P h ?sh* (ST,LT) *ins f′*
  by(*auto simp*: *conf-f-def2 fun-upd-apply*)
  have *confs′*: *conf-fs P h ?sh* Φ *C′ M′* (*size Ts*) *T frs′* **by**(*rule conf-fs-shupd*[*OF confs dist′*])

  have *confc′*: *conf-clinit P ?sh frs* **by**(*rule conf-clinit-shupd*[*OF confc dist*])

  with *h-ok sh-ok′ meth ty conf′ confs′ f′ Cons None* **show** *?thesis*
  by(*fastforce simp*: *correct-state-def*)
 qed(*simp add*: *correct-state-def*)
qed(*simp add*: *correct-state-def*)

lemma *correct-state-Throwing-ex*:
assumes *correct*: $P,\Phi \vdash (xp,h,(stk,loc,C,M,pc,ics)\#frs,sh)\surd$
shows $\bigwedge Cs\ a.\ ics = Throwing\ Cs\ a \Longrightarrow \exists\, obj.\ h\ a = Some\ obj$
using *correct* **by**(*clarsimp simp*: *correct-state-def conf-f-def*)

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
by(*unfold err-def*) *auto*

lemma *opt-mono*: $A \subseteq B \Longrightarrow$ *opt* $A \subseteq$ *opt* B
by(*unfold opt-def*) *auto*

— adding a class in the simplest way
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-fields FDTs* **and** *nc*: ¬*is-class P C*
shows *class-add P* (*C*, *cdec*) $\vdash D$ *has-fields FDTs*
using *assms*
proof(*induct rule*: *Fields.induct*)
  case (*has-fields-Object D fs ms FDTs*)
  from *has-fields-is-class-Object*[*OF fs*] *nc* **have** $C \neq$ *Object* **by** *fast*
  with *has-fields-Object* **show** *?case*
   by(*auto simp*: *class-def fun-upd-apply intro*!: *TypeRel.has-fields-Object*)
next
  case *rec*: (*has-fields-rec C1 D fs ms FDTs FDTs′*)
  with *has-fields-is-class* **have** [*simp*]: $D \neq C$ **by** *auto*
  with *rec* **have** *C1* $\neq C$ **by**(*clarsimp simp*: *is-class-def*)
  with *rec* **show** *?case*
   by(*auto simp*: *class-def fun-upd-apply intro*: *TypeRel.has-fields-rec*)
qed

lemma *class-add-has-fields-rev*:
⟦ *class-add P* (*C*, *cdec*) $\vdash D$ *has-fields FDTs*; $\neg P \vdash D \preceq^* C$ ⟧
$\Longrightarrow P \vdash D$ *has-fields FDTs*
proof(*induct rule*: *Fields.induct*)
  case (*has-fields-Object D fs ms FDTs*)
  then show *?case* **by**(*auto simp*: *class-def fun-upd-apply intro*!: *TypeRel.has-fields-Object*)
next
  case *rec*: (*has-fields-rec C1 D fs ms FDTs FDTs′*)
  then have *sub1*: $P \vdash C1 \prec^1 D$
   by(*auto simp*: *class-def fun-upd-apply intro*!: *subcls1I split*: *if-split-asm*)
  with *rec.prems* **have** *cls*: $\neg P \vdash D \preceq^* C$ **by** (*meson converse-rtrancl-into-rtrancl*)
  with *cls rec* **show** *?case*
   by(*auto simp*: *class-def fun-upd-apply*
      intro: *TypeRel.has-fields-rec split*: *if-split-asm*)
qed

lemma *class-add-has-field*:
assumes $P \vdash C_0$ *has* F,*b*:*T in D* **and** ¬ *is-class P C*
shows *class-add P* (*C*, *cdec*) $\vdash C_0$ *has* F,*b*:*T in D*
using *assms* **by**(*auto simp*: *has-field-def dest*!: *class-add-has-fields*[*of P* C_0])

lemma *class-add-has-field-rev*:
assumes *has*: *class-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
using *assms* **by**(*auto simp*: *has-field-def dest*!: *class-add-has-fields-rev*)

lemma *class-add-sees-field*:
assumes $P \vdash C_0$ *sees* $F,b{:}T$ *in* D **and** $\neg$ *is-class* P C
shows *class-add* P $(C,$ *cdec*$) \vdash C_0$ *sees* $F,b{:}T$ *in* D
using *assms* **by**(*auto simp*: *sees-field-def dest*!: *class-add-has-fields*[*of* P C_0])

lemma *class-add-sees-field-rev*:
assumes *has*: *class-add* P $(C,$ *cdec*$) \vdash C_0$ *sees* $F,b{:}T$ *in* D
and *ncp*: $\bigwedge D'. P \vdash C_0 \preceq^* D' \Longrightarrow D' \neq C$
shows $P \vdash C_0$ *sees* $F,b{:}T$ *in* D
using *assms* **by**(*auto simp*: *sees-field-def dest*!: *class-add-has-fields-rev*)

lemma *class-add-field*:
assumes *fd*: $P \vdash C_0$ *sees* $F,b{:}T$ *in* D **and** $\neg$ *is-class* P C
shows *field* P C_0 F = *field* (*class-add* P $(C,$ *cdec*$))$ C_0 F
using *class-add-sees-field*[*OF assms*, *of cdec*] *fd* **by** *simp*

2.16.2 Methods

lemma *class-add-sees-methods*:
assumes *ms*: $P \vdash D$ *sees-methods Mm* **and** *nc*: $\neg$*is-class* P C
shows *class-add* P $(C,$ *cdec*$) \vdash D$ *sees-methods Mm*
using *assms*
proof(*induct rule*: *Methods.induct*)
  case (*sees-methods-Object D fs ms Mm*)
  from *sees-methods-is-class-Object*[*OF ms*] *nc* **have** $C \neq$ *Object* **by** *fast*
  with *sees-methods-Object* **show** *?case*
   by(*auto simp*: *class-def fun-upd-apply intro*!: *TypeRel.sees-methods-Object*)
next
  case *rec*: (*sees-methods-rec C1 D fs ms Mm Mm'*)
  with *sees-methods-is-class* **have** [*simp*]: $D \neq C$ **by** *auto*
  with *rec* **have** *C1* $\neq C$ **by**(*clarsimp simp*: *is-class-def*)
  with *rec* **show** *?case*
   by(*auto simp*: *class-def fun-upd-apply intro*: *TypeRel.sees-methods-rec*)
qed

lemma *class-add-sees-methods-rev*:
$[\![$ *class-add* P $(C,$ *cdec*$) \vdash D$ *sees-methods Mm*;
   $\bigwedge D'. P \vdash D \preceq^* D' \Longrightarrow D' \neq C$ $]\!]$
$\Longrightarrow P \vdash D$ *sees-methods Mm*
proof(*induct rule*: *Methods.induct*)
  case (*sees-methods-Object D fs ms Mm*)
  then show *?case*
   by(*auto simp*: *class-def fun-upd-apply intro*!: *TypeRel.sees-methods-Object*)
next
  case *rec*: (*sees-methods-rec C1 D fs ms Mm Mm'*)
  then have *sub1*: $P \vdash$ *C1* $\prec^1 D$
   by(*auto simp*: *class-def fun-upd-apply intro*!: *subcls1I*)
  have *cls*: $\bigwedge D'. P \vdash D \preceq^* D' \Longrightarrow D' \neq C$
  proof −
    fix D' **assume** $P \vdash D \preceq^* D'$

with *sub1* **have** $P \vdash C1 \preceq^* D'$ **by** *simp*
with *rec.prems* **show** $D' \neq C$ **by** *simp*
qed
with *cls rec* **show** *?case*
by(*auto simp*: *class-def fun-upd-apply intro*: *TypeRel.sees-methods-rec*)
qed

lemma *class-add-sees-methods-Obj*:
assumes $P \vdash$ *Object sees-methods Mm* **and** *nObj*: $C \neq$ *Object*
shows *class-add P* $(C,$ *cdec*$) \vdash$ *Object sees-methods Mm*
proof −
from *assms* **obtain** C' *fs ms* **where** *cls*: *class P Object* = *Some*(C',*fs*,*ms*)
by(*auto elim*!: *Methods.cases*)
with *nObj* **have** *cls'*: *class* (*class-add P* $(C,$ *cdec*)) *Object* = *Some*(C',*fs*,*ms*)
by(*simp add*: *class-def fun-upd-apply*)
from *assms cls* **have** *Mm* = *map-option* ($\lambda m.$ (m, *Object*)) $\circ$ *map-of ms* **by**(*auto elim*!: *Methods.cases*)
with *assms cls'* **show** *?thesis*
by(*auto simp*: *is-class-def fun-upd-apply intro*!: *sees-methods-Object*)
qed

lemma *class-add-sees-methods-rev-Obj*:
assumes *class-add P* $(C,$ *cdec*$) \vdash$ *Object sees-methods Mm* **and** *nObj*: $C \neq$ *Object*
shows $P \vdash$ *Object sees-methods Mm*
proof −
from *assms* **obtain** C' *fs ms* **where** *cls*: *class* (*class-add P* $(C,$ *cdec*)) *Object* = *Some*(C',*fs*,*ms*)
by(*auto elim*!: *Methods.cases*)
with *nObj* **have** *cls'*: *class P Object* = *Some*(C',*fs*,*ms*)
by(*simp add*: *class-def fun-upd-apply*)
from *assms cls* **have** *Mm* = *map-option* ($\lambda m.$ (m, *Object*)) $\circ$ *map-of ms* **by**(*auto elim*!: *Methods.cases*)
with *assms cls'* **show** *?thesis*
by(*auto simp*: *is-class-def fun-upd-apply intro*!: *sees-methods-Object*)
qed

lemma *class-add-sees-method*:
assumes $P \vdash C_0$ *sees* M_0, b : $Ts \rightarrow T = m$ *in* D **and** $\neg$ *is-class P C*
shows *class-add P* $(C,$ *cdec*$) \vdash C_0$ *sees* M_0, b : $Ts \rightarrow T = m$ *in* D
using *assms* **by**(*auto simp*: *Method-def dest*!: *class-add-sees-methods*[*of P* C_0])

lemma *class-add-method*:
assumes *md*: $P \vdash C_0$ *sees* M_0, b : $Ts \rightarrow T = m$ *in* D **and** $\neg$ *is-class P C*
shows *method P* C_0 M_0 = *method* (*class-add P* $(C,$ *cdec*)) C_0 M_0
using *class-add-sees-method*[*OF assms*, *of cdec*] *md* **by** *simp*

lemma *class-add-sees-method-rev*:
$[\![$ *class-add P* $(C,$ *cdec*$) \vdash C_0$ *sees* M_0, b : $Ts \rightarrow T = m$ *in* D;
$\neg P \vdash C_0 \preceq^* C$ $]\!]$
$\Longrightarrow P \vdash C_0$ *sees* M_0, b : $Ts \rightarrow T = m$ *in* D
by(*auto simp*: *Method-def dest*!: *class-add-sees-methods-rev*)

lemma *class-add-sees-method-Obj*:
$[\![$ $P \vdash$ *Object sees* M_0, b : $Ts \rightarrow T = m$ *in* D; $C \neq$ *Object* $]\!]$
$\Longrightarrow$ *class-add P* $(C,$ *cdec*$) \vdash$ *Object sees* M_0, b : $Ts \rightarrow T = m$ *in* D

by(*auto simp*: *Method-def dest*!: *class-add-sees-methods-Obj*[**where** *P*=*P*])

lemma *class-add-sees-method-rev-Obj*:
$[\![$ *class-add P* (*C*, *cdec*) $\vdash$ *Object sees* M_0, *b* : *Ts*$\rightarrow$*T* = *m in D*; *C* $\neq$ *Object* $]\!]$
$\Longrightarrow$ *P* $\vdash$ *Object sees* M_0, *b* : *Ts*$\rightarrow$*T* = *m in D*
by(*auto simp*: *Method-def dest*!: *class-add-sees-methods-rev-Obj*[**where** *P*=*P*])

2.16.3 Types and states

lemma *class-add-is-type*:
is-type P T $\Longrightarrow$ *is-type* (*class-add P* (*C*, *cdec*)) *T*
by(*cases cdec*, *simp add*: *is-type-def is-class-def class-def fun-upd-apply split*: *ty.splits*)

lemma *class-add-types*:
types P $\subseteq$ *types* (*class-add P* (*C*, *cdec*))
using *class-add-is-type* **by**(*cases cdec*, *clarsimp*)

lemma *class-add-states*:
states P mxs mxl $\subseteq$ *states* (*class-add P* (*C*, *cdec*)) *mxs mxl*
proof –
let *?A* = *types P* **and** *?B* = *types* (*class-add P* (*C*, *cdec*))
have *ab*: *?A* $\subseteq$ *?B* **by**(*rule class-add-types*)
moreover have $\bigwedge$*n*. *nlists n ?A* $\subseteq$ *nlists n ?B* **using** *ab* **by**(*rule nlists-mono*)
moreover have *nlists mxl* (*err ?A*) $\subseteq$ *nlists mxl* (*err ?B*) **using** *err-mono*[*OF ab*] **by**(*rule nlists-mono*)
ultimately show *?thesis* **by**(*auto simp*: *JVM-states-unfold intro*!: *err-mono opt-mono*)
qed

lemma *class-add-check-types*:
check-types P mxs mxl τs $\Longrightarrow$ *check-types* (*class-add P* (*C*, *cdec*)) *mxs mxl* τs
using *class-add-states* **by**(*fastforce simp*: *check-types-def*)

2.16.4 Subclasses and subtypes

lemma *class-add-subcls*:
$[\![$ *P* $\vdash$ *D* $\preceq^*$ *D'*; $\neg$ *is-class P C* $]\!]$
$\Longrightarrow$ *class-add P* (*C*, *cdec*) $\vdash$ *D* $\preceq^*$ *D'*
proof(*induct rule*: *rtrancl.induct*)
case (*rtrancl-into-rtrancl a b c*)
then have *b* $\neq$ *C* **by**(*clarsimp simp*: *is-class-def dest*!: *subcls1D*)
with *rtrancl-into-rtrancl* **show** *?case*
by(*fastforce dest*!: *subcls1D simp*: *class-def fun-upd-apply*
intro!: *rtrancl-trans*[*of a b*] *subcls1I*)
qed(*simp*)

lemma *class-add-subcls-rev*:
$[\![$ *class-add P* (*C*, *cdec*) $\vdash$ *D* $\preceq^*$ *D'*; $\neg$*P* $\vdash$ *D* $\preceq^*$ *C* $]\!]$
$\Longrightarrow$ *P* $\vdash$ *D* $\preceq^*$ *D'*
proof(*induct rule*: *rtrancl.induct*)
case (*rtrancl-into-rtrancl a b c*)
then have *b* $\neq$ *C* **by**(*clarsimp simp*: *is-class-def dest*!: *subcls1D*)
with *rtrancl-into-rtrancl* **show** *?case*
by(*fastforce dest*!: *subcls1D simp*: *class-def fun-upd-apply*
intro!: *rtrancl-trans*[*of a b*] *subcls1I*)
qed(*simp*)

lemma *class-add-subtype*:
$[\![$ *subtype P x y*; $\neg$ *is-class P C* $]\!]$
$\Longrightarrow$ *subtype* (*class-add P* (*C*, *cdec*)) *x y*
proof(*induct rule*: *widen.induct*)
  case (*widen-subcls C D*)
  then show *?case* **using** *class-add-subcls* **by** *simp*
qed(*simp+*)

lemma *class-add-widens*:
$[\![$ $P \vdash Ts$ $[\leq]$ Ts'; $\neg$ *is-class P C* $]\!]$
$\Longrightarrow$ (*class-add P* (*C*, *cdec*)) $\vdash Ts$ $[\leq]$ Ts'
using *class-add-subtype* **by** (*metis* (*no-types*) *list-all2-mono*)

lemma *class-add-sup-ty-opt*:
$[\![$ $P \vdash l1 \leq_\top l2$; $\neg$ *is-class P C* $]\!]$
$\Longrightarrow$ *class-add P* (*C*, *cdec*) $\vdash l1 \leq_\top l2$
using *class-add-subtype* **by**(*auto simp*: *sup-ty-opt-def Err.le-def lesub-def split*: *err.splits*)

lemma *class-add-sup-loc*:
$[\![$ $P \vdash LT$ $[\leq_\top]$ LT'; $\neg$ *is-class P C* $]\!]$
$\Longrightarrow$ *class-add P* (*C*, *cdec*) $\vdash LT$ $[\leq_\top]$ LT'
using *class-add-sup-ty-opt*[**where** *P*=*P* **and** *C*=*C*] **by** (*simp add*: *list.rel-mono-strong*)

lemma *class-add-sup-state*:
$[\![$ $P \vdash \tau \leq_i \tau'$; $\neg$ *is-class P C* $]\!]$
$\Longrightarrow$ *class-add P* (*C*, *cdec*) $\vdash \tau \leq_i \tau'$
using *class-add-subtype class-add-sup-ty-opt*
by(*auto simp*: *sup-state-def Listn.le-def Product.le-def lesub-def class-add-widens*
    class-add-sup-ty-opt list-all2-mono)

lemma *class-add-sup-state-opt*:
$[\![$ $P \vdash \tau \leq' \tau'$; $\neg$ *is-class P C* $]\!]$
$\Longrightarrow$ *class-add P* (*C*, *cdec*) $\vdash \tau \leq' \tau'$
by(*auto simp*: *sup-state-opt-def Opt.le-def lesub-def class-add-widens*
    class-add-sup-ty-opt list-all2-mono)

2.16.5 Effect

lemma *class-add-is-relevant-class*:
$[\![$ *is-relevant-class i P* C_0; $\neg$ *is-class P C* $]\!]$
$\Longrightarrow$ *is-relevant-class i* (*class-add P* (*C*, *cdec*)) C_0
  by(*cases i*, *auto simp*: *class-add-subcls*)

lemma *class-add-is-relevant-class-rev*:
assumes *irc*: *is-relevant-class i* (*class-add P* (*C*, *cdec*)) C_0
  and *ncp*: $\bigwedge cd\ D'.\ cd \in set\ P \Longrightarrow \neg P \vdash fst\ cd \preceq^* C$
  and *wfxp*: *wf-syscls P*
shows *is-relevant-class i P* C_0
using *assms*
proof(*cases i*)
  case (*Getfield F D*) **with** *assms*
  show *?thesis* **by**(*fastforce simp*: *wf-syscls-def sys-xcpts-def dest*!: *class-add-subcls-rev*)
next

case (*Putfield F D*) **with** *assms*
show *?thesis* **by**(*fastforce simp*: *wf-syscls-def sys-xcpts-def dest*!: *class-add-subcls-rev*)
next
case (*Checkcast D*) **with** *assms*
show *?thesis* **by**(*fastforce simp*: *wf-syscls-def sys-xcpts-def dest*!: *class-add-subcls-rev*)
qed(*simp-all*)

lemma *class-add-is-relevant-entry*:
$[\![$ *is-relevant-entry P i pc e*; $\neg$ *is-class P C* $]\!]$
$\Longrightarrow$ *is-relevant-entry* (*class-add P* (*C*, *cdec*)) *i pc e*
by(*clarsimp simp*: *is-relevant-entry-def class-add-is-relevant-class*)

lemma *class-add-is-relevant-entry-rev*:
$[\![$ *is-relevant-entry* (*class-add P* (*C*, *cdec*)) *i pc e*;
$\bigwedge cd\ D'.\ cd \in set\ P \Longrightarrow \neg P \vdash fst\ cd \preceq^* C$;
wf-syscls P $]\!]$
$\Longrightarrow$ *is-relevant-entry P i pc e*
by(*auto simp*: *is-relevant-entry-def dest*!: *class-add-is-relevant-class-rev*)

lemma *class-add-relevant-entries*:
$\neg$ *is-class P C*
$\Longrightarrow$ *set* (*relevant-entries P i pc xt*) $\subseteq$ *set* (*relevant-entries* (*class-add P* (*C*, *cdec*)) *i pc xt*)
by(*clarsimp simp*: *relevant-entries-def class-add-is-relevant-entry*)

lemma *class-add-relevant-entries-eq*:
assumes *wf*: *wf-prog wf-md P* **and** *nclass*: $\neg$ *is-class P C*
shows *relevant-entries P i pc xt* = *relevant-entries* (*class-add P* (*C*, *cdec*)) *i pc xt*
proof −
have *ncp*: $\bigwedge cd\ D'.\ cd \in set\ P \Longrightarrow \neg P \vdash fst\ cd \preceq^* C$
by(*rule wf-subcls-nCls'*[*OF assms*])
moreover from *wf* **have** *wfsys*: *wf-syscls P* **by**(*simp add*: *wf-prog-def*)
moreover
note *class-add-is-relevant-entry*[*OF - nclass*, *of i pc - cdec*]
class-add-is-relevant-entry-rev[*OF - ncp wfsys*, *of cdec i pc*]
ultimately show *?thesis* **by** (*metis filter-cong relevant-entries-def*)
qed

lemma *class-add-norm-eff-pc*:
assumes *ne*: $\forall (pc',\tau') \in set\ (norm\text{-}eff\ i\ P\ pc\ \tau).\ pc' < mpc$
shows $\forall (pc',\tau') \in set\ (norm\text{-}eff\ i\ (class\text{-}add\ P\ (C,\ cdec))\ pc\ \tau).\ pc' < mpc$
using *assms* **by**(*cases i*, *auto simp*: *norm-eff-def*)

lemma *class-add-norm-eff-sup-state-opt*:
assumes *ne*: $\forall (pc',\tau') \in set\ (norm\text{-}eff\ i\ P\ pc\ \tau).\ P \vdash \tau' \le' \tau s!pc'$
and *nclass*: $\neg$ *is-class P C* **and** *app*: app_i (*i*, *P*, *pc*, *mxs*, *T*, τ)
shows $\forall (pc',\tau') \in set\ (norm\text{-}eff\ i\ (class\text{-}add\ P\ (C,\ cdec))\ pc\ \tau).\ (class\text{-}add\ P\ (C,\ cdec)) \vdash \tau' \le' \tau s!pc'$
proof −
obtain *ST LT* **where** $\tau = (ST,LT)$ **by**(*cases* τ)
with *assms* **show** *?thesis* **proof**(*cases i*)
qed(*fastforce simp*: *norm-eff-def*
dest!: *class-add-field*[**where** *cdec*=*cdec*] *class-add-method*[**where** *cdec*=*cdec*]
class-add-sup-loc[*OF - nclass*] *class-add-subtype*[*OF - nclass*]
class-add-widens[*OF - nclass*] *class-add-sup-state-opt*[*OF - nclass*])+
qed

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*
using *class-add-relevant-entries-eq*[*OF assms*, *of i pc xt cdec*] **by**(*cases* τ, *simp add*: *xcpt-eff-def*)

lemma *class-add-eff-pc*:
assumes *eff*: $\forall$ (*pc'*,τ') $\in$ *set* (*eff i P pc xt* (*Some* τ)). *pc'* < *mpc*
  and *wf*: *wf-prog wf-md P* **and** *nclass*: $\neg$ *is-class P C*
shows $\forall$ (*pc'*,τ') $\in$ *set* (*eff i* (*class-add P* (*C*, *cdec*)) *pc xt* (*Some* τ)). *pc'* < *mpc*
using *eff class-add-norm-eff-pc class-add-xcpt-eff-eq*[*OF wf nclass*]
  by(*auto simp*: *norm-eff-def eff-def*)

lemma *class-add-eff-sup-state-opt*:
assumes *eff*: $\forall$ (*pc'*,τ') $\in$ *set* (*eff i P pc xt* (*Some* τ)). $P \vdash \tau' \leq' \tau s!pc'$
  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'*,τ') $\in$ *set* (*eff i* (*class-add P* (*C*, *cdec*)) *pc xt* (*Some* τ)).
        (*class-add P* (*C*, *cdec*)) $\vdash \tau' \leq' \tau s!pc'$
proof –
  from *eff* **have** *ne*: $\forall$ (*pc'*, τ')$\in$*set* (*norm-eff i P pc* τ). $P \vdash \tau' \leq' \tau s$! *pc'*
   by(*simp add*: *norm-eff-def eff-def*)
  from *eff* **have** $\forall$ (*pc'*, τ')$\in$*set* (*xcpt-eff i P pc* τ *xt*). $P \vdash \tau' \leq' \tau s$! *pc'*
   by(*simp add*: *xcpt-eff-def eff-def*)
  with *class-add-norm-eff-sup-state-opt*[*OF ne nclass app*]
      class-add-xcpt-eff-eq[*OF wf nclass*]*class-add-sup-state-opt*[*OF - nclass*]
    show *?thesis* **by**(*cases cdec*, *auto simp*: *eff-def norm-eff-def xcpt-app-def*)
qed

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*)
using *assms*
proof(*cases i*)
  case *New* **then show** *?thesis* **using** *assms* **by**(*fastforce simp*: *is-class-def class-def fun-upd-apply*)
next
  case *Getfield* **then show** *?thesis* **using** *assms*
   by(*auto simp*: *class-add-subtype dest*!: *class-add-sees-field*[**where** *P*=*P*])
next
  case *Getstatic* **then show** *?thesis* **using** *assms* **by**(*auto dest*!: *class-add-sees-field*[**where** *P*=*P*])
next
  case *Putfield* **then show** *?thesis* **using** *assms*
   by(*auto dest*!: *class-add-subtype*[**where** *P*=*P*] *class-add-sees-field*[**where** *P*=*P*])
next
  case *Putstatic* **then show** *?thesis* **using** *assms*
   by(*auto dest*!: *class-add-subtype*[**where** *P*=*P*] *class-add-sees-field*[**where** *P*=*P*])
next
  case *Checkcast* **then show** *?thesis* **using** *assms*
   by(*clarsimp simp*: *is-class-def class-def fun-upd-apply*)
next
  case *Invoke* **then show** *?thesis* **using** *assms*
    by(*fastforce dest*!: *class-add-widens*[**where** *P*=*P*] *class-add-sees-method*[**where** *P*=*P*])
next

case *Invokestatic* **then show** *?thesis* **using** *assms*
by(*fastforce dest*!: *class-add-widens*[**where** *P*=*P*] *class-add-sees-method*[**where** *P*=*P*])
next
case *Return* **then show** *?thesis* **using** *assms* **by**(*clarsimp simp*: *class-add-subtype*)
qed(*simp*+)

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
using *xa class-add-relevant-entries-eq*[*OF wf nclass*] *nclass*
by(*auto simp*: *xcpt-app-def is-class-def class-def fun-upd-apply*) *auto*

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(*cases t*)
case (*Some* τ)
let *?P* = *class-add P* (*C*, *cdec*)
from *assms Some* **have** *eff*: $\forall (pc', \tau') \in set\ (eff\ i\ P\ pc\ xt\ \lfloor\tau\rfloor).\ pc' < mpc$ **by**(*simp add*: *app-def*)
from *assms Some* **have** app_i: $app_i\ (i,P,pc,mxs,T,\tau)$ **by**(*simp add*: *app-def*)
with $class\text{-}add\text{-}app_i$[*OF - nclass*] *Some* **have** $app_i\ (i,?P,pc,mxs,T,\tau)$ **by**(*cases* τ,*simp*)
moreover
from *app class-add-xcpt-app*[*OF - wf nclass*] *Some*
have *xcpt-app i ?P pc mxs xt* τ **by**(*simp add*: *app-def del*: *xcpt-app-def*)
moreover
from *app class-add-eff-pc*[*OF eff wf nclass*] *Some*
have $\forall (pc',\tau') \in set\ (eff\ i\ ?P\ pc\ xt\ t).\ pc' < mpc$ **by** *auto*
moreover note *app Some*
ultimately show *?thesis* **by**(*simp add*: *app-def*)
qed(*simp*)

2.16.6 Well-formedness and well-typedness

lemma *class-add-wf-mdecl*:
$\llbracket$ *wf-mdecl wf-md P* C_0 *md*;
$\bigwedge C_0\ md.\ wf\text{-}md\ P\ C_0\ md \Longrightarrow wf\text{-}md\ (class\text{-}add\ P\ (C,\ cdec))\ C_0\ md\ \rrbracket$
$\Longrightarrow$ *wf-mdecl wf-md* (*class-add P* (*C*, *cdec*)) C_0 *md*
by(*clarsimp simp*: *wf-mdecl-def class-add-is-type*)

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'*: $\bigwedge C_0\ S\ fs\ ms\ m.\llbracket(C_0,S,fs,ms) \in set\ P;\ m \in set\ ms\rrbracket \Longrightarrow wf\text{-}md'\ (class\text{-}add\ P\ (C,\ cdec))$
C_0 *m*
shows *wf-mdecl wf-md'* (*class-add P* (*C*, *cdec*)) C_0 *md*
using *assms* **by**(*clarsimp simp*: *wf-mdecl-def class-add-is-type*)

lemma *class-add-wf-cdecl*:
assumes *wfcd*: *wf-cdecl wf-md P cd* **and** *cdP*: $cd \in set\ P$
and *ncp*: $\neg\ P \vdash fst\ cd \preceq^* C$ **and** *dist*: *distinct-fst P*
and *wfmd*: $\bigwedge C_0\ md.\ wf\text{-}md\ P\ C_0\ md \Longrightarrow wf\text{-}md\ (class\text{-}add\ P\ (C,\ cdec))\ C_0\ md$
and *nclass*: $\neg$ *is-class P C*

shows *wf-cdecl wf-md* (*class-add P* (*C*, *cdec*)) *cd*
proof −
  let *?P* = *class-add P* (*C*, *cdec*)
  obtain *C1 D fs ms* **where** [*simp*]: *cd* = (*C1*,(*D*,*fs*,*ms*)) **by**(*cases cd*)
  from *wfcd*
  have $\forall f \in$*set fs*. *wf-fdecl ?P f* **by**(*auto simp*: *wf-cdecl-def wf-fdecl-def class-add-is-type*)
  moreover
  from *wfcd wfmd class-add-wf-mdecl*
  have $\forall m \in$*set ms*. *wf-mdecl wf-md ?P C1 m* **by**(*auto simp*: *wf-cdecl-def*)
  moreover
  have *C1* $\neq$ *Object* $\Longrightarrow$ *is-class ?P D* $\wedge$ $\neg$ *?P* $\vdash$ *D* $\preceq^*$ *C1*
   $\wedge$ ($\forall$ (*M*,*b*,*Ts*,*T*,*m*)$\in$*set ms*.
      $\forall$ *D′ b′ Ts′ T′ m′*. *?P* $\vdash$ *D sees M*,*b′*:*Ts′* $\rightarrow$ *T′* = *m′ in D′* $\longrightarrow$
            b = *b′* $\wedge$ *?P* $\vdash$ *Ts′* [$\leq$] *Ts* $\wedge$ *?P* $\vdash$ *T* $\leq$ *T′*)
  proof −
   assume *nObj*[*simp*]: *C1* $\neq$ *Object*
   with *cdP dist* **have** *sub1*: *P* $\vdash$ *C1* $\prec^1$ *D* **by**(*auto simp*: *class-def intro*!: *subcls1I map-of-SomeI*)
   with *ncp* **have** *ncp′*: $\neg$ *P* $\vdash$ *D* $\preceq^*$ *C* **by**(*auto simp*: *converse-rtrancl-into-rtrancl*)
   with *wfcd*
   have *clsD*: *is-class ?P D*
    by(*auto simp*: *wf-cdecl-def is-class-def class-def fun-upd-apply*)
   moreover
   from *wfcd sub1*
   have $\neg$ *?P* $\vdash$ *D* $\preceq^*$ *C1* **by**(*auto simp*: *wf-cdecl-def dest*!: *class-add-subcls-rev*[*OF - ncp′*])
   moreover
   have $\bigwedge$*M b Ts T m D′ b′ Ts′ T′ m′*. (*M*,*b*,*Ts*,*T*,*m*) $\in$ *set ms*
       $\Longrightarrow$ *?P* $\vdash$ *D sees M*,*b′*:*Ts′* $\rightarrow$ *T′* = *m′ in D′*
       $\Longrightarrow$ *b* = *b′* $\wedge$ *?P* $\vdash$ *Ts′* [$\leq$] *Ts* $\wedge$ *?P* $\vdash$ *T* $\leq$ *T′*
   proof −
    fix *M b Ts T m D′ b′ Ts′ T′ m′*
    assume *ms*: (*M*,*b*,*Ts*,*T*,*m*) $\in$ *set ms* **and** *meth′*: *?P* $\vdash$ *D sees M*,*b′*:*Ts′* $\rightarrow$ *T′* = *m′ in D′*
    with *sub1*
    have *P* $\vdash$ *D sees M*,*b′*:*Ts′* $\rightarrow$ *T′* = *m′ in D′*
     by(*fastforce dest*!: *class-add-sees-method-rev*[*OF - ncp′*])
    moreover
    with *wfcd ms meth′*
    have *b* = *b′* $\wedge$ *P* $\vdash$ *Ts′* [$\leq$] *Ts* $\wedge$ *P* $\vdash$ *T* $\leq$ *T′*
     by(*cases m′*, *fastforce simp*: *wf-cdecl-def elim*!: *ballE*[**where** *x*=(*M*,*b*,*Ts*,*T*,*m*)])
    ultimately show *b* = *b′* $\wedge$ *?P* $\vdash$ *Ts′* [$\leq$] *Ts* $\wedge$ *?P* $\vdash$ *T* $\leq$ *T′*
     by(*auto dest*!: *class-add-subtype*[*OF - nclass*] *class-add-widens*[*OF - nclass*])
   qed
   ultimately show *?thesis* **by** *clarsimp*
  qed
  moreover note *wfcd*
  ultimately show *?thesis* **by**(*simp add*: *wf-cdecl-def*)
qed

lemma *class-add-wf-cdecl′*:
assumes *wfcd*: *wf-cdecl wf-md P cd* **and** *cdP*: *cd* $\in$ *set P*
 and *ncp*: $\neg$*P* $\vdash$ *fst cd* $\preceq^*$ *C* **and** *dist*: *distinct-fst P*
 and *wfmd*: $\bigwedge C_0$ *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*
 and *nclass*: $\neg$ *is-class P C*
shows *wf-cdecl wf-md′* (*class-add P* (*C*, *cdec*)) *cd*

proof −
let *?P* = *class-add P* (*C*, *cdec*)
obtain *C1 D fs ms* **where** [*simp*]: *cd* = (*C1*,(*D*,*fs*,*ms*)) **by**(*cases cd*)
from *wfcd*
have ∀ *f*∈*set fs*. *wf-fdecl ?P f* **by**(*auto simp*: *wf-cdecl-def wf-fdecl-def class-add-is-type*)
moreover
from *cdP wfcd wfmd*
have ∀ *m*∈*set ms*. *wf-mdecl wf-md′ ?P C1 m*
by(*auto simp*: *wf-cdecl-def wf-mdecl-def class-add-is-type*)
moreover
have *C1* ≠ *Object* ⟹ *is-class ?P D* ∧ ¬ *?P* ⊢ *D* ⪯* *C1*
∧ (∀ (*M*,*b*,*Ts*,*T*,*m*)∈*set ms*.
∀ *D′ b′ Ts′ T′ m′*. *?P* ⊢ *D sees M*,*b′*:*Ts′* → *T′* = *m′ in D′* ⟶
b = *b′* ∧ *?P* ⊢ *Ts′* [≤] *Ts* ∧ *?P* ⊢ *T* ≤ *T′*)
proof −
assume *nObj*[*simp*]: *C1* ≠ *Object*
with *cdP dist* **have** *sub1*: *P* ⊢ *C1* ≺¹ *D* **by**(*auto simp*: *class-def intro*!: *subcls1I map-of-SomeI*)
with *ncp* **have** *ncp′*: ¬ *P* ⊢ *D* ⪯* *C* **by**(*auto simp*: *converse-rtrancl-into-rtrancl*)
with *wfcd*
have *clsD*: *is-class ?P D*
by(*auto simp*: *wf-cdecl-def is-class-def class-def fun-upd-apply*)
moreover
from *wfcd sub1*
have ¬ *?P* ⊢ *D* ⪯* *C1* **by**(*auto simp*: *wf-cdecl-def dest*!: *class-add-subcls-rev*[*OF - ncp′*])
moreover
have ⋀*M b Ts T m D′ b′ Ts′ T′ m′*. (*M*,*b*,*Ts*,*T*,*m*) ∈ *set ms*
⟹ *?P* ⊢ *D sees M*,*b′*:*Ts′* → *T′* = *m′ in D′*
⟹ *b* = *b′* ∧ *?P* ⊢ *Ts′* [≤] *Ts* ∧ *?P* ⊢ *T* ≤ *T′*
proof −
fix *M b Ts T m D′ b′ Ts′ T′ m′*
assume *ms*: (*M*,*b*,*Ts*,*T*,*m*) ∈ *set ms* **and** *meth′*: *?P* ⊢ *D sees M*,*b′*:*Ts′* → *T′* = *m′ in D′*
with *sub1*
have *P* ⊢ *D sees M*,*b′*:*Ts′* → *T′* = *m′ in D′*
by(*fastforce dest*!: *class-add-sees-method-rev*[*OF - ncp′*])
moreover
with *wfcd ms meth′*
have *b* = *b′* ∧ *P* ⊢ *Ts′* [≤] *Ts* ∧ *P* ⊢ *T* ≤ *T′*
by(*cases m′*, *fastforce simp*: *wf-cdecl-def elim*!: *ballE*[**where** *x*=(*M*,*b*,*Ts*,*T*,*m*)])
ultimately show *b* = *b′* ∧ *?P* ⊢ *Ts′* [≤] *Ts* ∧ *?P* ⊢ *T* ≤ *T′*
by(*auto dest*!: *class-add-subtype*[*OF - nclass*] *class-add-widens*[*OF - nclass*])
qed
ultimately show *?thesis* **by** *clarsimp*
qed
moreover note *wfcd*
ultimately show *?thesis* **by**(*simp add*: *wf-cdecl-def*)
qed

lemma *class-add-wt-start*:
⟦ *wt-start P* C_0 *b Ts mxl* τ*s*; ¬ *is-class P C* ⟧
⟹ *wt-start* (*class-add P* (*C*, *cdec*)) C_0 *b Ts mxl* τ*s*
using *class-add-sup-state-opt* **by**(*clarsimp simp*: *wt-start-def split*: *staticb.splits*)

lemma *class-add-wt-instr*:
assumes *wti*: *P*,*T*,*mxs*,*mpc*,*xt* ⊢ *i*,*pc* :: τ*s*

and *wf*: *wf-prog wf-md P* **and** *nclass*: $\neg$ *is-class P C*
shows *class-add P* $(C,$ *cdec*$), T,$*mxs*$,$*mpc*$,$*xt* $\vdash i,$*pc* $:: \tau s$
proof –
let *?P* = *class-add P* (*C*, *cdec*)
from *wti* **have** *eff*: $\forall(pc', \tau')\in$*set* (*eff i P pc xt* (τs ! *pc*)). $P \vdash \tau' \leq' \tau s$! pc'
by(*simp add*: *wt-instr-def*)
from *wti* **have** app_i: $\tau s!pc \neq$ *None* $\Longrightarrow$ app_i ($i,P,pc,mxs,T,$*the* ($\tau s!pc$))
by(*simp add*: *wt-instr-def app-def*)
from *wti class-add-app*[*OF* - *wf nclass*]
have *app i ?P mxs T pc mpc xt* ($\tau s!pc$) **by**(*simp add*: *wt-instr-def*)
moreover
have $\forall(pc',\tau') \in$ *set* (*eff i ?P pc xt* ($\tau s!pc$)). *?P* $\vdash \tau' \leq' \tau s!pc'$
proof(*cases* $\tau s!pc$)
case *Some* **with** *eff class-add-eff-sup-state-opt*[*OF* - *wf nclass* app_i] **show** *?thesis* **by** *auto*
qed(*simp add*: *eff-def*)
moreover note *wti*
ultimately show *?thesis* **by**(*clarsimp simp*: *wt-instr-def*)
qed

lemma *class-add-wt-method*:
assumes *wtm*: *wt-method P* C_0 *b Ts* T_r *mxs* mxl_0 *is xt* (Φ C_0 M_0)
and *wf*: *wf-prog wf-md P* **and** *nclass*: $\neg$ *is-class P C*
shows *wt-method* (*class-add P* (*C*, *cdec*)) C_0 *b Ts* T_r *mxs* mxl_0 *is xt* (Φ C_0 M_0)
proof –
let *?P* = *class-add P* (*C*, *cdec*)
let *?*τs = Φ C_0 M_0
from *wtm class-add-check-types*
have *check-types ?P mxs* ((*case b of Static* $\Rightarrow$ *0* | *NonStatic* $\Rightarrow$ *1*)+*size Ts*+mxl_0) (*map OK ?*τs)
by(*simp add*: *wt-method-def*)
moreover
from *wtm class-add-wt-start nclass*
have *wt-start ?P* C_0 *b Ts* mxl_0 *?*τs **by**(*simp add*: *wt-method-def*)
moreover
from *wtm class-add-wt-instr*[*OF* - *wf nclass*]
have $\forall$ *pc* < *size is*. *?P*$,T_r,$*mxs*$,$*size is*$,$*xt* $\vdash$ *is*!*pc*$,$*pc* :: *?*τs **by**(*clarsimp simp*: *wt-method-def*)
moreover note *wtm*
ultimately
show *?thesis* **by**(*clarsimp simp*: *wt-method-def*)
qed

lemma *class-add-wt-method'*:
$[\![$ (λP *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*)) *P* C_0 *md*;
wf-prog wf-md P; $\neg$ *is-class P C* $]\!]$
$\Longrightarrow$ (λP *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*))
(*class-add P* (*C*, *cdec*)) C_0 *md*
by(*clarsimp simp*: *class-add-wt-method*)

2.16.7 *distinct-fst*

lemma *class-add-distinct-fst*:
$[\![$ *distinct-fst P*; $\neg$ *is-class P C* $]\!]$
$\Longrightarrow$ *distinct-fst* (*class-add P* (*C*, *cdec*))
by(*clarsimp simp*: *distinct-fst-def is-class-def class-def*)

2.16.8 Conformance

lemma *class-add-conf*:
$[\![P,h \vdash v :\leq T; \neg \textit{is-class } P\ C]\!]$
$\Longrightarrow \textit{class-add } P\ (C,\ \textit{cdec}),h \vdash v :\leq T$
by(*clarsimp simp*: *conf-def class-add-subtype*)

lemma *class-add-oconf*:
fixes *obj*::*obj*
assumes *oc*: $P,h \vdash \textit{obj} \surd$ **and** *ns*: $\neg$ *is-class P C*
and *ncp*: $\bigwedge D'.\ P \vdash \textit{fst}(\textit{obj}) \preceq^* D' \Longrightarrow D' \neq C$
shows $(\textit{class-add } P\ (C,\ \textit{cdec})),h \vdash \textit{obj} \surd$
proof –
obtain C_0 *fs* **where** [*simp*]: $\textit{obj}=(C_0,\textit{fs})$ **by**(*cases obj*)
from *oc* **have**
oc': $\bigwedge F\ D\ T.\ P \vdash C_0\ \textit{has}\ F,\textit{NonStatic}{:}T\ \textit{in}\ D \Longrightarrow (\exists v.\ \textit{fs}\ (F,\ D) = \lfloor v \rfloor \wedge P,h \vdash v :\leq T)$
by(*simp add*: *oconf-def*)
have $\bigwedge F\ D\ T.\ \textit{class-add } P\ (C,\ \textit{cdec}) \vdash C_0\ \textit{has}\ F,\textit{NonStatic}{:}T\ \textit{in}\ D$
$\Longrightarrow \exists v.\ \textit{fs}(F,D) = \textit{Some } v \wedge \textit{class-add } P\ (C,\ \textit{cdec}),h \vdash v :\leq T$
proof –
fix *F D T* **assume** $\textit{class-add } P\ (C,\ \textit{cdec}) \vdash C_0\ \textit{has}\ F,\textit{NonStatic}{:}T\ \textit{in}\ D$
with *class-add-has-field-rev*[*OF - ncp*] **have** *meth*: $P \vdash C_0\ \textit{has}\ F,\textit{NonStatic}{:}T\ \textit{in}\ D$ **by** *simp*
then show $\exists v.\ \textit{fs}(F,D) = \textit{Some } v \wedge \textit{class-add } P\ (C,\ \textit{cdec}),h \vdash v :\leq T$
using *oc'*[*OF meth*] *class-add-conf*[*OF - ns*] **by**(*fastforce simp*: *oconf-def*)
qed
then show *?thesis* **by**(*simp add*: *oconf-def*)
qed

lemma *class-add-soconf*:
assumes *soc*: $P,h,C_0 \vdash_s \textit{sfs} \surd$ **and** *ns*: $\neg$ *is-class P C*
and *ncp*: $\bigwedge D'.\ P \vdash C_0 \preceq^* D' \Longrightarrow D' \neq C$
shows $(\textit{class-add } P\ (C,\ \textit{cdec})),h,C_0 \vdash_s \textit{sfs} \surd$
proof –
from *soc* **have**
oc': $\bigwedge F\ T.\ P \vdash C_0\ \textit{has}\ F,\textit{Static}{:}T\ \textit{in}\ C_0 \Longrightarrow (\exists v.\ \textit{sfs}\ F = \lfloor v \rfloor \wedge P,h \vdash v :\leq T)$
by(*simp add*: *soconf-def*)
have $\bigwedge F\ T.\ \textit{class-add } P\ (C,\ \textit{cdec}) \vdash C_0\ \textit{has}\ F,\textit{Static}{:}T\ \textit{in}\ C_0$
$\Longrightarrow \exists v.\ \textit{sfs}\ F = \textit{Some } v \wedge \textit{class-add } P\ (C,\ \textit{cdec}),h \vdash v :\leq T$
proof –
fix *F T* **assume** $\textit{class-add } P\ (C,\ \textit{cdec}) \vdash C_0\ \textit{has}\ F,\textit{Static}{:}T\ \textit{in}\ C_0$
with *class-add-has-field-rev*[*OF - ncp*] **have** *meth*: $P \vdash C_0\ \textit{has}\ F,\textit{Static}{:}T\ \textit{in}\ C_0$ **by** *simp*
then show $\exists v.\ \textit{sfs}\ F = \textit{Some } v \wedge \textit{class-add } P\ (C,\ \textit{cdec}),h \vdash v :\leq T$
using *oc'*[*OF meth*] *class-add-conf*[*OF - ns*] **by**(*fastforce simp*: *soconf-def*)
qed
then show *?thesis* **by**(*simp add*: *soconf-def*)
qed

lemma *class-add-hconf*:
assumes $P \vdash h \surd$ **and** $\neg$ *is-class P C*
and $\bigwedge a\ \textit{obj}\ D'.\ h\ a = \textit{Some obj} \Longrightarrow P \vdash \textit{fst}(\textit{obj}) \preceq^* D' \Longrightarrow D' \neq C$
shows $\textit{class-add } P\ (C,\ \textit{cdec}) \vdash h \surd$
using *assms* **by**(*auto simp*: *hconf-def intro*!: *class-add-oconf*)

lemma *class-add-hconf-wf*:

assumes *wf*: *wf-prog wf-md P* **and** $P \vdash h \surd$ **and** ¬ *is-class P C*
and $\bigwedge a$ *obj*. *h a* = *Some obj* $\Longrightarrow$ *fst*(*obj*) $\neq C$
shows *class-add P* (*C*, *cdec*) $\vdash h \surd$
using *wf-subcls-nCls*[*OF wf*] *assms* **by**(*fastforce simp*: *hconf-def intro*!: *class-add-oconf*)

lemma *class-add-shconf*:
assumes $P,h \vdash_s sh \surd$ **and** *ns*: ¬ *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*),$h \vdash_s sh \surd$
using *assms* **by**(*fastforce simp*: *shconf-def*)

lemma *class-add-shconf-wf*:
assumes *wf*: *wf-prog wf-md P* **and** $P,h \vdash_s sh \surd$ **and** ¬ *is-class P C*
and $\bigwedge C$ *sobj*. *sh C* = *Some sobj* $\Longrightarrow C \neq C$
shows *class-add P* (*C*, *cdec*),$h \vdash_s sh \surd$
using *wf-subcls-nCls*[*OF wf*] *assms* **by**(*fastforce simp*: *shconf-def*)

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* :: $ty_P \Rightarrow ty_P$ **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*: $\bigwedge C$. $C \neq$ *Start* $\Longrightarrow$ Φ-*start* Φ *C* = Φ *C*
Φ-*start* Φ *Start start-m* = *start*-φ_m Φ-*start* Φ *Start clinit* = *start*-φ_m
by *auto*

lemma *check-types*-φ_m: *check-types* (*start-prog P C M*) *1 0* (*map OK start*-φ_m)
by (*auto simp*: *check-types-def JVM-states-unfold*)

2.17.2 Some simple properties

lemma *preallocated-start-state*: *start-state P* = σ $\Longrightarrow$ *preallocated* (*fst*(*snd* σ))
using *preallocated-start*[*of P*] **by**(*auto simp*: *start-state-def split-beta*)

lemma *start-prog-Start-super*: *start-prog P C M* $\vdash$ *Start* $\prec^1$ *Object*
by(*auto intro*!: *subcls1I simp*: *class-def fun-upd-apply*)

lemma *start-prog-Start-fields*:
start-prog P C M $\vdash$ *Start has-fields FDTs* $\Longrightarrow$ *map-of FDTs* $(F, Start) = None$
by(*drule Fields.cases, auto simp*: *class-def fun-upd-apply Object-fields*)

lemma *start-prog-Start-soconf*:
(*start-prog P C M*),*h*,*Start* $\vdash_s$ *Map.empty* $\surd$
by(*simp add*: *soconf-def has-field-def start-prog-Start-fields*)

lemma *start-prog-start-shconf*:
start-prog P C M,*start-heap P* $\vdash_s$ *start-sheap* $\surd$

2.17.3 Well-typed and well-formed

lemma *start-wt-method*:
assumes $P \vdash C$ *sees M*, *Static* : $[] \rightarrow Void = m$ *in D* **and** $M \neq clinit$ **and** $\neg$ *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 −
  let *?cdec* = (*Object*, [], [*start-method C M*, *start-clinit*])
  obtain *mxs mxl ins xt* **where** *m*: *m* = (*mxs*,*mxl*,*ins*,*xt*) **by**(*cases m*)
  have *ca-sees*: *class-add P* (*Start*, *?cdec*) $\vdash C$ *sees M*, *Static* : $[] \rightarrow Void = m$ *in D*
    by(*rule class-add-sees-method*[*OF assms*(*1*,*3*)])
  have $\bigwedge pc.\ pc <$ *size ?is* $\Longrightarrow$ *?P*,*?*T_r,*?mxs*,*size ?is*,*?xt* $\vdash$ *?is*!*pc*,*pc* :: *?*τs
  proof −
    fix *pc* **assume** *pc*: *pc* < *size ?is*
    then show *?P*,*?*T_r,*?mxs*,*size ?is*,*?xt* $\vdash$ *?is*!*pc*,*pc* :: *?*τs
    proof(*cases pc* = *0*)
      case *True* **with** *assms m ca-sees* **show** *?thesis*
        by(*fastforce simp*: *wt-method-def wt-start-def relevant-entries-def*
                is-relevant-entry-def xcpt-eff-def)
    next
      case *False* **with** *pc* **show** *?thesis*
        by(*simp add*: *wt-method-def wt-start-def relevant-entries-def*
                is-relevant-entry-def xcpt-eff-def)
    qed
  qed
  with *assms check-types-*φ_m **show** *?thesis* **by**(*simp add*: *wt-method-def wt-start-def*)
qed

lemma *start-clinit-wt-method*:
assumes $P \vdash C$ *sees M*, *Static* : $[] \rightarrow Void = m$ *in D* **and** $M \neq clinit$ **and** $\neg$ *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 −
  let *?cdec* = (*Object*, [], [*start-method C M*, *start-clinit*])
  obtain *mxs mxl ins xt* **where** *m*: *m* = (*mxs*,*mxl*,*ins*,*xt*) **by**(*cases m*)
  have *ca-sees*: *class-add P* (*Start*, *?cdec*) $\vdash C$ *sees M*, *Static* : $[] \rightarrow Void = m$ *in D*
    by(*rule class-add-sees-method*[*OF assms*(*1*,*3*)])
  have $\bigwedge pc.\ pc <$ *size ?is* $\Longrightarrow$ *?P*,*?*T_r,*?mxs*,*size ?is*,*?xt* $\vdash$ *?is*!*pc*,*pc* :: *?*τs
  proof −
    fix *pc* **assume** *pc*: *pc* < *size ?is*
    then show *?P*,*?*T_r,*?mxs*,*size ?is*,*?xt* $\vdash$ *?is*!*pc*,*pc* :: *?*τs
    proof(*cases pc* = *0*)
      case *True* **with** *assms m ca-sees* **show** *?thesis*

by(*fastforce simp*: *wt-method-def wt-start-def relevant-entries-def*
is-relevant-entry-def xcpt-eff-def)
next
case *False* **with** *pc* **show** *?thesis*
by(*simp add*: *wt-method-def wt-start-def relevant-entries-def*
is-relevant-entry-def xcpt-eff-def)
qed
qed
with *assms check-types-*φ_m **show** *?thesis* **by**(*simp add*: *wt-method-def wt-start-def*)
qed

lemma *start-class-wf*:
assumes $P \vdash C$ *sees* M, *Static* : $[] \to$ *Void* $=$ m *in* D
and $M \neq$ *clinit* **and** $\neg$ *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-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-method* $P\ C\ b\ Ts\ T_r\ mxs\ mxl_0$ *is xt* (Φ C M))
(*start-prog P C M*) (*start-class C M*)
proof −
from *assms start-wt-method start-clinit-wt-method class-add-sees-method-rev-Obj*[**where** P=P **and** C=*Start*]
show *?thesis*
by(*auto simp*: *start-method-def wf-cdecl-def wf-fdecl-def wf-mdecl-def*
is-class-def class-def fun-upd-apply wf-clinit-def) *fast+*
qed

lemma *start-prog-wf-jvm-prog-phi*:
assumes *wtp*: *wf-jvm-prog*$_\Phi$ P
and *nstart*: $\neg$ *is-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': Φ' *Start start-m* $=$ *start-*φ_m Φ' *Start clinit* $=$ *start-*φ_m
and *Obj-start-m*: $\bigwedge b'\ Ts'\ T'\ m'\ D'.\ P \vdash$ *Object sees start-m*, b' : $Ts' \to T' = m'$ *in* D'
$\Longrightarrow b' = Static \wedge Ts' = [] \wedge T' = Void$
shows *wf-jvm-prog*$_{\Phi'}$ (*start-prog P C M*)
proof −
let *?wf-md* $=$ ($\lambda P\ C\ (M,b,Ts,T_r,(mxs,mxl_0,is,xt))$. *wt-method* $P\ C\ b\ Ts\ T_r\ mxs\ mxl_0$ *is xt* (Φ C M))
let *?wf-md'* $=$ ($\lambda P\ C\ (M,b,Ts,T_r,(mxs,mxl_0,is,xt))$. *wt-method* $P\ C\ b\ Ts\ T_r\ mxs\ mxl_0$ *is xt* (Φ' C M))
from *wtp* **have** *wf*: *wf-prog ?wf-md P* **by**(*simp add*: *wf-jvm-prog-phi-def*)
from *wf-subcls-nCls'*[*OF wf nstart*]
have *nsp*: $\bigwedge cd\ D'.\ cd \in set\ P \Longrightarrow \neg P \vdash fst\ cd \preceq^* Start$ **by** *simp*
have *wf-md'*:
$\bigwedge C_0\ S\ fs\ ms\ m.\ (C_0,\ S,\ fs,\ ms) \in set\ P \Longrightarrow m \in set\ ms \Longrightarrow$ *?wf-md'* (*start-prog P C M*) C_0 m
proof −
fix C_0 *S fs ms m* **assume** *asms*: $(C_0,\ S,\ fs,\ ms) \in set\ P$ $m \in set\ ms$
with *nstart* **have** *ns*: $C_0 \neq Start$ **by**(*auto simp*: *is-class-def class-def dest*: *weak-map-of-SomeI*)
from *wf asms* **have** *?wf-md* P C_0 m **by**(*auto simp*: *wf-prog-def wf-cdecl-def wf-mdecl-def*)

with Φ[*OF ns*] *class-add-wt-method*[*OF - wf nstart*]
show *?wf-md′* (*start-prog P C M*) C_0 *m* **by** *fastforce*
qed
from *wtp* **have** *a1*: *is-class P Object* **by** (*simp add*: *wf-jvm-prog-phi-def*)
with *wf-sees-clinit*[**where** *P*=*P* **and** *C*=*Object*] *wtp*
have *a2*: $\bigwedge b'\ Ts'\ T'\ m'\ D'.\ P \vdash$ *Object sees clinit*, b' : $Ts' \rightarrow T' = m'$ *in* D'
$\Longrightarrow b' = Static \wedge Ts' = [] \wedge T' = Void$
by(*fastforce simp*: *wf-jvm-prog-phi-def is-class-def dest*: *sees-method-fun*)
from *wf* **have** *dist*: *distinct-fst P* **by** (*simp add*: *wf-prog-def*)
with *class-add-distinct-fst*[*OF - nstart*] **have** *distinct-fst* (*start-prog P C M*) **by** *simp*
moreover from *wf* **have** *wf-syscls* (*start-prog P C M*) **by**(*simp add*: *wf-prog-def wf-syscls-def*)
moreover
from *class-add-wf-cdecl′*[**where** *wf-md′*=*?wf-md′*, *OF - - nsp dist*] *wf-md′ nstart wf*
have $\bigwedge c.\ c \in set\ P \Longrightarrow$ *wf-cdecl ?wf-md′* (*start-prog P C M*) *c* **by**(*fastforce simp*: *wf-prog-def*)
moreover from *start-class-wf*[*OF meth*] *nclinit nstart* Φ' *a1 Obj-start-m a2*
have *wf-cdecl ?wf-md′* (*start-prog P C M*) (*start-class C M*) **by** *simp*
ultimately show *?thesis* **by**(*simp add*: *wf-jvm-prog-phi-def wf-prog-def*)
qed

lemma *start-prog-wf-jvm-prog*:
assumes *wf*: *wf-jvm-prog P*
and *nstart*: $\neg$ *is-class P Start*
and *meth*: $P \vdash C$ *sees* M, *Static* : $[] \rightarrow Void = m$ *in* D **and** *nclinit*: $M \neq clinit$
and *Obj-start-m*: $\bigwedge b'\ Ts'\ T'\ m'\ D'.\ P \vdash$ *Object sees start-m*, b' : $Ts' \rightarrow T' = m'$ *in* D'
$\Longrightarrow b' = Static \wedge Ts' = [] \wedge T' = Void$
shows *wf-jvm-prog* (*start-prog P C M*)
proof −
from *wf* **obtain** Φ **where** *wtp*: *wf-jvm-prog*$_\Phi$ *P* **by**(*clarsimp simp*: *wf-jvm-prog-def*)

let $?\Phi' = \lambda C\, f.$ *if* $C = Start \wedge (f = start\text{-}m \vee f = clinit)$ *then* $start\text{-}\varphi_m$ *else* $\Phi\ C\ f$

from *start-prog-wf-jvm-prog-phi*[*OF wtp nstart meth nclinit - - - Obj-start-m*] **have**
wf-jvm-prog$_{?\Phi'}$ (*start-prog P C M*) **by** *simp*
then show *?thesis* **by**(*auto simp*: *wf-jvm-prog-def*)
qed

2.17.4 Methods and instructions

lemma *start-prog-Start-sees-methods*:
$P \vdash$ *Object sees-methods Mm*
$\Longrightarrow$ *start-prog P C M* $\vdash$
Start sees-methods Mm ++ (*map-option* ($\lambda m.$ (*m*,*Start*)) $\circ$ *map-of* [*start-method C M*, *start-clinit*])
by (*auto simp*: *class-def fun-upd-apply*
dest!: *class-add-sees-methods-Obj*[**where** *P*=*P* **and** *C*=*Start*] *intro*: *sees-methods-rec*)

lemma *start-prog-Start-sees-start-method*:
$P \vdash$ *Object sees-methods Mm*
$\Longrightarrow$ *start-prog P C M* $\vdash$
Start sees start-m, *Static* : $[] \rightarrow Void$ = (*1*, *0*, [*Invokestatic C M 0*,*Return*], []) *in Start*
by(*auto simp*: *start-method-def Method-def fun-upd-apply*
dest!: *start-prog-Start-sees-methods*)

lemma *wf-start-prog-Start-sees-start-method*:

assumes *wf*: *wf-prog wf-md P*
shows *start-prog P C M* ⊢
Start sees start-m, *Static* : []→*Void* = (*1*, *0*, [*Invokestatic C M 0*,*Return*], []) *in Start*
proof −
from *wf* **have** *is-class P Object* **by** *simp*
with *sees-methods-Object* **obtain** *Mm* **where** *P* ⊢ *Object sees-methods Mm*
by(*fastforce simp*: *is-class-def dest*: *sees-methods-Object*)
then show *?thesis* **by**(*rule start-prog-Start-sees-start-method*)
qed

lemma *start-prog-start-m-instrs*:
assumes *wf*: *wf-prog wf-md P*
shows (*instrs-of* (*start-prog P C M*) *Start start-m*) = [*Invokestatic C M 0*, *Return*]
proof −
from *wf-start-prog-Start-sees-start-method*[*OF wf*]
have *start-prog P C M* ⊢ *Start sees start-m*, *Static* :
[]→*Void* = (*1*,*0*,[*Invokestatic C M 0*,*Return*],[]) *in Start* **by** *simp*
then show *?thesis* **by** *simp*
qed

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'*) ⟹
∃(*f*,*t*,*D*,*h*,*d*) ∈ *set* (*relevant-entries P* (*Invoke n M*) *pc xt*).
$P \vdash C \preceq^* D \land pc \in \{f..<t\} \land pc' = h \land d' = d$
by (*induct xt*) (*auto simp*: *relevant-entries-def matches-ex-entry-def*
is-relevant-entry-def split: *if-split-asm*)

For the *Invokestatic* instruction the BV has checked all handlers that guard the current *pc*.

lemma *Invokestatic-handlers*:
$match\text{-}ex\text{-}table\ P\ C\ pc\ xt = Some\ (pc',d') \Longrightarrow$
$\exists (f,t,D,h,d) \in set\ (relevant\text{-}entries\ P\ (Invokestatic\ C_0\ n\ M)\ pc\ xt).$
$P \vdash C \preceq^* D \wedge pc \in \{f..<t\} \wedge pc' = h \wedge d' = d$
by (*induct xt*) (*auto simp*: *relevant-entries-def matches-ex-entry-def is-relevant-entry-def split*: *if-split-asm*)

For the instrs in *Called-set* the BV has checked all handlers that guard the current *pc*.

lemma *Called-set-handlers*:
$match\text{-}ex\text{-}table\ P\ C\ pc\ xt = Some\ (pc',d') \Longrightarrow i \in Called\text{-}set \Longrightarrow$
$\exists (f,t,D,h,d) \in set\ (relevant\text{-}entries\ P\ i\ pc\ xt).$
$P \vdash C \preceq^* D \wedge pc \in \{f..<t\} \wedge pc' = h \wedge d' = d$
by (*induct xt*) (*auto simp*: *relevant-entries-def matches-ex-entry-def is-relevant-entry-def split*: *if-split-asm*)

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*: $wf\text{-}jvm\text{-}prog_\Phi\ P$
assumes *h*: $h\ xcp = Some\ obj$
shows $\bigwedge f.\ P,\Phi \vdash (None,\ h,\ f\#frs,\ sh)\surd$
$\Longrightarrow curr\text{-}method\ f \neq clinit \Longrightarrow P,\Phi \vdash find\text{-}handler\ P\ xcp\ h\ frs\ sh\ \surd$
(**is** $\bigwedge f.\ ?correct\ (None,\ h,\ f\#frs,\ sh) \Longrightarrow ?prem\ f \Longrightarrow ?correct\ (?find\ frs)$)

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*:
$[\![\ fst\ (exec\text{-}instr\ (ins!pc)\ P\ h\ stk\ vars\ C\ M\ pc\ ics\ frs\ sh) = Some\ xcp;$
$P,T,mxs,size\ ins,xt \vdash ins!pc,pc :: \Phi\ C\ M;$
$P,\Phi \vdash (None,\ h,\ (stk,loc,C,M,pc,ics)\#frs,\ sh)\surd\ ]\!]$
$\Longrightarrow \exists\ obj.\ h\ xcp = Some\ obj$
(**is** $[\![\ ?xcpt;\ ?wt;\ ?correct\ ]\!] \Longrightarrow ?thesis$)
lemma *exec-step-xcpt-h*:
assumes *xcpt*: $fst\ (exec\text{-}step\ P\ h\ stk\ vars\ C\ M\ pc\ ics\ frs\ sh) = Some\ xcp$
and *ins*: $instrs\text{-}of\ P\ C\ M = ins$
and *wti*: $P,T,mxs,size\ ins,xt \vdash ins!pc,pc :: \Phi\ C\ M$
and *correct*: $P,\Phi \vdash (None,\ h,\ (stk,loc,C,M,pc,ics)\#frs,\ sh)\surd$
shows $\exists\ obj.\ h\ xcp = Some\ obj$
proof −
from *correct* **have** *pre*: *preallocated h* **by**(*simp add*: *defs1 hconf-def*)
{ fix C' *Cs* **assume** *ics*[*simp*]: $ics = Calling\ C'\ Cs$
with *xcpt* **have** *xcp = addr-of-sys-xcpt NoClassDefFoundError*
by(*cases ics*, *auto simp*: *split-beta split*: *init-state.splits if-split-asm*)
with *pre* **have** *?thesis* **using** *preallocated-def* **by** *force*
}
moreover
{ fix *Cs a* **assume** [*simp*]: *ics = Throwing Cs a*
with *xcpt* **have** *eq*: *a = xcp* **by**(*cases Cs*; *simp*)

from *correct* **have** $P,h,sh \vdash_i (C,M,pc,ics)$ **by**(*auto simp*: *defs1*)
with *eq* **have** *?thesis* **by** *simp*
}
moreover
{ fix *Cs* **assume** *ics*: $ics = No\text{-}ics \vee ics = Called\ Cs$
with *exec-instr-xcpt-h*[*OF - wti correct*] *xcpt ins* **have** *?thesis* **by**(*cases Cs*, *auto*)
}
ultimately show *?thesis* **by**(*cases ics*, *auto*)
qed

lemma *conf-sys-xcpt*:
$[\![preallocated\ h;\ C \in sys\text{-}xcpts]\!] \Longrightarrow P,h \vdash Addr\ (addr\text{-}of\text{-}sys\text{-}xcpt\ C) :\leq Class\ C$
by (*auto elim*: *preallocatedE*)

lemma *match-ex-table-SomeD*:
$match\text{-}ex\text{-}table\ P\ C\ pc\ xt = Some\ (pc',d') \Longrightarrow$
$\exists (f,t,D,h,d) \in set\ xt.\ matches\text{-}ex\text{-}entry\ P\ C\ pc\ (f,t,D,h,d) \wedge h = pc' \wedge d=d'$
by (*induct xt*) (*auto split*: *if-split-asm*)

Finally we can state that, whenever an exception occurs, the next state always conforms:

lemma *xcpt-correct*:
fixes σ' :: *jvm-state*
assumes *wtp*: $wf\text{-}jvm\text{-}prog_{\Phi}\ P$
assumes *meth*: $P \vdash C\ sees\ M,b{:}Ts{\rightarrow}T=(mxs,mxl_0,ins,xt)\ in\ C$
assumes *wt*: $P,T,mxs,size\ ins,xt \vdash ins!pc,pc :: \Phi\ C\ M$
assumes *xp*: $fst\ (exec\text{-}step\ P\ h\ stk\ loc\ C\ M\ pc\ ics\ frs\ sh) = Some\ xcp$
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 $P,\Phi \vdash \sigma'\surd$

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*: $wf\text{-}jvm\text{-}prog_{\Phi}\ P$
assumes *mC*: $P \vdash C\ sees\ M,b{:}Ts{\rightarrow}T=(mxs,mxl_0,ins,xt)\ in\ C$
assumes s': $Some\ \sigma' = exec\ (P,\ None,\ h,\ (stk,loc,C,M,pc,ics)\#frs,\ sh)$
assumes *cf*: $P,\Phi \vdash (None,\ h,\ (stk,loc,C,M,pc,ics)\#frs,\ sh)\surd$
assumes *xc*: $fst\ (exec\text{-}step\ P\ h\ stk\ loc\ C\ M\ pc\ ics\ frs\ sh) = None$
assumes *ics*: $ics = Calling\ C'\ Cs$

shows $P,\Phi \vdash \sigma'\surd$
proof −
from *wtprog* **obtain** *wfmb* **where** *wf*: *wf-prog wfmb P*
by (*simp add*: *wf-jvm-prog-phi-def*)

from *mC cf* **obtain** *ST LT* **where**
h-ok: $P \vdash h\ \surd$ **and**
sh-ok: $P,h \vdash_s sh\ \surd$ **and**
Φ: $\Phi\ C\ M\ !\ pc = Some\ (ST,LT)$ **and**

stk: $P,h \vdash stk\ [:\leq]\ ST$ **and** *loc*: $P,h \vdash loc\ [:\leq_\top]\ LT$ **and**
pc: $pc < size\ ins$ **and**
frame: *conf-f P h sh* (ST, LT) *ins* (stk,loc,C,M,pc,ics) **and**
fs: *conf-fs P h sh* Φ *C M* (*size Ts*) *T frs* **and**
confc: *conf-clinit P sh* ((*stk*,*loc*,*C*,*M*,*pc*,*ics*)#*frs*) **and**
vics: $P,h,sh \vdash_i (C,M,pc,ics)$
by (*fastforce dest*: *sees-method-fun*)

with *ics* **have** $confc_0$: *conf-clinit P sh* ((*stk*,*loc*,*C*,*M*,*pc*,*Calling* C' *Cs*)#*frs*) **by** *simp*

from *vics ics* **have** cls': *is-class P* C' **by** *auto*

{ assume *None*: *sh* $C' = None$

let $?sh = sh(C' \mapsto (sblank\ P\ C',\ Prepared))$

obtain *FDTs* **where**
flds: $P \vdash C'$ *has-fields FDTs* **using** *wf-Fields-Ex*[*OF wf* cls'] **by** *clarsimp*

from *shconf-upd-obj*[**where** $C=C'$, *OF sh-ok soconf-sblank*[*OF flds*]]
have *sh-ok*$'$: $P,h \vdash_s ?sh\ \surd$ **by** *simp*

from *None* **have** $\forall sfs.\ sh\ C' \neq Some(sfs,Processing)$ **by** *simp*
with *conf-clinit-nProc-dist*[*OF confc*] **have**
dist$'$: *distinct* (C' # *clinit-classes* ((*stk*, *loc*, *C*, *M*, *pc*, *ics*) # *frs*)) **by** *simp*
then have *dist*$''$: *distinct* (C' # *clinit-classes frs*) **by** *simp*

have *confc*$'$: *conf-clinit P ?sh* ((*stk*, *loc*, *C*, *M*, *pc*, *ics*) # *frs*)
by(*rule conf-clinit-shupd*[*OF confc dist*$'$])
have *fs*$'$: *conf-fs P h ?sh* Φ *C M* (*size Ts*) *T frs* **by**(*rule conf-fs-shupd*[*OF fs dist*$''$])
from *vics ics* **have** *vics*$'$: $P,h,?sh \vdash_i (C,\ M,\ pc,\ ics)$ **by** *auto*

from s' *ics None* **have** $\sigma' = (None,\ h,\ (stk,\ loc,\ C,\ M,\ pc,\ ics)\#frs,\ ?sh)$ **by** *auto*

with *mC h-ok sh-ok*$'$ Φ *stk loc pc fs*$'$ *confc vics*$'$ *confc*$'$ *frame None*
have *?thesis* **by** *fastforce*
}
moreover
{ fix *a* **assume** *sh* $C' = Some\ a$
then obtain *sfs i* **where** *shC*$'$[*simp*]: *sh* $C' = Some(sfs,i)$ **by**(*cases a*, *simp*)

from *confc ics* **have** *last*: $\exists sobj.\ sh\ (last(C'\#Cs)) = Some\ sobj$
by(*fastforce simp*: *conf-clinit-def*)

let $?f = \lambda ics'.\ (stk,\ loc,\ C,\ M,\ pc,\ ics'::init\text{-}call\text{-}status)$

{ assume *i*: $i = Done \lor i = Processing$
let $?ics = Called\ Cs$

from *last vics ics* **have** *vics*$'$: $P,h,sh \vdash_i (C,\ M,\ pc,\ ?ics)$ **by** *auto*
from *confc ics* **have** *confc*$'$: *conf-clinit P sh* (*?f ?ics*#*frs*)
by(*cases M=clinit*; *clarsimp simp*: *conf-clinit-def distinct-clinit-def*)

from *i* s' *ics* **have** $\sigma' = (None,\ h,\ ?f\ ?ics\#frs,\ sh)$ **by** *auto*

with *mC h-ok sh-ok* Φ *stk loc pc fs confc′ vics′ frame ics*
have *?thesis* **by** *fastforce*
}
moreover
{ assume *i*[*simp*]: *i* = *Error*
let *?a* = *addr-of-sys-xcpt NoClassDefFoundError*
let *?ics* = *Throwing Cs ?a*

from *h-ok* **have** *preh*: *preallocated h* **by** (*simp add*: *hconf-def*)
then obtain *obj* **where** *ha*: *h ?a* = *Some obj* **by**(*clarsimp simp*: *preallocated-def sys-xcpts-def*)
with *vics ics* **have** *vics′*: $P,h,sh \vdash_i (C, M, pc, ?ics)$ **by** *auto*

from *confc ics* **have** *confc′′*: *conf-clinit P sh* (*?f ?ics#frs*)
by(*cases M=clinit*; *clarsimp simp*: *conf-clinit-def distinct-clinit-def*)

from *s′ ics* **have** *σ′*: *σ′* = (*None*, *h*, *?f ?ics#frs*, *sh*) **by** *auto*

from *mC h-ok sh-ok* Φ *stk loc pc fs confc′′ vics σ′ ics ha*
have *?thesis* **by** *fastforce*
}
moreover
{ assume *i*[*simp*]: *i* = *Prepared*
let *?sh* = *sh*(*C′* ↦ (*sfs*,*Processing*))
let *?D* = *fst*(*the*(*class P C′*))
let *?ics* = *if C′ = Object then Called* (*C′#Cs*) *else Calling ?D* (*C′#Cs*)

from *shconf-upd-obj*[**where** *C=C′*, *OF sh-ok shconfD*[*OF sh-ok shC′*]]
have *sh-ok′*: $P,h \vdash_s ?sh \surd$ **by** *simp*

from *cls′* **have** $C' \neq Object \Longrightarrow P \vdash C' \preceq^* ?D$ **by**(*auto simp*: *is-class-def intro*!: *subcls1I*)
with *is-class-supclass*[*OF wf - cls′*] **have** *D*: $C' \neq Object \Longrightarrow$ *is-class P ?D* **by** *simp*

from *i* **have** ∀ *sfs*. *sh C′* ≠ *Some*(*sfs*,*Processing*) **by** *simp*
with *conf-clinit-nProc-dist*[*OF* $confc_0$] **have**
dist′: *distinct* (*C′* # *clinit-classes* ((*stk*, *loc*, *C*, *M*, *pc*, *Calling C′ Cs*) # *frs*)) **by** *fast*
then have *dist′′*: *distinct* (*C′* # *clinit-classes frs*) **by** *simp*

from *conf-clinit-shupd-Calling*[*OF* $confc_0$ *dist′ cls′*]
conf-clinit-shupd-Called[*OF* $confc_0$ *dist′ cls′*]
have *confc′*: *conf-clinit P ?sh* (*?f ?ics#frs*) **by** *clarsimp*
with *last ics* **have** ∃ *sobj*. *?sh* (*last*(*C′#Cs*)) = *Some sobj*
by(*auto simp*: *conf-clinit-def fun-upd-apply*)
with *D vics ics* **have** *vics′*: $P,h,?sh \vdash_i (C, M, pc, ?ics)$ **by** *auto*

have *fs′*: *conf-fs P h ?sh* Φ *C M* (*size Ts*) *T frs* **by**(*rule conf-fs-shupd*[*OF fs dist′′*])

from *frame vics′* **have** *frame′*: *conf-f P h ?sh* (*ST*, *LT*) *ins* (*?f ?ics*) **by** *simp*

from *i s′ ics* **have** *σ′* = (*None*, *h*, *?f ?ics#frs*, *?sh*) **by**(*auto simp*: *if-split-asm*)

with *mC h-ok sh-ok′* Φ *stk loc pc fs′ confc′ frame′ ics*
have *?thesis* **by** *fastforce*
}

ultimately have *?thesis* **by**(*cases i*, *auto*)
}
ultimately show *?thesis* **by**(*cases sh C'*, *auto*)
qed

lemma *Throwing-correct*:
fixes σ' :: *jvm-state*
assumes *wtprog*: $\textit{wf-jvm-prog}_\Phi$ *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* = *Throwing* (*C'*#*Cs*) *a*

shows $P,\Phi \vdash \sigma'\surd$
proof −
from *wtprog* **obtain** *wfmb* **where** *wf*: *wf-prog wfmb P*
by (*simp add*: *wf-jvm-prog-phi-def*)

from *mC cf* **obtain** *ST LT* **where**
h-ok: $P \vdash h\ \surd$ **and**
sh-ok: $P,h \vdash_s sh\ \surd$ **and**
Φ: Φ *C M* ! *pc* = *Some* (*ST*,*LT*) **and**
stk: $P,h \vdash stk\ [{:}\le]\ ST$ **and** *loc*: $P,h \vdash loc\ [{:}\le_\top]\ LT$ **and**
pc: *pc* < *size ins* **and**
frame: *conf-f P h sh* (*ST*, *LT*) *ins* (*stk*,*loc*,*C*,*M*,*pc*,*ics*) **and**
fs: *conf-fs P h sh* Φ *C M* (*size Ts*) *T frs* **and**
confc: *conf-clinit P sh* ((*stk*,*loc*,*C*,*M*,*pc*,*ics*)#*frs*) **and**
vics: $P,h,sh \vdash_i$ (*C*,*M*,*pc*,*ics*)
by (*fastforce dest*: *sees-method-fun*)

with *ics* **have** $\textit{confc}_0$: *conf-clinit P sh* ((*stk*,*loc*,*C*,*M*,*pc*,*Throwing* (*C'*#*Cs*) *a*)#*frs*) **by** *simp*

from *frame ics mC* **have**
cc: $\exists$ *C1*. *Called-context P C1* (*ins* ! *pc*) **by**(*clarsimp simp*: *conf-f-def2*)

from *frame ics* **obtain** *obj* **where** *ha*: *h a* = *Some obj* **by**(*auto simp*: *conf-f-def2*)

from *confc ics* **obtain** *sfs i* **where** *shC'*: *sh C'* = *Some*(*sfs*,*i*) **by**(*clarsimp simp*: *conf-clinit-def*)
then have *sfs*: $P,h,C' \vdash_s sfs\ \surd$ **by**(*rule shconfD*[*OF sh-ok*])

from *s' ics*
have σ': σ' = (*None*, *h*, (*stk*,*loc*,*C*,*M*,*pc*,*Throwing Cs a*)#*frs*, *sh*(*C'* $\mapsto$ (*fst*(*the*(*sh C'*)), *Error*)))
(**is** σ' = (*None*, *h*, *?f'*#*frs*, *?sh'*))
by *simp*

from *confc ics* **have** *dist*: *distinct* (*C'* # *clinit-classes* (*?f'* # *frs*))
by (*simp add*: *conf-clinit-def distinct-clinit-def*)
then have *dist'*: *distinct* (*C'* # *clinit-classes frs*) **by** *simp*

from *conf-clinit-Throwing confc ics* **have** *confc'*: *conf-clinit P sh* (*?f'* # *frs*) **by** *simp*

from *shconf-upd-obj*[*OF sh-ok sfs*] *shC'* **have** $P,h \vdash_s$ *?sh'* $\surd$ **by** *simp*
moreover

have *conf-fs P h ?sh′* Φ *C M* (*length Ts*) *T frs* **by**(*rule conf-fs-shupd*[*OF fs dist′*])
moreover
have *conf-clinit P ?sh′* (*?f′* # *frs*) **by**(*rule conf-clinit-shupd*[*OF confc′ dist*])
moreover note *σ′ h-ok mC* Φ *pc stk loc ha cc*
ultimately show *P*,Φ ⊢ *σ′* √ **by** *fastforce*
qed

lemma *Called-correct*:
fixes *σ′* :: *jvm-state*
assumes *wtprog*: $wf\text{-}jvm\text{-}prog_{\Phi}$ *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 −
from *wtprog* **obtain** *wfmb* **where** *wf*: *wf-prog wfmb P*
by (*simp add*: *wf-jvm-prog-phi-def*)

from *mC cf* **obtain** *ST LT* **where**
h-ok: *P* ⊢ *h* √ **and**
sh-ok: *P*,*h* $\vdash_s$ *sh* √ **and**
Φ: Φ *C M* ! *pc* = *Some* (*ST*,*LT*) **and**
stk: *P*,*h* ⊢ *stk* [:≤] *ST* **and** *loc*: *P*,*h* ⊢ *loc* [:$\leq_\top$] *LT* **and**
pc: *pc* < *size ins* **and**
frame: *conf-f P h sh* (*ST*, *LT*) *ins* (*stk*,*loc*,*C*,*M*,*pc*,*ics*) **and**
fs: *conf-fs P h sh* Φ *C M* (*size Ts*) *T frs* **and**
confc: *conf-clinit P sh* ((*stk*,*loc*,*C*,*M*,*pc*,*ics*)#*frs*) **and**
vics: *P*,*h*,*sh* $\vdash_i$ (*C*,*M*,*pc*,*ics*)
by (*fastforce dest*: *sees-method-fun*)

then have $confc_0$: *conf-clinit P sh* ((*stk*,*loc*,*C*,*M*,*pc*,*Called* (*C′*#*Cs*))#*frs*) **by** *simp*

from *frame mC* **obtain** *C1 sobj* **where**
ss: *Called-context P C1* (*ins* ! *pc*) **and**
shC1: *sh C1* = *Some sobj* **by**(*clarsimp simp*: *conf-f-def2*)

from *confc wf-sees-clinit*[*OF wf*] **obtain** *mxs′ mxl′ ins′ xt′* **where**
clinit: *P* ⊢ *C′ sees clinit*,*Static*: [] → *Void*=(*mxs′*,*mxl′*,*ins′*,*xt′*) *in C′*
by(*fastforce simp*: *conf-clinit-def is-class-def*)

let *?loc′* = *replicate mxl′ undefined*

from *s′ clinit*
have *σ′*: *σ′* = (*None*, *h*, ([],*?loc′*,*C′*,*clinit*,*0*,*No-ics*)#(*stk*,*loc*,*C*,*M*,*pc*,*Called Cs*)#*frs*, *sh*)
(**is** *σ′* = (*None*, *h*, *?if*#*?f′*#*frs*, *sh*))
by *simp*

with *wtprog clinit*
obtain *start*: *wt-start P C′ Static* [] *mxl′* (Φ *C′ clinit*) **and** *ins′*: *ins′* ≠ []
by (*auto dest*: *wt-jvm-prog-impl-wt-start*)
then obtain LT_0 **where** LT_0: Φ *C′ clinit* ! *0* = *Some* ([], LT_0)

by (*clarsimp simp*: *wt-start-def defs1 sup-state-opt-any-Some split*: *staticb.splits*)
moreover
have *conf-f P h sh* ([], LT_0) *ins′ ?if*
proof −
let *?LT* = *replicate mxl′ Err*
have *P*,*h* ⊢ *?loc′* [:≤⊤] *?LT* **by** *simp*
also from *start* LT_0 **have** *P* ⊢ . . . [≤⊤] LT_0 **by** (*simp add*: *wt-start-def*)
finally have *P*,*h* ⊢ *?loc′* [:≤⊤] LT_0 **.**
thus *?thesis* **using** *ins′* **by** *simp*
qed
moreover
from *conf-clinit-Called confc clinit* **have** *conf-clinit P sh* (*?if* # *?f′* # *frs*) **by** *simp*
moreover note *σ′ h-ok sh-ok mC* Φ *pc stk loc clinit ss shC1 fs*
ultimately show *P*,Φ ⊢ *σ′* √ **by** *fastforce*
qed

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*: $wf\text{-}jvm\text{-}prog_{\Phi}$ *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′*√
lemma *Invokestatic-nInit-correct*:
fixes *σ′* :: *jvm-state*
assumes *wtprog*: $wf\text{-}jvm\text{-}prog_{\Phi}$ *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*,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*
assumes *cs*: *ics* = *Called* [] ∨ (*ics* = *No-ics* ∧ (∃ *sfs*. *sh* (*fst*(*method P D M′*)) = *Some*(*sfs*, *Done*)))
shows *P*,Φ ⊢ *σ′*√
lemma *Invokestatic-Init-correct*:
fixes *σ′* :: *jvm-state*
assumes *wtprog*: $wf\text{-}jvm\text{-}prog_{\Phi}$ *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*,*No-ics*)#*frs*, *sh*)
assumes *approx*: *P*,Φ ⊢ (*None*, *h*, (*stk*,*loc*,*C*,*M*,*pc*,*No-ics*)#*frs*, *sh*)√
assumes *no-xcp*: *fst* (*exec-step P h stk loc C M pc No-ics frs sh*) = *None*
assumes *nDone*: ∀ *sfs*. *sh* (*fst*(*method P D M′*)) ≠ *Some*(*sfs*, *Done*)

shows $P,\Phi \vdash \sigma'\surd$
declare *list-all2-Cons2* [*iff*]

lemma *Return-correct*:
fixes σ' :: *jvm-state*
assumes *wt-prog*: $\mathit{wf\text{-}jvm\text{-}prog}_\Phi\ P$
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* 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 $P,\Phi \vdash \sigma'\surd$
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 σ' = *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,Φ |− σ' [*ok*]
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 σ' = *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,Φ |− σ' [*ok*]

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 σ' = *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,Φ |− σ' [*ok*]

2.19 Welltyped Programs produce no Type Errors

theory *BVNoTypeError*
imports *../JVM/JVMDefensive BVSpecTypeSafe*
begin

lemma *has-methodI*:
$P \vdash C$ *sees* $M,b{:}Ts{\rightarrow}T$ = *m in D* $\Longrightarrow P \vdash C$ *has* M,b
by (*unfold has-method-def*) *blast*

Some simple lemmas about the type testing functions of the defensive JVM:

lemma *typeof-NoneD* [*simp*,*dest*]: *typeof v* = *Some x* $\Longrightarrow$ ¬*is-Addr v*
  by (*cases v*) *auto*

lemma *is-Ref-def2*:
  is-Ref v = (*v* = *Null* ∨ (∃ *a*. *v* = *Addr a*))
  by (*cases v*) (*auto simp add*: *is-Ref-def*)

lemma [*iff*]: *is-Ref Null* **by** (*simp add*: *is-Ref-def2*)

lemma *is-RefI* [*intro*, *simp*]: *P*,*h* ⊢ *v* :≤ *T* $\Longrightarrow$ *is-refT T* $\Longrightarrow$ *is-Ref v*
lemma *is-IntgI* [*intro*, *simp*]: *P*,*h* ⊢ *v* :≤ *Integer* $\Longrightarrow$ *is-Intg v*
lemma *is-BoolI* [*intro*, *simp*]: *P*,*h* ⊢ *v* :≤ *Boolean* $\Longrightarrow$ *is-Bool v*
declare *defs1* [*simp del*]

lemma *wt-jvm-prog-states-NonStatic*:
assumes *wf*: *wf-jvm-prog*$_\Phi$ *P*
  and *mC*: *P* ⊢ *C sees M*,*NonStatic*: *Ts*→*T* = (*mxs*, *mxl*, *ins*, *et*) *in C*
  and Φ: Φ *C M* ! *pc* = τ **and** *pc*: *pc* < *size ins*
shows *OK* τ ∈ *states P mxs* (*1*+*size Ts*+*mxl*)
lemma *wt-jvm-prog-states-Static*:
assumes *wf*: *wf-jvm-prog*$_\Phi$ *P*
  and *mC*: *P* ⊢ *C sees M*,*Static*: *Ts*→*T* = (*mxs*, *mxl*, *ins*, *et*) *in C*
  and Φ: Φ *C M* ! *pc* = τ **and** *pc*: *pc* < *size ins*
shows *OK* τ ∈ *states P mxs* (*size Ts*+*mxl*)

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*,Φ ⊢ σ $\surd$
  shows *exec-d P* σ ≠ *TypeError*

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*,Φ ⊢ σ $\surd$ $\Longrightarrow$ *P* ⊢ σ −*jvm*→ σ′
  $\Longrightarrow$ *P* ⊢ (*Normal* σ) −*jvmd*→ (*Normal* σ′)

As corollary we get that the aggressive and the defensive machine are equivalent for welltyped 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*,Φ ⊢ σ $\surd$
  shows *P* ⊢ (*Normal* σ) −*jvmd*→ (*Normal* σ′) = *P* ⊢ σ −*jvm*→ σ′
proof(*rule iffI*)
  assume *P* ⊢ *Normal* σ −*jvmd*→ *Normal* σ′ **then show** *P* ⊢ σ −*jvm*→ σ′
    by (*rule defensive-imp-aggressive*)
next
  assume *P* ⊢ σ −*jvm*→ σ′ **then show** *P* ⊢ *Normal* σ −*jvmd*→ *Normal* σ′
    by (*rule welltyped-aggressive-imp-defensive* [*OF wf conforms*])
qed

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-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* σ) *−jvmd→* (*Normal* σ') = *start-prog P C M* $\vdash \sigma$ *−jvm→* σ'
proof −
 from *wf* **obtain** Φ **where** *wf'*: *wf-jvm-prog*$_\Phi$ *P* **by** (*auto simp*: *wf-jvm-prog-def*)
 let *?*Φ = Φ*-start* Φ
 from *start-prog-wf-jvm-prog-phi*[**where** Φ'=*?*Φ, *OF wf' nstart meth nclinit* Φ*-start Obj-start-m*]
 have *wf-jvm-prog*$_{?\Phi}$(*start-prog P C M*) **by** *simp*
 moreover
 from *wf' nstart meth nclinit* Φ*-start*(*2*) **have** *start-prog P C M*,*?*$\Phi \vdash \sigma \surd$
 unfolding *start* **by** (*rule BV-correct-initial*)
 ultimately show *?thesis* **by** (*rule welltyped-commutes*)
qed

lemma *not-TypeError-eq* [*iff*]:
 $x \neq$ *TypeError* = ($\exists$ *t*. x = *Normal t*)
 by (*cases x*) *auto*

locale *cnf* =
 fixes *P* **and** Φ **and** σ
 assumes *wf*: *wf-jvm-prog*$_\Phi$ *P*
 assumes *cnf*: *correct-state P* Φ σ

theorem (**in** *cnf*) *no-type-errors*:
 $P \vdash$ (*Normal* σ) *−jvmd→* $\sigma' \Longrightarrow \sigma' \neq$ *TypeError*
proof −
 assume $P \vdash$ (*Normal* σ) *−jvmd→* σ'
 then have (*Normal* σ, σ') $\in$ (*exec-1-d P*)* **by** (*unfold exec-all-d-def1*) *simp*
 then show *?thesis* **proof**(*induct rule*: *rtrancl-induct*)
 case (*step y z*)
 then obtain y_n **where** [*simp*]: y = *Normal* y_n **by** *clarsimp*
 have *n*σ*y*: $P \vdash$ *Normal* σ *−jvmd→* *Normal* y_n **using** *step.hyps*(*1*)
 by (*fold exec-all-d-def1*) *simp*
 have σ*y*: $P \vdash \sigma$ *−jvm→* y_n **using** *defensive-imp-aggressive*[*OF n*σ*y*] **by** *simp*
 show *?case* **using** *step no-type-error*[*OF wf BV-correct*[*OF wf* σ*y cnf*]]
 by (*auto simp add*: *exec-1-d-eq*)
 qed *simp*
qed

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-m*, b' : $Ts' \rightarrow T' = m'$ *in* D'

$\Longrightarrow b' = \mathit{Static} \wedge \mathit{Ts}' = [] \wedge T' = \mathit{Void})$
defines $\sigma \equiv \mathit{Normal}\ (\mathit{start\text{-}state}\ P)$
defines [*simp*]: $P_0 \equiv \mathit{start\text{-}prog}\ P\ C\ M$

corollary (**in** *start*) *bv-no-type-error*:
shows $P_0 \vdash \sigma\ \mathit{-jvmd}{\rightarrow}\ \sigma' \Longrightarrow \sigma' \neq \mathit{TypeError}$
proof −
from *wf* **obtain** Φ **where** *wf'*: $\mathit{wf\text{-}jvm\text{-}prog}_{\Phi}\ P$ **by** (*auto simp*: *wf-jvm-prog-def*)
let $?\Phi = \Phi\text{-}\mathit{start}\ \Phi$
from *start-prog-wf-jvm-prog-phi*[**where** $\Phi'{=}?\Phi$, *OF wf' nstart sees nclinit* Φ*-start Obj-start-m*]
have $\mathit{wf\text{-}jvm\text{-}prog}_{?\Phi} P_0$ **by** *simp*
moreover
from *BV-correct-initial*[**where** $\Phi'{=}?\Phi$, *OF wf' nstart sees nclinit* Φ*-start*(*2*)]
have *correct-state* $P_0\ ?\Phi$ (*start-state* P) **by** *simp*
ultimately have *cnf* $P_0\ ?\Phi$ (*start-state* P) **by** (*rule cnf.intro*)
moreover assume $P_0 \vdash \sigma\ \mathit{-jvmd}{\rightarrow}\ \sigma'$
ultimately show *?thesis* **by** (*unfold* σ*-def*) (*rule cnf.no-type-errors*)
qed

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*($C\bullet_s F\{D\}$) = *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*($C\bullet_s M(es)$) = *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*)

| $max\text{-}vars(throw\ e) = max\text{-}vars\ e$
| $max\text{-}vars(try\ e_1\ catch(C\ V)\ e_2) = max\ (max\text{-}vars\ e_1)\ (max\text{-}vars\ e_2 + 1)$
| $max\text{-}vars(INIT\ C\ (Cs,b) \leftarrow e) = max\text{-}vars\ e$
| $max\text{-}vars(RI(C,e);Cs \leftarrow e') = max\ (max\text{-}vars\ e)\ (max\text{-}vars\ e')$

| $max\text{-}varss\ [] = 0$
| $max\text{-}varss\ (e\#es) = max\ (max\text{-}vars\ e)\ (max\text{-}varss\ es)$

inductive
$eval_1 :: J_1\text{-}prog \Rightarrow expr_1 \Rightarrow state_1 \Rightarrow expr_1 \Rightarrow state_1 \Rightarrow bool$
$(\langle\text{-} \vdash_1 ((1\langle\text{-},/\text{-}\rangle) \Rightarrow/\ (1\langle\text{-},/\text{-}\rangle))\rangle\ [51,0,0,0,0]\ 81)$
and $evals_1 :: J_1\text{-}prog \Rightarrow expr_1\ list \Rightarrow state_1 \Rightarrow expr_1\ list \Rightarrow state_1 \Rightarrow bool$
$(\langle\text{-} \vdash_1 ((1\langle\text{-},/\text{-}\rangle)\ [\Rightarrow]/\ (1\langle\text{-},/\text{-}\rangle))\rangle\ [51,0,0,0,0]\ 81)$
for $P :: J_1\text{-}prog$
where

New_1:
$[\![\ sh\ C = Some\ (sfs,\ Done);\ new\text{-}Addr\ h = Some\ a;$
$P \vdash C\ has\text{-}fields\ FDTs;\ h' = h(a\mapsto blank\ P\ C)\ ]\!]$
$\Longrightarrow P \vdash_1 \langle new\ C,(h,l,sh)\rangle \Rightarrow \langle addr\ a,(h',l,sh)\rangle$
| $NewFail_1$:
$[\![\ sh\ C = Some\ (sfs,\ Done);\ new\text{-}Addr\ h = None\ ]\!] \Longrightarrow$
$P \vdash_1 \langle new\ C,\ (h,l,sh)\rangle \Rightarrow \langle THROW\ OutOfMemory,(h,l,sh)\rangle$
| $NewInit_1$:
$[\![\ \nexists sfs.\ sh\ C = Some\ (sfs,\ Done);\ P \vdash_1 \langle INIT\ C\ ([C],False) \leftarrow unit,(h,l,sh)\rangle \Rightarrow \langle Val\ v',(h',l',sh')\rangle;$
$new\text{-}Addr\ h' = Some\ a;\ P \vdash C\ has\text{-}fields\ FDTs;\ h'' = h'(a\mapsto blank\ P\ C)\ ]\!]$
$\Longrightarrow P \vdash_1 \langle new\ C,(h,l,sh)\rangle \Rightarrow \langle addr\ a,(h'',l',sh')\rangle$
| $NewInitOOM_1$:
$[\![\ \nexists sfs.\ sh\ C = Some\ (sfs,\ Done);\ P \vdash_1 \langle INIT\ C\ ([C],False) \leftarrow unit,(h,l,sh)\rangle \Rightarrow \langle Val\ v',(h',l',sh')\rangle;$
$new\text{-}Addr\ h' = None;\ is\text{-}class\ P\ C\ ]\!]$
$\Longrightarrow P \vdash_1 \langle new\ C,(h,l,sh)\rangle \Rightarrow \langle THROW\ OutOfMemory,(h',l',sh')\rangle$
| $NewInitThrow_1$:
$[\![\ \nexists sfs.\ sh\ C = Some\ (sfs,\ Done);\ P \vdash_1 \langle INIT\ C\ ([C],False) \leftarrow unit,(h,l,sh)\rangle \Rightarrow \langle throw\ a,s'\rangle;$
$is\text{-}class\ P\ C\ ]\!]$
$\Longrightarrow P \vdash_1 \langle new\ C,(h,l,sh)\rangle \Rightarrow \langle throw\ a,s'\rangle$

| $Cast_1$:
$[\![\ P \vdash_1 \langle e,s_0\rangle \Rightarrow \langle addr\ a,(h,l,sh)\rangle;\ h\ a = Some(D,fs);\ P \vdash D \preceq^* C\ ]\!]$
$\Longrightarrow P \vdash_1 \langle Cast\ C\ e,s_0\rangle \Rightarrow \langle addr\ a,(h,l,sh)\rangle$
| $CastNull_1$:
$P \vdash_1 \langle e,s_0\rangle \Rightarrow \langle null,s_1\rangle \Longrightarrow$
$P \vdash_1 \langle Cast\ C\ e,s_0\rangle \Rightarrow \langle null,s_1\rangle$
| $CastFail_1$:
$[\![\ P \vdash_1 \langle e,s_0\rangle \Rightarrow \langle addr\ a,(h,l,sh)\rangle;\ h\ a = Some(D,fs);\ \neg\ P \vdash D \preceq^* C\ ]\!]$
$\Longrightarrow P \vdash_1 \langle Cast\ C\ e,s_0\rangle \Rightarrow \langle THROW\ ClassCast,(h,l,sh)\rangle$
| $CastThrow_1$:
$P \vdash_1 \langle e,s_0\rangle \Rightarrow \langle throw\ e',s_1\rangle \Longrightarrow$
$P \vdash_1 \langle Cast\ C\ e,s_0\rangle \Rightarrow \langle throw\ e',s_1\rangle$

| Val_1:
$P \vdash_1 \langle Val\ v,s\rangle \Rightarrow \langle Val\ v,s\rangle$

| $BinOp_1$:
$[\![\ P \vdash_1 \langle e_1,s_0\rangle \Rightarrow \langle Val\ v_1,s_1\rangle;\ P \vdash_1 \langle e_2,s_1\rangle \Rightarrow \langle Val\ v_2,s_2\rangle;\ binop(bop,v_1,v_2) = Some\ v\ ]\!]$

$\Longrightarrow P \vdash_1 \langle e_1 \; «bop» \; e_2,s_0\rangle \Rightarrow \langle Val\ v,s_2\rangle$
| $BinOpThrow_{11}$:
$P \vdash_1 \langle e_1,s_0\rangle \Rightarrow \langle throw\ e,s_1\rangle \Longrightarrow$
$P \vdash_1 \langle e_1 \; «bop» \; e_2,\ s_0\rangle \Rightarrow \langle throw\ e,s_1\rangle$
| $BinOpThrow_{21}$:
$[\![P \vdash_1 \langle e_1,s_0\rangle \Rightarrow \langle Val\ v_1,s_1\rangle;\ P \vdash_1 \langle e_2,s_1\rangle \Rightarrow \langle throw\ e,s_2\rangle]\!]$
$\Longrightarrow P \vdash_1 \langle e_1 \; «bop» \; e_2,s_0\rangle \Rightarrow \langle throw\ e,s_2\rangle$

| Var_1:
$[\![ls!i = v;\ i < size\ ls]\!] \Longrightarrow$
$P \vdash_1 \langle Var\ i,(h,ls,sh)\rangle \Rightarrow \langle Val\ v,(h,ls,sh)\rangle$

| $LAss_1$:
$[\![P \vdash_1 \langle e,s_0\rangle \Rightarrow \langle Val\ v,(h,ls,sh)\rangle;\ i < size\ ls;\ ls' = ls[i := v]]\!]$
$\Longrightarrow P \vdash_1 \langle i{:=}\ e,s_0\rangle \Rightarrow \langle unit,(h,ls',sh)\rangle$
| $LAssThrow_1$:
$P \vdash_1 \langle e,s_0\rangle \Rightarrow \langle throw\ e',s_1\rangle \Longrightarrow$
$P \vdash_1 \langle i{:=}\ e,s_0\rangle \Rightarrow \langle throw\ e',s_1\rangle$

| $FAcc_1$:
$[\![P \vdash_1 \langle e,s_0\rangle \Rightarrow \langle addr\ a,(h,ls,sh)\rangle;\ h\ a = Some(C,fs);$
$P \vdash C\ has\ F,NonStatic{:}t\ in\ D;$
$fs(F,D) = Some\ v]\!]$
$\Longrightarrow P \vdash_1 \langle e{\bullet}F\{D\},s_0\rangle \Rightarrow \langle Val\ v,(h,ls,sh)\rangle$
| $FAccNull_1$:
$P \vdash_1 \langle e,s_0\rangle \Rightarrow \langle null,s_1\rangle \Longrightarrow$
$P \vdash_1 \langle e{\bullet}F\{D\},s_0\rangle \Rightarrow \langle THROW\ NullPointer,s_1\rangle$
| $FAccThrow_1$:
$P \vdash_1 \langle e,s_0\rangle \Rightarrow \langle throw\ e',s_1\rangle \Longrightarrow$
$P \vdash_1 \langle e{\bullet}F\{D\},s_0\rangle \Rightarrow \langle throw\ e',s_1\rangle$
| $FAccNone_1$:
$[\![P \vdash_1 \langle e,s_0\rangle \Rightarrow \langle addr\ a,(h,ls,sh)\rangle;\ h\ a = Some(C,fs);$
$\neg(\exists\, b\ t.\ P \vdash C\ has\ F,b{:}t\ in\ D)]\!]$
$\Longrightarrow P \vdash_1 \langle e{\bullet}F\{D\},s_0\rangle \Rightarrow \langle THROW\ NoSuchFieldError,(h,ls,sh)\rangle$
| $FAccStatic_1$:
$[\![P \vdash_1 \langle e,s_0\rangle \Rightarrow \langle addr\ a,(h,ls,sh)\rangle;\ h\ a = Some(C,fs);$
$P \vdash C\ has\ F,Static{:}t\ in\ D]\!]$
$\Longrightarrow P \vdash_1 \langle e{\bullet}F\{D\},s_0\rangle \Rightarrow \langle THROW\ IncompatibleClassChangeError,(h,ls,sh)\rangle$

| $SFAcc_1$:
$[\![P \vdash C\ has\ F,Static{:}t\ in\ D;$
$sh\ D = Some\ (sfs,Done);$
$sfs\ F = Some\ v]\!]$
$\Longrightarrow P \vdash_1 \langle C{\bullet}_sF\{D\},(h,ls,sh)\rangle \Rightarrow \langle Val\ v,(h,ls,sh)\rangle$
| $SFAccInit_1$:
$[\![P \vdash C\ has\ F,Static{:}t\ in\ D;$
$\nexists sfs.\ sh\ D = Some\ (sfs,Done);\ P \vdash_1 \langle INIT\ D\ ([D],False) \leftarrow unit,(h,ls,sh)\rangle \Rightarrow \langle Val\ v',(h',ls',sh')\rangle;$
$sh'\ D = Some\ (sfs,i);$
$sfs\ F = Some\ v]\!]$
$\Longrightarrow P \vdash_1 \langle C{\bullet}_sF\{D\},(h,ls,sh)\rangle \Rightarrow \langle Val\ v,(h',ls',sh')\rangle$
| $SFAccInitThrow_1$:
$[\![P \vdash C\ has\ F,Static{:}t\ in\ D;$
$\nexists sfs.\ sh\ D = Some\ (sfs,Done);\ P \vdash_1 \langle INIT\ D\ ([D],False) \leftarrow unit,(h,ls,sh)\rangle \Rightarrow \langle throw\ a,s'\rangle]\!]$
$\Longrightarrow P \vdash_1 \langle C{\bullet}_sF\{D\},(h,ls,sh)\rangle \Rightarrow \langle throw\ a,s'\rangle$

| $SFAccNone_1$:
$\llbracket\ \neg(\exists\, b\ t.\ P \vdash C\ has\ F{,}b{:}t\ in\ D)\ \rrbracket$
$\Longrightarrow P \vdash_1 \langle C{\boldsymbol{\cdot}}_s F\{D\}{,}s\rangle \Rightarrow \langle THROW\ NoSuchFieldError{,}s\rangle$
| $SFAccNonStatic_1$:
$\llbracket\ P \vdash C\ has\ F{,}NonStatic{:}t\ in\ D\ \rrbracket$
$\Longrightarrow P \vdash_1 \langle C{\boldsymbol{\cdot}}_s F\{D\}{,}s\rangle \Rightarrow \langle THROW\ IncompatibleClassChangeError{,}s\rangle$

| $FAss_1$:
$\llbracket\ P \vdash_1 \langle e_1{,}s_0\rangle \Rightarrow \langle addr\ a{,}s_1\rangle;\ P \vdash_1 \langle e_2{,}s_1\rangle \Rightarrow \langle Val\ v{,}(h_2{,}l_2{,}sh_2)\rangle;$
$h_2\ a = Some(C{,}fs);\ P \vdash C\ has\ F{,}NonStatic{:}t\ in\ D;$
$fs' = fs((F{,}D)\mapsto v);\ h_2{}' = h_2(a\mapsto(C{,}fs'))\ \rrbracket$
$\Longrightarrow P \vdash_1 \langle e_1{\boldsymbol{\cdot}}F\{D\}{:=}e_2{,}s_0\rangle \Rightarrow \langle unit{,}(h_2{}'{,}l_2{,}sh_2)\rangle$
| $FAssNull_1$:
$\llbracket\ P \vdash_1 \langle e_1{,}s_0\rangle \Rightarrow \langle null{,}s_1\rangle;\ \ P \vdash_1 \langle e_2{,}s_1\rangle \Rightarrow \langle Val\ v{,}s_2\rangle\ \rrbracket \Longrightarrow$
$P \vdash_1 \langle e_1{\boldsymbol{\cdot}}F\{D\}{:=}e_2{,}s_0\rangle \Rightarrow \langle THROW\ NullPointer{,}s_2\rangle$
| $FAssThrow_{11}$:
$P \vdash_1 \langle e_1{,}s_0\rangle \Rightarrow \langle throw\ e'{,}s_1\rangle \Longrightarrow$
$P \vdash_1 \langle e_1{\boldsymbol{\cdot}}F\{D\}{:=}e_2{,}s_0\rangle \Rightarrow \langle throw\ e'{,}s_1\rangle$
| $FAssThrow_{21}$:
$\llbracket\ P \vdash_1 \langle e_1{,}s_0\rangle \Rightarrow \langle Val\ v{,}s_1\rangle;\ P \vdash_1 \langle e_2{,}s_1\rangle \Rightarrow \langle throw\ e'{,}s_2\rangle\ \rrbracket$
$\Longrightarrow P \vdash_1 \langle e_1{\boldsymbol{\cdot}}F\{D\}{:=}e_2{,}s_0\rangle \Rightarrow \langle throw\ e'{,}s_2\rangle$
| $FAssNone_1$:
$\llbracket\ P \vdash_1 \langle e_1{,}s_0\rangle \Rightarrow \langle addr\ a{,}s_1\rangle;\ P \vdash_1 \langle e_2{,}s_1\rangle \Rightarrow \langle Val\ v{,}(h_2{,}l_2{,}sh_2)\rangle;$
$h_2\ a = Some(C{,}fs);\ \neg(\exists\, b\ t.\ P \vdash C\ has\ F{,}b{:}t\ in\ D)\ \rrbracket$
$\Longrightarrow P \vdash_1 \langle e_1{\boldsymbol{\cdot}}F\{D\}{:=}e_2{,}s_0\rangle \Rightarrow \langle THROW\ NoSuchFieldError{,}(h_2{,}l_2{,}sh_2)\rangle$
| $FAssStatic_1$:
$\llbracket\ P \vdash_1 \langle e_1{,}s_0\rangle \Rightarrow \langle addr\ a{,}s_1\rangle;\ P \vdash_1 \langle e_2{,}s_1\rangle \Rightarrow \langle Val\ v{,}(h_2{,}l_2{,}sh_2)\rangle;$
$h_2\ a = Some(C{,}fs);\ P \vdash C\ has\ F{,}Static{:}t\ in\ D\ \rrbracket$
$\Longrightarrow P \vdash_1 \langle e_1{\boldsymbol{\cdot}}F\{D\}{:=}e_2{,}s_0\rangle \Rightarrow \langle THROW\ IncompatibleClassChangeError{,}(h_2{,}l_2{,}sh_2)\rangle$

| $SFAss_1$:
$\llbracket\ P \vdash_1 \langle e_2{,}s_0\rangle \Rightarrow \langle Val\ v{,}(h_1{,}l_1{,}sh_1)\rangle;$
$P \vdash C\ has\ F{,}Static{:}t\ in\ D;$
$sh_1\ D = Some(sfs{,}Done);\ sfs' = sfs(F\mapsto v);\ sh_1{}' = sh_1(D\mapsto(sfs'{,}Done))\ \rrbracket$
$\Longrightarrow P \vdash_1 \langle C{\boldsymbol{\cdot}}_s F\{D\}{:=}e_2{,}s_0\rangle \Rightarrow \langle unit{,}(h_1{,}l_1{,}sh_1{}')\rangle$
| $SFAssInit_1$:
$\llbracket\ P \vdash_1 \langle e_2{,}s_0\rangle \Rightarrow \langle Val\ v{,}(h_1{,}l_1{,}sh_1)\rangle;$
$P \vdash C\ has\ F{,}Static{:}t\ in\ D;$
$\nexists sfs.\ sh_1\ D = Some(sfs{,}Done);\ P \vdash_1 \langle INIT\ D\ ([D]{,}False) \leftarrow unit{,}(h_1{,}l_1{,}sh_1)\rangle \Rightarrow \langle Val\ v'{,}(h'{,}l'{,}sh')\rangle;$
$sh'\ D = Some(sfs{,}i);$
$sfs' = sfs(F\mapsto v);\ sh'' = sh'(D\mapsto(sfs'{,}i))\ \rrbracket$
$\Longrightarrow P \vdash_1 \langle C{\boldsymbol{\cdot}}_s F\{D\}{:=}e_2{,}s_0\rangle \Rightarrow \langle unit{,}(h'{,}l'{,}sh'')\rangle$
| $SFAssInitThrow_1$:
$\llbracket\ P \vdash_1 \langle e_2{,}s_0\rangle \Rightarrow \langle Val\ v{,}(h_1{,}l_1{,}sh_1)\rangle;$
$P \vdash C\ has\ F{,}Static{:}t\ in\ D;$
$\nexists sfs.\ sh_1\ D = Some(sfs{,}Done);\ P \vdash_1 \langle INIT\ D\ ([D]{,}False) \leftarrow unit{,}(h_1{,}l_1{,}sh_1)\rangle \Rightarrow \langle throw\ a{,}s'\rangle\ \rrbracket$
$\Longrightarrow P \vdash_1 \langle C{\boldsymbol{\cdot}}_s F\{D\}{:=}e_2{,}s_0\rangle \Rightarrow \langle throw\ a{,}s'\rangle$
| $SFAssThrow_1$:
$P \vdash_1 \langle e_2{,}s_0\rangle \Rightarrow \langle throw\ e'{,}s_2\rangle$
$\Longrightarrow P \vdash_1 \langle C{\boldsymbol{\cdot}}_s F\{D\}{:=}e_2{,}s_0\rangle \Rightarrow \langle throw\ e'{,}s_2\rangle$
| $SFAssNone_1$:
$\llbracket\ P \vdash_1 \langle e_2{,}s_0\rangle \Rightarrow \langle Val\ v{,}(h_2{,}l_2{,}sh_2)\rangle;$
$\neg(\exists\, b\ t.\ P \vdash C\ has\ F{,}b{:}t\ in\ D)\ \rrbracket$

$\Longrightarrow P \vdash_1 \langle C\bullet_s F\{D\}{:=}e_2,s_0\rangle \Rightarrow \langle THROW\ NoSuchFieldError,(h_2,l_2,sh_2)\rangle$
| $SFAssNonStatic_1$:
$[\![\ P \vdash_1 \langle e_2,s_0\rangle \Rightarrow \langle Val\ v,(h_2,l_2,sh_2)\rangle;$
$P \vdash C\ has\ F,NonStatic{:}t\ in\ D\ ]\!]$
$\Longrightarrow P \vdash_1 \langle C\bullet_s F\{D\}{:=}e_2,s_0\rangle \Rightarrow \langle THROW\ IncompatibleClassChangeError,(h_2,l_2,sh_2)\rangle$

| $CallObjThrow_1$:
$P \vdash_1 \langle e,s_0\rangle \Rightarrow \langle throw\ e',s_1\rangle \Longrightarrow$
$P \vdash_1 \langle e\bullet M(es),s_0\rangle \Rightarrow \langle throw\ e',s_1\rangle$
| $CallNull_1$:
$[\![\ P \vdash_1 \langle e,s_0\rangle \Rightarrow \langle null,s_1\rangle;\ P \vdash_1 \langle es,s_1\rangle\ [\Rightarrow]\ \langle map\ Val\ vs,s_2\rangle\ ]\!]$
$\Longrightarrow P \vdash_1 \langle e\bullet M(es),s_0\rangle \Rightarrow \langle THROW\ NullPointer,s_2\rangle$
| $Call_1$:
$[\![\ P \vdash_1 \langle e,s_0\rangle \Rightarrow \langle addr\ a,s_1\rangle;\ P \vdash_1 \langle es,s_1\rangle\ [\Rightarrow]\ \langle map\ Val\ vs,(h_2,ls_2,sh_2)\rangle;$
$h_2\ a = Some(C,fs);\ P \vdash C\ sees\ M,NonStatic{:}Ts{\rightarrow}T = body\ in\ D;$
$size\ vs = size\ Ts;\ ls_2{}' = (Addr\ a)\ \#\ vs\ @\ replicate\ (max\text{-}vars\ body)\ undefined;$
$P \vdash_1 \langle body,(h_2,ls_2{}',sh_2)\rangle \Rightarrow \langle e',(h_3,ls_3,sh_3)\rangle\ ]\!]$
$\Longrightarrow P \vdash_1 \langle e\bullet M(es),s_0\rangle \Rightarrow \langle e',(h_3,ls_2,sh_3)\rangle$
| $CallParamsThrow_1$:
$[\![\ P \vdash_1 \langle e,s_0\rangle \Rightarrow \langle Val\ v,s_1\rangle;\ P \vdash_1 \langle es,s_1\rangle\ [\Rightarrow]\ \langle es',s_2\rangle;$
$es' = map\ Val\ vs\ @\ throw\ ex\ \#\ es_2\ ]\!]$
$\Longrightarrow P \vdash_1 \langle e\bullet M(es),s_0\rangle \Rightarrow \langle throw\ ex,s_2\rangle$
| $CallNone_1$:
$[\![\ P \vdash_1 \langle e,s_0\rangle \Rightarrow \langle addr\ a,s_1\rangle;\ \ P \vdash_1 \langle ps,s_1\rangle\ [\Rightarrow]\ \langle map\ Val\ vs,(h_2,ls_2,sh_2)\rangle;$
$h_2\ a = Some(C,fs);\ \neg(\exists\, b\ Ts\ T\ body\ D.\ P \vdash C\ sees\ M,b{:}Ts{\rightarrow}T = body\ in\ D)\ ]\!]$
$\Longrightarrow P \vdash_1 \langle e\bullet M(ps),s_0\rangle \Rightarrow \langle THROW\ NoSuchMethodError,(h_2,ls_2,sh_2)\rangle$
| $CallStatic_1$:
$[\![\ P \vdash_1 \langle e,s_0\rangle \Rightarrow \langle addr\ a,s_1\rangle;\ \ P \vdash_1 \langle ps,s_1\rangle\ [\Rightarrow]\ \langle map\ Val\ vs,(h_2,ls_2,sh_2)\rangle;$
$h_2\ a = Some(C,fs);\ P \vdash C\ sees\ M,Static{:}Ts{\rightarrow}T = body\ in\ D\ ]\!]$
$\Longrightarrow P \vdash_1 \langle e\bullet M(ps),s_0\rangle \Rightarrow \langle THROW\ IncompatibleClassChangeError,(h_2,ls_2,sh_2)\rangle$

| $SCallParamsThrow_1$:
$[\![\ P \vdash_1 \langle es,s_0\rangle\ [\Rightarrow]\ \langle es',s_2\rangle;\ es' = map\ Val\ vs\ @\ throw\ ex\ \#\ es_2\ ]\!]$
$\Longrightarrow P \vdash_1 \langle C\bullet_s M(es),s_0\rangle \Rightarrow \langle throw\ ex,s_2\rangle$
| $SCallNone_1$:
$[\![\ P \vdash_1 \langle ps,s_0\rangle\ [\Rightarrow]\ \langle map\ Val\ vs,s_2\rangle;$
$\neg(\exists\, b\ Ts\ T\ body\ D.\ P \vdash C\ sees\ M,b{:}Ts{\rightarrow}T = body\ in\ D)\ ]\!]$
$\Longrightarrow P \vdash_1 \langle C\bullet_s M(ps),s_0\rangle \Rightarrow \langle THROW\ NoSuchMethodError,s_2\rangle$
| $SCallNonStatic_1$:
$[\![\ P \vdash_1 \langle ps,s_0\rangle\ [\Rightarrow]\ \langle map\ Val\ vs,s_2\rangle;$
$P \vdash C\ sees\ M,NonStatic{:}Ts{\rightarrow}T = body\ in\ D\ ]\!]$
$\Longrightarrow P \vdash_1 \langle C\bullet_s M(ps),s_0\rangle \Rightarrow \langle THROW\ IncompatibleClassChangeError,s_2\rangle$
| $SCallInitThrow_1$:
$[\![\ P \vdash_1 \langle ps,s_0\rangle\ [\Rightarrow]\ \langle map\ Val\ vs,(h_1,ls_1,sh_1)\rangle;$
$P \vdash C\ sees\ M,Static{:}Ts{\rightarrow}T = body\ in\ D;$
$\nexists\, sfs.\ sh_1\ D = Some(sfs,Done);\ M \neq clinit;$
$P \vdash_1 \langle INIT\ D\ ([D],False) \leftarrow unit,(h_1,ls_1,sh_1)\rangle \Rightarrow \langle throw\ a,s'\rangle\ ]\!]$
$\Longrightarrow P \vdash_1 \langle C\bullet_s M(ps),s_0\rangle \Rightarrow \langle throw\ a,s'\rangle$
| $SCallInit_1$:
$[\![\ P \vdash_1 \langle ps,s_0\rangle\ [\Rightarrow]\ \langle map\ Val\ vs,(h_1,ls_1,sh_1)\rangle;$
$P \vdash C\ sees\ M,Static{:}Ts{\rightarrow}T = body\ in\ D;$
$\nexists\, sfs.\ sh_1\ D = Some(sfs,Done);\ M \neq clinit;$
$P \vdash_1 \langle INIT\ D\ ([D],False) \leftarrow unit,(h_1,ls_1,sh_1)\rangle \Rightarrow \langle Val\ v',(h_2,ls_2,sh_2)\rangle;$
$size\ vs = size\ Ts;\ ls_2{}' = vs\ @\ replicate\ (max\text{-}vars\ body)\ undefined;$

$P \vdash_1 \langle body,(h_2,ls_2{}',sh_2)\rangle \Rightarrow \langle e',(h_3,ls_3,sh_3)\rangle \rrbracket$
$\Longrightarrow P \vdash_1 \langle C\cdot_s M(ps),s_0\rangle \Rightarrow \langle e',(h_3,ls_2,sh_3)\rangle$
| $SCall_1$:
$\llbracket P \vdash_1 \langle ps,s_0\rangle [\Rightarrow] \langle map\ Val\ vs,(h_2,ls_2,sh_2)\rangle;$
$P \vdash C\ sees\ M,Static{:}Ts{\rightarrow}T = body\ in\ D;$
$sh_2\ D = Some(sfs,Done) \vee (M = clinit \wedge sh_2\ D = \lfloor(sfs,\ Processing)\rfloor);$
$size\ vs = size\ Ts;\ ls_2{}' = vs\ @\ replicate\ (max\text{-}vars\ body)\ undefined;$
$P \vdash_1 \langle body,(h_2,ls_2{}',sh_2)\rangle \Rightarrow \langle e',(h_3,ls_3,sh_3)\rangle \rrbracket$
$\Longrightarrow P \vdash_1 \langle C\cdot_s M(ps),s_0\rangle \Rightarrow \langle e',(h_3,ls_2,sh_3)\rangle$

| $Block_1$:
$P \vdash_1 \langle e,s_0\rangle \Rightarrow \langle e',s_1\rangle \Longrightarrow P \vdash_1 \langle Block\ i\ T\ e,s_0\rangle \Rightarrow \langle e',s_1\rangle$

| 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

| $InitFinal_1$:
$P \vdash_1 \langle e,s\rangle \Rightarrow \langle e',s'\rangle \Longrightarrow P \vdash_1 \langle INIT\ C\ (Nil,b) \leftarrow e,s\rangle \Rightarrow \langle e',s'\rangle$
| $InitNone_1$:
$[\![\ sh\ C = None;\ P \vdash_1 \langle INIT\ C'\ (C\#Cs,False) \leftarrow e,(h,l,sh(C \mapsto (sblank\ P\ C,\ Prepared)))\rangle \Rightarrow \langle e',s'\rangle$
$]\!]$
$\Longrightarrow P \vdash_1 \langle INIT\ C'\ (C\#Cs,False) \leftarrow e,(h,l,sh)\rangle \Rightarrow \langle e',s'\rangle$
| $InitDone_1$:
$[\![\ sh\ C = Some(sfs,Done);\ P \vdash_1 \langle INIT\ C'\ (Cs,True) \leftarrow e,(h,l,sh)\rangle \Rightarrow \langle e',s'\rangle\ ]\!]$
$\Longrightarrow P \vdash_1 \langle INIT\ C'\ (C\#Cs,False) \leftarrow e,(h,l,sh)\rangle \Rightarrow \langle e',s'\rangle$
| $InitProcessing_1$:
$[\![\ sh\ C = Some(sfs,Processing);\ P \vdash_1 \langle INIT\ C'\ (Cs,True) \leftarrow e,(h,l,sh)\rangle \Rightarrow \langle e',s'\rangle\ ]\!]$
$\Longrightarrow P \vdash_1 \langle INIT\ C'\ (C\#Cs,False) \leftarrow e,(h,l,sh)\rangle \Rightarrow \langle e',s'\rangle$
| $InitError_1$:
$[\![\ sh\ C = Some(sfs,Error);$
$P \vdash_1 \langle RI\ (C,\ THROW\ NoClassDefFoundError);Cs \leftarrow e,(h,l,sh)\rangle \Rightarrow \langle e',s'\rangle\ ]\!]$
$\Longrightarrow P \vdash_1 \langle INIT\ C'\ (C\#Cs,False) \leftarrow e,(h,l,sh)\rangle \Rightarrow \langle e',s'\rangle$
| $InitObject_1$:
$[\![\ sh\ C = Some(sfs,Prepared);$
$C = Object;$
$sh' = sh(C \mapsto (sfs,Processing));$
$P \vdash_1 \langle INIT\ C'\ (C\#Cs,True) \leftarrow e,(h,l,sh')\rangle \Rightarrow \langle e',s'\rangle\ ]\!]$
$\Longrightarrow P \vdash_1 \langle INIT\ C'\ (C\#Cs,False) \leftarrow e,(h,l,sh)\rangle \Rightarrow \langle e',s'\rangle$
| $InitNonObject_1$:
$[\![\ sh\ C = Some(sfs,Prepared);$
$C \neq Object;$
$class\ P\ C = Some\ (D,r);$
$sh' = sh(C \mapsto (sfs,Processing));$
$P \vdash_1 \langle INIT\ C'\ (D\#C\#Cs,False) \leftarrow e,(h,l,sh')\rangle \Rightarrow \langle e',s'\rangle\ ]\!]$
$\Longrightarrow P \vdash_1 \langle INIT\ C'\ (C\#Cs,False) \leftarrow e,(h,l,sh)\rangle \Rightarrow \langle e',s'\rangle$
| $InitRInit_1$:

$P \vdash_1 \langle RI\ (C, C{\bullet_s}clinit([])); Cs \leftarrow e,(h,l,sh)\rangle \Rightarrow \langle e',s'\rangle$
$\Longrightarrow P \vdash_1 \langle INIT\ C'\ (C\#Cs, True) \leftarrow e,(h,l,sh)\rangle \Rightarrow \langle e',s'\rangle$

| $RInit_1$:
$[\![\ P \vdash_1 \langle e,s\rangle \Rightarrow \langle Val\ v,\ (h',l',sh')\rangle;$
$sh'\ C = Some(sfs,\ i);\ sh'' = sh'(C \mapsto (sfs,\ Done));$
$C' = last(C\#Cs);$
$P \vdash_1 \langle INIT\ C'\ (Cs, True) \leftarrow e',\ (h',l',sh'')\rangle \Rightarrow \langle e_1,s_1\rangle\ ]\!]$
$\Longrightarrow P \vdash_1 \langle RI\ (C,e); Cs \leftarrow e',s\rangle \Rightarrow \langle e_1,s_1\rangle$
| $RInitInitFail_1$:
$[\![\ P \vdash_1 \langle e,s\rangle \Rightarrow \langle throw\ a,\ (h',l',sh')\rangle;$
$sh'\ C = Some(sfs,\ i);\ sh'' = sh'(C \mapsto (sfs,\ Error));$
$P \vdash_1 \langle RI\ (D, throw\ a); Cs \leftarrow e',\ (h',l',sh'')\rangle \Rightarrow \langle e_1,s_1\rangle\ ]\!]$
$\Longrightarrow P \vdash_1 \langle RI\ (C,e); D\#Cs \leftarrow e',s\rangle \Rightarrow \langle e_1,s_1\rangle$
| $RInitFailFinal_1$:
$[\![\ P \vdash_1 \langle e,s\rangle \Rightarrow \langle throw\ a,\ (h',l',sh')\rangle;$
$sh'\ C = Some(sfs,\ i);\ sh'' = sh'(C \mapsto (sfs,\ Error))\ ]\!]$
$\Longrightarrow P \vdash_1 \langle RI\ (C,e); Nil \leftarrow e',s\rangle \Rightarrow \langle throw\ a,\ (h',l',sh'')\rangle$

inductive-cases *eval₁-cases* [*cases set*]:
$P \vdash_1 \langle new\ C,s\rangle \Rightarrow \langle e',s'\rangle$
$P \vdash_1 \langle Cast\ C\ e,s\rangle \Rightarrow \langle e',s'\rangle$
$P \vdash_1 \langle Val\ v,s\rangle \Rightarrow \langle e',s'\rangle$
$P \vdash_1 \langle e_1\ «bop»\ e_2,s\rangle \Rightarrow \langle e',s'\rangle$
$P \vdash_1 \langle Var\ v,s\rangle \Rightarrow \langle e',s'\rangle$
$P \vdash_1 \langle V{:=}e,s\rangle \Rightarrow \langle e',s'\rangle$
$P \vdash_1 \langle e{\bullet}F\{D\},s\rangle \Rightarrow \langle e',s'\rangle$
$P \vdash_1 \langle C{\bullet_s}F\{D\},s\rangle \Rightarrow \langle e',s'\rangle$
$P \vdash_1 \langle e_1{\bullet}F\{D\}{:=}e_2,s\rangle \Rightarrow \langle e',s'\rangle$
$P \vdash_1 \langle C{\bullet_s}F\{D\}{:=}e_2,s\rangle \Rightarrow \langle e',s'\rangle$
$P \vdash_1 \langle e{\bullet}M(es),s\rangle \Rightarrow \langle e',s'\rangle$
$P \vdash_1 \langle C{\bullet_s}M(es),s\rangle \Rightarrow \langle e',s'\rangle$
$P \vdash_1 \langle \{V{:}T; e_1\},s\rangle \Rightarrow \langle e',s'\rangle$
$P \vdash_1 \langle e_1;;e_2,s\rangle \Rightarrow \langle e',s'\rangle$
$P \vdash_1 \langle if\ (e)\ e_1\ else\ e_2,s\rangle \Rightarrow \langle e',s'\rangle$
$P \vdash_1 \langle while\ (b)\ c,s\rangle \Rightarrow \langle e',s'\rangle$
$P \vdash_1 \langle throw\ e,s\rangle \Rightarrow \langle e',s'\rangle$
$P \vdash_1 \langle try\ e_1\ catch(C\ V)\ e_2,s\rangle \Rightarrow \langle e',s'\rangle$
$P \vdash_1 \langle INIT\ C\ (Cs,b) \leftarrow e,s\rangle \Rightarrow \langle e',s'\rangle$
$P \vdash_1 \langle RI\ (C,e); Cs \leftarrow e_0,s\rangle \Rightarrow \langle e',s'\rangle$

inductive-cases *evals₁-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 *eval₁-final*: $P \vdash_1 \langle e,s\rangle \Rightarrow \langle e',s'\rangle \Longrightarrow final\ e'$
and *evals₁-final*: $P \vdash_1 \langle es,s\rangle\ [\Rightarrow]\ \langle es',s'\rangle \Longrightarrow finals\ es'$

lemma *eval₁-final-same*:
assumes *eval*: $P \vdash_1 \langle e,s\rangle \Rightarrow \langle e',s'\rangle$ **shows** $final\ e \Longrightarrow e = e' \wedge s = s'$

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*:
  $P \vdash_1 \langle es_0,(h_0,ls_0,sh_0)\rangle\ [\Rightarrow]\ \langle es_1,(h_1,ls_1,sh_1)\rangle \Longrightarrow length\ ls_0 = length\ ls_1$

lemma *$evals_1$-preserves-elen*:
  $\bigwedge es'\ s\ s'.\ P \vdash_1 \langle es,s\rangle\ [\Rightarrow]\ \langle es',s'\rangle \Longrightarrow length\ es = length\ es'$

lemma *$clinit_1$-loc-pres*:
 $P \vdash_1 \langle C_0\bullet_s clinit([]),(h,l,sh)\rangle \Rightarrow \langle e',(h',l',sh')\rangle \Longrightarrow l = l'$
 by(*auto elim*!: *$eval_1$-cases*(*12*) *$evals_1$-cases*(*1*))

lemma
shows *$init_1$-ri_1-same-loc*: $P \vdash_1 \langle e,(h,l,sh)\rangle \Rightarrow \langle e',(h',l',sh')\rangle$
  $\Longrightarrow (\bigwedge C\ Cs\ b\ M\ a.\ e = INIT\ C\ (Cs,b) \leftarrow unit \vee e = C\bullet_s M([]) \vee e = RI\ (C,Throw\ a)\ ;\ Cs \leftarrow unit$
   $\vee\ e = RI\ (C,C\bullet_s clinit([]))\ ;\ Cs \leftarrow unit$
   $\Longrightarrow l = l')$
  and $P \vdash_1 \langle es,(h,l,sh)\rangle\ [\Rightarrow]\ \langle es',(h',l',sh')\rangle \Longrightarrow True$
proof(*induct rule*: *$eval_1$-$evals_1$-inducts*)
  case (*$RInitInitFail_1$ e h l sh a'*)
  then show *?case* **using** *$eval_1$-final*[*of - - - throw a'*]
    by(*fastforce dest*: *$eval_1$-final-same*[*of - Throw a*])
next
  case *$RInitFailFinal_1$* **then show** *?case* **by**(*auto dest*: *$eval_1$-final-same*)
qed(*auto dest*: *$evals_1$-cases $eval_1$-cases*(*17*) *$eval_1$-final-same*)

lemma *$init_1$-same-loc*: $P \vdash_1 \langle INIT\ C\ (Cs,b) \leftarrow unit,(h,l,sh)\rangle \Rightarrow \langle e',(h',l',sh')\rangle \Longrightarrow l = l'$
 by(*simp add*: *$init_1$-ri_1-same-loc*)

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*: $P \vdash_1 \langle es,(h,l,sh)\rangle\ [\Rightarrow]\ \langle es',(h',l',sh')\rangle \Longrightarrow h \trianglelefteq h'$

3.1.2 Initialization

lemma *$rinit_1$-throw*:
 $P_1 \vdash_1 \langle RI\ (D,Throw\ xa)\ ;\ Cs \leftarrow e,(h,\ l,\ sh)\rangle \Rightarrow \langle e',(h',\ l',\ sh')\rangle$
  $\Longrightarrow e' = Throw\ xa$
proof(*induct Cs arbitrary*: *D h l sh h' l' sh'*)
  case *Nil* **then show** *?case*
  proof(*rule $eval_1$-cases*(*20*)) **qed**(*auto elim*: *$eval_1$-cases*)
next
  case (*Cons C Cs*) **show** *?case* **using** *Cons.prems*
  proof(*induct rule*: *$eval_1$-cases*(*20*))
    case *$RInit_1$*: *1*
    then show *?case* **using** *Cons.hyps* **by**(*auto elim*: *$eval_1$-cases*)
  next
    case *$RInitInitFail_1$*: *2*
    then show *?case* **using** *Cons.hyps $eval_1$-final-same final-def* **by** *blast*
  next
    case *$RInitFailFinal_1$*: *3* **then show** *?case* **by** *simp*
  qed
qed

lemma *rinit*$_1$*-throwE*:
$P \vdash_1 \langle RI\ (C,throw\ e)\ ;\ Cs \leftarrow e_0,s\rangle \Rightarrow \langle e',s'\rangle$
$\Longrightarrow \exists\, a\ s_t.\ e' = throw\ a \wedge P \vdash_1 \langle throw\ e,s\rangle \Rightarrow \langle throw\ a,s_t\rangle$
proof(*induct Cs arbitrary*: *C e s*)
case *Nil*
then show *?case*
proof(*rule eval*$_1$*-cases*(*20*)) — RI
fix $v\ h'\ l'\ sh'$ **assume** $P \vdash_1 \langle throw\ e,s\rangle \Rightarrow \langle Val\ v,(h',\ l',\ sh')\rangle$
then show *?case* **using** *eval*$_1$*-cases*(*17*) **by** *blast*
qed(*auto*)
next
case (*Cons* C' Cs')
show *?case* **using** *Cons.prems*(*1*)
proof(*rule eval*$_1$*-cases*(*20*)) — RI
fix $v\ h'\ l'\ sh'$ **assume** $P \vdash_1 \langle throw\ e,s\rangle \Rightarrow \langle Val\ v,(h',\ l',\ sh')\rangle$
then show *?case* **using** *eval*$_1$*-cases*(*17*) **by** *blast*
next
fix $a\ h'\ l'\ sh'\ sfs\ i\ D\ Cs''$
assume $e''step$: $P \vdash_1 \langle throw\ e,s\rangle \Rightarrow \langle throw\ a,(h',\ l',\ sh')\rangle$
and *shC*: $sh'\ C = \lfloor(sfs,\ i)\rfloor$
and *riD*: $P \vdash_1 \langle RI\ (D,throw\ a)\ ;\ Cs'' \leftarrow e_0,(h',\ l',\ sh'(C \mapsto (sfs,\ Error)))\rangle \Rightarrow \langle e',s'\rangle$
and $C' \# Cs' = D \# Cs''$
then show *?thesis* **using** *Cons.hyps eval*$_1$*-final eval*$_1$*-final-same* **by** *blast*
qed(*simp*)
qed

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*-prog*,*env*$_1$, *expr*$_1$, *ty*] ⇒ *bool*
(‹(-,- $\vdash_1$/ - :: -)› [*51*,*51*,*51*]*50*)
and WTs_1 :: [J_1*-prog*,*env*$_1$, *expr*$_1$ *list*, *ty list*] ⇒ *bool*
(‹(-,- $\vdash_1$/ - [::] -)› [*51*,*51*,*51*]*50*)
for *P* :: J_1*-prog*
where

WTNew$_1$:
is-class P C $\Longrightarrow$
$P,E \vdash_1 new\ C :: Class\ C$

| *WTCast*$_1$:
$[\![\ P,E \vdash_1 e :: Class\ D;\ \ is\text{-}class\ P\ C;\ \ P \vdash C \preceq^* D \vee P \vdash D \preceq^* C\ ]\!]$

$\Longrightarrow P,E \vdash_1 \mathit{Cast}\ C\ e :: \mathit{Class}\ C$

| WTVal_1:
$\mathit{typeof}\ v = \mathit{Some}\ T \Longrightarrow$
$P,E \vdash_1 \mathit{Val}\ v :: T$

| WTVar_1:
$[\![\ E!i = T;\ i < \mathit{size}\ E\ ]\!]$
$\Longrightarrow P,E \vdash_1 \mathit{Var}\ i :: T$

| $\mathit{WTBinOp}_1$:
$[\![\ P,E \vdash_1 e_1 :: T_1;\ \ P,E \vdash_1 e_2 :: T_2;$
$\mathit{case}\ \mathit{bop}\ \mathit{of}\ \mathit{Eq} \Rightarrow (P \vdash T_1 \leq T_2 \vee P \vdash T_2 \leq T_1) \wedge T = \mathit{Boolean}$
$\mid \mathit{Add} \Rightarrow T_1 = \mathit{Integer} \wedge T_2 = \mathit{Integer} \wedge T = \mathit{Integer}\ ]\!]$
$\Longrightarrow P,E \vdash_1 e_1\ «\mathit{bop}»\ e_2 :: T$

| WTLAss_1:
$[\![\ E!i = T;\ \ i < \mathit{size}\ E;\ P,E \vdash_1 e :: T';\ \ P \vdash T' \leq T\ ]\!]$
$\Longrightarrow P,E \vdash_1 i{:=}e :: \mathit{Void}$

| WTFAcc_1:
$[\![\ P,E \vdash_1 e :: \mathit{Class}\ C;\ \ P \vdash C\ \mathit{sees}\ F{,}\mathit{NonStatic}{:}T\ \mathit{in}\ D\ ]\!]$
$\Longrightarrow P,E \vdash_1 e{\bullet}F\{D\} :: T$

| $\mathit{WTSFAcc}_1$:
$[\![\ P \vdash C\ \mathit{sees}\ F{,}\mathit{Static}{:}T\ \mathit{in}\ D\ ]\!]$
$\Longrightarrow P,E \vdash_1 C{\bullet}_s F\{D\} :: T$

| WTFAss_1:
$[\![\ P,E \vdash_1 e_1 :: \mathit{Class}\ C;\ \ P \vdash C\ \mathit{sees}\ F{,}\mathit{NonStatic}{:}T\ \mathit{in}\ D;\ \ P,E \vdash_1 e_2 :: T';\ \ P \vdash T' \leq T\ ]\!]$
$\Longrightarrow P,E \vdash_1 e_1{\bullet}F\{D\} := e_2 :: \mathit{Void}$

| $\mathit{WTSFAss}_1$:
$[\![\ \ P \vdash C\ \mathit{sees}\ F{,}\mathit{Static}{:}T\ \mathit{in}\ D;\ \ P,E \vdash_1 e_2 :: T';\ \ P \vdash T' \leq T\ ]\!]$
$\Longrightarrow P,E \vdash_1 C{\bullet}_s F\{D\}{:=}e_2 :: \mathit{Void}$

| WTCall_1:
$[\![\ P,E \vdash_1 e :: \mathit{Class}\ C;\ P \vdash C\ \mathit{sees}\ M{,}\mathit{NonStatic}{:}\mathit{Ts}' \rightarrow T = m\ \mathit{in}\ D;$
$P,E \vdash_1 \mathit{es}\ [::]\ \mathit{Ts};\ \ P \vdash \mathit{Ts}\ [\leq]\ \mathit{Ts}'\ ]\!]$
$\Longrightarrow P,E \vdash_1 e{\bullet}M(\mathit{es}) :: T$

| $\mathit{WTSCall}_1$:
$[\![\ P \vdash C\ \mathit{sees}\ M{,}\mathit{Static}{:}\mathit{Ts} \rightarrow T = m\ \mathit{in}\ D;$
$P,E \vdash_1 \mathit{es}\ [::]\ \mathit{Ts}';\ \ P \vdash \mathit{Ts}'\ [\leq]\ \mathit{Ts};\ M \neq \mathit{clinit}\ ]\!]$
$\Longrightarrow P,E \vdash_1 C{\bullet}_s M(\mathit{es}) :: T$

| $\mathit{WTBlock}_1$:
$[\![\ \mathit{is\text{-}type}\ P\ T;\ P,E@[T] \vdash_1 e{::}T'\ ]\!]$
$\Longrightarrow\ \ P,E \vdash_1 \{i{:}T;\ e\} :: T'$

| 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*]:$\neg P,E \vdash_1 INIT\ C\ (Cs,b) \leftarrow e :: T$
by(*auto elim*: WT_1*.cases*)
lemma *rinit-nWT*$_1$ [*simp*]:$\neg P,E \vdash_1 RI(C,e);Cs \leftarrow e' :: T$
by(*auto elim*: WT_1*.cases*)

lemma *WTs*$_1$*-same-size*: $\bigwedge Ts.\ P,E \vdash_1 es\ [::]\ Ts \Longrightarrow size\ es = size\ Ts$

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*: $P,E \vdash_1 es\ [::]\ Ts_1 \Longrightarrow (\bigwedge Ts_2.\ P,E \vdash_1 es\ [::]\ Ts_2 \Longrightarrow Ts_1 = Ts_2)$

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 $P,E \vdash_1 es\ [::]\ Ts \Longrightarrow True$
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(*induct rule*: WT_1*-WTs*$_1$*.inducts*) **qed**(*simp-all*)

3.2.2 Runtime Well-Typedness

inductive
WTrt$_1$ :: J_1*-prog* ⇒ *heap* ⇒ *sheap* ⇒ *env*$_1$ ⇒ *expr*$_1$ ⇒ *ty* ⇒ *bool*
and *WTrts*$_1$ :: J_1*-prog* ⇒ *heap* ⇒ *sheap* ⇒ *env*$_1$ ⇒ *expr*$_1$ *list* ⇒ *ty list* ⇒ *bool*
and *WTrt2*$_1$ :: [J_1*-prog*,*env*$_1$,*heap*,*sheap*,*expr*$_1$,*ty*] ⇒ *bool*
(‹-,-,-,- $\vdash_1$ - : -› [*51*,*51*,*51*,*51*]*50*)
and *WTrts2*$_1$ :: [J_1*-prog*,*env*$_1$,*heap*,*sheap*,*expr*$_1$ *list*, *ty list*] ⇒ *bool*
(‹-,-,-,- $\vdash_1$ - [:] -› [*51*,*51*,*51*,*51*]*50*)
for P :: J_1*-prog* **and** h :: *heap* **and** *sh* :: *sheap*
where

$P,E,h,sh \vdash_1 e : T \equiv WTrt_1\ P\ h\ sh\ E\ e\ T$
$\mid P,E,h,sh \vdash_1 es[:] Ts \equiv WTrts_1\ P\ h\ sh\ E\ es\ Ts$

$\mid WTrtNew_1$:
$is\text{-}class\ P\ C \Longrightarrow$
$P,E,h,sh \vdash_1 new\ C : Class\ C$

$\mid WTrtCast_1$:
$[\![\ P,E,h,sh \vdash_1 e : T;\ is\text{-}refT\ T;\ is\text{-}class\ P\ C\ ]\!]$
$\Longrightarrow P,E,h,sh \vdash_1 Cast\ C\ e : Class\ C$

$\mid WTrtVal_1$:
$typeof_h\ v = Some\ T \Longrightarrow$
$P,E,h,sh \vdash_1 Val\ v : T$

$\mid WTrtVar_1$:
$[\![\ E!i = T;\ i < size\ E\ ]\!] \Longrightarrow$
$P,E,h,sh \vdash_1 Var\ i : T$

$\mid WTrtBinOpEq_1$:
$[\![\ P,E,h,sh \vdash_1 e_1 : T_1;\ \ P,E,h,sh \vdash_1 e_2 : T_2\ ]\!]$
$\Longrightarrow P,E,h,sh \vdash_1 e_1 \ll Eq \gg e_2 : Boolean$

$\mid WTrtBinOpAdd_1$:
$[\![\ P,E,h,sh \vdash_1 e_1 : Integer;\ \ P,E,h,sh \vdash_1 e_2 : Integer\ ]\!]$
$\Longrightarrow P,E,h,sh \vdash_1 e_1 \ll Add \gg e_2 : Integer$

$\mid WTrtLAss_1$:
$[\![\ E!i = T;\ i < size\ E;\ P,E,h,sh \vdash_1 e : T';\ \ P \vdash T' \leq T\ ]\!]$
$\Longrightarrow P,E,h,sh \vdash_1 i{:=}e : Void$

$\mid WTrtFAcc_1$:
$[\![\ P,E,h,sh \vdash_1 e : Class\ C;\ P \vdash C\ has\ F,NonStatic{:}T\ in\ D\ ]\!] \Longrightarrow$
$P,E,h,sh \vdash_1 e{\cdot}F\{D\} : T$

$\mid WTrtFAccNT_1$:
$P,E,h,sh \vdash_1 e : NT \Longrightarrow$
$P,E,h,sh \vdash_1 e{\cdot}F\{D\} : T$

$\mid WTrtSFAcc_1$:
$[\![\ P \vdash C\ has\ F,Static{:}T\ in\ D\ ]\!] \Longrightarrow$
$P,E,h,sh \vdash_1 C{\cdot}_sF\{D\} : T$

$\mid WTrtFAss_1$:
$[\![\ P,E,h,sh \vdash_1 e_1 : Class\ C;\ \ P \vdash C\ has\ F,NonStatic{:}T\ in\ D;\ P,E,h,sh \vdash_1 e_2 : T_2;\ \ P \vdash T_2 \leq T\ ]\!]$
$\Longrightarrow P,E,h,sh \vdash_1 e_1{\cdot}F\{D\}{:=}e_2 : Void$

$\mid WTrtFAssNT_1$:
$[\![\ P,E,h,sh \vdash_1 e_1{:}NT;\ P,E,h,sh \vdash_1 e_2 : T_2\ ]\!]$
$\Longrightarrow P,E,h,sh \vdash_1 e_1{\cdot}F\{D\}{:=}e_2 : Void$

$\mid WTrtSFAss_1$:
$[\![\ P,E,h,sh \vdash_1 e_2 : T_2;\ P \vdash C\ has\ F,Static{:}T\ in\ D;\ P \vdash T_2 \leq T\ ]\!]$
$\Longrightarrow P,E,h,sh \vdash_1 C{\cdot}_sF\{D\}{:=}e_2 : Void$

| $WTrtCall_1$:
$[\![P,E,h,sh \vdash_1 e : Class\ C;\ P \vdash C\ sees\ M,NonStatic{:}Ts \rightarrow T = m\ in\ D;$
$P,E,h,sh \vdash_1 es\ [:]\ Ts';\ P \vdash Ts'\ [\leq]\ Ts\ ]\!]$
$\Longrightarrow P,E,h,sh \vdash_1 e{\bullet}M(es) : T$

| $WTrtCallNT_1$:
$[\![P,E,h,sh \vdash_1 e : NT;\ P,E,h,sh \vdash_1 es\ [:]\ Ts\ ]\!]$
$\Longrightarrow P,E,h,sh \vdash_1 e{\bullet}M(es) : T$

| $WTrtSCall_1$:
$[\![P \vdash C\ sees\ M,Static{:}Ts \rightarrow T = m\ in\ D;$
$P,E,h,sh \vdash_1 es\ [:]\ Ts';\ P \vdash Ts'\ [\leq]\ Ts;$
$M = clinit \longrightarrow sh\ D = \lfloor(sfs,Processing)\rfloor \wedge es = map\ Val\ vs\ ]\!]$
$\Longrightarrow P,E,h,sh \vdash_1 C{\bullet}_s M(es) : T$

| $WTrtBlock_1$:
$P,E@[T],h,sh \vdash_1 e : T' \Longrightarrow$
$P,E,h,sh \vdash_1 \{i{:}T;\ e\} : T'$

| $WTrtSeq_1$:
$[\![P,E,h,sh \vdash_1 e_1{:}T_1;\ \ P,E,h,sh \vdash_1 e_2{:}T_2\ ]\!]$
$\Longrightarrow P,E,h,sh \vdash_1 e_1;;e_2 : T_2$

| $WTrtCond_1$:
$[\![P,E,h,sh \vdash_1 e : Boolean;\ \ P,E,h,sh \vdash_1 e_1{:}T_1;\ \ P,E,h,sh \vdash_1 e_2{:}T_2;$
$P \vdash T_1 \leq T_2 \vee P \vdash T_2 \leq T_1;\ P \vdash T_1 \leq T_2 \longrightarrow T = T_2;\ P \vdash T_2 \leq T_1 \longrightarrow T = T_1\ ]\!]$
$\Longrightarrow P,E,h,sh \vdash_1 if\ (e)\ e_1\ else\ e_2 : T$

| $WTrtWhile_1$:
$[\![P,E,h,sh \vdash_1 e : Boolean;\ \ P,E,h,sh \vdash_1 c{:}T\ ]\!]$
$\Longrightarrow\ P,E,h,sh \vdash_1 while(e)\ c : Void$

| $WTrtThrow_1$:
$[\![P,E,h,sh \vdash_1 e : T_r;\ is\text{-}refT\ T_r\ ]\!] \Longrightarrow$
$P,E,h,sh \vdash_1 throw\ e : T$

| $WTrtTry_1$:
$[\![P,E,h,sh \vdash_1 e_1 : T_1;\ \ P,E@[Class\ C],h,sh \vdash_1 e_2 : T_2;\ P \vdash T_1 \leq T_2\ ]\!]$
$\Longrightarrow P,E,h,sh \vdash_1 try\ e_1\ catch(C\ i)\ e_2 : T_2$

| $WTrtInit_1$:
$[\![P,E,h,sh \vdash_1 e : T;\ \forall C' \in set\ (C\#Cs).\ is\text{-}class\ P\ C';\ \neg sub\text{-}RI\ e;$
$\forall C' \in set\ (tl\ Cs).\ \exists sfs.\ sh\ C' = \lfloor(sfs,Processing)\rfloor;$
$b \longrightarrow (\forall C' \in set\ Cs.\ \exists sfs.\ sh\ C' = \lfloor(sfs,Processing)\rfloor);$
$distinct\ Cs;\ supercls\text{-}lst\ P\ Cs\ ]\!]$
$\Longrightarrow P,E,h,sh \vdash_1 INIT\ C\ (Cs,\ b) \leftarrow e : T$

| $WTrtRI_1$:
$[\![P,E,h,sh \vdash_1 e : T;\ P,E,h,sh \vdash_1 e' : T';\ \forall C' \in set\ (C\#Cs).\ is\text{-}class\ P\ C';\ \neg sub\text{-}RI\ e';$
$\forall C' \in set\ (C\#Cs).\ not\text{-}init\ C'\ e;$
$\forall C' \in set\ Cs.\ \exists sfs.\ sh\ C' = \lfloor(sfs,Processing)\rfloor;$
$\exists sfs.\ sh\ C = \lfloor(sfs,\ Processing)\rfloor \vee (sh\ C = \lfloor(sfs,\ Error)\rfloor \wedge e = THROW\ NoClassDefFoundError);$
$distinct\ (C\#Cs);\ supercls\text{-}lst\ P\ (C\#Cs)\ ]\!]$

$\Longrightarrow P,E,h,sh \vdash_1 RI(C,\ e);Cs \leftarrow e' : T'$

— 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*$_1$*-implies-WTrt*$_1$: $P,E \vdash_1 e :: T \Longrightarrow P,E,h,sh \vdash_1 e : T$
and *WTs*$_1$*-implies-WTrts*$_1$: $P,E \vdash_1 es \ [::] \ Ts \Longrightarrow P,E,h,sh \vdash_1 es \ [:] \ Ts$

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**
$\mathcal{B}\ (new\ C)\ i = True\ |$
$\mathcal{B}\ (Cast\ C\ e)\ i = \mathcal{B}\ e\ i\ |$
$\mathcal{B}\ (Val\ v)\ i = True\ |$
$\mathcal{B}\ (e_1\ «bop»\ e_2)\ i = (\mathcal{B}\ e_1\ i \wedge \mathcal{B}\ e_2\ i)\ |$
$\mathcal{B}\ (Var\ j)\ i = True\ |$
$\mathcal{B}\ (e\cdot F\{D\})\ i = \mathcal{B}\ e\ i\ |$
$\mathcal{B}\ (C\cdot_s F\{D\})\ i = True\ |$
$\mathcal{B}\ (j{:=}e)\ i = \mathcal{B}\ e\ i\ |$
$\mathcal{B}\ (e_1\cdot F\{D\} := e_2)\ i = (\mathcal{B}\ e_1\ i \wedge \mathcal{B}\ e_2\ i)\ |$
$\mathcal{B}\ (C\cdot_s F\{D\} := e_2)\ i = \mathcal{B}\ e_2\ i\ |$
$\mathcal{B}\ (e\cdot M(es))\ i = (\mathcal{B}\ e\ i \wedge \mathcal{B}s\ es\ i)\ |$
$\mathcal{B}\ (C\cdot_s M(es))\ i = \mathcal{B}s\ es\ i\ |$
$\mathcal{B}\ (\{j{:}T\ ;\ e\})\ i = (i = j \wedge \mathcal{B}\ e\ (i{+}1))\ |$
$\mathcal{B}\ (e_1;;e_2)\ i = (\mathcal{B}\ e_1\ i \wedge \mathcal{B}\ e_2\ i)\ |$
$\mathcal{B}\ (if\ (e)\ e_1\ else\ e_2)\ i = (\mathcal{B}\ e\ i \wedge \mathcal{B}\ e_1\ i \wedge \mathcal{B}\ e_2\ i)\ |$
$\mathcal{B}\ (throw\ e)\ i = \mathcal{B}\ e\ i\ |$
$\mathcal{B}\ (while\ (e)\ c)\ i = (\mathcal{B}\ e\ i \wedge \mathcal{B}\ c\ i)\ |$
$\mathcal{B}\ (try\ e_1\ catch(C\ j)\ e_2)\ i = (\mathcal{B}\ e_1\ i \wedge i{=}j \wedge \mathcal{B}\ e_2\ (i{+}1))\ |$
$\mathcal{B}\ (INIT\ C\ (Cs,b) \leftarrow e)\ i = \mathcal{B}\ e\ i\ |$
$\mathcal{B}\ (RI(C,e);Cs \leftarrow e')\ i = (\mathcal{B}\ e\ i \wedge \mathcal{B}\ e'\ i)\ |$

$\mathcal{B}s\ []\ i = True\ |$
$\mathcal{B}s\ (e\#es)\ i = (\mathcal{B}\ e\ i \wedge \mathcal{B}s\ es\ i)$

definition *wf-J*$_1$*-mdecl* :: $J_1\text{-}prog \Rightarrow cname \Rightarrow expr_1\ mdecl \Rightarrow bool$
where
wf-J$_1$*-mdecl* $P\ C\ \equiv\ \lambda(M,b,Ts,T,body).$
$\neg sub\text{-}RI\ body \wedge$
$(case\ b\ of$
$NonStatic \Rightarrow$
$(\exists\, T'.\ P,Class\ C\#Ts \vdash_1 body :: T' \wedge P \vdash T' \leq T) \wedge$
$\mathcal{D}\ body\ \lfloor\{..size\ Ts\}\rfloor \wedge \mathcal{B}\ body\ (size\ Ts + 1)$
$|\ Static \Rightarrow (\exists\, T'.\ P,Ts \vdash_1 body :: T' \wedge P \vdash T' \leq T) \wedge$
$\mathcal{D}\ body\ \lfloor\{..{<}size\ Ts\}\rfloor \wedge \mathcal{B}\ body\ (size\ Ts))$

lemma *wf-J_1-mdecl-NonStatic*[*simp*]:
wf-J_1-mdecl P C $(M, NonStatic, Ts, T, body) \equiv$
$(\neg$*sub-RI body* $\wedge$
$(\exists T'. P, Class\ C\#Ts \vdash_1 body :: T' \wedge P \vdash T' \leq T) \wedge$
$\mathcal{D}\ body\ \lfloor\{..size\ Ts\}\rfloor \wedge \mathcal{B}\ body\ (size\ Ts + 1))$
lemma *wf-J_1-mdecl-Static*[*simp*]:
wf-J_1-mdecl P C $(M, Static, Ts, T, body) \equiv$
$(\neg$*sub-RI body* $\wedge$
$(\exists T'. P, Ts \vdash_1 body :: T' \wedge P \vdash T' \leq T) \wedge$
$\mathcal{D}\ body\ \lfloor\{..{<}size\ Ts\}\rfloor \wedge \mathcal{B}\ body\ (size\ Ts))$
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*
using *sees-wf-mdecl*[*OF wf cM*] **by**(*simp add*: *wf-J_1-mdecl-def wf-mdecl-def*)

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 $P,E,h,sh \vdash_1 C{\bullet_s}clinit([]) : Void$
proof −
from *ex* **obtain** *D fs ms* **where** $a = (D,fs,ms)$ **by**(*cases a*)
then have *sP*: $(C, D, fs, ms) \in set\ P$ **using** *ex map-of-SomeD*[*of P C a*] **by**(*simp add*: *class-def*)
then have *wf-clinit ms* **using** *assms* **by**(*unfold wf-prog-def wf-cdecl-def*, *auto*)
then obtain *m* **where** *sm*: $(clinit, Static, [], Void, m) \in set\ ms$
by(*unfold wf-clinit-def*) *auto*
then have $P \vdash C$ *sees clinit*,*Static*:$[] \rightarrow Void = m$ *in C*
using *mdecl-visible*[*OF wf sP sm*] **by** *simp*
then show *?thesis* **using** *WTrtSCall$_1$ proc* **by** *blast*
qed

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* $\Longrightarrow \exists sfs.\ sh\ C = \lfloor(sfs, Processing)\rfloor \Longrightarrow \exists sfs'.\ sh'\ C = \lfloor(sfs', Processing)\rfloor$
lemma *clinit$_1$-proc-pres*:
$[\![$ *wf-J_1-prog P*; $P \vdash_1 \langle C_0{\bullet_s}clinit([]),(h,l,sh)\rangle \Rightarrow \langle e',(h',l',sh')\rangle$;
$sh\ C' = \lfloor(sfs,Processing)\rfloor\ ]\!]$
$\Longrightarrow \exists sfs.\ sh'\ C' = \lfloor(sfs,Processing)\rfloor$
by(*auto dest*: *eval$_1$-proc-pres*)

end

3.3 Program Compilation

theory *PCompiler*
imports *../Common/WellForm*
begin

definition *compM* :: $(staticb \Rightarrow {'a} \Rightarrow {'b}) \Rightarrow {'a}\ mdecl \Rightarrow {'b}\ mdecl$
where

$compM\ f \equiv \lambda(M,\ b,\ Ts,\ T,\ m).\ (M,\ b,\ Ts,\ T,\ f\ b\ m)$

definition $compC :: (staticb \Rightarrow 'a \Rightarrow 'b) \Rightarrow 'a\ cdecl \Rightarrow 'b\ cdecl$
where
$compC\ f \equiv \lambda(C,D,Fdecls,Mdecls).\ (C,D,Fdecls,\ map\ (compM\ f)\ Mdecls)$

definition $compP :: (staticb \Rightarrow 'a \Rightarrow 'b) \Rightarrow 'a\ prog \Rightarrow 'b\ prog$
where
$compP\ f \equiv map\ (compC\ f)$

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*:
$map\text{-}of\ (map\ (\lambda(x,a,b,c).(x,a,b,f\ c))\ ts) =$
$map\text{-}option\ (\lambda(a,b,c).(a,b,f\ c)) \circ (map\text{-}of\ ts)$
lemma *map-of-map245*:
$map\text{-}of\ (map\ (\lambda(x,a,b,c,d).(x,a,b,c,f\ a\ c\ d))\ ts) =$
$map\text{-}option\ (\lambda(a,b,c,d).(a,b,c,f\ a\ c\ d)) \circ (map\text{-}of\ ts)$

lemma *class-compP*:
$class\ P\ C = Some\ (D,\ fs,\ ms)$
$\Longrightarrow class\ (compP\ f\ P)\ C = Some\ (D,\ fs,\ map\ (compM\ f)\ ms)$

lemma *class-compPD*:
$class\ (compP\ f\ P)\ C = Some\ (D,\ fs,\ cms)$
$\Longrightarrow \exists\, ms.\ class\ P\ C = Some(D,fs,ms) \wedge cms = map\ (compM\ f)\ ms$

lemma [*simp*]: $is\text{-}class\ (compP\ f\ P)\ C = is\text{-}class\ P\ C$

lemma [*simp*]: $class\ (compP\ f\ P)\ C = map\text{-}option\ (\lambda c.\ snd(compC\ f\ (C,c)))\ (class\ P\ C)$

lemma *sees-methods-compP*:
$P \vdash C\ sees\text{-}methods\ Mm \Longrightarrow$
$compP\ f\ P \vdash C\ sees\text{-}methods\ (map\text{-}option\ (\lambda((b,Ts,T,m),D).\ ((b,Ts,T,f\ b\ m),D)) \circ Mm)$

lemma *sees-method-compP*:
$P \vdash C\ sees\ M,b\colon Ts{\rightarrow}T = m\ in\ D \Longrightarrow$
$compP\ f\ P \vdash C\ sees\ M,b\colon Ts{\rightarrow}T = (f\ b\ m)\ in\ D$

lemma [*simp*]:
$P \vdash C\ sees\ M,b\colon Ts{\rightarrow}T = m\ in\ D \Longrightarrow$
$method\ (compP\ f\ P)\ C\ M = (D,b,Ts,T,f\ b\ m)$

lemma *sees-methods-compPD*:
$[\![\ cP \vdash C\ sees\text{-}methods\ Mm';\ cP = compP\ f\ P\ ]\!] \Longrightarrow$
$\exists\, Mm.\ P \vdash C\ sees\text{-}methods\ Mm\ \wedge$
$Mm' = (map\text{-}option\ (\lambda((b,Ts,T,m),D).\ ((b,Ts,T,f\ b\ m),D)) \circ Mm)$

lemma *sees-method-compPD*:
$compP\ f\ P \vdash C\ sees\ M,b\colon Ts{\rightarrow}T = fm\ in\ D \Longrightarrow$
$\exists\, m.\ P \vdash C\ sees\ M,b\colon Ts{\rightarrow}T = m\ in\ D \wedge f\ b\ m = fm$

lemma [*simp*]: $subcls1(compP\ f\ P) = subcls1\ P$

lemma *compP-widen*[*simp*]: $(compP\ f\ P \vdash T \leq T') = (P \vdash T \leq T')$

lemma [*simp*]: $(compP\ f\ P \vdash Ts\ [\leq]\ Ts') = (P \vdash Ts\ [\leq]\ Ts')$

lemma [*simp*]: $is\text{-}type\ (compP\ f\ P)\ T = is\text{-}type\ P\ T$

lemma [*simp*]: $(compP\ (f{::}staticb{\Rightarrow}'a{\Rightarrow}'b)\ P \vdash C\ has\text{-}fields\ FDTs) = (P \vdash C\ has\text{-}fields\ FDTs)$

lemma *fields-compP* [*simp*]: $fields\ (compP\ f\ P)\ C = fields\ P\ C$
lemma *ifields-compP* [*simp*]: $ifields\ (compP\ f\ P)\ C = ifields\ P\ C$
lemma *blank-compP* [*simp*]: $blank\ (compP\ f\ P)\ C = blank\ P\ C$
lemma *isfields-compP* [*simp*]: $isfields\ (compP\ f\ P)\ C = isfields\ P\ C$
lemma *sblank-compP* [*simp*]: $sblank\ (compP\ f\ P)\ C = sblank\ P\ C$
lemma *sees-fields-compP* [*simp*]: $(compP\ f\ P \vdash C\ sees\ F,b{:}T\ in\ D) = (P \vdash C\ sees\ F,b{:}T\ in\ D)$
lemma *has-field-compP* [*simp*]: $(compP\ f\ P \vdash C\ has\ F,b{:}T\ in\ D) = (P \vdash C\ has\ F,b{:}T\ in\ D)$
lemma *field-compP* [*simp*]: $field\ (compP\ f\ P)\ F\ D = field\ P\ F\ D$

3.3.1 Invariance of *wf-prog* under compilation

lemma [*iff*]: $distinct\text{-}fst\ (compP\ f\ P) = distinct\text{-}fst\ P$

lemma [*iff*]: $distinct\text{-}fst\ (map\ (compM\ f)\ ms) = distinct\text{-}fst\ ms$

lemma [*iff*]: $wf\text{-}syscls\ (compP\ f\ P) = wf\text{-}syscls\ P$

lemma [*iff*]: $wf\text{-}fdecl\ (compP\ f\ P) = wf\text{-}fdecl\ P$

lemma *wf-clinit-compM* [*iff*]: $wf\text{-}clinit\ (map\ (compM\ f)\ ms) = wf\text{-}clinit\ ms$
lemma *set-compP*:
$((C,D,fs,ms') \in set(compP\ f\ P)) =$
$(\exists ms.\ (C,D,fs,ms) \in set\ P \wedge ms' = map\ (compM\ f)\ ms)$
lemma *wf-cdecl-compPI*:
$[\![\bigwedge C\ M\ b\ Ts\ T\ m.$
$[\![wf\text{-}mdecl\ wf_1\ P\ C\ (M,b,Ts,T,m);\ P \vdash C\ sees\ M,b{:}Ts{\rightarrow}T = m\ in\ C]\!]$
$\Longrightarrow wf\text{-}mdecl\ wf_2\ (compP\ f\ P)\ C\ (M,b,Ts,T,\ f\ b\ m);$
$\forall x{\in}set\ P.\ wf\text{-}cdecl\ wf_1\ P\ x;\ x \in set\ (compP\ f\ P);\ wf\text{-}prog\ p\ P]\!]$
$\Longrightarrow wf\text{-}cdecl\ wf_2\ (compP\ f\ P)\ x$

lemma *wf-prog-compPI*:
assumes *lift*:
$\bigwedge C\ M\ b\ Ts\ T\ m.$
$[\![P \vdash C\ sees\ M,b{:}Ts{\rightarrow}T = m\ in\ C;\ wf\text{-}mdecl\ wf_1\ P\ C\ (M,b,Ts,T,m)]\!]$
$\Longrightarrow wf\text{-}mdecl\ wf_2\ (compP\ f\ P)\ C\ (M,b,Ts,T,\ f\ b\ m)$
and *wf*: $wf\text{-}prog\ wf_1\ P$
shows $wf\text{-}prog\ wf_2\ (compP\ f\ P)$

end
theory *Hidden*
imports *List−Index.List-Index*
begin

definition $hidden :: 'a\ list \Rightarrow nat \Rightarrow bool$ **where**
$hidden\ xs\ i \equiv i < size\ xs \wedge xs!i \in set(drop\ (i+1)\ xs)$

lemma *hidden-last-index*: $x \in set\ xs \Longrightarrow hidden\ (xs\ @\ [x])\ (last\text{-}index\ xs\ x)$
by(*auto simp add*: *hidden-def nth-append rev-nth*[*symmetric*]
  dest: *last-index-less*[*OF - le-refl*])

lemma *hidden-inacc*: $hidden\ xs\ i \Longrightarrow last\text{-}index\ xs\ x \neq i$
by(*auto simp add*: *hidden-def last-index-drop last-index-less-size-conv*)

lemma [*simp*]: $hidden\ xs\ i \Longrightarrow hidden\ (xs@[x])\ i$
by(*auto simp add*:*hidden-def nth-append*)

lemma *fun-upds-apply*:
 $(m(xs[\mapsto]ys))\ x =$
 $(let\ xs' = take\ (size\ ys)\ xs$
  $in\ if\ x \in set\ xs'\ then\ Some(ys\ !\ last\text{-}index\ xs'\ x)\ else\ m\ x)$
proof(*induct xs arbitrary*: *m ys*)
 case *Nil* **then show** *?case* **by**(*simp add*: *Let-def*)
next
 case *Cons* **show** *?case*
 proof(*cases ys*)
  case *Nil*
  then show *?thesis* **by**(*simp add*:*Let-def*)
 next
  case *Cons'*: *Cons*
  then show *?thesis* **using** *Cons* **by**(*simp add*: *Let-def last-index-Cons*)
 qed
qed

lemma *map-upds-apply-eq-Some*:
 $((m(xs[\mapsto]ys))\ x = Some\ y) =$
 $(let\ xs' = take\ (size\ ys)\ xs$
  $in\ if\ x \in set\ xs'\ then\ ys\ !\ last\text{-}index\ xs'\ x = y\ else\ m\ x = Some\ y)$
by(*simp add*:*fun-upds-apply Let-def*)

lemma *map-upds-upd-conv-last-index*:
 $[\![x \in set\ xs;\ size\ xs \leq size\ ys\ ]\!]$
 $\Longrightarrow m(xs[\mapsto]ys,\ x\mapsto y) = m(xs[\mapsto]ys[last\text{-}index\ xs\ x := y])$
by(*rule ext*) (*simp add*:*fun-upds-apply eq-sym-conv Let-def*)

end

3.4 Compilation Stage 1

theory *Compiler1* **imports** *PCompiler J1 Hidden* **begin**

Replacing variable names by indices.

primrec $compE_1$ :: $vname\ list \Rightarrow expr \Rightarrow expr_1$
 and $compEs_1$ :: $vname\ list \Rightarrow expr\ list \Rightarrow expr_1\ list$ **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 $(e_1 \ll bop \gg e_2) = (compE_1\ Vs\ e_1) \ll bop \gg (compE_1\ Vs\ e_2)$
| $compE_1$ Vs $(Var\ V) = Var(last\text{-}index\ Vs\ V)$
| $compE_1$ Vs $(V{:=}e) = (last\text{-}index\ Vs\ V){:=}\ (compE_1\ Vs\ e)$
| $compE_1$ Vs $(e{\bullet}F\{D\}) = (compE_1\ Vs\ e){\bullet}F\{D\}$
| $compE_1$ Vs $(C{\bullet}_sF\{D\}) = C{\bullet}_sF\{D\}$
| $compE_1$ Vs $(e_1{\bullet}F\{D\}{:=}e_2) = (compE_1\ Vs\ e_1){\bullet}F\{D\} := (compE_1\ Vs\ e_2)$
| $compE_1$ Vs $(C{\bullet}_sF\{D\}{:=}e_2) = C{\bullet}_sF\{D\} := (compE_1\ Vs\ e_2)$
| $compE_1$ Vs $(e{\bullet}M(es)) = (compE_1\ Vs\ e){\bullet}M(compEs_1\ Vs\ es)$
| $compE_1$ Vs $(C{\bullet}_sM(es)) = C{\bullet}_sM(compEs_1\ Vs\ es)$
| $compE_1$ 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*]: $compEs_1\ Vs\ es = map\ (compE_1\ Vs)\ es$
lemma [*simp*]: $\bigwedge Vs.\ sub\text{-}RI\ (compE_1\ Vs\ e) = sub\text{-}RI\ e$
and [*simp*]: $\bigwedge Vs.\ sub\text{-}RIs\ (compEs_1\ Vs\ es) = sub\text{-}RIs\ es$
proof(*induct rule*: *sub-RI-sub-RIs-induct*) **qed**(*auto*)

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*: $final\ e \Longrightarrow compE_1\ Vs\ e = fin_1\ e$

lemma [*simp*]:
$\bigwedge Vs.\ max\text{-}vars\ (compE_1\ Vs\ e) = max\text{-}vars\ e$
and $\bigwedge Vs.\ max\text{-}varss\ (compEs_1\ Vs\ es) = max\text{-}varss\ es$

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**
$unmod\ (new\ C)\ i = True\ |$
$unmod\ (Cast\ C\ e)\ i = unmod\ e\ i\ |$
$unmod\ (Val\ v)\ i = True\ |$
$unmod\ (e_1\ «bop»\ e_2)\ i = (unmod\ e_1\ i \wedge unmod\ e_2\ i)\ |$
$unmod\ (Var\ i)\ j = True\ |$
$unmod\ (i{:=}e)\ j = (i \neq j \wedge unmod\ e\ j)\ |$
$unmod\ (e{\cdot}F\{D\})\ i = unmod\ e\ i\ |$
$unmod\ (C{\cdot}_sF\{D\})\ i = True\ |$
$unmod\ (e_1{\cdot}F\{D\}{:=}e_2)\ i = (unmod\ e_1\ i \wedge unmod\ e_2\ i)\ |$
$unmod\ (C{\cdot}_sF\{D\}{:=}e_2)\ i = unmod\ e_2\ i\ |$
$unmod\ (e{\cdot}M(es))\ i = (unmod\ e\ i \wedge unmods\ es\ i)\ |$
$unmod\ (C{\cdot}_sM(es))\ i = unmods\ es\ i\ |$
$unmod\ \{j{:}T;\ e\}\ i = unmod\ e\ i\ |$
$unmod\ (e_1;;e_2)\ i = (unmod\ e_1\ i \wedge\ \ unmod\ e_2\ i)\ |$
$unmod\ (if\ (e)\ e_1\ else\ e_2)\ i = (unmod\ e\ i \wedge unmod\ e_1\ i \wedge unmod\ e_2\ i)\ |$
$unmod\ (while\ (e)\ c)\ i = (unmod\ e\ i \wedge unmod\ c\ i)\ |$
$unmod\ (throw\ e)\ i = unmod\ e\ i\ |$
$unmod\ (try\ e_1\ catch(C\ i)\ e_2)\ j = (unmod\ e_1\ j \wedge (if\ i{=}j\ then\ False\ else\ unmod\ e_2\ j))\ |$
$unmod\ (INIT\ C\ (Cs,b) \leftarrow e)\ i = unmod\ e\ i\ |$
$unmod\ (RI(C,e);Cs \leftarrow e')\ i = (unmod\ e\ i \wedge unmod\ e'\ i)\ |$

$unmods\ ([])\ i = True\ |$
$unmods\ (e\#es)\ i = (unmod\ e\ i \wedge unmods\ es\ i)$

lemma *hidden-unmod*: $\bigwedge Vs.\ hidden\ Vs\ i \Longrightarrow unmod\ (compE_1\ Vs\ e)\ i$ **and**
$\bigwedge Vs.\ hidden\ Vs\ i \Longrightarrow unmods\ (compEs_1\ Vs\ es)\ i$

lemma *$eval_1$-preserves-unmod*:
$[\![\ P \vdash_1 \langle e,(h,ls,sh)\rangle \Rightarrow \langle e',(h',ls',sh')\rangle;\ unmod\ e\ i;\ i < size\ ls\ ]\!]$
$\Longrightarrow ls\ !\ i = ls'\ !\ i$
and $[\![\ P \vdash_1 \langle es,(h,ls,sh)\rangle\ [\Rightarrow]\ \langle es',(h',ls',sh')\rangle;\ unmods\ es\ i;\ i < size\ ls\ ]\!]$
$\Longrightarrow ls\ !\ i = ls'\ !\ i$

lemma *LAss-lem*:
$[\![x \in set\ xs;\ size\ xs \leq size\ ys\ ]\!]$
$\Longrightarrow m_1 \subseteq_m m_2(xs[\mapsto]ys) \Longrightarrow m_1(x\mapsto y) \subseteq_m m_2(xs[\mapsto]ys[last\text{-}index\ xs\ x := y])$**lemma** *Block-lem*:
fixes $l :: 'a \rightharpoonup 'b$
assumes *0*: $l \subseteq_m [Vs\ [\mapsto]\ ls]$
and *1*: $l' \subseteq_m [Vs\ [\mapsto]\ ls',\ V\mapsto v]$
and *hidden*: $V \in set\ Vs \Longrightarrow ls\ !\ last\text{-}index\ Vs\ V = ls'\ !\ last\text{-}index\ Vs\ V$
and *size*: $size\ ls = size\ ls' \quad size\ Vs < size\ ls'$
shows $l'(V := l\ V) \subseteq_m [Vs\ [\mapsto]\ ls']$

The main theorem:

theorem assumes *wf*: *wwf-J-prog P*
shows *$eval_1$-eval*: $P \vdash \langle e,(h,l,sh)\rangle \Rightarrow \langle e',(h',l',sh')\rangle$
$\Longrightarrow (\bigwedge Vs\ ls.\ [\![\ fv\ e \subseteq set\ Vs;\ \ l \subseteq_m [Vs[\mapsto]ls];\ size\ Vs + max\text{-}vars\ e \leq size\ ls\ ]\!]$
$\Longrightarrow \exists ls'.\ compP_1\ P \vdash_1 \langle compE_1\ Vs\ e,(h,ls,sh)\rangle \Rightarrow \langle fin_1\ e',(h',ls',sh')\rangle \wedge l' \subseteq_m [Vs[\mapsto]ls'])$
and *$evals_1$-evals*: $P \vdash \langle es,(h,l,sh)\rangle\ [\Rightarrow]\ \langle es',(h',l',sh')\rangle$
$\Longrightarrow (\bigwedge Vs\ ls.\ [\![\ fvs\ es \subseteq set\ Vs;\ \ l \subseteq_m [Vs[\mapsto]ls];\ size\ Vs + max\text{-}varss\ es \leq size\ ls\ ]\!]$
$\Longrightarrow \exists ls'.\ compP_1\ P \vdash_1 \langle compEs_1\ Vs\ es,(h,ls,sh)\rangle\ [\Rightarrow]\ \langle compEs_1\ Vs\ es',(h',ls',sh')\rangle \wedge$

$l' \subseteq_m [Vs[\mapsto]ls'])$

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 *compE₁-pres-wt*: $\bigwedge Vs\ Ts\ U.$
$[\![\ P,[Vs[\mapsto]Ts] \vdash e :: U;\ size\ Ts = size\ Vs\ ]\!]$
$\Longrightarrow compP\ f\ P,Ts \vdash_1 compE_1\ Vs\ e :: U$
and $\bigwedge Vs\ Ts\ Us.$
$[\![\ P,[Vs[\mapsto]Ts] \vdash es\ [::]\ Us;\ size\ Ts = size\ Vs\ ]\!]$
$\Longrightarrow compP\ f\ P,Ts \vdash_1 compEs_1\ Vs\ es\ [::]\ Us$

and the correct block numbering:

lemma $\mathcal{B}$: $\bigwedge Vs\ n.\ size\ Vs = n \Longrightarrow \mathcal{B}\ (compE_1\ Vs\ e)\ n$
and $\mathcal{B}s$: $\bigwedge Vs\ n.\ size\ Vs = n \Longrightarrow \mathcal{B}s\ (compEs_1\ Vs\ es)\ n$

The main complication is preservation of definite assignment $\mathcal{D}$.

lemma *image-last-index*: $A \subseteq set(xs@[x]) \Longrightarrow last\text{-}index\ (xs\ @\ [x])\ `\ A =$
$(if\ x \in A\ then\ insert\ (size\ xs)\ (last\text{-}index\ xs\ `\ (A-\{x\}))\ else\ last\text{-}index\ xs\ `\ A)$

lemma *A-compE₁-None*[*simp*]:
$\bigwedge Vs.\ \mathcal{A}\ e = None \Longrightarrow \mathcal{A}\ (compE_1\ Vs\ e) = None$
and $\bigwedge Vs.\ \mathcal{A}s\ es = None \Longrightarrow \mathcal{A}s\ (compEs_1\ Vs\ es) = None$

lemma *A-compE₁*:
$\bigwedge A\ Vs.\ [\![\ \mathcal{A}\ e = \lfloor A\rfloor;\ fv\ e \subseteq set\ Vs\ ]\!] \Longrightarrow \mathcal{A}\ (compE_1\ Vs\ e) = \lfloor last\text{-}index\ Vs\ `\ A\rfloor$
and $\bigwedge A\ Vs.\ [\![\ \mathcal{A}s\ es = \lfloor A\rfloor;\ fvs\ es \subseteq set\ Vs\ ]\!] \Longrightarrow \mathcal{A}s\ (compEs_1\ Vs\ es) = \lfloor last\text{-}index\ Vs\ `\ A\rfloor$

lemma *D-None*[*iff*]: $\mathcal{D}\ (e::'a\ exp)\ None$ **and** [*iff*]: $\mathcal{D}s\ (es::'a\ exp\ list)\ None$

lemma *D-last-index-compE₁*:
$\bigwedge A\ Vs.\ [\![\ A \subseteq set\ Vs;\ fv\ e \subseteq set\ Vs\ ]\!] \Longrightarrow$
$\mathcal{D}\ e\ \lfloor A\rfloor \Longrightarrow \mathcal{D}\ (compE_1\ Vs\ e)\ \lfloor last\text{-}index\ Vs\ `\ A\rfloor$
and $\bigwedge A\ Vs.\ [\![\ A \subseteq set\ Vs;\ fvs\ es \subseteq set\ Vs\ ]\!] \Longrightarrow$
$\mathcal{D}s\ es\ \lfloor A\rfloor \Longrightarrow \mathcal{D}s\ (compEs_1\ Vs\ es)\ \lfloor last\text{-}index\ Vs\ `\ A\rfloor$

lemma *last-index-image-set*: $distinct\ xs \Longrightarrow last\text{-}index\ xs\ `\ set\ xs = \{..<size\ xs\}$

lemma *D-compE₁*:
$[\![\ \mathcal{D}\ e\ \lfloor set\ Vs\rfloor;\ fv\ e \subseteq set\ Vs;\ distinct\ Vs\ ]\!] \Longrightarrow \mathcal{D}\ (compE_1\ Vs\ e)\ \lfloor\{..<length\ Vs\}\rfloor$

lemma *D-compE₁'*:
assumes $\mathcal{D}\ e\ \lfloor set(V\#Vs)\rfloor$ **and** $fv\ e \subseteq set(V\#Vs)$ **and** $distinct(V\#Vs)$
shows $\mathcal{D}\ (compE_1\ (V\#Vs)\ e)\ \lfloor\{..length\ Vs\}\rfloor$
lemma *compP₁-pres-wf*: $wf\text{-}J\text{-}prog\ P \Longrightarrow wf\text{-}J_1\text{-}prog\ (compP_1\ P)$

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 ⇒ [CmpEq] | Add ⇒ [IAdd]) = Suc 0
 by(cases bop, simp+)

primrec compE₂ :: expr₁ ⇒ instr list
  and compEs₂ :: expr₁ list ⇒ instr list where
  compE₂ (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 = []
```

$\mid \mathit{compxE}_2\ (\mathit{Cast}\ C\ e)\ \mathit{pc}\ d = \mathit{compxE}_2\ e\ \mathit{pc}\ d$
$\mid \mathit{compxE}_2\ (\mathit{Val}\ v)\ \mathit{pc}\ d = []$
$\mid \mathit{compxE}_2\ (e_1\ «\mathit{bop}»\ e_2)\ \mathit{pc}\ d =$
$\quad \mathit{compxE}_2\ e_1\ \mathit{pc}\ d\ @\ \mathit{compxE}_2\ e_2\ (\mathit{pc} + \mathit{size}(\mathit{compE}_2\ e_1))\ (d{+}1)$
$\mid \mathit{compxE}_2\ (\mathit{Var}\ i)\ \mathit{pc}\ d = []$
$\mid \mathit{compxE}_2\ (i{:=}e)\ \mathit{pc}\ d = \mathit{compxE}_2\ e\ \mathit{pc}\ d$
$\mid \mathit{compxE}_2\ (e{\cdot}F\{D\})\ \mathit{pc}\ d = \mathit{compxE}_2\ e\ \mathit{pc}\ d$
$\mid \mathit{compxE}_2\ (C{\cdot}_s F\{D\})\ \mathit{pc}\ d = []$
$\mid \mathit{compxE}_2\ (e_1{\cdot}F\{D\} := e_2)\ \mathit{pc}\ d =$
$\quad \mathit{compxE}_2\ e_1\ \mathit{pc}\ d\ @\ \mathit{compxE}_2\ e_2\ (\mathit{pc} + \mathit{size}(\mathit{compE}_2\ e_1))\ (d{+}1)$
$\mid \mathit{compxE}_2\ (C{\cdot}_s F\{D\} := e_2)\ \mathit{pc}\ d = \mathit{compxE}_2\ e_2\ \mathit{pc}\ d$
$\mid \mathit{compxE}_2\ (e{\cdot}M(\mathit{es}))\ \mathit{pc}\ d =$
$\quad \mathit{compxE}_2\ e\ \mathit{pc}\ d\ @\ \mathit{compxEs}_2\ \mathit{es}\ (\mathit{pc} + \mathit{size}(\mathit{compE}_2\ e))\ (d{+}1)$
$\mid \mathit{compxE}_2\ (C{\cdot}_s M(\mathit{es}))\ \mathit{pc}\ d = \mathit{compxEs}_2\ \mathit{es}\ \mathit{pc}\ d$
$\mid \mathit{compxE}_2\ (\{i{:}T;\ e\})\ \mathit{pc}\ d = \mathit{compxE}_2\ e\ \mathit{pc}\ d$
$\mid \mathit{compxE}_2\ (e_1;;e_2)\ \mathit{pc}\ d =$
$\quad \mathit{compxE}_2\ e_1\ \mathit{pc}\ d\ @\ \mathit{compxE}_2\ e_2\ (\mathit{pc}{+}\mathit{size}(\mathit{compE}_2\ e_1){+}1)\ d$
$\mid \mathit{compxE}_2\ (\mathit{if}\ (e)\ e_1\ \mathit{else}\ e_2)\ \mathit{pc}\ d =$
$\qquad (\mathit{let}\ \mathit{pc}_1\ = \mathit{pc} + \mathit{size}(\mathit{compE}_2\ e) + 1;$
$\qquad\qquad \mathit{pc}_2\ = \mathit{pc}_1 + \mathit{size}(\mathit{compE}_2\ e_1) + 1$
$\qquad\ \ \mathit{in}\ \mathit{compxE}_2\ e\ \mathit{pc}\ d\ @\ \mathit{compxE}_2\ e_1\ \mathit{pc}_1\ d\ @\ \mathit{compxE}_2\ e_2\ \mathit{pc}_2\ d)$
$\mid \mathit{compxE}_2\ (\mathit{while}\ (b)\ e)\ \mathit{pc}\ d =$
$\quad \mathit{compxE}_2\ b\ \mathit{pc}\ d\ @\ \mathit{compxE}_2\ e\ (\mathit{pc}{+}\mathit{size}(\mathit{compE}_2\ b){+}1)\ d$
$\mid \mathit{compxE}_2\ (\mathit{throw}\ e)\ \mathit{pc}\ d = \mathit{compxE}_2\ e\ \mathit{pc}\ d$
$\mid \mathit{compxE}_2\ (\mathit{try}\ e_1\ \mathit{catch}(C\ i)\ e_2)\ \mathit{pc}\ d =$
$\quad (\mathit{let}\ \mathit{pc}_1 = \mathit{pc} + \mathit{size}(\mathit{compE}_2\ e_1)$
$\quad\ \ \mathit{in}\ \mathit{compxE}_2\ e_1\ \mathit{pc}\ d\ @\ \mathit{compxE}_2\ e_2\ (\mathit{pc}_1{+}2)\ d\ @\ [(\mathit{pc},\mathit{pc}_1,C,\mathit{pc}_1{+}1,d)])$
$\mid \mathit{compxE}_2\ (\mathit{INIT}\ C\ (\mathit{Cs},\ b) \leftarrow e)\ \mathit{pc}\ d = []$
$\mid \mathit{compxE}_2\ (\mathit{RI}(C,\ e);\mathit{Cs} \leftarrow e')\ \mathit{pc}\ d = []$

$\mid \mathit{compxEs}_2\ []\ \mathit{pc}\ d\quad = []$
$\mid \mathit{compxEs}_2\ (e\#\mathit{es})\ \mathit{pc}\ d = \mathit{compxE}_2\ e\ \mathit{pc}\ d\ @\ \mathit{compxEs}_2\ \mathit{es}\ (\mathit{pc}{+}\mathit{size}(\mathit{compE}_2\ e))\ (d{+}1)$

primrec $\mathit{max\text{-}stack} :: \mathit{expr}_1 \Rightarrow \mathit{nat}$
  and $\mathit{max\text{-}stacks} :: \mathit{expr}_1\ \mathit{list} \Rightarrow \mathit{nat}$ **where**
$\ \ \mathit{max\text{-}stack}\ (\mathit{new}\ C) = 1$
$\mid \mathit{max\text{-}stack}\ (\mathit{Cast}\ C\ e) = \mathit{max\text{-}stack}\ e$
$\mid \mathit{max\text{-}stack}\ (\mathit{Val}\ v) = 1$
$\mid \mathit{max\text{-}stack}\ (e_1\ «\mathit{bop}»\ e_2) = \mathit{max}\ (\mathit{max\text{-}stack}\ e_1)\ (\mathit{max\text{-}stack}\ e_2) + 1$
$\mid \mathit{max\text{-}stack}\ (\mathit{Var}\ i) = 1$
$\mid \mathit{max\text{-}stack}\ (i{:=}e) = \mathit{max\text{-}stack}\ e$
$\mid \mathit{max\text{-}stack}\ (e{\cdot}F\{D\}) = \mathit{max\text{-}stack}\ e$
$\mid \mathit{max\text{-}stack}\ (C{\cdot}_s F\{D\}) = 1$
$\mid \mathit{max\text{-}stack}\ (e_1{\cdot}F\{D\} := e_2) = \mathit{max}\ (\mathit{max\text{-}stack}\ e_1)\ (\mathit{max\text{-}stack}\ e_2) + 1$
$\mid \mathit{max\text{-}stack}\ (C{\cdot}_s F\{D\} := e_2) = \mathit{max\text{-}stack}\ e_2$
$\mid \mathit{max\text{-}stack}\ (e{\cdot}M(\mathit{es})) = \mathit{max}\ (\mathit{max\text{-}stack}\ e)\ (\mathit{max\text{-}stacks}\ \mathit{es}) + 1$
$\mid \mathit{max\text{-}stack}\ (C{\cdot}_s M(\mathit{es})) = \mathit{max\text{-}stacks}\ \mathit{es} + 1$
$\mid \mathit{max\text{-}stack}\ (\{i{:}T;\ e\}) = \mathit{max\text{-}stack}\ e$
$\mid \mathit{max\text{-}stack}\ (e_1;;e_2) = \mathit{max}\ (\mathit{max\text{-}stack}\ e_1)\ (\mathit{max\text{-}stack}\ e_2)$
$\mid \mathit{max\text{-}stack}\ (\mathit{if}\ (e)\ e_1\ \mathit{else}\ e_2) =$
$\ \ \mathit{max}\ (\mathit{max\text{-}stack}\ e)\ (\mathit{max}\ (\mathit{max\text{-}stack}\ e_1)\ (\mathit{max\text{-}stack}\ e_2))$
$\mid \mathit{max\text{-}stack}\ (\mathit{while}\ (e)\ c) = \mathit{max}\ (\mathit{max\text{-}stack}\ e)\ (\mathit{max\text{-}stack}\ c)$
$\mid \mathit{max\text{-}stack}\ (\mathit{throw}\ e) = \mathit{max\text{-}stack}\ e$
$\mid \mathit{max\text{-}stack}\ (\mathit{try}\ e_1\ \mathit{catch}(C\ i)\ e_2) = \mathit{max}\ (\mathit{max\text{-}stack}\ e_1)\ (\mathit{max\text{-}stack}\ e_2)$

| $\mathit{max\text{-}stacks}\ [] = 0$
| $\mathit{max\text{-}stacks}\ (e\#es) = \mathit{max}\ (\mathit{max\text{-}stack}\ e)\ (1 + \mathit{max\text{-}stacks}\ es)$

lemma $\mathit{max\text{-}stack1}'$: $\neg\mathit{sub\text{-}RI}\ e \Longrightarrow 1 \le \mathit{max\text{-}stack}\ e$
lemma $\mathit{compE_2\text{-}not\text{-}Nil}'$: $\neg\mathit{sub\text{-}RI}\ e \Longrightarrow \mathit{compE_2}\ e \neq []$
lemma $\mathit{compE_2\text{-}nRet}$: $\bigwedge i.\ i \in \mathit{set}\ (\mathit{compE_2}\ e_1) \Longrightarrow i \neq \mathit{Return}$
and $\bigwedge i.\ i \in \mathit{set}\ (\mathit{compEs_2}\ es_1) \Longrightarrow i \neq \mathit{Return}$
by(*induct rule*: $\mathit{compE_2.induct}\ \mathit{compEs_2.induct}$, *auto simp*: *nth-append split*: *bop.splits*)

definition $\mathit{compMb_2} :: \mathit{staticb} \Rightarrow \mathit{expr_1} \Rightarrow \mathit{jvm\text{-}method}$
where
$\mathit{compMb_2} \equiv \lambda b\ \mathit{body}.$
let $\mathit{ins} = \mathit{compE_2}\ \mathit{body}\ @\ [\mathit{Return}]$;
$\mathit{xt} = \mathit{compxE_2}\ \mathit{body}\ 0\ 0$
in (*max-stack body*, *max-vars body*, *ins*, *xt*)

definition $\mathit{compP_2} :: J_1\text{-}\mathit{prog} \Rightarrow \mathit{jvm\text{-}prog}$
where
$\mathit{compP_2} \equiv \mathit{compP}\ \mathit{compMb_2}$

lemma $\mathit{compMb_2}$ [*simp*]:
$\mathit{compMb_2}\ b\ e = (\mathit{max\text{-}stack}\ e, \mathit{max\text{-}vars}\ e,$
$\mathit{compE_2}\ e\ @\ [\mathit{Return}], \mathit{compxE_2}\ e\ 0\ 0)$
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**
$P,C,M,pc \triangleright is \longleftrightarrow \mathit{prefix}\ is\ (\mathit{drop}\ pc\ (\mathit{instrs\text{-}of}\ P\ C\ M))$

definition *at* :: *jvm-prog* ⇒ *cname* ⇒ *mname* ⇒ *nat* ⇒ *instr* ⇒ *bool*
(‹(-,-,-,-/ ▹ -)› [51,0,0,0,51] 50) **where**
$P,C,M,pc \triangleright i \longleftrightarrow (\exists is.\ \mathit{drop}\ pc\ (\mathit{instrs\text{-}of}\ P\ C\ M) = i\#is)$

lemma [*simp*]: $P,C,M,pc \triangleright []$

lemma [*simp*]: $P,C,M,pc \triangleright (i\#is) = (P,C,M,pc \triangleright i \wedge P,C,M,pc + 1 \triangleright is)$

lemma [*simp*]: $P,C,M,pc \triangleright (is_1\ @\ is_2) = (P,C,M,pc \triangleright is_1 \wedge P,C,M,pc + \mathit{size}\ is_1 \triangleright is_2)$

lemma [*simp*]: $P,C,M,pc \triangleright i \Longrightarrow \mathit{instrs\text{-}of}\ P\ C\ M\ !\ pc = i$

lemma *beforeM*:
$P \vdash C$ *sees* M,b: $Ts \to T$ = *body in* $D \Longrightarrow$
$compP_2\ P,D,M,0 \rhd compE_2\ body$ @ [*Return*]

This lemma executes a single instruction by rewriting:

lemma [*simp*]:
$P,C,M,pc \rhd instr \Longrightarrow$
$(P \vdash (None,\ h,\ (vs,ls,C,M,pc,ics)\ \#\ frs,\ sh)\ -jvm\to \sigma') =$
$((None,\ h,\ (vs,ls,C,M,pc,ics)\ \#\ frs,\ sh) = \sigma' \vee$
$(\exists \sigma.\ exec(P,(None,\ h,\ (vs,ls,C,M,pc,ics)\ \#\ frs,\ sh)) = Some\ \sigma \wedge P \vdash \sigma\ -jvm\to \sigma'))$

3.7.2 Exception tables

definition $pcs :: ex\text{-}table \Rightarrow nat\ set$
where
$pcs\ xt\ \equiv\ \bigcup (f,t,C,h,d) \in set\ xt.\ \{f\ ..< t\}$

lemma *pcs-subset*:
shows $(\bigwedge pc\ d.\ pcs(compxE_2\ e\ pc\ d) \subseteq \{pc..<pc+size(compE_2\ e)\})$
and $(\bigwedge pc\ d.\ pcs(compxEs_2\ es\ pc\ d) \subseteq \{pc..<pc+size(compEs_2\ es)\})$

lemma [*simp*]: $pcs\ [] = \{\}$

lemma [*simp*]: $pcs\ (x\#xt) = \{fst\ x\ ..< fst(snd\ x)\} \cup pcs\ xt$

lemma [*simp*]: $pcs(xt_1\ @\ xt_2) = pcs\ xt_1 \cup pcs\ xt_2$

lemma [*simp*]: $pc < pc_0 \vee pc_0+size(compE_2\ e) \le pc \Longrightarrow pc \notin pcs(compxE_2\ e\ pc_0\ d)$

lemma [*simp*]: $pc < pc_0 \vee pc_0+size(compEs_2\ es) \le pc \Longrightarrow pc \notin pcs(compxEs_2\ es\ pc_0\ d)$

lemma [*simp*]: $pc_1 + size(compE_2\ e_1) \le pc_2 \Longrightarrow pcs(compxE_2\ e_1\ pc_1\ d_1) \cap pcs(compxE_2\ e_2\ pc_2\ d_2) = \{\}$

lemma [*simp*]: $pc_1 + size(compE_2\ e) \le pc_2 \Longrightarrow pcs(compxE_2\ e\ pc_1\ d_1) \cap pcs(compxEs_2\ es\ pc_2\ d_2) = \{\}$

lemma [*simp*]:
$pc \notin pcs\ xt_0 \Longrightarrow match\text{-}ex\text{-}table\ P\ C\ pc\ (xt_0\ @\ xt_1) = match\text{-}ex\text{-}table\ P\ C\ pc\ xt_1$

lemma [*simp*]: $[\![\ x \in set\ xt;\ pc \notin pcs\ xt\ ]\!] \Longrightarrow \neg\ matches\text{-}ex\text{-}entry\ P\ D\ pc\ x$

lemma [*simp*]:
assumes *xe*: $xe \in set(compxE_2\ e\ pc\ d)$ **and** *outside*: $pc' < pc \vee pc+size(compE_2\ e) \le pc'$
shows $\neg\ matches\text{-}ex\text{-}entry\ P\ C\ pc'\ xe$

lemma [*simp*]:
assumes *xe*: $xe \in set(compxEs_2\ es\ pc\ d)$ **and** *outside*: $pc' < pc \vee pc+size(compEs_2\ es) \le pc'$
shows $\neg\ matches\text{-}ex\text{-}entry\ P\ C\ pc'\ xe$

lemma *match-ex-table-app*[*simp*]:
$\forall xte \in set\ xt_1.\ \neg\ matches\text{-}ex\text{-}entry\ P\ D\ pc\ xte \Longrightarrow$
$match\text{-}ex\text{-}table\ P\ D\ pc\ (xt_1\ @\ xt) = match\text{-}ex\text{-}table\ P\ D\ pc\ xt$

lemma [*simp*]:
$\forall x \in set\ xtab.\ \neg\ matches\text{-}ex\text{-}entry\ P\ C\ pc\ x \Longrightarrow$
$match\text{-}ex\text{-}table\ P\ C\ pc\ xtab = None$

lemma *match-ex-entry*:
$matches\text{-}ex\text{-}entry\ P\ C\ pc\ (start,\ end,\ catch\text{-}type,\ handler) =$
$(start \le pc \wedge pc < end \wedge\ P \vdash C \preceq^* catch\text{-}type)$

definition *caught* :: $jvm\text{-}prog \Rightarrow pc \Rightarrow heap \Rightarrow addr \Rightarrow ex\text{-}table \Rightarrow bool$ **where**
$caught\ P\ pc\ h\ a\ xt \longleftrightarrow$
$(\exists\ entry \in set\ xt.\ matches\text{-}ex\text{-}entry\ P\ (cname\text{-}of\ h\ a)\ pc\ entry)$

definition *beforex* :: $jvm\text{-}prog \Rightarrow cname \Rightarrow mname \Rightarrow ex\text{-}table \Rightarrow nat\ set \Rightarrow nat \Rightarrow bool$
(‹(2-,/-,/- ▷/ - /'/ -,/-)› [51,0,0,0,0,51] 50) **where**
$P,C,M \rhd xt\ /\ I,d \longleftrightarrow$
$(\exists xt_0\ xt_1.\ ex\text{-}table\text{-}of\ P\ C\ M = xt_0\ @\ xt\ @\ xt_1 \wedge pcs\ xt_0 \cap I = \{\} \wedge pcs\ xt \subseteq I \wedge$
$(\forall pc \in I.\ \forall C\ pc'\ d'.\ match\text{-}ex\text{-}table\ P\ C\ pc\ xt_1 = \lfloor(pc',d')\rfloor \longrightarrow d' \le d))$

definition *dummyx* :: $jvm\text{-}prog \Rightarrow cname \Rightarrow mname \Rightarrow ex\text{-}table \Rightarrow nat\ set \Rightarrow nat \Rightarrow bool$ (‹(2-,-,- ▷/ - '/-,-)› [51,0,0,0,0,51] 50) **where**
$P,C,M \triangleright xt/I,d \longleftrightarrow P,C,M \rhd xt/I,d$

abbreviation
$beforex_0\ P\ C\ M\ d\ I\ xt\ xt_0\ xt_1$
$\equiv ex\text{-}table\text{-}of\ P\ C\ M = xt_0\ @\ xt\ @\ xt_1 \wedge pcs\ xt_0 \cap I = \{\}$
$\wedge\ pcs\ xt \subseteq I \wedge (\forall pc \in I.\ \forall C\ pc'\ d'.\ match\text{-}ex\text{-}table\ P\ C\ pc\ xt_1 = \lfloor(pc',d')\rfloor \longrightarrow d' \le d)$

lemma *beforex-beforex*$_0$*-eq*:
$P,C,M \rhd xt\ /\ I,d \equiv \exists xt_0\ xt_1.\ beforex_0\ P\ C\ M\ d\ I\ xt\ xt_0\ xt_1$
using *beforex-def* **by** *auto*

lemma *beforexD1*: $P,C,M \rhd xt\ /\ I,d \Longrightarrow pcs\ xt \subseteq I$

lemma *beforex-mono*: $[\![\ P,C,M \rhd xt/I,d';\ d' \le d\ ]\!] \Longrightarrow P,C,M \rhd xt/I,d$

lemma [*simp*]: $P,C,M \rhd xt/I,d \Longrightarrow P,C,M \rhd xt/I,Suc\ d$

lemma *beforex-append*[*simp*]:
$pcs\ xt_1 \cap pcs\ xt_2 = \{\} \Longrightarrow$
$P,C,M \rhd xt_1\ @\ xt_2/I,d =$
$(P,C,M \rhd xt_1/I{-}pcs\ xt_2,d\ \wedge\ P,C,M \rhd xt_2/I{-}pcs\ xt_1,d \wedge P,C,M \triangleright xt_1@xt_2/I,d)$

lemma *beforex-appendD1*:
assumes *bx*: $P,C,M \rhd xt_1\ @\ xt_2\ @\ [(f,t,D,h,d)]\ /\ I,d$
and *pcs*: $pcs\ xt_1 \subseteq J$ **and** *JI*: $J \subseteq I$ **and** *Jpcs*: $J \cap pcs\ xt_2 = \{\}$
shows $P,C,M \rhd xt_1\ /\ J,d$

lemma *beforex-appendD2*:
assumes *bx*: $P,C,M \rhd xt_1\ @\ xt_2\ @\ [(f,t,D,h,d)]\ /\ I,d$
and *pcs*: $pcs\ xt_2 \subseteq J$ **and** *JI*: $J \subseteq I$ **and** *Jpcs*: $J \cap pcs\ xt_1 = \{\}$
shows $P,C,M \rhd xt_2\ /\ J,d$

lemma *beforexM*:
$P \vdash C\ sees\ M,b\colon\ Ts{\rightarrow}T = body\ in\ D \Longrightarrow compP_2\ P,D,M \rhd compxE_2\ body\ 0\ 0/\{..{<}size(compE_2$

body)}*,0*

lemma *match-ex-table-SomeD2*:
assumes *met*: *match-ex-table P D pc* (*ex-table-of P C M*) = ⌊(*pc*′,*d*′)⌋
and *bx*: *P*,*C*,*M* ▷ *xt*/*I*,*d*
and *nmet*: ∀ *x* ∈ *set xt*. ¬ *matches-ex-entry P D pc x* **and** *pcI*: *pc* ∈ *I*
shows *d*′ ≤ *d*

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*

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*
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*′
by(*simp add*: *handle-def find-handler-frs-tl-neq del*: *find-handler.simps*)

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*′)

$= ((\exists\, T.\ P,E,h,sh \vdash_1 \mathit{INIT}\ C_0\ (Cs,b) \leftarrow e' : T) \wedge \mathit{unit} = e' \wedge \mathit{ics} = \mathit{No\text{-}ics})$ |
$\mathit{Jcc\text{-}cond}\ P\ E\ C\ M\ vs\ pc\ ics\ I\ h\ sh\ (\mathit{RI}(C',e_0);Cs \leftarrow e')$
$= (((e_0 = C'\bullet_s \mathit{clinit}([]) \wedge (\exists\, T.\ P,E,h,sh \vdash_1 \mathit{RI}(C',e_0);Cs \leftarrow e'{:}T))$
$\vee ((\exists\, a.\ e_0 = \mathit{Throw}\ a) \wedge (\forall\, C \in \mathit{set}(C'\#Cs).\ \mathit{is\text{-}class}\ P\ C)))$
$\wedge\ \mathit{unit} = e' \wedge \mathit{ics} = \mathit{No\text{-}ics})$ |
$\mathit{Jcc\text{-}cond}\ P\ E\ C\ M\ vs\ pc\ ics\ I\ h\ sh\ (C'\bullet_s M'(es))$
$= (\mathit{let}\ e = (C'\bullet_s M'(es))$
$\mathit{in\ if}\ M' = \mathit{clinit} \wedge es = []\ \mathit{then}\ (\exists\, T.\ P,E,h,sh \vdash_1 e{:}T) \wedge (\exists\, Cs.\ \mathit{ics} = \mathit{Called}\ Cs)$
$\mathit{else}\ (\mathit{compP}_2\ P,C,M,pc \rhd \mathit{compE}_2\ e \wedge \mathit{compP}_2\ P,C,M \rhd \mathit{compxE}_2\ e\ pc\ (\mathit{size}\ vs)/I,\mathit{size}\ vs$
$\wedge \{pc..{<}pc{+}\mathit{size}(\mathit{compE}_2\ e)\} \subseteq I \wedge \neg \mathit{sub\text{-}RI}\ e \wedge \mathit{ics} = \mathit{No\text{-}ics})$
) |
$\mathit{Jcc\text{-}cond}\ P\ E\ C\ M\ vs\ pc\ ics\ I\ h\ sh\ e$
$= (\mathit{compP}_2\ P,C,M,pc \rhd \mathit{compE}_2\ e \wedge \mathit{compP}_2\ P,C,M \rhd \mathit{compxE}_2\ e\ pc\ (\mathit{size}\ vs)/I,\mathit{size}\ vs$
$\wedge \{pc..{<}pc{+}\mathit{size}(\mathit{compE}_2\ e)\} \subseteq I \wedge \neg \mathit{sub\text{-}RI}\ e \wedge \mathit{ics} = \mathit{No\text{-}ics})$

fun $\mathit{Jcc\text{-}frames} :: \mathit{jvm\text{-}prog} \Rightarrow \mathit{cname} \Rightarrow \mathit{mname} \Rightarrow \mathit{val\ list} \Rightarrow \mathit{val\ list} \Rightarrow pc \Rightarrow \mathit{init\text{-}call\text{-}status}$
$\Rightarrow \mathit{frame\ list} \Rightarrow \mathit{expr}_1 \Rightarrow \mathit{frame\ list}$ **where**
$\mathit{Jcc\text{-}frames}\ P\ C\ M\ vs\ ls\ pc\ ics\ frs\ (\mathit{INIT}\ C_0\ (C'\#Cs,b) \leftarrow e')$
$= (\mathit{case}\ b\ \mathit{of}\ \mathit{False} \Rightarrow (vs,ls,C,M,pc,\mathit{Calling}\ C'\ Cs)\ \#\ frs$
$\mid \mathit{True} \Rightarrow (vs,ls,C,M,pc,\mathit{Called}\ (C'\#Cs))\ \#\ frs$
) |
$\mathit{Jcc\text{-}frames}\ P\ C\ M\ vs\ ls\ pc\ ics\ frs\ (\mathit{INIT}\ C_0\ (\mathit{Nil},b) \leftarrow e')$
$= (vs,ls,C,M,pc,\mathit{Called}\ [])\#frs$ |
$\mathit{Jcc\text{-}frames}\ P\ C\ M\ vs\ ls\ pc\ ics\ frs\ (\mathit{RI}(C',e_0);Cs \leftarrow e')$
$= (\mathit{case}\ e_0\ \mathit{of}\ \mathit{Throw}\ a \Rightarrow (vs,ls,C,M,pc,\mathit{Throwing}\ (C'\#Cs)\ a)\ \#\ frs$
$\mid \text{-} \Rightarrow (vs,ls,C,M,pc,\mathit{Called}\ (C'\#Cs))\ \#\ frs\ )$ |
$\mathit{Jcc\text{-}frames}\ P\ C\ M\ vs\ ls\ pc\ ics\ frs\ (C'\bullet_s M'(es))$
$= (\mathit{if}\ M' = \mathit{clinit} \wedge es = []$
$\mathit{then\ create\text{-}init\text{-}frame}\ P\ C'\#(vs,ls,C,M,pc,ics)\#frs$
$\mathit{else}\ (vs,ls,C,M,pc,ics)\#frs$
) |
$\mathit{Jcc\text{-}frames}\ P\ C\ M\ vs\ ls\ pc\ ics\ frs\ e$
$= (vs,ls,C,M,pc,ics)\#frs$

fun $\mathit{Jcc\text{-}rhs} :: J_1\text{-}\mathit{prog} \Rightarrow \mathit{ty\ list} \Rightarrow \mathit{cname} \Rightarrow \mathit{mname} \Rightarrow \mathit{val\ list} \Rightarrow \mathit{val\ list} \Rightarrow pc \Rightarrow \mathit{init\text{-}call\text{-}status}$
$\Rightarrow \mathit{frame\ list} \Rightarrow \mathit{heap} \Rightarrow \mathit{val\ list} \Rightarrow \mathit{sheap} \Rightarrow \mathit{val} \Rightarrow \mathit{expr}_1 \Rightarrow \mathit{jvm\text{-}state}$ **where**
$\mathit{Jcc\text{-}rhs}\ P\ E\ C\ M\ vs\ ls\ pc\ ics\ frs\ h'\ ls'\ sh'\ v\ (\mathit{INIT}\ C_0\ (Cs,b) \leftarrow e')$
$= (\mathit{None},h',(vs,ls,C,M,pc,\mathit{Called}\ [])\#frs,sh')$ |
$\mathit{Jcc\text{-}rhs}\ P\ E\ C\ M\ vs\ ls\ pc\ ics\ frs\ h'\ ls'\ sh'\ v\ (\mathit{RI}(C',e_0);Cs \leftarrow e')$
$= (\mathit{None},h',(vs,ls,C,M,pc,\mathit{Called}\ [])\#frs,sh')$ |
$\mathit{Jcc\text{-}rhs}\ P\ E\ C\ M\ vs\ ls\ pc\ ics\ frs\ h'\ ls'\ sh'\ v\ (C'\bullet_s M'(es))$
$= (\mathit{let}\ e = (C'\bullet_s M'(es))$
$\mathit{in\ if}\ M' = \mathit{clinit} \wedge es = []$
$\mathit{then}\ (\mathit{None},h',(vs,ls,C,M,pc,ics)\#frs,sh'(C'\mapsto(\mathit{fst}(\mathit{the}(sh'\ C')),\mathit{Done})))$
$\mathit{else}\ (\mathit{None},h',(v\#vs,ls',C,M,pc{+}\mathit{size}(\mathit{compE}_2\ e),ics)\#frs,sh')$
) |
$\mathit{Jcc\text{-}rhs}\ P\ E\ C\ M\ vs\ ls\ pc\ ics\ frs\ h'\ ls'\ sh'\ v\ e$
$= (\mathit{None},h',(v\#vs,ls',C,M,pc{+}\mathit{size}(\mathit{compE}_2\ e),ics)\#frs,sh')$

fun $\mathit{Jcc\text{-}err} :: \mathit{jvm\text{-}prog} \Rightarrow \mathit{cname} \Rightarrow \mathit{mname} \Rightarrow \mathit{heap} \Rightarrow \mathit{val\ list} \Rightarrow \mathit{val\ list} \Rightarrow pc \Rightarrow \mathit{init\text{-}call\text{-}status}$
$\Rightarrow \mathit{frame\ list} \Rightarrow \mathit{sheap} \Rightarrow \mathit{nat\ set} \Rightarrow \mathit{heap} \Rightarrow \mathit{val\ list} \Rightarrow \mathit{sheap} \Rightarrow \mathit{addr} \Rightarrow \mathit{expr}_1$
$\Rightarrow \mathit{bool}$ **where**
$\mathit{Jcc\text{-}err}\ P\ C\ M\ h\ vs\ ls\ pc\ ics\ frs\ sh\ I\ h'\ ls'\ sh'\ xa\ (\mathit{INIT}\ C_0\ (Cs,b) \leftarrow e')$

= (∃ vs′. P ⊢ (None,h,Jcc-frames P C M vs ls pc ics frs (INIT C_0 (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′,e_0);Cs ← e′)
= (∃ vs′. P ⊢ (None,h,Jcc-frames P C M vs ls pc ics frs (RI(C′,e_0);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′$\bullet_s$M′(es))
= (let e = (C′$\bullet_s$M′(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 −
have *NC*: ∀ C′. e ≠ C′$\bullet_s$clinit([]) **using** *assms(2)* **proof**(*cases e*) **qed**(*simp-all*)
then show *?thesis* **using** *assms*
proof(*cases e*)
case (*SCall C M es*)
then have *M ≠ clinit* **using** *nsub* **by** *simp*
then show *?thesis* **using** *SCall nsub* **proof**(*cases es*) **qed**(*simp-all*)

qed(*simp-all*)
qed

lemma *Jcc-pieces-Cast*:
assumes [*simp*]: $P \equiv compP_2\ P_1$
and $Jcc\text{-}pieces\ P_1\ E\ C\ M\ h_0\ vs\ ls_0\ pc\ ics\ frs\ sh_0\ I\ h_1\ ls_1\ sh_1\ v\ xa\ (Cast\ C'\ e)$
$= (True,\ frs_0,\ (xp',h',(v\#vs',ls',C_0,M',pc',ics')\#frs',sh'),\ err)$
shows $Jcc\text{-}pieces\ P_1\ E\ C\ M\ h_0\ vs\ ls_0\ pc\ ics\ frs\ sh_0\ I\ h_1\ ls_1\ sh_1\ v'\ xa\ e$
$= (True,\ frs_0,\ (xp',h',(v'\#vs',ls',C_0,M',pc' - 1,ics')\#frs',sh'),$
$(\exists pc_1.\ pc \le pc_1 \wedge pc_1 < pc + size(compE_2\ e) \wedge$
$\neg\ caught\ P\ pc_1\ h_1\ xa\ (compxE_2\ e\ pc\ (size\ vs)) \wedge$
$(\exists vs'.\ P \vdash (None,h_0,frs_0,sh_0)\ -jvm\rightarrow\ handle\ P\ C\ M\ xa\ h_1\ (vs'@vs)\ ls_1\ pc_1\ ics\ frs\ sh_1)))$
proof −
have *pc*: $\{pc..<pc + length\ (compE_2\ e)\} \subseteq I$ **using** *assms* **by** *clarsimp*
show *?thesis* **using** *assms nsub-RI-Jcc-pieces*[**where** $e=e$] *pc* **by** *clarsimp*
qed

lemma *Jcc-pieces-BinOp1*:
assumes
$Jcc\text{-}pieces\ P\ E\ C\ M\ h_0\ vs\ ls_0\ pc\ ics\ frs\ sh_0\ I\ h_2\ ls_2\ sh_2\ v\ xa\ (e\ «bop»\ e')$
$= (True,\ frs_0,\ (xp',h',(v\#vs',ls',C_0,M',pc',ics')\#frs',sh'),\ err)$
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 −
have *bef*: $compP\ compMb_2\ P,C_0,M' \rhd compxE_2\ e\ pc\ (length\ vs)$
$/\ I - pcs\ (compxE_2\ e'\ (pc + length\ (compE_2\ e))\ (Suc\ (length\ vs'))),length\ vs$
using *assms* **by** *clarsimp*
have *vs*: $vs = vs'$ **using** *assms* **by** *simp*
show *?thesis* **using** *assms nsub-RI-Jcc-pieces*[**where** $e=e$] *bef vs* **by** *clarsimp*
qed

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 −
have *bef*: $compP\ compMb_2\ P_1,C_0,M' \rhd compxE_2\ e\ pc\ (length\ vs)$
$/\ I - pcs\ (compxE_2\ e'\ (pc + length\ (compE_2\ e))\ (Suc\ (length\ vs'))),length\ vs$
using *assms* **by** *clarsimp*
have *vs*: $vs = vs'$ **using** *assms* **by** *simp*
show *?thesis* **using** *assms nsub-RI-Jcc-pieces*[**where** $e=e'$] *bef vs* **by** *clarsimp*
qed

lemma *Jcc-pieces-FAcc*:
assumes

$\mathit{Jcc\text{-}pieces}\ P\ E\ C\ M\ h_0\ vs\ ls_0\ pc\ ics\ frs\ sh_0\ I\ h_1\ ls_1\ sh_1\ v\ xa\ (e{\bullet}F\{D\})$
$= (\mathit{True},\ frs_0,\ (xp',h',(v\#vs',ls',C_0,M',pc',ics')\#frs',sh'),\ err)$
shows $\exists\, err.\ \mathit{Jcc\text{-}pieces}\ P\ E\ C\ M\ h_0\ vs\ ls_0\ pc\ ics\ frs\ sh_0\ I\ h_1\ ls_1\ sh_1\ v'\ xa\ e$
$= (\mathit{True},\ frs_0,\ (xp',h',(v'\#vs',ls',C_0,M',pc' - 1,ics')\#frs',sh'),\ err)$
proof −
have *pc*: $\{pc..{<}pc + \mathit{length}\ (\mathit{compE}_2\ e)\} \subseteq I$ **using** *assms* **by** *clarsimp*
then show *?thesis* **using** *assms nsub-RI-Jcc-pieces*[**where** *e=e*] **by** *clarsimp*
qed

lemma *Jcc-pieces-LAss*:
assumes [*simp*]: $P \equiv \mathit{compP}_2\ P_1$
and $\mathit{Jcc\text{-}pieces}\ P_1\ E\ C\ M\ h_0\ vs\ ls_0\ pc\ ics\ frs\ sh_0\ I\ h_1\ ls_1\ sh_1\ v\ xa\ (i{:=}e)$
$= (\mathit{True},\ frs_0,\ (xp',h',(v\#vs',ls',C_0,M',pc',ics')\#frs',sh'),\ err)$
shows $\mathit{Jcc\text{-}pieces}\ P_1\ E\ C\ M\ h_0\ vs\ ls_0\ pc\ ics\ frs\ sh_0\ I\ h_1\ ls_1\ sh_1\ v'\ xa\ e$
$= (\mathit{True},\ frs_0,\ (xp',h',(v'\#vs',ls',C_0,M',pc' - 2,ics')\#frs',sh'),$
$(\exists\, pc_1.\ pc \le pc_1 \wedge pc_1 < pc + \mathit{size}(\mathit{compE}_2\ e) \wedge$
$\neg\ \mathit{caught}\ P\ pc_1\ h_1\ xa\ (\mathit{compxE}_2\ e\ pc\ (\mathit{size}\ vs)) \wedge$
$(\exists\, vs'.\ P \vdash (\mathit{None},h_0,frs_0,sh_0)\ -jvm\rightarrow\ \mathit{handle}\ P\ C\ M\ xa\ h_1\ (vs'@vs)\ ls_1\ pc_1\ ics\ frs\ sh_1)))$
proof −
have *pc*: $\{pc..{<}pc + \mathit{length}\ (\mathit{compE}_2\ e)\} \subseteq I$ **using** *assms* **by** *clarsimp*
show *?thesis* **using** *assms nsub-RI-Jcc-pieces*[**where** *e=e*] *pc* **by** *clarsimp*
qed

lemma *Jcc-pieces-FAss1*:
assumes
$\mathit{Jcc\text{-}pieces}\ P\ E\ C\ M\ h_0\ vs\ ls_0\ pc\ ics\ frs\ sh_0\ I\ h_2\ ls_2\ sh_2\ v\ xa\ (e{\bullet}F\{D\}{:=}e')$
$= (\mathit{True},\ frs_0,\ (xp',h',(v\#vs',ls',C_0,M',pc',ics')\#frs',sh'),\ err)$
shows $\exists\, err.\ \mathit{Jcc\text{-}pieces}\ P\ E\ C\ M\ h_0\ vs\ ls_0\ pc\ ics\ frs\ sh_0$
$(I - \mathit{pcs}\ (\mathit{compxE}_2\ e'\ (pc + \mathit{length}\ (\mathit{compE}_2\ e))\ (\mathit{Suc}\ (\mathit{length}\ vs'))))\ h_1\ ls_1\ sh_1\ v'\ xa\ e$
$= (\mathit{True},\ frs_0,\ (xp',h_1,(v'\#vs',ls_1,C_0,M',pc' - \mathit{size}\ (\mathit{compE}_2\ e') - 2,ics')\#frs',sh_1),\ err)$
proof −
show *?thesis* **using** *assms nsub-RI-Jcc-pieces*[**where** *e=e*] **by** *clarsimp*
qed

lemma *Jcc-pieces-FAss2*:
assumes
$\mathit{Jcc\text{-}pieces}\ P\ E\ C\ M\ h_0\ vs\ ls_0\ pc\ ics\ frs\ sh_0\ I\ h_2\ ls_2\ sh_2\ v\ xa\ (e{\bullet}F\{D\}{:=}e')$
$= (\mathit{True},\ frs_0,\ (xp',h',(v\#vs',ls',C_0,M',pc',ics')\#frs',sh'),\ err)$
shows $\mathit{Jcc\text{-}pieces}\ P\ E\ C\ M\ h_1\ (v_1\#vs)\ ls_1\ (pc + \mathit{size}\ (\mathit{compE}_2\ e))\ ics\ frs\ sh_1$
$(I - \mathit{pcs}\ (\mathit{compxE}_2\ e\ pc\ (\mathit{length}\ vs')))\ h_2\ ls_2\ sh_2\ v'\ xa\ e'$
$= (\mathit{True},\ (v_1\#vs,ls_1,C,M,pc + \mathit{size}\ (\mathit{compE}_2\ e),ics)\#frs,$
$(xp',h',(v'\#v_1\#vs',ls',C_0,M',pc' - 2,ics')\#frs',sh'),$
$(\exists\, pc_1.\ (pc + \mathit{size}\ (\mathit{compE}_2\ e)) \le pc_1 \wedge pc_1 < pc + \mathit{size}\ (\mathit{compE}_2\ e) + \mathit{size}(\mathit{compE}_2\ e') \wedge$
$\neg\ \mathit{caught}\ (\mathit{compP}_2\ P)\ pc_1\ h_2\ xa\ (\mathit{compxE}_2\ e'\ (pc + \mathit{size}\ (\mathit{compE}_2\ e))\ (\mathit{size}\ (v_1\#vs))) \wedge$
$(\exists\, vs'.\ (\mathit{compP}_2\ P) \vdash (\mathit{None},h_1,(v_1\#vs,ls_1,C,M,pc + \mathit{size}\ (\mathit{compE}_2\ e),ics)\#frs,sh_1)$
$-jvm\rightarrow\ \mathit{handle}\ (\mathit{compP}_2\ P)\ C\ M\ xa\ h_2\ (vs'@v_1\#vs)\ ls_2\ pc_1\ ics\ frs\ sh_2)))$
proof −
show *?thesis* **using** *assms nsub-RI-Jcc-pieces*[**where** *e=e'*] **by** *clarsimp*
qed

lemma *Jcc-pieces-SFAss*:
assumes
$\mathit{Jcc\text{-}pieces}\ P\ E\ C\ M\ h_0\ vs\ ls_0\ pc\ ics\ frs\ sh_0\ I\ h'\ ls'\ sh'\ v\ xa\ (C'{\bullet_s}F\{D\}{:=}e)$
$= (\mathit{True},\ frs_0,\ (xp',h',(v\#vs',ls',C_0,M',pc',ics')\#frs',sh'),\ err)$

shows $\exists$ *err. Jcc-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 −
have *pc*: {*pc..<pc + length* (*compE*$_2$ *e*)} ⊆ *I* **using** *assms* **by** *clarsimp*
show *?thesis* **using** *assms nsub-RI-Jcc-pieces*[**where** *e=e*] *pc* **by** *clarsimp*
qed

lemma *Jcc-pieces-Call1*:
assumes
Jcc-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*•M_0(*es*))
= (*True*, frs_0, (xp',h',(*v*#vs',ls',C',M',pc',ics')#frs',sh'), *err*)
shows $\exists$ *err. Jcc-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 −
show *?thesis* **using** *assms nsub-RI-Jcc-pieces*[**where** *e=e*] **by** *clarsimp*
qed

lemma *Jcc-pieces-clinit*:
assumes [*simp*]: *P* ≡ *compP*$_2$ P_1
and *cond*: *Jcc-cond* P_1 *E C M vs pc ics I h sh* (*C1*•$_s$*clinit*([]))
shows *Jcc-pieces* P_1 *E C M h vs ls pc ics frs sh I* h' ls' sh' *v xa* (*C1*•$_s$*clinit*([]))
= (*True*, *create-init-frame P C1* # (*vs,ls,C,M,pc,ics*)#*frs*,
(*None*, h', (*vs,ls,C,M,pc,ics*)#*frs*, sh'(*C1*↦(*fst*(*the*(sh' *C1*)),*Done*))),
P ⊢ (*None,h,create-init-frame P C1* # (*vs,ls,C,M,pc,ics*)#*frs,sh*) −*jvm*→
(*case ics of Called Cs* ⇒ (*None*,h',(*vs,ls,C,M,pc,Throwing Cs xa*)#*frs*,(sh'(*C1* ↦ (*fst*(*the*(sh' *C1*)),*Error*))))))
using *assms* **by**(*auto split*: *init-call-status.splits list.splits bool.splits*)

lemma *Jcc-pieces-SCall-clinit-body*:
assumes [*simp*]: *P* ≡ *compP*$_2$ P_1 **and** *wf*: *wf-J*$_1$*-prog* P_1
and *Jcc-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*•$_s$*clinit*([]))
= (*True*, frs', rhs', err')
and *method*: P_1 ⊢ *C1 sees clinit,Static*: []→*Void* = *body in D*
shows *Jcc-pieces* P_1 [] *D clinit* h_2 [] (*replicate* (*max-vars body*) *undefined*) *0*
No-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-ics*)#*tl* frs',sh_3),
$\exists$ pc_1. *0* ≤ pc_1 ∧ pc_1 < *size*(*compE*$_2$ *body*) ∧
¬ *caught P* pc_1 h_3 *xa* (*compxE*$_2$ *body 0 0*) ∧
($\exists$ vs'. *P* ⊢ (*None*,h_2,frs',sh_2) −*jvm*→ *handle P D clinit xa* h_3 vs' ls_3 pc_1
No-ics (*tl* frs') sh_3))
proof −
have *M-in-D*: P_1 ⊢ *D sees clinit,Static*: []→*Void* = *body in D*
using *method* **by**(*rule sees-method-idemp*)
hence *M-code*: *compP*$_2$ P_1,*D,clinit,0* ▷ *compE*$_2$ *body* @ [*Return*]
and *M-xtab*: *compP*$_2$ P_1,*D,clinit* ▷ *compxE*$_2$ *body 0 0*/{..<*size*(*compE*$_2$ *body*)},*0*
by(*rule beforeM*, *rule beforexM*)
have *nsub*: ¬*sub-RI body* **by**(*rule sees-wf*$_1$*-nsub-RI*[*OF wf method*])
then show *?thesis* **using** *assms nsub-RI-Jcc-pieces M-code M-xtab* **by** *clarsimp*
qed

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 $Jcc\text{-}pieces\ P_1\ E\ C\ M\ h\ vs\ ls\ pc\ ics\ frs\ sh$
$(I - pcs\ (compxEs_2\ es\ (pc + length\ (compE_2\ e))\ (Suc\ (length\ vs))))\ h'\ ls'\ sh'\ v\ xa\ e$
$= (True,\ (vs,\ ls,\ C,\ M,\ pc,\ ics)\ \#\ frs,$
$(None,\ h',\ (v\#vs,\ ls',\ C,\ M,\ pc + length\ (compE_2\ e),\ ics)\ \#\ frs,\ sh'),$
$\exists\, pc_1{\geq}pc.\ pc_1 < pc + length\ (compE_2\ e) \wedge \neg\ caught\ P\ pc_1\ h'\ xa\ (compxE_2\ e\ pc\ (length\ vs))$
$\wedge (\exists\, vs'.\ P \vdash (None,\ h,\ (vs,\ ls,\ C,\ M,\ pc,\ ics)\ \#\ frs,\ sh)$
$-jvm\rightarrow\ handle\ P\ C\ M\ xa\ h'\ (vs'@vs)\ ls'\ pc_1\ ics\ frs\ sh'))$
proof −
show *?thesis* **using** *assms nsub-RI-Jcc-pieces*[**where** *e=e*] **by** *auto*
qed

lemma *Jcc-pieces-InitNone*:
assumes [*simp*]: $P \equiv compP_2\ P_1$
and $Jcc\text{-}pieces\ P_1\ E\ C\ M\ h\ vs\ l\ pc\ ics\ frs\ sh\ I\ h'\ l'\ sh'\ v\ xa\ (INIT\ C'\ (C_0\ \#\ Cs,False) \leftarrow e)$
$= (True,\ frs',\ (None,\ h',\ (vs,\ l,\ C,\ M,\ pc,\ Called\ [])\ \#\ frs,\ sh'),\ err)$
shows
$Jcc\text{-}pieces\ P_1\ E\ C\ M\ h\ vs\ l\ pc\ ics\ frs\ (sh(C_0 \mapsto (sblank\ P\ C_0,\ Prepared)))$
$I\ h'\ l'\ sh'\ v\ xa\ (INIT\ C'\ (C_0\ \#\ Cs,False) \leftarrow e)$
$= (True,\ frs',\ (None,\ h',\ (vs,\ l,\ C,\ M,\ pc,\ Called\ [])\ \#\ frs,\ sh'),$
$\exists\, vs'.\ P \vdash (None,h,frs',(sh(C_0 \mapsto (sblank\ P_1\ C_0,\ Prepared))))$
$-jvm\rightarrow\ handle\ P\ C\ M\ xa\ h'\ (vs'@vs)\ l\ pc\ ics\ frs\ sh')$
proof −
have $Jcc\text{-}cond\ P_1\ E\ C\ M\ vs\ pc\ ics\ I\ h\ sh\ (INIT\ C'\ (C_0\ \#\ Cs,False) \leftarrow e)$ **using** *assms* **by** *simp*
then obtain T **where** $P_1,E,h,sh \vdash_1 INIT\ C'\ (C_0\ \#\ Cs,False) \leftarrow unit : T$ **by** *fastforce*
then have $P_1,E,h,sh(C_0 \mapsto (sblank\ P_1\ C_0,\ Prepared)) \vdash_1 INIT\ C'\ (C_0\ \#\ Cs,False) \leftarrow unit : T$
by(*auto simp*: *fun-upd-apply*)
then have $Ex\ (WTrt2_1\ P_1\ E\ h\ (sh(C_0 \mapsto (sblank\ P_1\ C_0,\ Prepared)))\ (INIT\ C'\ (C_0\ \#\ Cs,False)$
$\leftarrow unit))$
by(*simp only*: *exI*)
then show *?thesis* **using** *assms* **by** *clarsimp*
qed

lemma *Jcc-pieces-InitDP*:
assumes [*simp*]: $P \equiv compP_2\ P_1$
and $Jcc\text{-}pieces\ P_1\ E\ C\ M\ h\ vs\ l\ pc\ ics\ frs\ sh\ I\ h'\ l'\ sh'\ v\ xa\ (INIT\ C'\ (C_0\ \#\ Cs,False) \leftarrow e)$
$= (True,\ frs',\ (None,\ h',\ (vs,\ l,\ C,\ M,\ pc,\ Called\ [])\ \#\ frs,\ sh'),\ err)$
shows
$Jcc\text{-}pieces\ P_1\ E\ C\ M\ h\ vs\ l\ pc\ ics\ frs\ sh\ I\ h'\ l'\ sh'\ v\ xa\ (INIT\ C'\ (Cs,True) \leftarrow e)$
$= (True,\ (calling\text{-}to\text{-}scalled\ (hd\ frs'))\#(tl\ frs'),$
$(None,\ h',\ (vs,\ l,\ C,\ M,\ pc,\ Called\ [])\ \#\ frs,\ sh'),$
$\exists\, vs'.\ P \vdash (None,h,calling\text{-}to\text{-}scalled\ (hd\ frs')\#(tl\ frs'),sh)$
$-jvm\rightarrow\ handle\ P\ C\ M\ xa\ h'\ (vs'@vs)\ l\ pc\ ics\ frs\ sh')$
proof −
have $Jcc\text{-}cond\ P_1\ E\ C\ M\ vs\ pc\ ics\ I\ h\ sh\ (INIT\ C'\ (C_0\ \#\ Cs,False) \leftarrow e)$ **using** *assms* **by** *simp*
then obtain T **where** $P_1,E,h,sh \vdash_1 INIT\ C'\ (C_0\ \#\ Cs,False) \leftarrow unit : T$ **by** *fastforce*
then have $P_1,E,h,sh \vdash_1 INIT\ C'\ (Cs,True) \leftarrow unit : T$
by (*auto*; *metis list.sel*(*2*) *list.set-sel*(*2*))
then have *wtrt*: $Ex\ (WTrt2_1\ P_1\ E\ h\ sh\ (INIT\ C'\ (Cs,True) \leftarrow unit))$ **by**(*simp only*: *exI*)
show *?thesis* **using** *assms wtrt*

proof(*cases Cs*)
case (*Cons C1 Cs1*)
then show *?thesis* **using** *assms wtrt*
by(*case-tac method P C1 clinit*) *clarsimp*
qed(*clarsimp*)
qed

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*) ← *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* ← *e*)
= (*True*, (*calling-to-throwing* (*hd frs′*) (*addr-of-sys-xcpt NoClassDefFoundError*))#(*tl frs′*),
(*None*, *h′*, (*vs*, *l*, *C*, *M*, *pc*, *Called* []) # *frs*, *sh′*),
∃ *vs′*. *P* ⊢ (*None*,*h*, (*calling-to-throwing* (*hd frs′*) (*addr-of-sys-xcpt NoClassDefFoundError*))#(*tl frs′*),*sh*)
−*jvm*→ *handle P C M xa h′* (*vs′*@*vs*) *l pc ics frs sh′*)
proof −
show *?thesis* **using** *assms*
proof(*cases Cs*)
case (*Cons C1 Cs1*)
then show *?thesis* **using** *assms*
by(*case-tac method P C1 clinit*, *case-tac method P* C_0 *clinit*) *clarsimp*
qed(*clarsimp*)
qed

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 ↦ (*sfs*,*Processing*))) *v xa* (*INIT C′* (C_0 # *Cs*,*False*) ← *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 ↦ (*sfs*,*Processing*))) *I h′ l′ sh″ v xa* (*INIT C′* (C_0 # *Cs*,*True*) ← *e*)
= (*True*, *calling-to-called* (*hd frs′*)#(*tl frs′*),
(*None*, *h′*, (*vs*, *l*, *C*, *M*, *pc*, *Called* []) # *frs*, *sh″*),
∃ *vs′*. *P* ⊢ (*None*,*h*,*calling-to-called* (*hd frs′*)#(*tl frs′*),*sh′*)
−*jvm*→ *handle P C M xa h′* (*vs′*@*vs*) *l pc ics frs sh″*)
proof −
have *Jcc-cond* P_1 *E C M vs pc ics I h sh* (*INIT C′* (C_0 # *Cs*,*False*) ← *e*) **using** *assms* **by** *simp*
then obtain *T* **where** P_1,*E*,*h*,*sh* $\vdash_1$ *INIT C′* (C_0 # *Cs*,*False*) ← *unit* : *T* **by** *fastforce*
then have P_1,*E*,*h*,*sh*(C_0 ↦ (*sfs*,*Processing*)) $\vdash_1$ *INIT C′* (C_0 # *Cs*,*True*) ← *unit* : *T*
using *assms* **by** *clarsimp* (*auto simp*: *fun-upd-apply*)
then have *wtrt*: *Ex* (*WTrt2*$_1$ P_1 *E h* (*sh*(C_0 ↦ (*sfs*,*Processing*))) (*INIT C′* (C_0 # *Cs*,*True*) ← *unit*))
by(*simp only*: *exI*)
show *?thesis* **using** *assms wtrt* **by** *clarsimp*
qed

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-to-calling* (*hd* frs') *D*#(*tl* frs'),
(*None*, h', (*vs*, *l*, *C*, *M*, *pc*, *Called* []) # *frs*, sh''),
$\exists\, vs'$. $P \vdash$ (*None*,*h*,*calling-to-calling* (*hd* frs') *D*#(*tl* frs'),sh')
$-jvm\rightarrow$ *handle P C M xa* h' (vs'@*vs*) *l pc ics frs* sh'')
proof −
have *Jcc-cond* P_1 *E C M vs pc ics I h sh* (*INIT* C' $(C_0 \# Cs,False) \leftarrow e$) **using** *assms* **by** *simp*
then obtain *T* **where** P_1,*E*,*h*,*sh* $\vdash_1$ *INIT* C' $(C_0 \# Cs,False) \leftarrow unit : T$ **by** *fastforce*
then have P_1,*E*,*h*,$sh(C_0 \mapsto (sfs,Processing))$ $\vdash_1$ *INIT* C' $(D \# C_0 \# Cs,False) \leftarrow unit : T$
using *assms* **by** *clarsimp* (*auto simp*: *fun-upd-apply*)
then have *wtrt*: *Ex* (*WTrt2*$_1$ P_1 *E h* $(sh(C_0 \mapsto (sfs,Processing)))$ (*INIT* C' $(D \# C_0 \# Cs,False)$ $\leftarrow unit$))
by(*simp only*: *exI*)
show *?thesis* **using** *assms wtrt* **by** *clarsimp*
qed

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* $(C_0,C_0\bullet_s clinit([]))$; $Cs \leftarrow e$)
= (*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 −
have *cond*: *Jcc-cond* P_1 *E C M vs pc ics I h sh* (*INIT* C' $(C_0 \# Cs,True) \leftarrow e$) **using** *assms* **by** *simp*
then have *clinit*: $\exists\, T$. P_1,*E*,*h*,*sh* $\vdash_1$ $C_0\bullet_s clinit([]) : T$ **using** *wf*
by *clarsimp* (*auto simp*: *is-class-def intro*: *wf*$_1$*-types-clinit*)
then obtain *T* **where** *cT*: P_1,*E*,*h*,*sh* $\vdash_1$ $C_0\bullet_s clinit([]) : T$ **by** *blast*
obtain *T* **where** P_1,*E*,*h*,*sh* $\vdash_1$ *INIT* C' $(C_0 \# Cs,True) \leftarrow unit : T$ **using** *cond* **by** *fastforce*
then have P_1,*E*,*h*,*sh* $\vdash_1$ *RI* $(C_0,C_0\bullet_s clinit([]))$; $Cs \leftarrow unit : T$
using *assms* **by** (*auto intro*: *cT*)
then have *wtrt*: *Ex* (*WTrt2*$_1$ P_1 *E h sh* (*RI* $(C_0,C_0\bullet_s clinit([]))$; $Cs \leftarrow unit$))
by(*simp only*: *exI*)
then show *?thesis* **using** *assms* **by** *simp*
qed

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* $(C_0,C_0\bullet_s clinit([]))$;$Cs \leftarrow e$)
= (*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* $(C_0\bullet_s clinit([]))$
= (*True*, *create-init-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 −
have *cond*: *Jcc-cond* P_1 *E C M vs pc ics I h sh* $(RI\ (C_0,C_0{\cdot}_s clinit([]));Cs \leftarrow e)$ **using** *assms* **by** *simp*
then have *wtrt*: $\exists\, T.\ P_1,E,h,sh \vdash_1 C_0{\cdot}_s clinit([]) : T$ **using** *wf*
by *clarsimp* (*auto simp*: *is-class-def intro*: wf_1*-types-clinit*)
then show *?thesis* **using** *assms* **by** *clarsimp*
qed

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 −
have *Jcc-cond* P_1 *E C M vs pc ics I h sh* $(RI\ (C_0,C_0{\cdot}_s clinit([]));Cs \leftarrow e)$ **using** *assms* **by** *simp*
then have *Ex* $(WTrt2_1\ P_1\ E\ h\ sh\ (RI\ (C_0,C_0{\cdot}_s clinit([]))\ ;\ Cs \leftarrow unit))$ **by** *simp*
then obtain *T* **where** *riwt*: $P_1,E,h,sh \vdash_1 RI\ (C_0,C_0{\cdot}_s clinit([]));Cs \leftarrow unit : T$ **by** *meson*
then have $P_1,E,h',sh'' \vdash_1 INIT\ (last\ (C_0\#Cs))\ (Cs,True) \leftarrow unit : T$ **using** *proc*
proof(*cases Cs*) **qed**(*auto*)
then have *wtrt*: *Ex* $(WTrt2_1\ P_1\ E\ h'\ sh''\ (INIT\ (last\ (C_0\#Cs))\ (Cs,True) \leftarrow unit))$ **by**(*simp only*: *exI*)
show *?thesis* **using** *assms wtrt*
proof(*cases Cs*)
case (*Cons C1 Cs1*)
then show *?thesis* **using** *assms wtrt*
by(*case-tac method P C1 clinit*) *clarsimp*
qed(*clarsimp*)
qed

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* $(RI\ (D,Throw\ xa)\ ;\ Cs \leftarrow e')$
$= (True,\ (vs1,\ l1,\ C1,\ M1,\ pc1,\ Throwing\ (D\#Cs)\ xa)\ \#\ tl\ frs',$
$(None,\ h_1,\ (vs,\ l,\ C,\ M,\ pc,\ Called\ [])\ \#\ frs,\ sh_1),$
$\exists\, vs'.\ P \vdash (None,h',(vs1,\ l1,\ C1,\ M1,\ pc1,\ Throwing\ (D\#Cs)\ xa)\ \#\ tl\ frs',sh'')$
$-jvm\rightarrow handle\ P\ C\ M\ xa\ h_1\ (vs'@vs)\ l\ pc\ ics\ frs\ sh_1)$
using *assms* **by**(*case-tac method P D clinit*, *cases* $e = C_0{\cdot}_s clinit([])$) *clarsimp+*

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 –
have *cname*: $cname\text{-}of\ h_2\ (the\text{-}Addr\ ((rev\ pvs\ @\ Addr\ a\ \#\ vs)\ !\ length\ Ts)) = Ca$
using *ha method len* **by**(*auto simp*: *nth-append*)
have *r*: $(rev\ pvs\ @\ Addr\ a\ \#\ vs)\ !\ (length\ Ts) = Addr\ a$ **using** *len* **by**(*auto simp*: *nth-append*)
have *exm*: $\exists\ Ts\ T\ m\ D\ b.\ P \vdash Ca\ sees\ M',b{:}Ts \rightarrow T = m\ in\ D$
using *sees-method-compP*[*OF method*] **by** *fastforce*
show *?thesis* **using** *assms cname r exm* **by** *simp*
qed

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 –
have *exm*: $\exists\ Ts\ T\ m\ D\ b.\ P \vdash C'\ sees\ M',b{:}Ts \rightarrow T = m\ in\ D$
using *sees-method-compP*[*OF method*] **by** *fastforce*
show *?thesis* **using** *assms exm* **by** *simp*
qed

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 –
have *exm*: $\exists\ Ts\ T\ m\ D\ b.\ P \vdash C'\ sees\ M',b{:}Ts \rightarrow T = m\ in\ D$
using *sees-method-compP*[*OF method*] **by** *fastforce*
show *?thesis* **using** *assms exm* **by** *simp*
qed

lemma *jvm-Return-Init*:
$P,D,clinit,0 \triangleright compE_2\ body\ @\ [Return]$
$\Longrightarrow P \vdash (None,\ h,\ (vs,\ ls,\ D,\ clinit,\ size(compE_2\ body),\ No\text{-}ics)\ \#\ frs,\ sh)$
${-jvm\rightarrow}\ (None,\ h,\ frs,\ sh(D{\mapsto}(fst(the(sh\ D)),Done)))$
(**is** $?P \Longrightarrow P \vdash\ ?s1\ {-jvm\rightarrow}\ ?s2$)
proof –
assume *?P*
then have $exec\ (P,\ ?s1) = \lfloor ?s2\rfloor$ **by**(*cases frs*) *auto*

then have $(?s1, ?s2) \in (exec\text{-}1\ P)^*$
by(*rule exec-1I*[*THEN r-into-rtrancl*])
then show *?thesis* **by**(*simp add*: *exec-all-def1*)
qed

lemma *jvm-InitNone*:
$[\![$ *ics-of f* = *Calling C Cs*;
sh C = *None* $]\!]$
$\Longrightarrow P \vdash (None,h,f\#frs,sh)\ -jvm\rightarrow\ (None,h,f\#frs,sh(C \mapsto (sblank\ P\ C,\ Prepared)))$
(**is** $[\![$ *?P*; *?Q* $]\!] \Longrightarrow P \vdash ?s1\ -jvm\rightarrow\ ?s2$)
proof −
assume *assms*: *?P ?Q*
then obtain *stk1 loc1 C1 M1 pc1 ics1* **where** *f* = (*stk1*,*loc1*,*C1*,*M1*,*pc1*,*ics1*)
by(*cases f*) *simp*
then have *exec* (*P*, *?s1*) = ⌊*?s2*⌋ **using** *assms*
by(*case-tac ics1*) *simp-all*
then have $(?s1, ?s2) \in (exec\text{-}1\ P)^*$
by(*rule exec-1I*[*THEN r-into-rtrancl*])
then show *?thesis* **by**(*simp add*: *exec-all-def1*)
qed

lemma *jvm-InitDP*:
$[\![$ *ics-of f* = *Calling C Cs*;
sh C = ⌊(*sfs*,*i*)⌋; *i* = *Done* ∨ *i* = *Processing* $]\!]$
$\Longrightarrow P \vdash (None,h,f\#frs,sh)\ -jvm\rightarrow\ (None,h,(calling\text{-}to\text{-}scalled\ f)\#frs,sh)$
(**is** $[\![$ *?P*; *?Q*; *?R* $]\!] \Longrightarrow P \vdash ?s1\ -jvm\rightarrow\ ?s2$)
proof −
assume *assms*: *?P ?Q ?R*
then obtain *stk1 loc1 C1 M1 pc1 ics1* **where** *f* = (*stk1*,*loc1*,*C1*,*M1*,*pc1*,*ics1*)
by(*cases f*) *simp*
then have *exec* (*P*, *?s1*) = ⌊*?s2*⌋ **using** *assms*
by(*case-tac i*) *simp-all*
then have $(?s1, ?s2) \in (exec\text{-}1\ P)^*$
by(*rule exec-1I*[*THEN r-into-rtrancl*])
then show *?thesis* **by**(*simp add*: *exec-all-def1*)
qed

lemma *jvm-InitError*:
sh C = ⌊(*sfs*,*Error*)⌋
$\Longrightarrow P \vdash (None,h,(vs,ls,C_0,M,pc,Calling\ C\ Cs)\#frs,sh)$
$-jvm\rightarrow\ (None,h,(vs,ls,C_0,M,pc,Throwing\ Cs\ (addr\text{-}of\text{-}sys\text{-}xcpt\ NoClassDefFoundError))\#frs,sh)$
by(*clarsimp simp*: *exec-all-def1 intro*!: *r-into-rtrancl exec-1I*)

lemma *exec-ErrorThrowing*:
sh C = ⌊(*sfs*,*Error*)⌋
$\Longrightarrow$ *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*)
by(*clarsimp simp*: *exec-all-def1 fun-upd-idem-iff intro*!: *r-into-rtrancl exec-1I*)

lemma *jvm-InitObj*:
$[\![$ *sh C* = *Some*(*sfs*,*Prepared*);
C = *Object*;
sh′ = *sh*(*C* ↦ (*sfs*,*Processing*)) $]\!]$
$\Longrightarrow P \vdash (None,\ h,\ (vs,ls,C_0,M,pc,Calling\ C\ Cs)\#frs,\ sh)\ -jvm\rightarrow$

(*None*, *h*, (*vs*,*ls*,C_0,*M*,*pc*,*Called* (*C*#*Cs*))#*frs*,*sh*′)
(**is** ⟦ *?P*; *?Q*; *?R* ⟧ ⟹ *P* ⊢ *?s1* −*jvm*→ *?s2*)
proof −
  assume *assms*: *?P ?Q ?R*
  then have *exec* (*P*, *?s1*) = ⌊*?s2*⌋
    by(*case-tac method P C clinit*) *simp*
  then have (*?s1*, *?s2*) ∈ (*exec-1 P*)*
    by(*rule exec-1I*[*THEN r-into-rtrancl*])
  then show *?thesis* **by**(*simp add*: *exec-all-def1*)
qed

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 −
  assume *assms*: *?P ?Q ?R ?S*
  then have *exec* (*P*, *?s1*) = ⌊*?s2*⌋
    by(*case-tac method P C clinit*) *simp*
  then have (*?s1*, *?s2*) ∈ (*exec-1 P*)*
    by(*rule exec-1I*[*THEN r-into-rtrancl*])
  then show *?thesis* **by**(*simp add*: *exec-all-def1*)
qed

lemma *jvm-RInit-throw*:
P ⊢ (*None*,*h*,(*vs*,*l*,*C*,*M*,*pc*,*Throwing* [] *xa*) # *frs*,*sh*)
    −*jvm*→ *handle P C M xa h vs l pc No-ics frs sh*
(**is** *P* ⊢ *?s1* −*jvm*→ *?s2*)
proof −
  have *exec* (*P*, *?s1*) = ⌊*?s2*⌋
    by(*simp add*: *handle-def split*: *bool.splits*)
  then have (*?s1*, *?s2*) ∈ (*exec-1 P*)*
    by(*rule exec-1I*[*THEN r-into-rtrancl*])
  then show *?thesis* **by**(*simp add*: *exec-all-def1*)
qed

lemma *jvm-RInit-throw*′:
P ⊢ (*None*,*h*,(*vs*,*l*,*C*,*M*,*pc*,*Throwing* [*C*′] *xa*) # *frs*,*sh*)
    −*jvm*→ *handle P C M xa h vs l pc No-ics frs* (*sh*(*C*′:=*Some*(*fst*(*the*(*sh C*′)), *Error*)))
(**is** *P* ⊢ *?s1* −*jvm*→ *?s2*)
proof −
  let *?sy* = (*None*,*h*,(*vs*,*l*,*C*,*M*,*pc*,*Throwing* [] *xa*) # *frs*,*sh*(*C*′:=*Some*(*fst*(*the*(*sh C*′)), *Error*)))
  have *exec* (*P*, *?s1*) = ⌊*?sy*⌋ **by** *simp*
  then have (*?s1*, *?sy*) ∈ (*exec-1 P*)*
    by(*rule exec-1I*[*THEN r-into-rtrancl*])
  also have (*?sy*, *?s2*) ∈ (*exec-1 P*)*
    using *jvm-RInit-throw* **by**(*simp add*: *exec-all-def1*)
  ultimately show *?thesis* **by**(*simp add*: *exec-all-def1*)
qed

lemma *jvm-Called*:
$P \vdash (None,\ h,\ (vs,\ l,\ C,\ M,\ pc,\ Called\ (C_0\ \#\ Cs))\ \#\ frs,\ sh)\ -jvm\rightarrow$
$(None,\ h,\ create\text{-}init\text{-}frame\ P\ C_0\ \#\ (vs,\ l,\ C,\ M,\ pc,\ Called\ Cs)\ \#\ frs,\ sh)$
by(*simp add*: *exec-all-def1 r-into-rtrancl exec-1I*)

lemma *jvm-Throwing*:
$P \vdash (None,\ h,\ (vs,\ l,\ C,\ M,\ pc,\ Throwing\ (C_0\#Cs)\ xa')\ \#\ frs,\ sh)\ -jvm\rightarrow$
$(None,\ h,\ (vs,\ l,\ C,\ M,\ pc,\ Throwing\ Cs\ xa')\ \#\ frs,\ sh(C_0 \mapsto (fst\ (the\ (sh\ C_0)),\ Error)))$
by(*simp add*: *exec-all-def1 r-into-rtrancl exec-1I*)

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*: $create\text{-}init\text{-}frame\ (compP_2\ P)\ C = (stk,loc,D,M,pc,ics) \Longrightarrow D=C$
using *wf-sees-clinit*[*OF wf ex*] **by**(*cases method P C clinit*, *auto*)

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 −
obtain $xt_0\ xt_1$ **where**
$beforex_0\ P\ C\ M\ (size\ vs)\ I\ (compxE_2\ (try\ e_1\ catch(Ci\ i)\ e_2)\ pc\ (size\ vs))\ xt_0\ xt_1$
using *bx* **by**(*clarsimp simp*:*beforex-def*)
then have $\exists xt1.\ beforex_0\ P\ C\ M\ (size\ vs)\ \{pc..<pc + length\ (compE_2\ e_1)\}$
$(compxE_2\ e_1\ pc\ (size\ vs))\ xt_0\ xt1$
using *pcI pcs-subset*(*1*) *atLeastLessThan-iff* **by** *simp blast*
then show *?thesis* **using** *beforex-def* **by** *blast*
qed

— Evaluation of initialization expressions

lemma
shows $eval_1$*-init-return*: $P \vdash_1 \langle e,s\rangle \Rightarrow \langle e',s'\rangle$
$\Longrightarrow iconf\ (shp_1\ s)\ e$
$\Longrightarrow (\exists Cs\ b.\ e = INIT\ C'\ (Cs,b) \leftarrow unit) \vee (\exists C\ e_0\ Cs\ e_i.\ e = RI(C,e_0);Cs@[C'] \leftarrow unit)$
$\vee (\exists e_0.\ e = RI(C',e_0);Nil \leftarrow unit)$
$\Longrightarrow (val\text{-}of\ e' = Some\ v \longrightarrow (\exists sfs\ i.\ shp_1\ s'\ C' = \lfloor(sfs,i)\rfloor \wedge (i = Done \vee i = Processing)))$
$\wedge (throw\text{-}of\ e' = Some\ a \longrightarrow (\exists sfs\ i.\ shp_1\ s'\ C' = \lfloor(sfs,Error)\rfloor))$
and $P \vdash_1 \langle es,s\rangle\ [\Rightarrow]\ \langle es',s'\rangle \Longrightarrow True$
proof(*induct rule*: $eval_1$-$evals_1$*.inducts*)
case ($InitFinal_1\ e\ s\ e'\ s'\ C\ b$) **then show** *?case*
by(*auto simp*: *initPD-def dest*: $eval_1$*-final-same*)
next
case ($InitDone_1\ sh\ C\ sfs\ C'\ Cs\ e\ h\ l\ e'\ s'$)
then have *final* e' **using** $eval_1$*-final* **by** *simp*
then show *?case*
proof(*rule finalE*)
fix *v* **assume** e': $e' = Val\ v$ **then show** *?thesis* **using** $InitDone_1$ *initPD-def*
proof(*cases Cs*) **qed**(*auto*)
next
fix *a* **assume** e': $e' = throw\ a$ **then show** *?thesis* **using** $InitDone_1$ *initPD-def*

```
    proof(cases Cs) qed(auto)
  qed
next
  case (InitProcessing₁ sh C sfs C' Cs e h l e' s')
  then have final e' using eval₁-final by simp
  then show ?case
  proof(rule finalE)
    fix v assume e': e' = Val v then show ?thesis using InitProcessing₁ initPD-def
    proof(cases Cs) qed(auto)
  next
    fix a assume e': e' = throw a then show ?thesis using InitProcessing₁ initPD-def
    proof(cases Cs) qed(auto)
  qed
next
  case (InitError₁ sh C sfs Cs e h l e' s' C') show ?case
  proof(cases Cs)
    case Nil then show ?thesis using InitError₁ by simp
  next
    case (Cons C2 list)
    then have final e' using InitError₁ eval₁-final by simp
    then show ?thesis
    proof(rule finalE)
      fix v assume e': e' = Val v show ?thesis
        using InitError₁.hyps(2) e' rinit₁-throwE by blast
    next
      fix a assume e': e' = throw a
      then show ?thesis using Cons InitError₁ cons-to-append[of list] by clarsimp
    qed
  qed
next
  case (InitRInit₁ C Cs h l sh e' s' C') show ?case
  proof(cases Cs)
    case Nil then show ?thesis using InitRInit₁ by simp
  next
    case (Cons C' list) then show ?thesis
      using InitRInit₁ Cons cons-to-append[of list] by clarsimp
  qed
next
  case (RInit₁ e s v h' l' sh' C sfs i sh'' C' Cs e' e₁ s₁)
  then have final: final e₁ using eval₁-final by simp
  then show ?case
  proof(cases Cs)
    case Nil show ?thesis using final
    proof(rule finalE)
      fix v assume e': e₁ = Val v show ?thesis
      using RInit₁ Nil by(clarsimp, meson fun-upd-same initPD-def)
    next
      fix a assume e': e₁ = throw a show ?thesis
      using RInit₁ Nil by(clarsimp, meson fun-upd-same initPD-def)
    qed
  next
    case (Cons a list) show ?thesis using final
    proof(rule finalE)
      fix v assume e': e₁ = Val v then show ?thesis
```

using $RInit_1$ *Cons* **by**(*clarsimp*, *metis last.simps last-appendR list.distinct*(*1*))
next
fix *a* **assume** e': $e_1 = throw\ a$ **then show** *?thesis*
using $RInit_1$ *Cons* **by**(*clarsimp*, *metis last.simps last-appendR list.distinct*(*1*))
qed
qed
next
case ($RInitInitFail_1\ e\ s\ a\ h'\ l'\ sh'\ C\ sfs\ i\ sh''\ D\ Cs\ e'\ e_1\ s_1$)
then have *final*: $final\ e_1$ **using** $eval_1$*-final* **by** *simp*
then show *?case*
proof(*rule finalE*)
fix *v* **assume** e': $e_1 = Val\ v$ **then show** *?thesis*
using $RInitInitFail_1$ **by**(*clarsimp*, *meson exp.distinct*(*101*) $rinit_1$*-throwE*)
next
fix a' **assume** e': $e_1 = Throw\ a'$
then have $iconf\ (sh'(C \mapsto (sfs,\ Error)))\ a$
using $RInitInitFail_1$*.hyps*(*1*) $eval_1$*-final* **by** *fastforce*
then show *?thesis* **using** $RInitInitFail_1\ e'$
by(*clarsimp*, *meson Cons-eq-append-conv list.inject*)
qed
qed(*auto simp*: *fun-upd-same*)

lemma $init_1$*-Val-PD*: $P \vdash_1 \langle INIT\ C'\ (Cs,b) \leftarrow unit,s\rangle \Rightarrow \langle Val\ v,s'\rangle$
$\Longrightarrow iconf\ (shp_1\ s)\ (INIT\ C'\ (Cs,b) \leftarrow unit)$
$\Longrightarrow \exists\, sfs\ i.\ shp_1\ s'\ C' = \lfloor(sfs,i)\rfloor \wedge (i = Done \vee i = Processing)$
by(*drule-tac* $v = v$ **in** $eval_1$*-init-return*, *simp+*)

lemma $init_1$*-throw-PD*: $P \vdash_1 \langle INIT\ C'\ (Cs,b) \leftarrow unit,s\rangle \Rightarrow \langle throw\ a,s'\rangle$
$\Longrightarrow iconf\ (shp_1\ s)\ (INIT\ C'\ (Cs,b) \leftarrow unit)$
$\Longrightarrow \exists\, sfs\ i.\ shp_1\ s'\ C' = \lfloor(sfs,Error)\rfloor$
by(*drule-tac* $a = a$ **in** $eval_1$*-init-return*, *simp+*)

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(*cases Cs*)
case *Nil*
then show *?thesis* **using** $eval_1$*-init-return*[*OF eval iconf*] *last* **by** *simp*
next
case (*Cons a list*)
then have *nNil*: $Cs \neq []$ **by** *simp*
then have $\exists\, Cs'.\ Cs = Cs'\ @\ [C']$ **using** *last append-butlast-last-id*[*OF nNil*]
by(*rule-tac x=butlast Cs* **in** *exI*) *simp*
then show *?thesis* **using** $eval_1$*-init-return*[*OF eval iconf*] **by** *simp*
qed

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(*cases Cs*)
case *Nil*
then show *?thesis* **using** $eval_1$*-init-return*[*OF eval iconf*] *last* **by** *simp*

next
  case (*Cons a list*)
  then have *nNil*: $Cs \neq []$ **by** *simp*
  then have $\exists\, Cs'.\ Cs = Cs'\ @\ [C']$ **using** *last append-butlast-last-id*[*OF nNil*]
    by(*rule-tac x=butlast Cs* **in** *exI*) *simp*
  then show *?thesis* **using** $eval_1$*-init-return*[*OF eval iconf*] **by** *simp*
qed

The proof

lemma fixes P_1 **defines** [*simp*]: $P \equiv compP_2\ P_1$
assumes *wf*: $wf\text{-}J_1\text{-}prog\ P_1$
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** $P_1 \vdash_1 \langle es,(h_0,ls_0,sh_0)\rangle\ [\Rightarrow]\ \langle fs,(h_1,ls_1,sh_1)\rangle \Longrightarrow$
   $(\bigwedge C\ M\ pc\ ics\ ws\ xa\ es'\ vs\ frs\ I.$
    $[\![\ P,C,M,pc \rhd compEs_2\ es;\ P,C,M \rhd compxEs_2\ es\ pc\ (size\ vs)/I,size\ vs;$
     $\{pc..{<}pc{+}size(compEs_2\ es)\} \subseteq I;\ ics = No\text{-}ics;$
     $\neg sub\text{-}RIs\ es\ ]\!] \Longrightarrow$
    $(fs = map\ Val\ ws \longrightarrow$
     $P \vdash (None,h_0,(vs,ls_0,C,M,pc,ics)\#frs,sh_0)\ -jvm\rightarrow$
          $(None,h_1,(rev\ ws\ @\ vs,ls_1,C,M,pc{+}size(compEs_2\ es),ics)\#frs,sh_1))$
    $\wedge$
    $(fs = map\ Val\ ws\ @\ Throw\ xa\ \#\ es' \longrightarrow$
     $(\exists\, pc_1.\ pc \leq pc_1 \wedge pc_1 < pc + size(compEs_2\ es) \wedge$
           $\neg\ caught\ P\ pc_1\ h_1\ xa\ (compxEs_2\ es\ pc\ (size\ vs)) \wedge$
           $(\exists\, vs'.\ P \vdash (None,h_0,(vs,ls_0,C,M,pc,ics)\#frs,sh_0)$
                          $-jvm\rightarrow\ handle\ P\ C\ M\ xa\ h_1\ (vs'@vs)\ ls_1\ pc_1\ ics\ frs\ sh_1))))$

lemma *atLeast0AtMost*[*simp*]: $\{0{::}nat..n\} = \{..n\}$
by *auto*

lemma *atLeast0LessThan*[*simp*]: $\{0{::}nat..{<}n\} = \{..{<}n\}$
by *auto*

fun *exception* :: $'a\ exp \Rightarrow addr\ option$ **where**
  $exception\ (Throw\ a) = Some\ a$
| $exception\ e = None$

lemma $comp_2$*-correct*:
assumes *wf*: $wf\text{-}J_1\text{-}prog\ P_1$
  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 $compP_2\ P_1 \vdash (None,h,[([],ls,C,M,0,No\text{-}ics)],sh)\ -jvm\rightarrow\ (exception\ e',h',[],sh')$
end

3.8 Combining Stages 1 and 2

theory *Compiler*
imports *Correctness1 Correctness2*
begin

definition *J2JVM* :: *J-prog* ⇒ *jvm-prog*
where
$J2JVM \ \equiv \ compP_2 \circ compP_1$

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*   *size rest* = *max-vars body*
shows $J2JVM\ P \vdash (None,h,[([],vs@rest,C,M,0,No\text{-}ics)],sh)\ -jvm\rightarrow\ (exception\ e',h',[],sh')$
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*   *size rest* = *max-vars body*
and *nclinit*: $M \neq clinit$
shows $J2JVM\ P \vdash (None,h,[([],vs@rest,C,M,0,No\text{-}ics)],sh)\ -jvm\rightarrow\ (exception\ e',h',[],sh')$
end

3.9 Preservation of Well-Typedness

theory *TypeComp*
imports *Compiler ../BV/BVSpec*
begin

lemma *max-stack1*: $P,E \vdash_1 e :: T \Longrightarrow 1 \leq max\text{-}stack\ e$
locale *TC0* =
fixes $P :: J_1\text{-}prog$ **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*]: $P,E \vdash_1 e :: T \Longrightarrow ty\ E\ e = T$
lemma (**in** *TC0*) [*simp*]: $ty_i'\ ST\ E\ None = None$
lemma (**in** *TC0*) ty_l-*app-diff*[*simp*]:
$ty_l\ (E@[T])\ (A - \{size\ E\}) = ty_l\ E\ A$

lemma (**in** *TC0*) ty_i'-*app-diff*[*simp*]:
$ty_i'\ ST\ (E\ @\ [T])\ (A \ominus size\ E) = ty_i'\ ST\ E\ A$

lemma (**in** *TC0*) ty_l*-antimono*:
$A \subseteq A' \Longrightarrow P \vdash ty_l\ E\ A'\ [\le_\top]\ ty_l\ E\ A$

lemma (**in** *TC0*) ty_i'*-antimono*:
$A \subseteq A' \Longrightarrow P \vdash ty_i'\ ST\ E\ \lfloor A'\rfloor \le' ty_i'\ ST\ E\ \lfloor A\rfloor$

lemma (**in** *TC0*) ty_l*-env-antimono*:
$P \vdash ty_l\ (E@[T])\ A\ [\le_\top]\ ty_l\ E\ A$

lemma (**in** *TC0*) ty_i'*-env-antimono*:
$P \vdash ty_i'\ ST\ (E@[T])\ A \le' ty_i'\ ST\ E\ A$

lemma (**in** *TC0*) ty_i'*-incr*:
$P \vdash ty_i'\ ST\ (E\ @\ [T])\ \lfloor insert\ (size\ E)\ A\rfloor \le' ty_i'\ ST\ E\ \lfloor A\rfloor$

lemma (**in** *TC0*) ty_l*-incr*:
$P \vdash ty_l\ (E\ @\ [T])\ (insert\ (size\ E)\ A)\ [\le_\top]\ ty_l\ E\ A$

lemma (**in** *TC0*) ty_l*-in-types*:
$set\ E \subseteq types\ P \Longrightarrow ty_l\ E\ A \in nlists\ mxl\ (err\ (types\ P))$
locale *TC1* = *TC0*
begin

primrec $compT :: ty\ list \Rightarrow nat\ hyperset \Rightarrow ty\ list \Rightarrow expr_1 \Rightarrow ty_i'\ list$ **and**
$compTs :: ty\ list \Rightarrow nat\ hyperset \Rightarrow ty\ list \Rightarrow expr_1\ list \Rightarrow ty_i'\ list$ **where**
$compT\ E\ A\ ST\ (new\ C) = []$
$|\ compT\ E\ A\ ST\ (Cast\ C\ e) =$
$compT\ E\ A\ ST\ e\ @\ [after\ E\ A\ ST\ e]$
$|\ compT\ E\ A\ ST\ (Val\ v) = []$
$|\ compT\ E\ A\ ST\ (e_1\ «bop»\ e_2) =$
$(let\ ST_1 = ty\ E\ e_1\#ST;\ A_1 = A \sqcup \mathcal{A}\ e_1\ in$
$compT\ E\ A\ ST\ e_1\ @\ [after\ E\ A\ ST\ e_1]\ @$
$compT\ E\ A_1\ ST_1\ e_2\ @\ [after\ E\ A_1\ ST_1\ e_2])$
$|\ compT\ E\ A\ ST\ (Var\ i) = []$
$|\ compT\ E\ A\ ST\ (i := e) = compT\ E\ A\ ST\ e\ @$
$[after\ E\ A\ ST\ e,\ ty_i'\ ST\ E\ (A \sqcup \mathcal{A}\ e \sqcup \lfloor\{i\}\rfloor)]$
$|\ compT\ E\ A\ ST\ (e\cdot F\{D\}) =$
$compT\ E\ A\ ST\ e\ @\ [after\ E\ A\ ST\ e]$
$|\ compT\ E\ A\ ST\ (C\cdot_s F\{D\}) = []$
$|\ compT\ E\ A\ ST\ (e_1\cdot F\{D\} := e_2) =$
$(let\ ST_1 = ty\ \ E\ e_1\#ST;\ A_1 = A \sqcup \mathcal{A}\ e_1;\ A_2 = A_1 \sqcup \mathcal{A}\ e_2\ in$
$compT\ E\ A\ ST\ e_1\ @\ [after\ E\ A\ ST\ e_1]\ @$
$compT\ E\ A_1\ ST_1\ e_2\ @\ [after\ E\ A_1\ ST_1\ e_2]\ @$
$[ty_i'\ ST\ E\ A_2])$
$|\ compT\ E\ A\ ST\ (C\cdot_s F\{D\} := e_2) = compT\ E\ A\ ST\ e_2\ @\ [after\ E\ A\ ST\ e_2]\ @\ [ty_i'\ ST\ E\ (A \sqcup \mathcal{A}$
$e_2)]$
$|\ compT\ E\ A\ ST\ \{i{:}T;\ e\} = compT\ (E@[T])\ (A{\ominus}i)\ ST\ e$
$|\ compT\ E\ A\ ST\ (e_1;;e_2) =$
$(let\ A_1 = A \sqcup \mathcal{A}\ e_1\ in$
$compT\ E\ A\ ST\ e_1\ @\ [after\ E\ A\ ST\ e_1,\ ty_i'\ ST\ E\ A_1]\ @$
$compT\ E\ A_1\ ST\ e_2)$
$|\ compT\ E\ A\ ST\ (if\ (e)\ e_1\ else\ e_2) =$
$(let\ A_0 = A \sqcup \mathcal{A}\ e;\ \tau = ty_i'\ ST\ E\ A_0\ in$
$compT\ E\ A\ ST\ e\ @\ [after\ E\ A\ ST\ e,\ \tau]\ @$

$compT\ E\ A_0\ ST\ e_1\ @\ [after\ E\ A_0\ ST\ e_1,\ \tau]\ @$
$compT\ E\ A_0\ ST\ e_2)$
$|\ compT\ E\ A\ ST\ (while\ (e)\ c) =$
$(let\ A_0 = A \sqcup \mathcal{A}\ e;\ \ A_1 = A_0 \sqcup \mathcal{A}\ c;\ \tau = ty_i{}'\ ST\ E\ A_0\ in$
$compT\ E\ A\ ST\ e\ @\ [after\ E\ A\ ST\ e,\ \tau]\ @$
$compT\ E\ A_0\ ST\ c\ @\ [after\ E\ A_0\ ST\ c,\ ty_i{}'\ ST\ E\ A_1,\ ty_i{}'\ ST\ E\ A_0])$
$|\ compT\ E\ A\ ST\ (throw\ e) = compT\ E\ A\ ST\ e\ @\ [after\ E\ A\ ST\ e]$
$|\ compT\ E\ A\ ST\ (e{\bullet}M(es)) =$
$compT\ E\ A\ ST\ e\ @\ [after\ E\ A\ ST\ e]\ @$
$compTs\ E\ (A \sqcup \mathcal{A}\ e)\ (ty\quad E\ e\ \#\ ST)\ es$
$|\ compT\ E\ A\ ST\ (C{\bullet}_sM(es)) = compTs\ E\ A\ ST\ es$
$|\ compT\ E\ A\ ST\ (try\ e_1\ catch(C\ i)\ e_2) =$
$compT\ E\ A\ ST\ e_1\ @\ [after\ E\ A\ ST\ e_1]\ @$
$[ty_i{}'\ (Class\ C\#ST)\ E\ A,\ ty_i{}'\ ST\ (E@[Class\ C])\ (A \sqcup \lfloor\{i\}\rfloor)]\ @$
$compT\ (E@[Class\ C])\ (A \sqcup \lfloor\{i\}\rfloor)\ ST\ e_2$
$|\ compT\ E\ A\ ST\ (INIT\ C\ (Cs,b) \leftarrow e) = []$
$|\ compT\ E\ A\ ST\ (RI(C,e');Cs \leftarrow e) = []$
$|\ compTs\ E\ A\ ST\ [] = []$
$|\ compTs\ \ E\ A\ ST\ (e\#es) = compT\ E\ A\ ST\ e\ @\ [after\ E\ A\ ST\ e]\ @$
$compTs\ E\ (A \sqcup (\mathcal{A}\ e))\ (ty\ E\ e\ \#\ ST)\ es$

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-$Nil[simp]$: $P,E \vdash_1 e :: T \Longrightarrow compE_2\ e \neq []$
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 $\bigwedge E\ A\ ST.\ \neg sub\text{-}RIs\ es \Longrightarrow size(compTs\ E\ A\ ST\ es) = size(compEs_2\ es)$
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 $\bigwedge E\ A\ ST.\ P,E \vdash_1 es\ [::]\ Ts \Longrightarrow size(compTs\ E\ A\ ST\ es) = size(compEs_2\ es)$

lemma (**in** $TC1$) $[simp]$: $\bigwedge ST\ E.\ \lfloor\tau\rfloor \notin set\ (compT\ E\ None\ ST\ e)$
and $[simp]$: $\bigwedge ST\ E.\ \lfloor\tau\rfloor \notin set\ (compTs\ E\ None\ ST\ es)$

lemma (**in** $TC0$) $pair$-eq-$ty_i{}'$-$conv$:
$(\lfloor(ST,\ LT)\rfloor = ty_i{}'\ ST_0\ E\ A) =$
$(case\ A\ of\ None \Rightarrow False\ |\ Some\ A \Rightarrow (ST = ST_0 \wedge LT = ty_l\ E\ A))$

lemma (**in** $TC0$) $pair$-$conv$-$ty_i{}'$:
$\lfloor(ST,\ ty_l\ E\ A)\rfloor = ty_i{}'\ ST\ E\ \lfloor A\rfloor$

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
$\bigwedge E\ A\ ST_0.\ [\![\ \lfloor(ST,LT)\rfloor \in set(compTs\ E\ A\ ST_0\ es);\ \mathcal{B}s\ es\ (size\ E)\ ]\!]$
$\Longrightarrow P \vdash \lfloor(ST,LT)\rfloor \leq'\ ty_i{}'\ ST\ E\ A$

lemma $[iff]$: $OK\ None \in states\ P\ mxs\ mxl$
lemma (**in** $TC0$) $after$-in-$states$:
assumes wf: $wf\text{-}prog\ p\ P$ **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 $OK\ (after\ E\ A\ ST\ e) \in states\ P\ mxs\ mxl$

lemma (**in** *TC0*) *OK-ty$_i$′-in-statesI*[*simp*]:
$\llbracket\ set\ E \subseteq types\ P;\ set\ ST \subseteq types\ P;\ size\ ST \leq mxs\ \rrbracket$
$\Longrightarrow OK\ (ty_i{}'\ ST\ E\ A) \in states\ P\ mxs\ mxl$

lemma *is-class-type-aux*: $is\text{-}class\ P\ C \Longrightarrow is\text{-}type\ P\ (Class\ C)$
theorem (**in** *TC1*) *compT-states*:
assumes *wf*: *wf-prog p P*
shows $\bigwedge E\ T\ A\ ST.$
$\llbracket\ P,E \vdash_1 e :: T;\ set\ E \subseteq types\ P;\ set\ ST \subseteq types\ P;$
$size\ ST + max\text{-}stack\ e \leq mxs;\ size\ E + max\text{-}vars\ e \leq mxl\ \rrbracket$
$\Longrightarrow OK\ `\ set(compT\ E\ A\ ST\ e) \subseteq states\ P\ mxs\ mxl$
and $\bigwedge E\ Ts\ A\ ST.$
$\llbracket\ P,E \vdash_1 es[::]Ts;\ \ set\ E \subseteq types\ P;\ set\ ST \subseteq types\ P;$
$size\ ST + max\text{-}stacks\ es \leq mxs;\ size\ E + max\text{-}varss\ es \leq mxl\ \rrbracket$
$\Longrightarrow OK\ `\ set(compTs\ E\ A\ ST\ es) \subseteq states\ P\ mxs\ mxl$

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*]: $shift\ 0\ xt = xt$
lemma [*simp*]: $shift\ n\ [] = []$
lemma [*simp*]: $shift\ n\ (xt_1\ @\ xt_2) = shift\ n\ xt_1\ @\ shift\ n\ xt_2$
lemma [*simp*]: $shift\ m\ (shift\ n\ xt) = shift\ (m+n)\ xt$
lemma [*simp*]: $pcs\ (shift\ n\ xt) = \{pc+n|pc.\ pc \in pcs\ xt\}$

lemma *shift-compxE$_2$*:
shows $\bigwedge pc\ pc'\ d.\ shift\ pc\ (compxE_2\ e\ pc'\ d) = compxE_2\ e\ (pc' + pc)\ d$
and $\bigwedge pc\ pc'\ d.\ shift\ pc\ (compxEs_2\ es\ pc'\ d) = compxEs_2\ es\ (pc' + pc)\ d$

lemma *compxE$_2$-size-convs*[*simp*]:
shows $n \neq 0 \Longrightarrow compxE_2\ e\ n\ d = shift\ n\ (compxE_2\ e\ 0\ d)$
and $n \neq 0 \Longrightarrow compxEs_2\ es\ n\ d = shift\ n\ (compxEs_2\ es\ 0\ d)$
locale $TC2 = TC1 +$
fixes $T_r :: ty$ **and** $mxs :: pc$
begin

definition
$wt\text{-}instrs :: instr\ list \Rightarrow ex\text{-}table \Rightarrow ty_i{}'\ list \Rightarrow bool$
(‹(⊢ -, - /[::]/ -)› [*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*]: $\tau s \neq [] \Longrightarrow \vdash [],[]\ [::]\ \tau s$
lemma [*simp*]: $eff\ i\ P\ pc\ et\ None = []$

lemma *wt-instr-appR*:
$[\![$ $P,T,m,mpc,xt \vdash is!pc,pc :: \tau s$;
$pc < size\ is$; $size\ is < size\ \tau s$; $mpc \leq size\ \tau s$; $mpc \leq mpc'$ $]\!]$
$\Longrightarrow P,T,m,mpc',xt \vdash is!pc,pc :: \tau s@\tau s'$

lemma *relevant-entries-shift* [*simp*]:
$relevant\text{-}entries\ P\ i\ (pc+n)\ (shift\ n\ xt) = shift\ n\ (relevant\text{-}entries\ P\ i\ pc\ xt)$

lemma [*simp*]:
$xcpt\text{-}eff\ i\ P\ (pc+n)\ \tau\ (shift\ n\ xt) =$
$map\ (\lambda(pc,\tau).\ (pc + n,\ \tau))\ (xcpt\text{-}eff\ i\ P\ pc\ \tau\ xt)$

lemma [*simp*]:
$app_i\ (i,\ P,\ pc,\ m,\ T,\ \tau) \Longrightarrow$
$eff\ i\ P\ (pc+n)\ (shift\ n\ xt)\ (Some\ \tau) =$
$map\ (\lambda(pc,\tau).\ (pc+n,\tau))\ (eff\ i\ P\ pc\ xt\ (Some\ \tau))$

lemma [*simp*]:
$xcpt\text{-}app\ i\ P\ (pc+n)\ mxs\ (shift\ n\ xt)\ \tau = xcpt\text{-}app\ i\ P\ pc\ mxs\ xt\ \tau$

lemma *wt-instr-appL*:
assumes $P,T,m,mpc,xt \vdash i,pc :: \tau s$ **and** $pc < size\ \tau s$ **and** $mpc \leq size\ \tau s$
shows $P,T,m,mpc + size\ \tau s',shift\ (size\ \tau s')\ xt \vdash i,pc+size\ \tau s' :: \tau s'@\tau s$

lemma *wt-instr-Cons*:
assumes *wti*: $P,T,m,mpc - 1,[] \vdash i,pc - 1 :: \tau s$
and *pcl*: $0 < pc$ **and** *mpcl*: $0 < mpc$
and *pcu*: $pc < size\ \tau s + 1$ **and** *mpcu*: $mpc \leq size\ \tau s + 1$
shows $P,T,m,mpc,[] \vdash i,pc :: \tau\#\tau s$

lemma *wt-instr-append*:
assumes *wti*: $P,T,m,mpc - size\ \tau s',[] \vdash i,pc - size\ \tau s' :: \tau s$
and *pcl*: $size\ \tau s' \leq pc$ **and** *mpcl*: $size\ \tau s' \leq mpc$
and *pcu*: $pc < size\ \tau s + size\ \tau s'$ **and** *mpcu*: $mpc \leq size\ \tau s + size\ \tau s'$
shows $P,T,m,mpc,[] \vdash i,pc :: \tau s'@\tau s$

lemma *xcpt-app-pcs*:
$pc \notin pcs\ xt \Longrightarrow xcpt\text{-}app\ i\ P\ pc\ mxs\ xt\ \tau$

lemma *xcpt-eff-pcs*:
$pc \notin pcs\ xt \Longrightarrow xcpt\text{-}eff\ i\ P\ pc\ \tau\ xt = []$

lemma *pcs-shift*:
$pc < n \Longrightarrow pc \notin pcs\ (shift\ n\ xt)$

lemma *wt-instr-appRx*:
$[\![$ $P,T,m,mpc,xt \vdash is!pc,pc :: \tau s$; $pc < size\ is$; $size\ is < size\ \tau s$; $mpc \leq size\ \tau s$ $]\!]$
$\Longrightarrow P,T,m,mpc,xt\ @\ shift\ (size\ is)\ xt' \vdash is!pc,pc :: \tau s$

lemma *wt-instr-appLx*:
$[\![$ $P,T,m,mpc,xt \vdash i,pc :: \tau s$; $pc \notin pcs\ xt'$ $]\!]$
$\Longrightarrow P,T,m,mpc,xt'@xt \vdash i,pc :: \tau s$

lemma (**in** *TC2*) *wt-instrs-extR*:

$\vdash is,xt\ [::]\ \tau s \Longrightarrow \vdash is,xt\ [::]\ \tau s\ @\ \tau s'$

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*: $size\ \tau s_1 = size\ is_1$
shows $\vdash is_1 @ is_2,\ xt_1\ @\ shift\ (size\ is_1)\ xt_2\ [::]\ \tau s_1 @ \tau s_2$
corollary (**in** *TC2*) *wt-instrs-ext2*:
$[\![\vdash is_2,xt_2\ [::]\ \tau s_2;\ \vdash is_1,xt_1\ [::]\ \tau s_1 @ \tau s_2;\ size\ \tau s_1 = size\ is_1\ ]\!]$
$\Longrightarrow \vdash is_1 @ is_2,\ xt_1\ @\ shift\ (size\ is_1)\ xt_2\ [::]\ \tau s_1 @ \tau s_2$

corollary (**in** *TC2*) *wt-instrs-ext-prefix* [*trans*]:
$[\![\vdash is_1,xt_1\ [::]\ \tau s_1 @ \tau s_2;\ \vdash is_2,xt_2\ [::]\ \tau s_3;$
$size\ \tau s_1 = size\ is_1;\ prefix\ \tau s_3\ \tau s_2\ ]\!]$
$\Longrightarrow \vdash is_1 @ is_2,\ xt_1\ @\ shift\ (size\ is_1)\ xt_2\ [::]\ \tau s_1 @ \tau s_2$

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: $size\ \tau s_1 = size\ is_1$
shows $\vdash is_1 @ is_2,\ xt_1 @ shift\ (size\ is_1)\ xt_2\ [::]\ \tau s_1 @ \tau \# \tau s_2$

corollary (**in** *TC2*) *wt-instrs-app-last*[*trans*]:
assumes $\vdash is_2,xt_2\ [::]\ \tau \# \tau s_2$ $\vdash is_1,xt_1\ [::]\ \tau s_1$
$last\ \tau s_1 = \tau$ $size\ \tau s_1 = size\ is_1 + 1$
shows $\vdash is_1 @ is_2,\ xt_1 @ shift\ (size\ is_1)\ xt_2\ [::]\ \tau s_1 @ \tau s_2$

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*: $mpc = size\ \tau s$ **and** *is-τs*: $size\ is + 1 < size\ \tau s$
shows $\vdash is @ [i],xt\ [::]\ \tau s$

corollary (**in** *TC2*) *wt-instrs-app2*:
$[\![\vdash is_2,xt_2\ [::]\ \tau' \# \tau s_2;\ \vdash is_1,xt_1\ [::]\ \tau \# \tau s_1 @ [\tau'];$
$xt' = xt_1\ @\ shift\ (size\ is_1)\ xt_2;\ size\ \tau s_1 + 1 = size\ is_1\ ]\!]$
$\Longrightarrow \vdash is_1 @ is_2,xt'\ [::]\ \tau \# \tau s_1 @ \tau' \# \tau s_2$

corollary (**in** *TC2*) *wt-instrs-app2-simp*[*trans*,*simp*]:
$[\![\vdash is_2,xt_2\ [::]\ \tau' \# \tau s_2;\ \vdash is_1,xt_1\ [::]\ \tau \# \tau s_1 @ [\tau'];\ size\ \tau s_1 + 1 = size\ is_1\ ]\!]$
$\Longrightarrow \vdash is_1 @ is_2,\ xt_1 @ shift\ (size\ is_1)\ xt_2\ [::]\ \tau \# \tau s_1 @ \tau' \# \tau s_2$

corollary (**in** *TC2*) *wt-instrs-Cons*[*simp*]:
$[\![\tau s \neq [];\ \vdash [i],[]\ [::]\ [\tau,\tau'];\ \vdash is,xt\ [::]\ \tau' \# \tau s\ ]\!]$
$\Longrightarrow \vdash i \# is,shift\ 1\ xt\ [::]\ \tau \# \tau' \# \tau s$
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.