A Machine-Checked Model for a Java-like Language, Virtual Machine and Compiler

Gerwin Klein Tobias Nipkow

14th April 2004

Contents

Chapter 1

Preface

This document contains the automatically generated listings of the Isabelle sources for the theories defining and analysing Jinja (a Java-like programming language), the Jinja Virtual Machine, and the compiler. To shorten the document, all proofs have been hidden. For a detailed exposition of these theories see the paper by Klein and Nipkow [1].

1.1 Theory Dependencies

Figure 1.1 shows the dependencies between the Isabelle theories in the following sections.

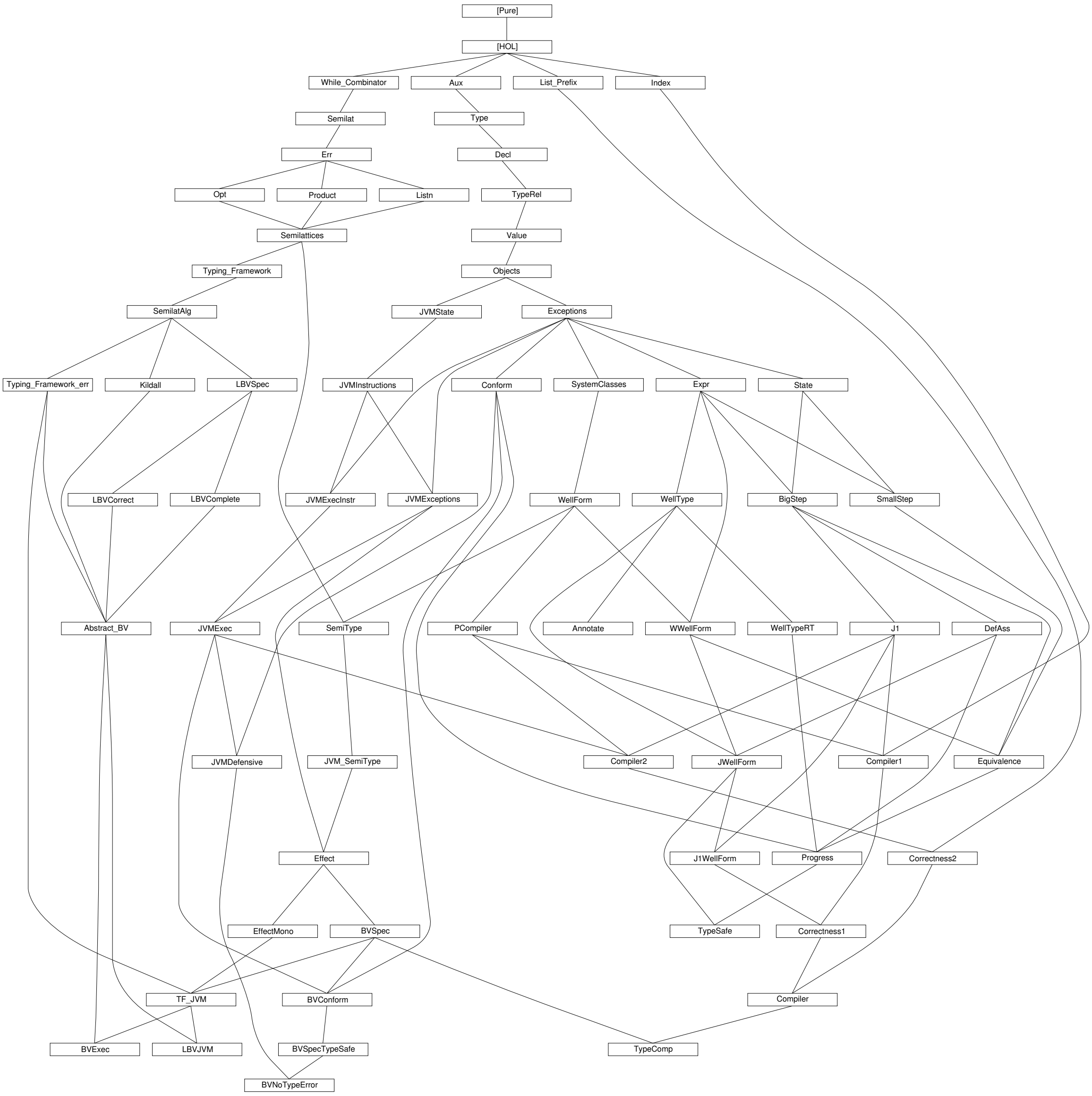

Figure 1.1: Theory Dependency Graph

Chapter 2

Jinja Source Language

2.1 Auxiliary Definitions

theory *Aux* = *Main*:

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*]:
$(Suc(n + max\ i\ j) \le m) = (Suc(n + i) \le m \wedge Suc(n + j) \le m)$

translations $\lfloor x \rfloor$ == *Some x*

2.1.1 *distinct-fst*

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

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

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

lemma *map-of-SomeI*:
$[\![$ *distinct-fst kxs*; $(k,x) \in set\ kxs$ $]\!] \Longrightarrow$ *map-of kxs k* = *Some x*

2.1.2 Using *list-all2* for relations

constdefs
fun-of :: $('a \times 'b)$ *set* $\Rightarrow$ $'a \Rightarrow 'b \Rightarrow$ *bool*
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-all2 (*fun-of S*) $(x\#xs)$ *ys* =
$(\exists z\ zs.\ ys = z\#zs \wedge (x,z) \in S \wedge$ *list-all2* (*fun-of S*) *xs zs*)

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

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

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

lemma *rel-list-all2-trans*:

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

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

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

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

end

2.2 Jinja types

theory *Type* = *Aux*:

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

constdefs
Object :: *cname*
Object ≡ ''*Object*''
this :: *vname*
this ≡ ''*this*''

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

constdefs
is-refT :: *ty* ⇒ *bool*
$\textit{is-refT}\ T \ \equiv\ T = NT \lor (\exists C.\ T = \textit{Class}\ C)$

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

2.3 Class Declarations and Programs

theory *Decl* = *Type*:

types
fdecl = *vname* × *ty* — field declaration

′*m mdecl* = *mname* × *ty list* × *ty* × ′*m* — method = name, arg. types, return type, body

′*m class* = *cname* × *fdecl list* × ′*m mdecl list* — class = superclass, fields, methods

′*m cdecl* = *cname* × ′*m class* — class declaration

′*m prog* = ′*m cdecl list* — program

constdefs
class :: ′*m prog* ⇒ *cname* ⇀ ′*m class*
class ≡ *map-of*

is-class :: ′*m prog* ⇒ *cname* ⇒ *bool*
is-class P C ≡ *class P C* ≠ *None*

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

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

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

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

end

2.4 Relations between Jinja Types

theory *TypeRel* = *Decl*:

2.4.1 The subclass relations

consts
$subcls1$ :: $'m\ prog \Rightarrow (cname \times cname)\ set$ — subclass

translations
$P \vdash C \prec^1 D$ == $(C,D) \in subcls1\ P$
$P \vdash C \preceq^* D$ == $(C,D) \in (subcls1\ P)^*$

inductive *subcls1 P*
intros *subcls1I*: $[\![class\ P\ C = Some\ (D,rest);\ C \neq Object]\!] \Longrightarrow P \vdash C \prec^1 D$

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 *finite-subcls1*: $finite\ (subcls1\ P)$

2.4.2 The subtype relations

consts
$widen$ :: $'m\ prog \Rightarrow (ty \times ty)\ set$ — widening

translations
$P \vdash S \le T$ == $(S,T) \in widen\ P$
$P \vdash Ts\ [\le]\ Ts'$ == $list\text{-}all2\ (fun\text{-}of\ (widen\ P))\ Ts\ Ts'$

inductive *widen P*
intros
widen-refl[*iff*]: $P \vdash T \le T$
widen-subcls: $P \vdash C \preceq^* D \Longrightarrow P \vdash Class\ C \le Class\ D$
widen-null[*iff*]: $P \vdash NT \le Class\ C$

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

lemma *Class-widen*: $P \vdash Class\ C \le T \Longrightarrow \exists D.\ T = Class\ D$
lemma [*iff*]: $(P \vdash T \le NT) = (T = NT)$
lemma *Class-widen-Class* [*iff*]: $(P \vdash Class\ C \le Class\ D) = (P \vdash C \preceq^* D)$
lemma *widen-Class*: $(P \vdash T \le Class\ C) = (T = NT \vee (\exists D.\ T = Class\ D \wedge P \vdash D \preceq^* C))$

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

2.4.3 Method lookup

consts

$Methods :: {}'m\ prog \Rightarrow (cname \times (mname \rightharpoonup (ty\ list \times ty \times {}'m) \times cname))set$

translations $P \vdash C\ sees\text{-}methods\ Mm\ \ ==\ \ (C,Mm) \in Methods\ P$

inductive *Methods P*
intros
sees-methods-Object:
$[\![\ class\ P\ Object = Some(D,fs,ms);\ Mm = option\text{-}map\ (\lambda m.\ (m,Object)) \circ map\text{-}of\ ms\ ]\!]$
$\Longrightarrow P \vdash Object\ sees\text{-}methods\ Mm$
sees-methods-rec:
$[\![\ class\ P\ C = Some(D,fs,ms);\ C \neq Object;\ P \vdash D\ sees\text{-}methods\ Mm;$
$Mm' = Mm\ {+}{+}\ (option\text{-}map\ (\lambda m.\ (m,C)) \circ map\text{-}of\ ms)\ ]\!]$
$\Longrightarrow P \vdash C\ sees\text{-}methods\ Mm'$

lemma *sees-methods-fun*:
assumes *1*: $P \vdash C\ sees\text{-}methods\ Mm$
shows $\bigwedge Mm'.\ P \vdash C\ sees\text{-}methods\ Mm' \Longrightarrow Mm' = Mm$

lemma *visible-methods-exist*:
$P \vdash C\ sees\text{-}methods\ Mm \Longrightarrow Mm\ M = Some(m,D) \Longrightarrow$
$(\exists D'\ fs\ ms.\ class\ P\ D = Some(D',fs,ms) \wedge map\text{-}of\ ms\ M = Some\ m)$

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

lemma *sees-methods-idemp*:
assumes *Cmethods*: $P \vdash C\ sees\text{-}methods\ Mm$
shows $\bigwedge m\ D.\ Mm\ M = Some(m,D) \Longrightarrow$
$\exists Mm'.\ (P \vdash D\ sees\text{-}methods\ Mm') \wedge Mm'\ M = Some(m,D)$

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

constdefs
$Method :: {}'m\ prog \Rightarrow cname \Rightarrow mname \Rightarrow ty\ list \Rightarrow ty \Rightarrow {}'m \Rightarrow cname \Rightarrow bool$
$(\text{-} \vdash \text{-}\ sees\ \text{-}:\ \text{-}{\rightarrow}\text{-} = \text{-}\ in\ \text{-}\ [51,51,51,51,51,51,51]\ 50)$
$P \vdash C\ sees\ M:\ Ts{\rightarrow}T = m\ in\ D\ \equiv$
$\exists Mm.\ P \vdash C\ sees\text{-}methods\ Mm \wedge Mm\ M = Some((Ts,T,m),D)$

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

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

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

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

lemma *sees-method-decl-mono*:
$[\![\ P \vdash C' \preceq^* C;\ P \vdash C$ *sees* $M{:}Ts{\rightarrow}T = m$ *in* $D;$
$P \vdash C'$ *sees* $M{:}Ts'{\rightarrow}T' = m'$ *in* $D'\ ]\!] \Longrightarrow P \vdash D' \preceq^* D$

lemma *sees-method-is-class*:
$[\![\ P \vdash C$ *sees* $M{:}Ts{\rightarrow}T = m$ *in* $D\ ]\!] \Longrightarrow$ *is-class* $P\ C$

2.4.4 Field lookup

consts
Fields :: $'m$ *prog* $\Rightarrow$ (*cname* $\times$ ((*vname* $\times$ *cname*) $\times$ *ty*) *list*)*set*

translations
$P \vdash C$ *has-fields* $FDTs$ == $(C,FDTs) \in$ *Fields* P

inductive *Fields P*
intros
has-fields-rec:
$[\![$ *class* $P\ C =$ *Some*(D,fs,ms); $C \neq$ *Object*; $P \vdash D$ *has-fields* $FDTs$;
$FDTs' =$ *map* $(\lambda(F,T).\ ((F,C),T))$ *fs* @ $FDTs\ ]\!]$
$\Longrightarrow P \vdash C$ *has-fields* $FDTs'$
has-fields-Object:
$[\![$ *class* P *Object* $=$ *Some*(D,fs,ms); $FDTs =$ *map* $(\lambda(F,T).\ ((F,Object),T))$ *fs* $]\!]$
$\Longrightarrow P \vdash$ *Object has-fields* $FDTs$

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); $(F,T) \in$ *set fs* $]\!]$
$\Longrightarrow ((F,D),T) \in$ *set* $FDTs$

lemma *has-fields-decl-above*:
assumes *fields*: $P \vdash C$ *has-fields* $FDTs$
shows $((F,D),T) \in$ *set* $FDTs \Longrightarrow P \vdash C \preceq^* D$

lemma *subcls-notin-has-fields*:
assumes *fields*: $P \vdash C$ *has-fields* $FDTs$
shows $((F,D),T) \in$ *set* $FDTs \Longrightarrow (D,C) \notin$ (*subcls1* $P)^+$

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*@$FDTs$ $\wedge$ *dom*(*map-of pre*) $\cap$ *dom*(*map-of* $FDTs$) $= \{\}$

constdefs
has-field :: $'m$ *prog* $\Rightarrow$ *cname* $\Rightarrow$ *vname* $\Rightarrow$ *ty* $\Rightarrow$ *cname* $\Rightarrow$ *bool*
(- $\vdash$ - *has* -:- *in* - [51,51,51,51,51] 50)

$P \vdash C\ \mathit{has}\ F{:}T\ \mathit{in}\ D \ \equiv$
$\exists\, \mathit{FDTs}.\ P \vdash C\ \mathit{has\text{-}fields}\ \mathit{FDTs} \wedge \mathit{map\text{-}of}\ \mathit{FDTs}\ (F,D) = \mathit{Some}\ T$

lemma *has-field-mono*:
$[\![\ P \vdash C\ \mathit{has}\ F{:}T\ \mathit{in}\ D;\ P \vdash C' \preceq^* C\ ]\!] \Longrightarrow P \vdash C'\ \mathit{has}\ F{:}T\ \mathit{in}\ D$

constdefs
$\mathit{sees\text{-}field} :: {}'m\ \mathit{prog} \Rightarrow \mathit{cname} \Rightarrow \mathit{vname} \Rightarrow \mathit{ty} \Rightarrow \mathit{cname} \Rightarrow \mathit{bool}$
$(\text{-} \vdash \text{-}\ \mathit{sees}\ \text{-:-}\ \mathit{in}\ \text{-}\ [51,51,51,51,51]\ 50)$
$P \vdash C\ \mathit{sees}\ F{:}T\ \mathit{in}\ D \ \equiv$
$\exists\, \mathit{FDTs}.\ P \vdash C\ \mathit{has\text{-}fields}\ \mathit{FDTs} \wedge$
$\mathit{map\text{-}of}\ (\mathit{map}\ (\lambda((F,D),T).\ (F,(D,T)))\ \mathit{FDTs})\ F = \mathit{Some}(D,T)$

lemma *map-of-remap-SomeD*:
$\mathit{map\text{-}of}\ (\mathit{map}\ (\lambda((k,k'),x).\ (k,(k',x)))\ t)\ k = \mathit{Some}\ (k',x) \Longrightarrow \mathit{map\text{-}of}\ t\ (k,\ k') = \mathit{Some}\ x$

lemma *has-visible-field*:
$P \vdash C\ \mathit{sees}\ F{:}T\ \mathit{in}\ D \Longrightarrow P \vdash C\ \mathit{has}\ F{:}T\ \mathit{in}\ D$

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

lemma *sees-field-decl-above*:
$P \vdash C\ \mathit{sees}\ F{:}T\ \mathit{in}\ D \Longrightarrow P \vdash C \preceq^* D$

lemma *sees-field-idemp*:
$P \vdash C\ \mathit{sees}\ F{:}T\ \mathit{in}\ D \Longrightarrow P \vdash D\ \mathit{sees}\ F{:}T\ \mathit{in}\ D$

2.4.5 Functional lookup

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

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

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

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

2.5 Jinja Values

theory *Value* = *TypeRel*:

types *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

consts
  the-Intg :: *val* ⇒ *int*
  the-Addr :: *val* ⇒ *addr*

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

primrec
  the-Addr (*Addr a*) = *a*

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

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

end

2.6 Objects and the Heap

theory *Objects* = *TypeRel* + *Value*:

2.6.1 Objects

types
fields = *vname* × *cname* ⇀ *val* — field name, defining class, value
obj = *cname* × *fields* — class instance with class name and fields

constdefs
obj-ty :: *obj* ⇒ *ty*
obj-ty obj ≡ *Class* (*fst obj*)

init-fields :: ((*vname* × *cname*) × *ty*) *list* ⇒ *fields*
init-fields ≡ *map-of* ∘ *map* (λ(*F*,*T*). (*F*,*default-val T*))

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

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

2.6.2 Heap

types *heap* = *addr* ⇀ *obj*

syntax
cname-of :: *heap* ⇒ *addr* ⇒ *cname*
translations
cname-of hp a == *fst* (*the* (*hp a*))

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

cast-ok :: *'m prog* ⇒ *cname* ⇒ *heap* ⇒ *val* ⇒ *bool*
cast-ok P C h v ≡ *v* = *Null* ∨ *P* ⊢ *cname-of h* (*the-Addr v*) $\preceq^*$ *C*

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

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

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

lemma [*simp*]: ($typeof_h$ *v* = *Some Boolean*) = (∃ *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* $a \wedge h\ a$ = *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 *empty* because they do not contain addresses:

consts
typeof :: *val* $\Rightarrow$ *ty option*

translations
typeof v == *typeof-h 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-type* P T

2.6.3 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

end

2.7 Exceptions

theory *Exceptions* = *Objects*:

constdefs
NullPointer :: *cname*
NullPointer ≡ *''NullPointer''*

ClassCast :: *cname*
ClassCast ≡ *''ClassCast''*

OutOfMemory :: *cname*
OutOfMemory ≡ *''OutOfMemory''*

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

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

start-heap :: *'c prog* ⇒ *heap*
start-heap G ≡ *empty* (*addr-of-sys-xcpt NullPointer* ↦ *blank G NullPointer*)
(*addr-of-sys-xcpt ClassCast* ↦ *blank G ClassCast*)
(*addr-of-sys-xcpt OutOfMemory* ↦ *blank G OutOfMemory*)

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

2.7.1 System exceptions

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

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

2.7.2 *preallocated*

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

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

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

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

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

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

lemma *typeof-NullPointer* [*simp*]:
$preallocated\ h \Longrightarrow typeof_h\ (Addr(addr\text{-}of\text{-}sys\text{-}xcpt\ NullPointer)) = Some(Class\ NullPointer)$

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

lemma *preallocated-start*:
$preallocated\ (start\text{-}heap\ P)$

end

2.8 Expressions

theory *Expr* = *Exceptions*:

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
| *FAss ('a exp) vname cname ('a exp)* (-•-{-} := - [*10*,*90*,*99*,*90*]*90*) — field assignment
| *Call ('a exp) mname ('a exp list)* (-•-'(-') [*90*,*99*,*0*] *90*) — 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*)

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

The semantics of binary operators:

consts
binop :: *bop* × *val* × *val* ⇒ *val option*
recdef *binop* {}
$binop(Eq,v_1,v_2) = Some(Bool\ (v_1 = v_2))$
$binop(Add,Intg\ i_1,Intg\ i_2) = Some(Intg(i_1+i_2))$
$binop(bop,v_1,v_2) = None$

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

2.8.1 Syntactic sugar

syntax
InitBlock:: *vname* ⇒ *ty* ⇒ *'a exp* ⇒ *'a exp* ⇒ *'a exp* ((*1*'{-:- := -;/ -}))
translations
InitBlock V T e1 e2 => {*V*:*T*; *V* := *e1*;; *e2*}

syntax
unit :: *'a exp*
null :: *'a exp*
addr :: *addr* ⇒ *'a exp*
true :: *'a exp*
false :: *'a exp*
translations
unit == *Val Unit*

$null == Val\ Null$
$addr\ a == Val(Addr\ a)$
$true == Val(Bool\ True)$
$false == Val(Bool\ False)$

syntax
$Throw :: addr \Rightarrow 'a\ exp$
$THROW :: cname \Rightarrow 'a\ exp$
translations
$Throw\ a == throw(Val(Addr\ a))$
$THROW\ xc == Throw(addr\text{-}of\text{-}sys\text{-}xcpt\ xc)$

2.8.2 Free Variables

consts
$fv :: expr \Rightarrow vname\ set$
$fvs :: expr\ list \Rightarrow vname\ set$
primrec
$fv(new\ C) = \{\}$
$fv(Cast\ C\ e) = fv\ e$
$fv(Val\ v) = \{\}$
$fv(e_1 \ll bop \gg e_2) = fv\ e_1 \cup fv\ e_2$
$fv(Var\ V) = \{V\}$
$fv(LAss\ V\ e) = \{V\} \cup fv\ e$
$fv(e{\cdot}F\{D\}) = fv\ e$
$fv(e_1{\cdot}F\{D\}{:=}e_2) = fv\ e_1 \cup fv\ e_2$
$fv(e{\cdot}M(es)) = fv\ e \cup fvs\ es$
$fv(\{V{:}T;\ e\}) = fv\ e - \{V\}$
$fv(e_1;;e_2) = fv\ e_1 \cup fv\ e_2$
$fv(if\ (b)\ e_1\ else\ e_2) = fv\ b \cup fv\ e_1 \cup fv\ e_2$
$fv(while\ (b)\ e) = fv\ b \cup fv\ e$
$fv(throw\ e) = fv\ e$
$fv(try\ e_1\ catch(C\ V)\ e_2) = fv\ e_1 \cup (fv\ e_2 - \{V\})$

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

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

end

2.9 Program State

theory *State* = *Exceptions*:

types
  $locals = vname \rightharpoonup val$ — local vars, incl. params and "this"
  $state \;= heap \times locals$

constdefs
  $hp :: state \Rightarrow heap$
  $hp \;\equiv\; fst$
  $lcl :: state \Rightarrow locals$
  $lcl \;\equiv\; snd$
end

2.10 Big Step Semantics

theory *BigStep* = *Expr* + *State*:

consts
$eval$ :: $J\text{-}prog \Rightarrow ((expr \times state) \times (expr \times state))\ set$
$evals$ :: $J\text{-}prog \Rightarrow ((expr\ list \times state) \times (expr\ list \times state))\ set$

translations
$P \vdash \langle e,s\rangle \Rightarrow \langle e',s'\rangle$ == $((e,s),\ e',s') \in eval\ P$
$P \vdash \langle es,s\rangle\ [\Rightarrow]\ \langle es',s'\rangle$ == $((es,s),\ es',s') \in evals\ P$

inductive *eval P evals P*
intros

New:
$[\![\ new\text{-}Addr\ h = Some\ a;\ P \vdash C\ has\text{-}fields\ FDTs;\ h' = h(a \mapsto (C,init\text{-}fields\ FDTs))\ ]\!]$
$\Longrightarrow P \vdash \langle new\ C,(h,l)\rangle \Rightarrow \langle addr\ a,(h',l)\rangle$

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

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

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

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

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

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

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

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

FAss:
$[\![\ P \vdash \langle e_1,s_0\rangle \Rightarrow \langle addr\ a,s_1\rangle;\ P \vdash \langle e_2,s_1\rangle \Rightarrow \langle Val\ v,(h_2,l_2)\rangle;$
$h_2\ a = Some(C,fs);\ fs' = fs((F,D)\mapsto v);\ h_2' = h_2(a\mapsto(C,fs'))\ ]\!]$
$\Longrightarrow P \vdash \langle e_1{\cdot}F\{D\}{:=}e_2,s_0\rangle \Rightarrow \langle unit,(h_2',l_2)\rangle$

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

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

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

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

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

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

Call:

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

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

Seq:
$[\![P \vdash \langle e_0,s_0\rangle \Rightarrow \langle Val\ v,s_1\rangle;\ P \vdash \langle e_1,s_1\rangle \Rightarrow \langle e_2,s_2\rangle\ ]\!]$
$\Longrightarrow P \vdash \langle e_0;;e_1,s_0\rangle \Rightarrow \langle e_2,s_2\rangle$

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

2.10.1 Final expressions

constdefs
$final :: 'a\ exp \Rightarrow bool$
$final\ e\ \equiv\ (\exists v.\ e = Val\ v) \lor (\exists a.\ e = Throw\ a)$
$finals :: 'a\ exp\ list \Rightarrow bool$
$finals\ es\ \equiv\ (\exists vs.\ es = map\ Val\ vs) \lor (\exists vs\ a\ es'.\ es = map\ Val\ vs\ @\ Throw\ a\ \#\ es')$

lemma [*simp*]: $final(Val\ v)$
lemma [*simp*]: $final(throw\ e) = (\exists a.\ e = addr\ a)$
lemma *finalE*: $[\![final\ e;\ \ \bigwedge v.\ e = Val\ v \Longrightarrow R;\ \ \bigwedge a.\ e = Throw\ a \Longrightarrow R]\!] \Longrightarrow R$
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*]: $finals\ (throw\ e\ \#\ es) = (\exists a.\ e = addr\ a)$
lemma *not-finals-ConsI*: $\neg\ final\ e \Longrightarrow \neg\ finals(e\#es)$

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

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

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

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

end

2.11 Small Step Semantics

theory *SmallStep* = *Expr* + *State*:

consts *blocks* :: *vname list* ∗ *ty list* ∗ *val list* ∗ *expr* ⇒ *expr*
recdef *blocks measure*(λ(*Vs*,*Ts*,*vs*,*e*). *size Vs*)
blocks(*V*#*Vs*, *T*#*Ts*, *v*#*vs*, *e*) = {*V*:*T* := *Val v*; *blocks*(*Vs*,*Ts*,*vs*,*e*)}
blocks([],[],[],*e*) = *e*

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

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

consts
red :: *J-prog* ⇒ ((*expr* × *state*) × (*expr* × *state*)) *set*
reds :: *J-prog* ⇒ ((*expr list* × *state*) × (*expr list* × *state*)) *set*

translations
P ⊢ ⟨*e*,*s*⟩ → ⟨*e′*,*s′*⟩ == ((*e*,*s*), *e′*,*s′*) ∈ *red P*
P ⊢ ⟨*es*,*s*⟩ [→] ⟨*es′*,*s′*⟩ == ((*es*,*s*), *es′*,*s′*) ∈ *reds P*

inductive *red P reds P*
intros

RedNew:
⟦ *new-Addr h* = *Some a*; *P* ⊢ *C has-fields FDTs*; *h′* = *h*(*a*↦(*C*,*init-fields FDTs*)) ⟧
⟹ *P* ⊢ ⟨*new C*, (*h*,*l*)⟩ → ⟨*addr a*, (*h′*,*l*)⟩

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

CastRed:
P ⊢ ⟨*e*,*s*⟩ → ⟨*e′*,*s′*⟩ ⟹
P ⊢ ⟨*Cast C e*, *s*⟩ → ⟨*Cast C e′*, *s′*⟩

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

RedCast:
⟦ *hp s a* = *Some*(*D*,*fs*); *P* ⊢ *D* $\preceq^*$ *C* ⟧
⟹ *P* ⊢ ⟨*Cast C* (*addr a*), *s*⟩ → ⟨*addr a*, *s*⟩

RedCastFail:
⟦ *hp s a* = *Some*(*D*,*fs*); ¬ *P* ⊢ *D* $\preceq^*$ *C* ⟧
⟹ *P* ⊢ ⟨*Cast C* (*addr a*), *s*⟩ → ⟨*THROW ClassCast*, *s*⟩

BinOpRed1:
P ⊢ ⟨*e*,*s*⟩ → ⟨*e′*,*s′*⟩ ⟹
P ⊢ ⟨*e* «*bop*» e_2, *s*⟩ → ⟨*e′* «*bop*» e_2, *s′*⟩

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

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

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

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

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

FAccRed:
$P \vdash \langle e,s\rangle \to \langle e',s'\rangle \Longrightarrow$
$P \vdash \langle e{\bullet}F\{D\},\ s\rangle \to \langle e'{\bullet}F\{D\},\ s'\rangle$

RedFAcc:
$[\![\ hp\ s\ a = Some(C,fs);\ fs(F,D) = Some\ v\ ]\!]$
$\Longrightarrow P \vdash \langle (addr\ a){\bullet}F\{D\},\ s\rangle \to \langle Val\ v,s\rangle$

RedFAccNull:
$P \vdash \langle null{\bullet}F\{D\},\ s\rangle \to \langle THROW\ NullPointer,\ s\rangle$

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

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

RedFAss:
$h\ a = Some(C,fs) \Longrightarrow$
$P \vdash \langle (addr\ a){\bullet}F\{D\}{:=}(Val\ v),\ (h,l)\rangle \to \langle unit,\ (h(a \mapsto (C,fs((F,D) \mapsto v))),l)\rangle$

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

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

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

RedCall:
$\llbracket$ *hp s a = Some(C,fs)*; $P \vdash C$ *sees* $M{:}Ts{\to}T$ *= (pns,body) in D*; *size vs = size pns*; *size Ts = size pns* $\rrbracket$
$\Longrightarrow P \vdash \langle(addr\ a){\cdot}M(map\ Val\ vs),\ s\rangle \to \langle blocks(this\#pns,\ Class\ D\#Ts,\ Addr\ a\#vs,\ body),\ s\rangle$

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

BlockRedNone:
$\llbracket\ P \vdash \langle e,\ (h,l(V{:=}None))\rangle \to \langle e',\ (h',l')\rangle;\ l'\ V = None;\ \neg\ assigned\ V\ e\ \rrbracket$
$\Longrightarrow P \vdash \langle\{V{:}T;\ e\},\ (h,l)\rangle \to \langle\{V{:}T;\ e'\},\ (h',l'(V := l\ V))\rangle$

BlockRedSome:
$\llbracket\ P \vdash \langle e,\ (h,l(V{:=}None))\rangle \to \langle e',\ (h',l')\rangle;\ l'\ V = Some\ v;\neg\ assigned\ V\ e\ \rrbracket$
$\Longrightarrow P \vdash \langle\{V{:}T;\ e\},\ (h,l)\rangle \to \langle\{V{:}T := Val\ v;\ e'\},\ (h',l'(V := l\ V))\rangle$

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

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

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

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

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

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

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

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

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

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

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

TryRed:

$P \vdash \langle e,s\rangle \to \langle e',s'\rangle \Longrightarrow$
$P \vdash \langle \mathit{try}\ e\ \mathit{catch}(C\ V)\ e_2,\ s\rangle \to \langle \mathit{try}\ e'\ \mathit{catch}(C\ V)\ e_2,\ s'\rangle$

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

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

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

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

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

— Exception propagation

CastThrow: $P \vdash \langle \mathit{Cast}\ C\ (\mathit{throw}\ e),\ s\rangle \to \langle \mathit{throw}\ e,\ s\rangle$
BinOpThrow1: $P \vdash \langle (\mathit{throw}\ e)\ «\mathit{bop}»\ e_2,\ s\rangle \to \langle \mathit{throw}\ e,\ s\rangle$
BinOpThrow2: $P \vdash \langle (\mathit{Val}\ v_1)\ «\mathit{bop}»\ (\mathit{throw}\ e),\ s\rangle \to \langle \mathit{throw}\ e,\ s\rangle$
LAssThrow: $P \vdash \langle V{:=}(\mathit{throw}\ e),\ s\rangle \to \langle \mathit{throw}\ e,\ s\rangle$
FAccThrow: $P \vdash \langle (\mathit{throw}\ e){\cdot}F\{D\},\ s\rangle \to \langle \mathit{throw}\ e,\ s\rangle$
FAssThrow1: $P \vdash \langle (\mathit{throw}\ e){\cdot}F\{D\}{:=}e_2,\ s\rangle \to \langle \mathit{throw}\ e,s\rangle$
FAssThrow2: $P \vdash \langle \mathit{Val}\ v{\cdot}F\{D\}{:=}(\mathit{throw}\ e),\ s\rangle \to \langle \mathit{throw}\ e,\ s\rangle$
CallThrowObj: $P \vdash \langle (\mathit{throw}\ e){\cdot}M(es),\ s\rangle \to \langle \mathit{throw}\ e,\ s\rangle$
CallThrowParams: $[\![\ es = \mathit{map}\ \mathit{Val}\ \mathit{vs}\ @\ \mathit{throw}\ e\ \#\ es'\ ]\!] \Longrightarrow P \vdash \langle (\mathit{Val}\ v){\cdot}M(es),\ s\rangle \to \langle \mathit{throw}\ e,\ s\rangle$
BlockThrow: $P \vdash \langle\{V{:}T;\ \mathit{Throw}\ a\},\ s\rangle \to \langle \mathit{Throw}\ a,\ s\rangle$
InitBlockThrow: $P \vdash \langle\{V{:}T := \mathit{Val}\ v;\ \mathit{Throw}\ a\},\ s\rangle \to \langle \mathit{Throw}\ a,\ s\rangle$
SeqThrow: $P \vdash \langle (\mathit{throw}\ e);;e_2,\ s\rangle \to \langle \mathit{throw}\ e,\ s\rangle$
CondThrow: $P \vdash \langle \mathit{if}\ (\mathit{throw}\ e)\ e_1\ \mathit{else}\ e_2,\ s\rangle \to \langle \mathit{throw}\ e,\ s\rangle$
ThrowThrow: $P \vdash \langle \mathit{throw}(\mathit{throw}\ e),\ s\rangle \to \langle \mathit{throw}\ e,\ s\rangle$

2.11.1 The reflexive transitive closure

translations
$P \vdash \langle e,s\rangle \to\!*\ \langle e',s'\rangle\ \ ==\ \ ((e,s),\ e',s') \in (\mathit{red}\ P)^*$
$P \vdash \langle es,s\rangle\ [\to]*\ \langle es',s'\rangle\ \ ==\ \ ((es,s),\ es',s') \in (\mathit{reds}\ P)^*$

2.12 System Classes

theory *SystemClasses* = *Decl* + *Exceptions*:

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

constdefs

$ObjectC :: {}'m\ cdecl$
$ObjectC \equiv (Object,\ (arbitrary,[],[]))$

$NullPointerC :: {}'m\ cdecl$
$NullPointerC \equiv (NullPointer,\ (Object,[],[]))$

$ClassCastC :: {}'m\ cdecl$
$ClassCastC \equiv (ClassCast,\ (Object,[],[]))$

$OutOfMemoryC :: {}'m\ cdecl$
$OutOfMemoryC \equiv (OutOfMemory,\ (Object,[],[]))$

$SystemClasses :: {}'m\ cdecl\ list$
$SystemClasses \equiv [ObjectC,\ NullPointerC,\ ClassCastC,\ OutOfMemoryC]$

end

2.13 Generic Well-formedness of programs

theory *WellForm* = *TypeRel* + *SystemClasses*:

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.

types $'m\ \textit{wf-mdecl-test} = {'m}\ \textit{prog} \Rightarrow \textit{cname} \Rightarrow {'m}\ \textit{mdecl} \Rightarrow \textit{bool}$

constdefs

$\textit{wf-fdecl} :: {'m}\ \textit{prog} \Rightarrow \textit{fdecl} \Rightarrow \textit{bool}$
$\textit{wf-fdecl}\ P \equiv \lambda(F,T).\ \textit{is-type}\ P\ T$

$\textit{wf-mdecl} :: {'m}\ \textit{wf-mdecl-test} \Rightarrow {'m}\ \textit{wf-mdecl-test}$
$\textit{wf-mdecl}\ \textit{wf-md}\ P\ C \equiv \lambda(M,Ts,T,mb).$
$(\forall T{\in}\textit{set}\ Ts.\ \textit{is-type}\ P\ T) \wedge \textit{is-type}\ P\ T \wedge \textit{wf-md}\ P\ C\ (M,Ts,T,mb)$

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

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

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

2.13.1 Well-formedness lemmas

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

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

lemma *is-class-Object* [*simp*]:
$\textit{wf-prog}\ \textit{wf-md}\ P \Longrightarrow \textit{is-class}\ P\ \textit{Object}$

lemma *is-class-xcpt*:
$[\![\ C \in \textit{sys-xcpts};\ \textit{wf-prog}\ \textit{wf-md}\ P\ ]\!] \Longrightarrow \textit{is-class}\ P\ C$

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

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

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

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

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

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

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*:
$[\![\ is\text{-}class\ P\ C;\ wf\text{-}prog\ wf\text{-}md\ P\ ]\!] \Longrightarrow P \vdash C \preceq^* Object$

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

2.13.2 Well-formedness and method lookup

lemma *sees-wf-mdecl*:
$[\![\ wf\text{-}prog\ wf\text{-}md\ P;\ P \vdash C\ sees\ M{:}Ts{\rightarrow}T = m\ in\ D\ ]\!] \Longrightarrow wf\text{-}mdecl\ wf\text{-}md\ P\ D\ (M,Ts,T,m)$

lemma *sees-method-mono* [*rule-format* (*no-asm*)]:
$[\![\ P \vdash C' \preceq^* C;\ wf\text{-}prog\ wf\text{-}md\ P\ ]\!] \Longrightarrow$
$\forall D\ Ts\ T\ m.\ P \vdash C\ sees\ M{:}Ts{\rightarrow}T = m\ in\ D \longrightarrow$
$(\exists D'\ Ts'\ T'\ m'.\ P \vdash C'\ sees\ M{:}Ts'{\rightarrow}T' = m'\ in\ D' \wedge P \vdash Ts\ [\leq]\ Ts' \wedge P \vdash T' \leq T)$

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

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

lemma *mdecl-visible*:
assumes *wf*: *wf-prog wf-md P* **and** *C*: $(C,S,fs,ms) \in set\ P$ **and** *m*: $(M,Ts,T,m) \in set\ ms$
shows $P \vdash C\ sees\ M{:}Ts{\rightarrow}T = m\ in\ C$

lemma *Call-lemma*:
$[\![\ P \vdash C\ sees\ M{:}Ts{\rightarrow}T = m\ in\ D;\ P \vdash C' \preceq^* C;\ wf\text{-}prog\ wf\text{-}md\ P\ ]\!]$
$\Longrightarrow \exists D'\ Ts'\ T'\ m'.$
$P \vdash C'\ sees\ M{:}Ts'{\rightarrow}T' = m'\ in\ D' \wedge P \vdash Ts\ [\leq]\ Ts' \wedge P \vdash T' \leq T \wedge P \vdash C' \preceq^* D'$
$\wedge\ is\text{-}type\ P\ T' \wedge (\forall T{\in}set\ Ts'.\ is\text{-}type\ P\ T) \wedge wf\text{-}md\ P\ D'\ (M,Ts',T',m')$

lemma *wf-prog-lift*:
assumes *wf*: $wf\text{-}prog\ (\lambda P\ C\ bd.\ A\ P\ C\ bd)\ P$
and *rule*:
$\bigwedge wf\text{-}md\ C\ M\ Ts\ C\ T\ m\ bd.$
$wf\text{-}prog\ wf\text{-}md\ P \Longrightarrow$
$P \vdash C\ sees\ M{:}Ts{\rightarrow}T = m\ in\ C \Longrightarrow$
$set\ Ts \subseteq\ types\ P \Longrightarrow$
$bd = (M,Ts,T,m) \Longrightarrow$
$A\ P\ C\ bd \Longrightarrow$
$B\ P\ C\ bd$
shows $wf\text{-}prog\ (\lambda P\ C\ bd.\ B\ P\ C\ bd)\ P$

2.13.3 Well-formedness and field lookup

lemma *wf-Fields-Ex*:
$[\![\ wf\text{-}prog\ wf\text{-}md\ P;\ is\text{-}class\ P\ C\ ]\!] \Longrightarrow \exists FDTs.\ P \vdash C\ has\text{-}fields\ FDTs$

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

lemma *sees-field-is-type*:
$[\![\ P \vdash C\ sees\ F{:}T\ in\ D;\ wf\text{-}prog\ wf\text{-}md\ P\ ]\!] \Longrightarrow is\text{-}type\ P\ T$

end

2.14 Weak well-formedness of Jinja programs

theory *WWellForm* = *WellForm* + *Expr*:

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

lemma *wwf-J-mdecl*[*simp*]:
$wwf\text{-}J\text{-}mdecl\ P\ C\ (M,Ts,T,pns,body) =$
$(length\ Ts = length\ pns \wedge distinct\ pns \wedge this \notin set\ pns \wedge fv\ body \subseteq \{this\} \cup set\ pns)$

syntax
$wwf\text{-}J\text{-}prog :: J\text{-}prog \Rightarrow bool$
translations
$wwf\text{-}J\text{-}prog\ ==\ wf\text{-}prog\ wwf\text{-}J\text{-}mdecl$

end

2.15 Equivalence of Big Step and Small Step Semantics

theory *Equivalence* = *BigStep* + *SmallStep* + *WWellForm*:

2.15.1 Small steps simulate big step

Cast

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

LAss

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

BinOp

lemma *BinOp1Reds*:
$P \vdash \langle e,s\rangle \to\!* \langle e',s'\rangle \Longrightarrow P \vdash \langle\ e \ll bop \gg e_2,\ s\rangle \to\!* \langle e' \ll bop \gg e_2,\ s'\rangle$
lemma *BinOp2Reds*:
$P \vdash \langle e,s\rangle \to\!* \langle e',s'\rangle \Longrightarrow P \vdash \langle (Val\ v) \ll bop \gg e,\ s\rangle \to\!* \langle (Val\ v) \ll bop \gg e',\ s'\rangle$
lemma *BinOpRedsVal*:
$[\![\ P \vdash \langle e_1,s_0\rangle \to\!* \langle Val\ v_1,s_1\rangle;\ P \vdash \langle e_2,s_1\rangle \to\!* \langle Val\ v_2,s_2\rangle;\ binop(bop,v_1,v_2) = Some\ v\ ]\!]$
$\Longrightarrow P \vdash \langle e_1 \ll bop \gg e_2,\ s_0\rangle \to\!* \langle Val\ v,s_2\rangle$
lemma *BinOpRedsThrow1*:
$P \vdash \langle e,s\rangle \to\!* \langle throw\ e',s'\rangle \Longrightarrow P \vdash \langle e \ll bop \gg e_2,\ s\rangle \to\!* \langle throw\ e',\ s'\rangle$
lemma *BinOpRedsThrow2*:
$[\![\ P \vdash \langle e_1,s_0\rangle \to\!* \langle Val\ v_1,s_1\rangle;\ P \vdash \langle e_2,s_1\rangle \to\!* \langle throw\ e,s_2\rangle]\!]$
$\Longrightarrow P \vdash \langle e_1 \ll bop \gg e_2,\ s_0\rangle \to\!* \langle throw\ e,s_2\rangle$

FAcc

lemma *FAccReds*:
$P \vdash \langle e,s\rangle \to\!* \langle e',s'\rangle \Longrightarrow P \vdash \langle e{\cdot}F\{D\},\ s\rangle \to\!* \langle e'{\cdot}F\{D\},\ s'\rangle$
lemma *FAccRedsVal*:
$[\![P \vdash \langle e,s\rangle \to\!* \langle addr\ a,s'\rangle;\ hp\ s'\ a = Some(C,fs);\ fs(F,D) = Some\ v\ ]\!]$
$\Longrightarrow P \vdash \langle e{\cdot}F\{D\},s\rangle \to\!* \langle Val\ v,s'\rangle$
lemma *FAccRedsNull*:
$P \vdash \langle e,s\rangle \to\!* \langle null,s'\rangle \Longrightarrow P \vdash \langle e{\cdot}F\{D\},s\rangle \to\!* \langle THROW\ NullPointer,s'\rangle$
lemma *FAccRedsThrow*:

$P \vdash \langle e,s\rangle \to* \langle throw\ a,s'\rangle \Longrightarrow P \vdash \langle e\cdot F\{D\},s\rangle \to* \langle throw\ a,s'\rangle$

FAss

lemma *FAssReds1*:
$P \vdash \langle e,s\rangle \to* \langle e',s'\rangle \Longrightarrow P \vdash \langle e\cdot F\{D\}{:=}e_2,\ s\rangle \to* \langle e'\cdot F\{D\}{:=}e_2,\ s'\rangle$
lemma *FAssReds2*:
$P \vdash \langle e,s\rangle \to* \langle e',s'\rangle \Longrightarrow P \vdash \langle Val\ v\cdot F\{D\}{:=}e,\ s\rangle \to* \langle Val\ v\cdot F\{D\}{:=}e',\ s'\rangle$
lemma *FAssRedsVal*:
$[\![\ P \vdash \langle e_1,s_0\rangle \to* \langle addr\ a,s_1\rangle;\ P \vdash \langle e_2,s_1\rangle \to* \langle Val\ v,(h_2,l_2)\rangle;\ Some(C,fs) = h_2\ a\ ]\!] \Longrightarrow$
$P \vdash \langle e_1\cdot F\{D\}{:=}e_2,\ s_0\rangle \to* \langle unit,\ (h_2(a\mapsto(C,fs((F,D)\mapsto v))),l_2)\rangle$
lemma *FAssRedsNull*:
$[\![\ P \vdash \langle e_1,s_0\rangle \to* \langle null,s_1\rangle;\ P \vdash \langle e_2,s_1\rangle \to* \langle Val\ v,s_2\rangle\ ]\!] \Longrightarrow$
$P \vdash \langle e_1\cdot F\{D\}{:=}e_2,\ s_0\rangle \to* \langle THROW\ NullPointer,\ s_2\rangle$
lemma *FAssRedsThrow1*:
$P \vdash \langle e,s\rangle \to* \langle throw\ e',s'\rangle \Longrightarrow P \vdash \langle e\cdot F\{D\}{:=}e_2,\ s\rangle \to* \langle throw\ e',\ s'\rangle$
lemma *FAssRedsThrow2*:
$[\![\ P \vdash \langle e_1,s_0\rangle \to* \langle Val\ v,s_1\rangle;\ P \vdash \langle e_2,s_1\rangle \to* \langle throw\ e,s_2\rangle\ ]\!]$
$\Longrightarrow P \vdash \langle e_1\cdot F\{D\}{:=}e_2,s_0\rangle \to* \langle throw\ e,s_2\rangle$

;;

lemma *SeqReds*:
$P \vdash \langle e,s\rangle \to* \langle e',s'\rangle \Longrightarrow P \vdash \langle e;;e_2,\ s\rangle \to* \langle e';;e_2,\ s'\rangle$
lemma *SeqRedsThrow*:
$P \vdash \langle e,s\rangle \to* \langle throw\ e',s'\rangle \Longrightarrow P \vdash \langle e;;e_2,\ s\rangle \to* \langle throw\ e',\ s'\rangle$
lemma *SeqReds2*:
$[\![\ P \vdash \langle e_1,s_0\rangle \to* \langle Val\ v_1,s_1\rangle;\ P \vdash \langle e_2,s_1\rangle \to* \langle e_2',s_2\rangle\ ]\!] \Longrightarrow P \vdash \langle e_1;;e_2,\ s_0\rangle \to* \langle e_2',s_2\rangle$

If

lemma *CondReds*:
$P \vdash \langle e,s\rangle \to* \langle e',s'\rangle \Longrightarrow P \vdash \langle if\ (e)\ e_1\ else\ e_2,s\rangle \to* \langle if\ (e')\ e_1\ else\ e_2,s'\rangle$
lemma *CondRedsThrow*:
$P \vdash \langle e,s\rangle \to* \langle throw\ a,s'\rangle \Longrightarrow P \vdash \langle if\ (e)\ e_1\ else\ e_2,\ s\rangle \to* \langle throw\ a,s'\rangle$
lemma *CondReds2T*:
$[\![P \vdash \langle e,s_0\rangle \to* \langle true,s_1\rangle;\ P \vdash \langle e_1,\ s_1\rangle \to* \langle e',s_2\rangle\ ]\!] \Longrightarrow P \vdash \langle if\ (e)\ e_1\ else\ e_2,\ s_0\rangle \to* \langle e',s_2\rangle$
lemma *CondReds2F*:
$[\![P \vdash \langle e,s_0\rangle \to* \langle false,s_1\rangle;\ P \vdash \langle e_2,\ s_1\rangle \to* \langle e',s_2\rangle\ ]\!] \Longrightarrow P \vdash \langle if\ (e)\ e_1\ else\ e_2,\ s_0\rangle \to* \langle e',s_2\rangle$

While

lemma *WhileFReds*:
$P \vdash \langle b,s\rangle \to* \langle false,s'\rangle \Longrightarrow P \vdash \langle while\ (b)\ c,s\rangle \to* \langle unit,s'\rangle$
lemma *WhileRedsThrow*:
$P \vdash \langle b,s\rangle \to* \langle throw\ e,s'\rangle \Longrightarrow P \vdash \langle while\ (b)\ c,s\rangle \to* \langle throw\ e,s'\rangle$
lemma *WhileTReds*:
$[\![\ P \vdash \langle b,s_0\rangle \to* \langle true,s_1\rangle;\ P \vdash \langle c,s_1\rangle \to* \langle Val\ v_1,s_2\rangle;\ P \vdash \langle while\ (b)\ c,s_2\rangle \to* \langle e,s_3\rangle\ ]\!]$
$\Longrightarrow P \vdash \langle while\ (b)\ c,s_0\rangle \to* \langle e,s_3\rangle$
lemma *WhileTRedsThrow*:
$[\![\ P \vdash \langle b,s_0\rangle \to* \langle true,s_1\rangle;\ P \vdash \langle c,s_1\rangle \to* \langle throw\ e,s_2\rangle\ ]\!]$
$\Longrightarrow P \vdash \langle while\ (b)\ c,s_0\rangle \to* \langle throw\ e,s_2\rangle$

Throw

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

InitBlock

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

Block

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

try-catch

lemma *TryReds*:
$P \vdash \langle e,s\rangle \to\ast \langle e',s'\rangle \Longrightarrow P \vdash \langle \mathit{try}\ e\ \mathit{catch}(C\ V)\ e_2,s\rangle \to\ast \langle \mathit{try}\ e'\ \mathit{catch}(C\ V)\ e_2,s'\rangle$
lemma *TryRedsVal*:
$P \vdash \langle e,s\rangle \to\ast \langle \mathit{Val}\ v,s'\rangle \Longrightarrow P \vdash \langle \mathit{try}\ e\ \mathit{catch}(C\ V)\ e_2,s\rangle \to\ast \langle \mathit{Val}\ v,s'\rangle$
lemma *TryCatchRedsFinal*:
$[\![\ P \vdash \langle e_1,s_0\rangle \to\ast \langle \mathit{Throw}\ a,(h_1,l_1)\rangle;\ \ h_1\ a = \mathit{Some}(D,\mathit{fs});\ P \vdash D \preceq^*\ C;$
$P \vdash \langle e_2,\ (h_1,\ l_1(V \mapsto \mathit{Addr}\ a))\rangle \to\ast \langle e_2',\ (h_2,l_2)\rangle;\ \mathit{final}\ e_2'\ ]\!]$
$\Longrightarrow P \vdash \langle \mathit{try}\ e_1\ \mathit{catch}(C\ V)\ e_2,\ s_0\rangle \to\ast \langle e_2',\ (h_2,\ l_2(V := l_1\ V))\rangle$
lemma *TryRedsFail*:
$[\![\ P \vdash \langle e_1,s\rangle \to\ast \langle \mathit{Throw}\ a,(h,l)\rangle;\ h\ a = \mathit{Some}(D,\mathit{fs});\ \neg\ P \vdash D \preceq^*\ C\ ]\!]$
$\Longrightarrow P \vdash \langle \mathit{try}\ e_1\ \mathit{catch}(C\ V)\ e_2,s\rangle \to\ast \langle \mathit{Throw}\ a,(h,l)\rangle$

List

lemma *ListReds1*:
$P \vdash \langle e,s\rangle \to\ast \langle e',s'\rangle \Longrightarrow P \vdash \langle e\#es,s\rangle\ [\to]\ast \langle e'\ \#\ es,s'\rangle$
lemma *ListReds2*:
$P \vdash \langle es,s\rangle\ [\to]\ast \langle es',s'\rangle \Longrightarrow P \vdash \langle \mathit{Val}\ v\ \#\ es,s\rangle\ [\to]\ast \langle \mathit{Val}\ v\ \#\ es',s'\rangle$
lemma *ListRedsVal*:
$[\![\ P \vdash \langle e,s_0\rangle \to\ast \langle \mathit{Val}\ v,s_1\rangle;\ P \vdash \langle es,s_1\rangle\ [\to]\ast \langle es',s_2\rangle\ ]\!]$
$\Longrightarrow P \vdash \langle e\#es,s_0\rangle\ [\to]\ast \langle \mathit{Val}\ v\ \#\ es',s_2\rangle$

Call

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

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

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

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

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

lemma *overwrite-upd-lemma*:
$(f(g(a{\mapsto}b) \mid A))(a := g\ a) = f(g|insert\ a\ A)$

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

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

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

An now the actual method call reduction lemmas.

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

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

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

lemma *CallRedsThrowParams*:
$[\![\ P \vdash \langle e,s0\rangle \to\!*\ \langle Val\ v,s_1\rangle;\ \ P \vdash \langle es,s_1\rangle\ [\to]*\ \langle map\ Val\ vs_1\ @\ throw\ a\ \#\ es_2,s_2\rangle\ ]\!]$
$\Longrightarrow P \vdash \langle e{\cdot}M(es),s0\rangle \to\!*\ \langle throw\ a,s_2\rangle$

lemma *CallRedsThrowObj*:
$P \vdash \langle e,s0\rangle \to\!*\ \langle throw\ a,s_1\rangle \Longrightarrow P \vdash \langle e{\cdot}M(es),s0\rangle \to\!*\ \langle throw\ a,s_1\rangle$

lemma *CallRedsNull*:
$[\![P \vdash \langle e,s_0\rangle \to\!* \langle null,s_1\rangle;\ P \vdash \langle es,s_1\rangle\ [\to]\!*\ \langle map\ Val\ vs,s_2\rangle]\!]$
$\Longrightarrow P \vdash \langle e{\cdot}M(es),s_0\rangle \to\!* \langle THROW\ NullPointer,s_2\rangle$

The main Theorem

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

2.15.2 Big steps simulates small step

This direction was carried out by Norbert Schirmer and Daniel Wasserrab.

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

lemma
assumes *wf*: *wwf-J-prog P*
shows *eval-restrict-lcl*:
$P \vdash \langle e,(h,l)\rangle \Rightarrow \langle e',(h',l')\rangle \Longrightarrow (\bigwedge W.\ fv\ e \subseteq W \Longrightarrow P \vdash \langle e,(h,l\lfloor W)\rangle \Rightarrow \langle e',(h',l'\lfloor W)\rangle)$
and $P \vdash \langle es,(h,l)\rangle\ [\Rightarrow]\ \langle es',(h',l')\rangle \Longrightarrow (\bigwedge W.\ fvs\ es \subseteq W \Longrightarrow P \vdash \langle es,(h,l\lfloor W)\rangle\ [\Rightarrow]\ \langle es',(h',l'\lfloor W)\rangle)$

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

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

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

The key lemma:

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

Its extension to $\to\!*$:

lemma *extend-eval*:
assumes *wf*: *wwf-J-prog P*
and *reds*: $P \vdash \langle e,s\rangle \to\!* \langle e'',s''\rangle$ **and** *eval-rest*: $P \vdash \langle e'',s''\rangle \Rightarrow \langle e',s'\rangle$
shows $P \vdash \langle e,s\rangle \Rightarrow \langle e',s'\rangle$

lemma *extend-evals*:
assumes *wf*: *wwf-J-prog P*
and *reds*: $P \vdash \langle es,s\rangle$ $[\rightarrow]*$ $\langle es'',s''\rangle$ **and** *eval-rest*: $P \vdash \langle es'',s''\rangle$ $[\Rightarrow]$ $\langle es',s'\rangle$
shows $P \vdash \langle es,s\rangle$ $[\Rightarrow]$ $\langle es',s'\rangle$

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

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

2.15.3 Equivalence

And now, the crowning achievement:

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

end

2.16 Well-typedness of Jinja expressions

theory *WellType* = *Objects* + *Expr*:

types
$env \;\; = vname \rightharpoonup ty$

consts
$WT :: J\text{-}prog \Rightarrow (env \times expr \quad\;\; \times ty \quad\;\;)\ set$
$WTs :: J\text{-}prog \Rightarrow (env \times expr\ list \times ty\ list)\ set$

translations
$P,E \vdash e :: T \;\; == \;\; (E,e,T) \in WT\ P$
$P,E \vdash es\ [::]\ Ts \;\; == \;\; (E,es,Ts) \in WTs\ P$

inductive *WT P WTs P*
intros

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

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

WTVal:
$typeof\ v = Some\ T \implies$
$P,E \vdash Val\ v :: T$

WTVar:
$E\ v = Some\ T \implies$
$P,E \vdash Var\ v :: T$

WTBinOp:
$[\![\ P,E \vdash e_1 :: T_1;\;\; P,E \vdash e_2 :: T_2;$
$\quad case\ bop\ of\ Eq \Rightarrow (P \vdash T_1 \leq T_2 \vee P \vdash T_2 \leq T_1) \wedge T = Boolean$
$\qquad\qquad\quad |\ Add \Rightarrow T_1 = Integer \wedge T_2 = Integer \wedge T = Integer\ ]\!]$
$\implies P,E \vdash e_1 \ll bop \gg e_2 :: T$

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

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

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

WTCall:
$[\![\ P,E \vdash e :: Class\ C;\;\; P \vdash C\ sees\ M{:}Ts \rightarrow T = (pns,body)\ in\ D;$

$P,E \vdash es\ [::]\ Ts'$; $P \vdash Ts'\ [\le]\ Ts$ $]\!]$
$\Longrightarrow P,E \vdash e\cdot M(es) :: T$

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

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

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

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

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

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

— well-typed expression lists

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

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

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

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 $P,E \vdash es\ [::]\ Ts \Longrightarrow fvs\ es \subseteq dom\ E$

2.17 Runtime Well-typedness

theory *WellTypeRT* = *WellType*:

consts
$WTrt$ :: $J\text{-}prog \Rightarrow (env \times heap \times expr \times ty)set$
$WTrts$:: $J\text{-}prog \Rightarrow (env \times heap \times expr\ list \times ty\ list)set$

translations
$P,E,h \vdash e : T \ == \ (E,h,e,T) \in WTrt\ P$
$P,E,h \vdash es[:]Ts \ == \ (E,h,es,Ts) \in WTrts\ P$

inductive *WTrt P WTrts P*
intros

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

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

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

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

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

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

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

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

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

WTrtFAssNT:

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

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

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

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

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

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

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

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

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

— well-typed expression lists

WTrtNil:
$P,E,h \vdash []\ [:]\ []$

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

2.17.1 Easy consequences

lemma [*iff*]: $(P,E,h \vdash []\ [:]\ Ts) = (Ts = [])$
lemma [*iff*]: $(P,E,h \vdash e\#es\ [:]\ T\#Ts) = (P,E,h \vdash e : T \wedge P,E,h \vdash es\ [:]\ Ts)$
lemma [*iff*]: $(P,E,h \vdash (e\#es)\ [:]\ Ts) =$
$(\exists\, U\ Us.\ Ts = U\#Us \wedge P,E,h \vdash e : U \wedge P,E,h \vdash es\ [:]\ Us)$
lemma [*simp*]: $\forall\, Ts.\ (P,E,h \vdash es_1\ @\ es_2\ [:]\ Ts) =$

$(\exists\, Ts_1\ Ts_2.\ Ts = Ts_1\ @\ Ts_2 \wedge P,E,h \vdash es_1\ [:]\ Ts_1\ \&\ P,E,h \vdash es_2[:]Ts_2)$
lemma [*iff*]: $P,E,h \vdash \mathit{Val}\ v : T = (\mathit{typeof}_h\ v = \mathit{Some}\ T)$
lemma [*iff*]: $P,E,h \vdash \mathit{Var}\ v : T = (E\ v = \mathit{Some}\ T)$
lemma [*iff*]: $P,E,h \vdash e_1;;e_2 : T_2 = (\exists\, T_1.\ P,E,h \vdash e_1{:}T_1 \wedge P,E,h \vdash e_2{:}T_2)$
lemma [*iff*]: $P,E,h \vdash \{V{:}T;\ e\} : T' = (P,E(V \mapsto T),h \vdash e : T')$

2.17.2 Some interesting lemmas

lemma *WTrts-Val*[*simp*]:
$\bigwedge Ts.\ (P,E,h \vdash \mathit{map}\ \mathit{Val}\ \mathit{vs}\ [:]\ Ts) = (\mathit{map}\ (\mathit{typeof}_h)\ \mathit{vs} = \mathit{map}\ \mathit{Some}\ Ts)$

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

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

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

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

end

2.18 Definite assignment

theory *DefAss* = *BigStep*:

2.18.1 Hypersets

types *'a hyperset* = *'a set option*

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

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

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

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

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

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* ⊔ *A* = *None* ∧ *A* ⊔ *None* = *None*
lemma [*simp*]: *a* ∈∈ *None* ∧ *None* ⊖ *a* = *None*
lemma *hyperUn-assoc*: $(A \sqcup B) \sqcup C = A \sqcup (B \sqcup C)$
lemma *hyper-insert-comm*: $A \sqcup \lfloor\{a\}\rfloor = \lfloor\{a\}\rfloor \sqcup A \wedge A \sqcup (\lfloor\{a\}\rfloor \sqcup B) = \lfloor\{a\}\rfloor \sqcup (A \sqcup B)$

2.18.2 Definite assignment

consts
$\mathcal{A}$ :: *'a exp* ⇒ *'a hyperset*
$\mathcal{A}s$ :: *'a exp list* ⇒ *'a hyperset*
$\mathcal{D}$ :: *'a exp* ⇒ *'a hyperset* ⇒ *bool*
$\mathcal{D}s$ :: *'a exp list* ⇒ *'a hyperset* ⇒ *bool*

primrec
$\mathcal{A}$ (*new C*) = $\lfloor\{\}\rfloor$
$\mathcal{A}$ (*Cast C e*) = $\mathcal{A}$ *e*
$\mathcal{A}$ (*Val v*) = $\lfloor\{\}\rfloor$
$\mathcal{A}\ (e_1 \ll bop \gg e_2) = \mathcal{A}\ e_1 \sqcup \mathcal{A}\ e_2$
$\mathcal{A}$ (*Var V*) = $\lfloor\{\}\rfloor$
$\mathcal{A}$ (*LAss V e*) = $\lfloor\{V\}\rfloor \sqcup \mathcal{A}$ *e*
$\mathcal{A}$ (*e*•*F*{*D*}) = $\mathcal{A}$ *e*
$\mathcal{A}\ (e_1 \bullet F\{D\} := e_2) = \mathcal{A}\ e_1 \sqcup \mathcal{A}\ e_2$
$\mathcal{A}$ (*e*•*M*(*es*)) = $\mathcal{A}$ *e* ⊔ $\mathcal{A}s$ *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) $= \mathcal{A}\ e_1 \sqcap (\mathcal{A}\ e_2 \ominus V)$

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

primrec
$\mathcal{D}$ (*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* $V\ e$) $A = \mathcal{D}\ e\ A$
$\mathcal{D}$ ($e{\cdot}F\{D\}$) $A = \mathcal{D}\ e\ A$
$\mathcal{D}$ ($e_1{\cdot}F\{D\}{:=}e_2$) $A = (\mathcal{D}\ e_1\ A \wedge \mathcal{D}\ e_2\ (A \sqcup \mathcal{A}\ e_1))$
$\mathcal{D}$ ($e{\cdot}M(es)$) $A = (\mathcal{D}\ e\ A \wedge \mathcal{D}s\ es\ (A \sqcup \mathcal{A}\ e))$
$\mathcal{D}$ ($\{V{:}T;\ e\}$) $A = \mathcal{D}\ e\ (A \ominus V)$
$\mathcal{D}$ ($e_1;;e_2$) $A = (\mathcal{D}\ e_1\ A \wedge \mathcal{D}\ e_2\ (A \sqcup \mathcal{A}\ e_1))$
$\mathcal{D}$ (*if* (e) e_1 *else* e_2) $A =$
  $(\mathcal{D}\ e\ A \wedge \mathcal{D}\ e_1\ (A \sqcup \mathcal{A}\ e) \wedge \mathcal{D}\ e_2\ (A \sqcup \mathcal{A}\ e))$
$\mathcal{D}$ (*while* (e) c) $A = (\mathcal{D}\ e\ A \wedge \mathcal{D}\ c\ (A \sqcup \mathcal{A}\ e))$
$\mathcal{D}$ (*throw* e) $A = \mathcal{D}\ e\ A$
$\mathcal{D}$ (*try* e_1 *catch*($C\ V$) e_2) $A = (\mathcal{D}\ e_1\ A \wedge \mathcal{D}\ e_2\ (A \sqcup \lfloor\{V\}\rfloor))$

$\mathcal{D}s$ ([]) $A =$ *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'*: $\mathcal{D}s\ es\ A \Longrightarrow A \sqsubseteq A' \Longrightarrow \mathcal{D}s\ es\ A'$

end

2.19 Conformance Relations for Type Soundness Proofs

theory *Conform* = *Exceptions*:

constdefs
$conf :: {'m}\ prog \Rightarrow heap \Rightarrow val \Rightarrow ty \Rightarrow bool \quad (\text{-,-} \vdash \text{-} :\leq \text{-}\ \ [51,51,51,51]\ 50)$
$P,h \vdash v :\leq T \ \equiv$
$\exists T'.\ typeof_h\ v = Some\ T' \wedge P \vdash T' \leq T$

$fconf :: {'m}\ prog \Rightarrow heap \Rightarrow ({'a} \rightharpoonup val) \Rightarrow ({'a} \rightharpoonup ty) \Rightarrow bool \quad (\text{-,-} \vdash \text{-}\ {'}(:\leq{'})\ \text{-}\ [51,51,51,51]\ 50)$
$P,h \vdash v_m\ (:\leq)\ T_m \ \equiv$
$\forall FD\ T.\ T_m\ FD = Some\ T \longrightarrow (\exists v.\ v_m\ FD = Some\ v \wedge P,h \vdash v :\leq T)$

$oconf :: {'m}\ prog \Rightarrow heap \Rightarrow obj \Rightarrow bool \quad (\text{-,-} \vdash \text{-} \surd\ [51,51,51]\ 50)$
$P,h \vdash obj \surd \ \equiv$
$let\ (C,v_m) = obj\ in\ \exists FDTs.\ P \vdash C\ has\text{-}fields\ FDTs \wedge P,h \vdash v_m\ (:\leq)\ map\text{-}of\ FDTs$

$hconf :: {'m}\ prog \Rightarrow heap \Rightarrow bool \quad (\text{-} \vdash \text{-} \surd\ [51,51]\ 50)$
$P \vdash h \surd \ \equiv$
$(\forall a\ obj.\ h\ a = Some\ obj \longrightarrow P,h \vdash obj \surd) \wedge preallocated\ h$

$lconf :: {'m}\ prog \Rightarrow heap \Rightarrow ({'a} \rightharpoonup val) \Rightarrow ({'a} \rightharpoonup ty) \Rightarrow bool \quad (\text{-,-} \vdash \text{-}\ {'}(:\leq{'})_w\ \text{-}\ [51,51,51,51]\ 50)$
$P,h \vdash v_m\ (:\leq)_w\ T_m \ \equiv$
$\forall V\ v.\ v_m\ V = Some\ v \longrightarrow (\exists T.\ T_m\ V = Some\ T \wedge P,h \vdash v :\leq T)$

translations
$P,h \vdash vs\ [:\leq]\ Ts \ == \ list\text{-}all2\ (conf\ P\ h)\ vs\ Ts$

2.19.1 Value conformance :≤

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*]: $is\text{-}type\ P\ T \Longrightarrow P,h \vdash default\text{-}val\ T :\leq T$
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*: $[\![\ v \neq Null;\ P,h \vdash v :\leq Class\ C\ ]\!]$
$\Longrightarrow \exists a\ C'\ fs.\ v = Addr\ a \wedge h\ a = Some(C',fs) \wedge P \vdash C' \preceq^* C$

2.19.2 Value list conformance [:≤]

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*: $P,h \vdash xs\ [:\leq]\ y\#ys = (\exists z\ zs.\ xs = z\#zs \wedge P,h \vdash z :\leq y \wedge P,h \vdash zs\ [:\leq]\ ys)$

2.19.3 Field conformance $(:\le)$

lemma *fconf-hext*: $[\![\ P,h \vdash \mathit{fvs}\ (:\le)\ E;\ h \unlhd h'\ ]\!] \Longrightarrow P,h' \vdash \mathit{fvs}\ (:\le)\ E$
lemma *fconf-init-fields*: $P,h \vdash \mathit{init\text{-}fields}\ \mathit{fs}\ (:\le)\ \mathit{map\text{-}of}\ \mathit{fs}$

2.19.4 Object conformance

lemma *oconf-hext*: $P,h \vdash \mathit{obj}\ \surd \Longrightarrow h \unlhd h' \Longrightarrow P,h' \vdash \mathit{obj}\ \surd$
lemma *oconf-fupd* [*intro?*]:
$[\![\ P \vdash C\ \mathit{has}\ F{:}T\ \mathit{in}\ D;\ P,h \vdash v :\le T;\ P,h \vdash (C,\mathit{fs})\ \surd\ ]\!]$
$\Longrightarrow P,h \vdash (C,\ \mathit{fs}((F,D)\mapsto v))\ \surd$

2.19.5 Heap conformance

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

2.19.6 Local variable conformance

lemma *lconf-hext*: $[\![\ P,h \vdash l\ (:\le)_w\ E;\ h \unlhd h'\ ]\!] \Longrightarrow P,h' \vdash l\ (:\le)_w\ E$
lemma *lconf-upd*:
$[\![\ P,h \vdash l\ (:\le)_w\ E;\ P,h \vdash v :\le T;\ E\ V = \mathit{Some}\ T\ ]\!] \Longrightarrow P,h \vdash l(V\mapsto v)\ (:\le)_w\ E$
lemma *lconf-empty*[*iff*]: $P,h \vdash \mathit{empty}\ (:\le)_w\ E$
lemma *lconf-upd2*: $[\![P,h \vdash l\ (:\le)_w\ E;\ P,h \vdash v :\le T]\!] \Longrightarrow P,h \vdash l(V\mapsto v)\ (:\le)_w\ E(V\mapsto T)$

end

2.20 Progress of Small Step Semantics

theory *Progress* = *Equivalence* + *WellTypeRT* + *DefAss* + *Conform*:

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

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

Derivation of new induction scheme for well typing:

consts
$WTrt' :: J\text{-}prog \Rightarrow (env \times heap \times expr \quad \times ty \quad)set$
$WTrts' :: J\text{-}prog \Rightarrow (env \times heap \times expr\ list \times ty\ list)set$

translations
$P,E,h \vdash e :' T\ ==\ (E,h,e,T) \in WTrt'\ P$
$P,E,h \vdash es\ [:']\ Ts\ ==\ (E,h,es,Ts) \in WTrts'\ P$

inductive $WTrt'\ P\ WTrts'\ P$
intros
$is\text{-}class\ P\ C \Longrightarrow P,E,h \vdash new\ C :' Class\ C$
$[\![P,E,h \vdash e :' T;\ is\text{-}refT\ T;\ is\text{-}class\ P\ C]\!]$
$\Longrightarrow P,E,h \vdash Cast\ C\ e :' Class\ C$
$typeof_h\ v = Some\ T \Longrightarrow P,E,h \vdash Val\ v :' T$
$E\ v = Some\ T \Longrightarrow P,E,h \vdash Var\ v :' T$
$[\![P,E,h \vdash e_1 :' T_1;\ P,E,h \vdash e_2 :' T_2;$
$case\ bop\ of\ Eq \Rightarrow T' = Boolean$
$\quad |\ Add \Rightarrow T_1 = Integer \wedge T_2 = Integer \wedge T' = Integer]\!]$
$\Longrightarrow P,E,h \vdash e_1 \ll bop \gg e_2 :' T'$
$[\![P,E,h \vdash Var\ V :' T;\ P,E,h \vdash e :' T';\ P \vdash T' \leq T\ (*\ V \neq This*)]\!]$
$\Longrightarrow P,E,h \vdash V{:=}e :' Void$
$[\![P,E,h \vdash e :' Class\ C;\ P \vdash C\ has\ F{:}T\ in\ D]\!] \Longrightarrow P,E,h \vdash e{\cdot}F\{D\} :' T$
$P,E,h \vdash e :' NT \Longrightarrow P,E,h \vdash e{\cdot}F\{D\} :' T$
$[\![P,E,h \vdash e_1 :' Class\ C;\ P \vdash C\ has\ F{:}T\ in\ D;$
$P,E,h \vdash e_2 :' T_2;\ P \vdash T_2 \leq T]\!]$
$\Longrightarrow P,E,h \vdash e_1{\cdot}F\{D\}{:=}e_2 :' Void$
$[\![P,E,h \vdash e_1{:}'NT;\ P,E,h \vdash e_2 :' T_2]\!] \Longrightarrow P,E,h \vdash e_1{\cdot}F\{D\}{:=}e_2 :' Void$
$[\![P,E,h \vdash e :' Class\ C;\ P \vdash C\ sees\ M{:}Ts \rightarrow T = (pns,body)\ in\ D;$
$P,E,h \vdash es\ [:']\ Ts';\ P \vdash Ts'\ [\leq]\ Ts]\!]$
$\Longrightarrow P,E,h \vdash e{\cdot}M(es) :' T$
$[\![P,E,h \vdash e :' NT;\ P,E,h \vdash es\ [:']\ Ts]\!] \Longrightarrow P,E,h \vdash e{\cdot}M(es) :' T$
$P,E,h \vdash []\ [:']\ []$
$[\![P,E,h \vdash e :' T;\ P,E,h \vdash es\ [:']\ Ts]\!] \Longrightarrow P,E,h \vdash e\#es\ [:']\ T\#Ts$
$[\![typeof_h\ v = Some\ T_1;\ P \vdash T_1 \leq T;\ P,E(V \mapsto T),h \vdash e_2 :' T_2]\!]$
$\Longrightarrow P,E,h \vdash \{V{:}T := Val\ v;\ e_2\} :' T_2$
$[\![P,E(V \mapsto T),h \vdash e :' T';\ \neg\ assigned\ V\ e]\!] \Longrightarrow P,E,h \vdash \{V{:}T;\ e\} :' T'$
$[\![P,E,h \vdash e_1{:}' T_1;\ P,E,h \vdash e_2{:}'T_2]\!] \Longrightarrow P,E,h \vdash e_1;;e_2 :' T_2$
$[\![P,E,h \vdash e :' Boolean;\ P,E,h \vdash e_1{:}' T_1;\ P,E,h \vdash e_2{:}' T_2;$

$P \vdash T_1 \leq T_2 \vee P \vdash T_2 \leq T_1;$
$P \vdash T_1 \leq T_2 \longrightarrow T = T_2;\ P \vdash T_2 \leq T_1 \longrightarrow T = T_1 \,]\!]$
$\Longrightarrow P,E,h \vdash \mathit{if}\ (e)\ e_1\ \mathit{else}\ e_2 :' T$
$[\![\ P,E,h \vdash e :' \mathit{Boolean};\ \ P,E,h \vdash c :' T\ ]\!]$
$\Longrightarrow\ P,E,h \vdash \mathit{while}(e)\ c :' \mathit{Void}$
$[\![\ P,E,h \vdash e :' T_r;\ \mathit{is\text{-}refT}\ T_r\ ]\!] \Longrightarrow\ P,E,h \vdash \mathit{throw}\ e :' T$
$[\![\ P,E,h \vdash e_1 :' T_1;\ \ P,E(V \mapsto \mathit{Class}\ C),h \vdash e_2 :' T_2;\ P \vdash T_1 \leq T_2\ ]\!]$
$\Longrightarrow P,E,h \vdash \mathit{try}\ e_1\ \mathit{catch}(C\ V)\ e_2 :' T_2$

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

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

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

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

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

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

end

2.21 Well-formedness Constraints

theory *JWellForm* = *WellForm* + *WWellForm* + *WellType* + *DefAss*:

constdefs
$wf\text{-}J\text{-}mdecl :: J\text{-}prog \Rightarrow cname \Rightarrow J\text{-}mb\ mdecl \Rightarrow bool$
$wf\text{-}J\text{-}mdecl\ P\ C\ \equiv\ \lambda(M,Ts,T,(pns,body)).$
$length\ Ts = length\ pns\ \wedge$
$distinct\ pns\ \wedge$
$this \notin set\ pns\ \wedge$
$(\exists\, T'.\ P,[this\mapsto Class\ C,pns[\mapsto]Ts] \vdash body :: T' \wedge P \vdash T' \leq T) \wedge$
$\mathcal{D}\ body\ \lfloor\{this\} \cup set\ pns\rfloor$

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

syntax
wf-J-prog :: *J-prog* ⇒ *bool*
translations
wf-J-prog == *wf-prog wf-J-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

2.22 Type Safety Proof

theory *TypeSafe* = *Progress* + *JWellForm*:

2.22.1 Basic preservation lemmas

First two easy preservation lemmas.

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

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

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

Now preservation of definite assignment.

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

Combining conformance of heap and local variables:

constdefs
$sconf :: J\text{-}prog \Rightarrow env \Rightarrow state \Rightarrow bool\quad (\text{-},\text{-} \vdash \text{-} \surd\quad [51,51,51]50)$
$P,E \vdash s \surd\ \equiv\ let\ (h,l) = s\ in\ P \vdash h \surd \wedge P,h \vdash l\ (:\leq)_w\ E$

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

2.22.2 Subject reduction

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

theorem assumes *wf*: *wf-J-prog P*
shows *subject-reduction2*: $P \vdash \langle e,(h,l)\rangle \rightarrow \langle e',(h',l')\rangle \Longrightarrow$
$(\bigwedge E\ T.\ [\![\ P,E \vdash (h,l) \surd;\ P,E,h \vdash e{:}T\ ]\!]$
$\Longrightarrow \exists T'.\ P,E,h' \vdash e'{:}T' \wedge P \vdash T' \leq T)$

and *subjects-reduction2*: $P \vdash \langle es,(h,l)\rangle\ [\to]\ \langle es',(h',l')\rangle \Longrightarrow$
$(\bigwedge E\ Ts.\ [\![\ P,E \vdash (h,l)\ \surd;\ P,E,h \vdash es\ [:]\ Ts\ ]\!]$
$\Longrightarrow \exists\, Ts'.\ P,E,h' \vdash es'\ [:]\ Ts' \wedge P \vdash Ts'\ [\le]\ Ts)$

corollary *subject-reduction*:
$[\![\ wf\text{-}J\text{-}prog\ P;\ P \vdash \langle e,s\rangle \to \langle e',s'\rangle;\ P,E \vdash s\ \surd;\ P,E,hp\ s \vdash e{:}T\ ]\!]$
$\Longrightarrow \exists\, T'.\ P,E,hp\ s' \vdash e'{:}T' \wedge P \vdash T' \le T$
corollary *subjects-reduction*:
$[\![\ wf\text{-}J\text{-}prog\ P;\ P \vdash \langle es,s\rangle\ [\to]\ \langle es',s'\rangle;\ P,E \vdash s\ \surd;\ P,E,hp\ s \vdash es[:]Ts\ ]\!]$
$\Longrightarrow \exists\, Ts'.\ P,E,hp\ s' \vdash es'[:]Ts' \wedge P \vdash Ts'\ [\le]\ Ts$

2.22.3 Lifting to $\to*$

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

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

lemma *Red-preserves-defass*:
assumes *wf*: *wf-J-prog P* **and** *reds*: $P \vdash \langle e,s\rangle \to* \langle e',s'\rangle$
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-induct2*)
  case *refl* **thus** *?case* **.**
next
  case (*step e s e' s'*) **thus** *?case*
    by(*cases s*,*cases s'*)(*auto dest*:*red-preserves-defass*[*OF wf*])
qed

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

2.22.4 Lifting to $\Rightarrow$

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

lemma *eval-preserves-sconf*:
$[\![\ wf\text{-}J\text{-}prog\ P;\ P \vdash \langle e,s\rangle \Rightarrow \langle e',s'\rangle;\ P,E \vdash e{::}T;\ P,E \vdash s\surd\ ]\!] \Longrightarrow P,E \vdash s'\surd$

lemma *eval-preserves-type*: **assumes** *wf*: *wf-J-prog P*
shows $[\![\ P \vdash \langle e,s\rangle \Rightarrow \langle e',s'\rangle;\ P,E \vdash s\surd;\ P,E \vdash e{::}T\ ]\!]$
$\Longrightarrow \exists\, T'.\ P \vdash T' \le T \wedge P,E,hp\ s' \vdash e'{:}T'$

2.22.5 The final polish

The above preservation lemmas are now combined and packed nicely.

constdefs
  wf-config :: *J-prog* $\Rightarrow$ *env* $\Rightarrow$ *state* $\Rightarrow$ *expr* $\Rightarrow$ *ty* $\Rightarrow$ *bool* (-,-,- $\vdash$ - : - $\surd$ [*51,0,0,0,0*]*50*)
  $P,E,s \vdash e{:}T\ \surd\ \equiv\ P,E \vdash s\ \surd \wedge P,E,hp\ s \vdash e{:}T$

theorem *Subject-reduction*: **assumes** *wf*: *wf-J-prog P*

shows $P \vdash \langle e,s\rangle \to \langle e',s'\rangle \Longrightarrow P,E,s \vdash e : T \surd$
$\Longrightarrow \exists\, T'.\ P,E,s' \vdash e' : T' \surd \wedge P \vdash T' \leq T$

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

corollary *Progress*: **assumes** *wf*: *wf-J-prog P*
shows $[\![\ P,E,s \vdash e : T \surd;\ \mathcal{D}\ e\ \lfloor dom(lcl\ s)\rfloor;\ \neg\ \mathit{final}\ e\ ]\!] \Longrightarrow \exists\, e'\ s'.\ P \vdash \langle e,s\rangle \to \langle e',s'\rangle$

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

end

2.23 Program annotation

theory *Annotate* = *WellType*:

consts
$Anno :: J\text{-}prog \Rightarrow (env \times expr \times expr)\ set$
$Annos :: J\text{-}prog \Rightarrow (env \times expr\ list \times expr\ list)\ set$

translations
$P,E \vdash e \leadsto e' == (E,e,e') \in Anno\ P$
$P,E \vdash es\ [\leadsto]\ es' == (E,es,es') \in Annos\ P$

inductive *Anno P Annos P*
intros

$AnnoNew$: $is\text{-}class\ P\ C \implies P,E \vdash new\ C \leadsto new\ C$
$AnnoCast$: $P,E \vdash e \leadsto e' \implies P,E \vdash Cast\ C\ e \leadsto Cast\ C\ e'$
$AnnoVal$: $P,E \vdash Val\ v \leadsto Val\ v$
$AnnoVarVar$: $E\ V = \lfloor T \rfloor \implies P,E \vdash Var\ V \leadsto Var\ V$
$AnnoVarField$: $[\![E\ V = None;\ E\ this = \lfloor Class\ C \rfloor;\ P \vdash C\ sees\ V{:}T\ in\ D]\!]$
$\implies P,E \vdash Var\ V \leadsto Var\ this{\cdot}V\{D\}$
$AnnoBinOp$:
$[\![P,E \vdash e1 \leadsto e1';\ P,E \vdash e2 \leadsto e2']\!]$
$\implies P,E \vdash e1 \ll bop \gg e2 \leadsto e1' \ll bop \gg e2'$
$AnnoLAss$:
$P,E \vdash e \leadsto e' \implies P,E \vdash V{:=}e \leadsto V{:=}e'$
$AnnoFAcc$:
$[\![P,E \vdash e \leadsto e';\ P,E \vdash e' :: Class\ C;\ P \vdash C\ sees\ F{:}T\ in\ D]\!]$
$\implies P,E \vdash e{\cdot}F\{[]\} \leadsto e'{\cdot}F\{D\}$
$AnnoFAss$: $[\![P,E \vdash e1 \leadsto e1';\ P,E \vdash e2 \leadsto e2';$
$P,E \vdash e1' :: Class\ C;\ P \vdash C\ sees\ F{:}T\ in\ D]\!]$
$\implies P,E \vdash e1{\cdot}F\{[]\} := e2 \leadsto e1'{\cdot}F\{D\} := e2'$
$AnnoCall$:
$[\![P,E \vdash e \leadsto e';\ P,E \vdash es\ [\leadsto]\ es']\!]$
$\implies P,E \vdash Call\ e\ M\ es \leadsto Call\ e'\ M\ es'$
$AnnoBlock$:
$P,E(V \mapsto T) \vdash e \leadsto e' \implies P,E \vdash \{V{:}T;\ e\} \leadsto \{V{:}T;\ e'\}$
$AnnoComp$: $[\![P,E \vdash e1 \leadsto e1';\ P,E \vdash e2 \leadsto e2']\!]$
$\implies P,E \vdash e1;;e2 \leadsto e1';;e2'$
$AnnoCond$: $[\![P,E \vdash e \leadsto e';\ P,E \vdash e1 \leadsto e1';\ P,E \vdash e2 \leadsto e2']\!]$
$\implies P,E \vdash if\ (e)\ e1\ else\ e2 \leadsto if\ (e')\ e1'\ else\ e2'$
$AnnoLoop$: $[\![P,E \vdash e \leadsto e';\ P,E \vdash c \leadsto c']\!]$
$\implies P,E \vdash while\ (e)\ c \leadsto while\ (e')\ c'$
$AnnoThrow$: $P,E \vdash e \leadsto e' \implies P,E \vdash throw\ e \leadsto throw\ e'$
$AnnoTry$: $[\![P,E \vdash e1 \leadsto e1';\ P,E(V \mapsto Class\ C) \vdash e2 \leadsto e2']\!]$
$\implies P,E \vdash try\ e1\ catch(C\ V)\ e2 \leadsto try\ e1'\ catch(C\ V)\ e2'$

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

end

Chapter 3

Jinja Virtual Machine

3.1 State of the JVM

theory *JVMState* = *Objects*:

3.1.1 Frame Stack

types
pc = *nat*

frame = *val list* × *val list* × *cname* × *mname* × *pc*
— operand stack
— registers (including this pointer, method parameters, and local variables)
— name of class where current method is defined
— parameter types
— program counter within frame

3.1.2 Runtime State

types
jvm-state = *addr option* × *heap* × *frame list*
— exception flag, heap, frames

end

3.2 Instructions of the JVM

theory *JVMInstructions* = *JVMState*:

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
| *Putfield vname cname* — Set field in object
| *Checkcast cname* — Check whether object is of given type
| *Invoke mname nat* — inv. instance meth of an object
| *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

types
bytecode = *instr list*

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

ex-table = *ex-entry list*

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

jvm-prog = *jvm-method prog*

end

3.3 JVM Instruction Semantics

theory *JVMExecInstr* = *JVMInstructions* + *JVMState* + *Exceptions*:

consts
exec-instr :: [*instr*, *jvm-prog*, *heap*, *val list*, *val list*,
cname, *mname*, *pc*, *frame list*] => *jvm-state*
primrec
exec-instr-Load:
exec-instr (*Load n*) *P h stk loc* C_0 M_0 *pc frs* =
(*None*, *h*, ((*loc* ! *n*) # *stk*, *loc*, C_0, M_0, *pc+1*)#*frs*)

exec-instr (*Store n*) *P h stk loc* C_0 M_0 *pc frs* =
(*None*, *h*, (*tl stk*, *loc*[*n*:=*hd stk*], C_0, M_0, *pc+1*)#*frs*)

exec-instr-Push:
exec-instr (*Push v*) *P h stk loc* C_0 M_0 *pc frs* =
(*None*, *h*, (*v* # *stk*, *loc*, C_0, M_0, *pc+1*)#*frs*)

exec-instr-New:
exec-instr (*New C*) *P h stk loc* C_0 M_0 *pc frs* =
(*case new-Addr h of*
None ⇒ (*Some* (*addr-of-sys-xcpt OutOfMemory*), *h*, (*stk*, *loc*, C_0, M_0, *pc*)#*frs*)
| *Some a* ⇒ (*None*, *h*(*a*↦*blank P C*), (*Addr a*#*stk*, *loc*, C_0, M_0, *pc+1*)#*frs*))

exec-instr (*Getfield F C*) *P h stk loc* C_0 M_0 *pc frs* =
(*let v* = *hd stk*;
xp′ = *if v=Null then* ⌊*addr-of-sys-xcpt NullPointer*⌋ *else None*;
(*D*,*fs*) = *the*(*h*(*the-Addr v*))
in (*xp′*, *h*, (*the*(*fs*(*F*,*C*))#(*tl stk*), *loc*, C_0, M_0, *pc+1*)#*frs*))

exec-instr (*Putfield F C*) *P h stk loc* C_0 M_0 *pc frs* =
(*let v* = *hd stk*;
r = *hd* (*tl stk*);
xp′ = *if r=Null then* ⌊*addr-of-sys-xcpt NullPointer*⌋ *else None*;
a = *the-Addr r*;
(*D*,*fs*) = *the* (*h a*);
h′ = *h*(*a* ↦ (*D*, *fs*((*F*,*C*) ↦ *v*)))
in (*xp′*, *h′*, (*tl* (*tl stk*), *loc*, C_0, M_0, *pc+1*)#*frs*))

exec-instr (*Checkcast C*) *P h stk loc* C_0 M_0 *pc frs* =
(*let v* = *hd stk*;
xp′ = *if* ¬*cast-ok P C h v then* ⌊*addr-of-sys-xcpt ClassCast*⌋ *else None*
in (*xp′*, *h*, (*stk*, *loc*, C_0, M_0, *pc+1*)#*frs*))

exec-instr-Invoke:
exec-instr (*Invoke M n*) *P h stk loc* C_0 M_0 *pc frs* =
(*let ps* = *take n stk*;
r = *stk*!*n*;
xp′ = *if r=Null then* ⌊*addr-of-sys-xcpt NullPointer*⌋ *else None*;
C = *fst*(*the*(*h*(*the-Addr r*)));
(*D*,*M′*,*Ts*,*mxs*,mxl_0,*ins*,*xt*)= *method P C M*;
f′ = ([],[*r*]@(*rev ps*)@(*replicate* mxl_0 *arbitrary*),*D*,*M*,*0*)
in (*xp′*, *h*, *f′*#(*stk*, *loc*, C_0, M_0, *pc*)#*frs*))

exec-instr Return P h stk_0 loc_0 C_0 M_0 *pc frs* =
(*if frs*=[] *then* (*None*, *h*, []) *else*
let v = *hd* stk_0;
(*stk*,*loc*,*C*,*m*,*pc*) = *hd frs*;
n = *length* (*fst* (*snd* (*method P* C_0 M_0)))
in (*None*, *h*, (*v*#(*drop* (*n*+*1*) *stk*),*loc*,*C*,*m*,*pc*+*1*)#*tl frs*))

exec-instr Pop P h stk loc C_0 M_0 *pc frs* =
(*None*, *h*, (*tl stk*, *loc*, C_0, M_0, *pc*+*1*)#*frs*)

exec-instr IAdd P h stk loc C_0 M_0 *pc frs* =
(*let* i_2 = *the-Intg* (*hd stk*);
i_1 = *the-Intg* (*hd* (*tl stk*))
in (*None*, *h*, (*Intg* (i_1+i_2)#(*tl* (*tl stk*)), *loc*, C_0, M_0, *pc*+*1*)#*frs*))

exec-instr (*IfFalse i*) *P h stk loc* C_0 M_0 *pc frs* =
(*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')#*frs*))

exec-instr CmpEq P h stk loc C_0 M_0 *pc frs* =
(*let* v_2 = *hd stk*;
v_1 = *hd* (*tl stk*)
in (*None*, *h*, (*Bool* (v_1=v_2) # *tl* (*tl stk*), *loc*, C_0, M_0, *pc*+*1*)#*frs*))

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

exec-instr Throw P h stk loc C_0 M_0 *pc frs* =
(*let* xp' = *if hd stk* = *Null then* $\lfloor$*addr-of-sys-xcpt NullPointer*$\rfloor$ *else* $\lfloor$*the-Addr*(*hd stk*)$\rfloor$
in (xp', *h*, (*stk*, *loc*, C_0, M_0, *pc*)#*frs*))

lemma *exec-instr-Store*:
exec-instr (*Store n*) *P h* (*v*#*stk*) *loc* C_0 M_0 *pc frs* =
(*None*, *h*, (*stk*, *loc*[*n*:=*v*], C_0, M_0, *pc*+*1*)#*frs*)
by *simp*

lemma *exec-instr-Getfield*:
exec-instr (*Getfield F C*) *P h* (*v*#*stk*) *loc* C_0 M_0 *pc frs* =
(*let* xp' = *if v*=*Null then* $\lfloor$*addr-of-sys-xcpt NullPointer*$\rfloor$ *else None*;
(*D*,*fs*) = *the*(*h*(*the-Addr v*))
in (xp', *h*, (*the*(*fs*(*F*,*C*))#*stk*, *loc*, C_0, M_0, *pc*+*1*)#*frs*))
by *simp*

lemma *exec-instr-Putfield*:
exec-instr (*Putfield F C*) *P h* (*v*#*r*#*stk*) *loc* C_0 M_0 *pc frs* =
(*let* xp' = *if r*=*Null then* $\lfloor$*addr-of-sys-xcpt NullPointer*$\rfloor$ *else None*;
a = *the-Addr r*;
(*D*,*fs*) = *the* (*h a*);
h' = *h*(*a* $\mapsto$ (*D*, *fs*((*F*,*C*) $\mapsto$ *v*)))
in (xp', h', (*stk*, *loc*, C_0, M_0, *pc*+*1*)#*frs*))
by *simp*

lemma *exec-instr-Checkcast*:
$exec\text{-}instr\ (Checkcast\ C)\ P\ h\ (v\#stk)\ loc\ C_0\ M_0\ pc\ frs =$
$(let\ xp' = if\ \neg cast\text{-}ok\ P\ C\ h\ v\ then\ \lfloor addr\text{-}of\text{-}sys\text{-}xcpt\ ClassCast \rfloor\ else\ None$
$in\ (xp',\ h,\ (v\#stk,\ loc,\ C_0,\ M_0,\ pc+1)\#frs))$
by *simp*

lemma *exec-instr-Return*:
$exec\text{-}instr\ Return\ P\ h\ (v\#stk_0)\ loc_0\ C_0\ M_0\ pc\ frs =$
$(if\ frs=[]\ then\ (None,\ h,\ [])\ else$
$let\ (stk,loc,C,m,pc) = hd\ frs;$
$n = length\ (fst\ (snd\ (method\ P\ C_0\ M_0)))$
$in\ (None,\ h,\ (v\#(drop\ (n+1)\ stk),loc,C,m,pc+1)\#tl\ frs))$
by *simp*

lemma *exec-instr-IPop*:
$exec\text{-}instr\ Pop\ P\ h\ (v\#stk)\ loc\ C_0\ M_0\ pc\ frs =$
$(None,\ h,\ (stk,\ loc,\ C_0,\ M_0,\ pc+1)\#frs)$
by *simp*

lemma *exec-instr-IAdd*:
$exec\text{-}instr\ IAdd\ P\ h\ (Intg\ i_2\ \#\ Intg\ i_1\ \#\ stk)\ loc\ C_0\ M_0\ pc\ frs =$
$(None,\ h,\ (Intg\ (i_1+i_2)\#stk,\ loc,\ C_0,\ M_0,\ pc+1)\#frs)$
by *simp*

lemma *exec-instr-IfFalse*:
$exec\text{-}instr\ (IfFalse\ i)\ P\ h\ (v\#stk)\ loc\ C_0\ M_0\ pc\ frs =$
$(let\ pc' = if\ v = Bool\ False\ then\ nat(int\ pc+i)\ else\ pc+1$
$in\ (None,\ h,\ (stk,\ loc,\ C_0,\ M_0,\ pc')\#frs))$
by *simp*

lemma *exec-instr-CmpEq*:
$exec\text{-}instr\ CmpEq\ P\ h\ (v_2\#v_1\#stk)\ loc\ C_0\ M_0\ pc\ frs =$
$(None,\ h,\ (Bool\ (v_1=v_2)\ \#\ stk,\ loc,\ C_0,\ M_0,\ pc+1)\#frs)$
by *simp*

lemma *exec-instr-Throw*:
$exec\text{-}instr\ Throw\ P\ h\ (v\#stk)\ loc\ C_0\ M_0\ pc\ frs =$
$(let\ xp' = if\ v = Null\ then\ \lfloor addr\text{-}of\text{-}sys\text{-}xcpt\ NullPointer \rfloor\ else\ \lfloor the\text{-}Addr\ v \rfloor$
$in\ (xp',\ h,\ (v\#stk,\ loc,\ C_0,\ M_0,\ pc)\#frs))$
by *simp*

end

3.4 Exception handling in the JVM

```
theory JVMExceptions = JVMInstructions + Exceptions:

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


consts
  match-ex-table :: 'm prog ⇒ cname ⇒ pc ⇒ ex-table ⇒ (pc × nat) option
primrec
  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)

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


consts
  find-handler :: jvm-prog ⇒ addr ⇒ heap ⇒ frame list ⇒ jvm-state
primrec
  find-handler P a h [] = (Some a, h, [])
  find-handler P a h (fr#frs) =
      (let (stk,loc,C,M,pc) = fr in
       case match-ex-table P (cname-of h a) pc (ex-table-of P C M) of
         None ⇒ find-handler P a h frs
       | Some pc-d ⇒ (None, h, (Addr a # drop (size stk − snd pc-d) stk, loc, C, M, fst pc-d)#frs))

end
```

3.5 Program Execution in the JVM

theory *JVMExec* = *JVMExecInstr* + *JVMExceptions*:

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

— single step execution:
consts
  exec :: *jvm-prog* × *jvm-state* => *jvm-state option*
recdef *exec* {}
  exec (*P*, *xp*, *h*, []) = *None*

  exec (*P*, *None*, *h*, (*stk*,*loc*,*C*,*M*,*pc*)#*frs*) =
  (*let*
    i = *instrs-of P C M* ! *pc*;
    (*xcpt′*, *h′*, *frs′*) = *exec-instr i P h stk loc C M pc frs*
   in Some(*case xcpt′ of*
      None ⇒ (*None*,*h′*,*frs′*)
     | *Some a* ⇒ *find-handler P a h* ((*stk*,*loc*,*C*,*M*,*pc*)#*frs*)))

  exec (*P*, *Some xa*, *h*, *frs*) = *None*

lemma [*simp*]: *exec* (*P*, *x*, *h*, []) = *None*
by(*cases x*) *simp*+

— relational view
consts
  exec-1 :: *jvm-prog* ⇒ (*jvm-state* × *jvm-state*) *set*
syntax
  @*exec-1* :: *jvm-prog* ⇒ *jvm-state* ⇒ *jvm-state* ⇒ *bool*
  (- |−/ - −*jvm*−>*1*/ - [*61*,*61*,*61*] *60*)
syntax (*xsymbols*)
  @*exec-1* :: *jvm-prog* ⇒ *jvm-state* ⇒ *jvm-state* ⇒ *bool*
  (- ⊢/ - −*jvm*→$_1$/ - [*61*,*61*,*61*] *60*)
translations
  $P \vdash \sigma$ −*jvm*→$_1$ σ' == $(\sigma,\sigma') \in$ *exec-1 P*
inductive *exec-1 P* **intros**
  exec-1I: *exec* (P,σ) = *Some* $\sigma' \Longrightarrow P \vdash \sigma$ −*jvm*→$_1$ σ'

— reflexive transitive closure:
consts
  exec-all :: *jvm-prog* ⇒ *jvm-state* ⇒ *jvm-state* ⇒ *bool*
      (- |−/ - −*jvm*−>/ - [*61*,*61*,*61*]*60*)

syntax (*xsymbols*)
  exec-all :: *jvm-prog* ⇒ *jvm-state* ⇒ *jvm-state* ⇒ *bool*
      ((- ⊢/ - −*jvm*→/ -) [*61*,*61*,*61*]*60*)
defs
  exec-all-def1: $P \vdash \sigma$ −*jvm*→ $\sigma' \equiv (\sigma,\sigma') \in$ (*exec-1 P*)*

lemma *exec-1-def*:
$exec\text{-}1\ P = \{(\sigma,\sigma').\ exec\ (P,\sigma) = Some\ \sigma'\}$
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-all-finalD*: $P \vdash (x,\ h,\ [])\ -jvm\rightarrow\ \sigma \Longrightarrow \sigma = (x,\ h,\ [])$
lemma *exec-all-deterministic*:
$[\![\ P \vdash \sigma\ -jvm\rightarrow\ (x,h,[]);\ P \vdash \sigma\ -jvm\rightarrow\ \sigma'\ ]\!] \Longrightarrow P \vdash \sigma'\ -jvm\rightarrow\ (x,h,[])$

The start configuration of the JVM: in the start heap, we call a method m of class C in program P. The *this* pointer of the frame is set to *Null* to simulate a static method invokation.

constdefs
$start\text{-}state :: jvm\text{-}prog \Rightarrow cname \Rightarrow mname \Rightarrow jvm\text{-}state$
$start\text{-}state\ P\ C\ M \equiv$
$let\ (D,Ts,T,mxs,mxl_0,b) = method\ P\ C\ M\ in$
$(None,\ start\text{-}heap\ P,\ [([],\ Null\ \#\ replicate\ mxl_0\ arbitrary,\ C,\ M,\ 0)])$

end

3.6 A Defensive JVM

theory *JVMDefensive* = *JVMExec* + *Conform*:

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

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

consts *is-Addr* :: *val* ⇒ *bool*
recdef *is-Addr* {}
is-Addr (*Addr a*) = *True*
is-Addr v = *False*

consts *is-Intg* :: *val* ⇒ *bool*
recdef *is-Intg* {}
is-Intg (*Intg i*) = *True*
is-Intg v = *False*

consts *is-Bool* :: *val* ⇒ *bool*
recdef *is-Bool* {}
is-Bool (*Bool b*) = *True*
is-Bool v = *False*

constdefs
is-Ref :: *val* ⇒ *bool*
is-Ref v ≡ *v* = *Null* ∨ *is-Addr v*

constdefs
has- :: *'c prog* ⇒ *cname* ⇒ *mname* ⇒ *bool* (- ⊢ - *has* -)
P ⊢ *C has M* ≡ ∃ *Ts T m D*. *P* ⊢ *C sees M*:*Ts*→*T* = *m in D*

consts
check-instr :: [*instr*, *jvm-prog*, *heap*, *val list*, *val list*,
cname, *mname*, *pc*, *frame list*] ⇒ *bool*
primrec
check-instr-Load:
check-instr (*Load n*) *P hp stk loc C* M_0 *pc frs* =
(*n* < *length loc*)

check-instr-Store:
check-instr (*Store n*) *P hp stk loc* C_0 M_0 *pc frs* =
(*0* < *length stk* ∧ *n* < *length loc*)

check-instr-Push:
check-instr (*Push v*) *P hp stk loc* C_0 M_0 *pc frs* =
(¬*is-Addr v*)

check-instr-New:
check-instr (*New C*) *P hp stk loc* C_0 M_0 *pc frs* =
is-class P C

check-instr-Getfield:
check-instr (*Getfield F C*) *P hp stk loc* C_0 M_0 *pc frs* =

$(0 < length\ stk \land (\exists C'\ T.\ P \vdash C\ sees\ F{:}T\ in\ C') \land$
$(let\ (C',\ T) = field\ P\ C\ F;\ ref = hd\ stk\ in$
$C' = C \land is\text{-}Ref\ ref \land (ref \neq Null \longrightarrow$
$hp\ (the\text{-}Addr\ ref) \neq None \land$
$(let\ (D,vs) = the\ (hp\ (the\text{-}Addr\ ref))\ in$
$P \vdash D \preceq^* C \land vs\ (F,C) \neq None \land P,hp \vdash the\ (vs\ (F,C)) :\leq T))))$

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

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

check-instr-Invoke:
$check\text{-}instr\ (Invoke\ M\ n)\ P\ hp\ stk\ loc\ C_0\ M_0\ pc\ frs =$
$(n < length\ stk \land is\text{-}Ref\ (stk!n) \land$
$(stk!n \neq Null \longrightarrow$
$(let\ a = the\text{-}Addr\ (stk!n);$
$C = cname\text{-}of\ hp\ a;$
$Ts = fst\ (snd\ (method\ P\ C\ M))$
$in\ hp\ a \neq None \land P \vdash C\ has\ M \land$
$P,hp \vdash rev\ (take\ n\ stk)\ [:\leq]\ Ts)))$

check-instr-Return:
$check\text{-}instr\ Return\ P\ hp\ stk\ loc\ C_0\ M_0\ pc\ frs =$
$(0 < length\ stk \land ((0 < length\ frs) \longrightarrow$
$(P \vdash C_0\ has\ M_0) \land$
$(let\ v = hd\ stk;$
$T = fst\ (snd\ (snd\ (method\ P\ C_0\ M_0)))$
$in\ P,hp \vdash v :\leq T)))$

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

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

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

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

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

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

constdefs
$check :: jvm\text{-}prog \Rightarrow jvm\text{-}state \Rightarrow bool$
$check\ P\ \sigma \equiv let\ (xcpt,\ hp,\ frs) = \sigma\ in$
$(case\ frs\ of\ [] \Rightarrow True \mid (stk,loc,C,M,pc)\#frs' \Rightarrow$
$(let\ (C',Ts,T,mxs,mxl_0,ins,xt) = method\ P\ C\ M;\ i = ins!pc\ in$
$pc < size\ ins \wedge size\ stk \le mxs \wedge$
$check\text{-}instr\ i\ P\ hp\ stk\ loc\ C\ M\ pc\ frs'))$

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

consts
$exec\text{-}1\text{-}d :: jvm\text{-}prog \Rightarrow (jvm\text{-}state\ type\text{-}error \times jvm\text{-}state\ type\text{-}error)\ set$
syntax (*xsymbols*)
$@exec\text{-}1\text{-}d :: jvm\text{-}prog \Rightarrow jvm\text{-}state\ type\text{-}error \Rightarrow jvm\text{-}state\ type\text{-}error \Rightarrow bool$
$(\text{-} \vdash \text{-} -jvmd\rightarrow_1 \text{-}\ [61,61,61]60)$
translations
$P \vdash \sigma\ -jvmd\rightarrow_1\ \sigma' == (\sigma,\sigma') \in exec\text{-}1\text{-}d\ P$
inductive *exec-1-d P* **intros**
$exec\text{-}1\text{-}d\text{-}ErrorI$: $exec\text{-}d\ P\ \sigma = TypeError \Longrightarrow P \vdash Normal\ \sigma\ -jvmd\rightarrow_1\ TypeError$
$exec\text{-}1\text{-}d\text{-}NormalI$: $exec\text{-}d\ P\ \sigma = Normal\ (Some\ \sigma') \Longrightarrow P \vdash Normal\ \sigma\ -jvmd\rightarrow_1\ Normal\ \sigma'$

— reflexive transitive closure:
consts
$exec\text{-}all\text{-}d :: jvm\text{-}prog \Rightarrow jvm\text{-}state\ type\text{-}error \Rightarrow jvm\text{-}state\ type\text{-}error \Rightarrow bool$
$(\text{-}\ |-\ \text{-}\ -jvmd->\ \text{-}\ [61,61,61]60)$
syntax (*xsymbols*)
$exec\text{-}all\text{-}d :: jvm\text{-}prog \Rightarrow jvm\text{-}state\ type\text{-}error \Rightarrow jvm\text{-}state\ type\text{-}error \Rightarrow bool$
$(\text{-} \vdash \text{-} -jvmd\rightarrow \text{-}\ [61,61,61]60)$
defs
$exec\text{-}all\text{-}d\text{-}def1$: $P \vdash \sigma\ -jvmd\rightarrow\ \sigma' \equiv (\sigma,\sigma') \in (exec\text{-}1\text{-}d\ P)^*$

lemma *exec-1-d-def*:
$exec\text{-}1\text{-}d\ P = \{(s,t).\ \exists\sigma.\ s = Normal\ \sigma \wedge t = TypeError \wedge exec\text{-}d\ P\ \sigma = TypeError\} \cup$
$\{(s,t).\ \exists\sigma\ \sigma'.\ s = Normal\ \sigma \wedge t = Normal\ \sigma' \wedge exec\text{-}d\ P\ \sigma = Normal\ (Some\ \sigma')\}$
by (*auto elim*!: *exec-1-d.elims intro*!: *exec-1-d.intros*)

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

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

lemma *exec-d-no-errorI* [*intro*]:
$check\ P\ \sigma \Longrightarrow exec\text{-}d\ P\ \sigma \neq TypeError$
by (*unfold exec-d-def*) *simp*

theorem *no-type-error-commutes*:
$exec\text{-}d\ P\ \sigma \neq TypeError \Longrightarrow exec\text{-}d\ P\ \sigma = Normal\ (exec\ (P,\ \sigma))$
by (*unfold exec-d-def*, *auto*)

lemma *defensive-imp-aggressive*:
$P \vdash (Normal\ \sigma)\ -jvmd\rightarrow\ (Normal\ \sigma') \Longrightarrow P \vdash \sigma\ -jvm\rightarrow\ \sigma'$
end

Chapter 4

Bytecode Verifier

4.1 Semilattices

theory *Semilat* = *While-Combinator*:

types
$'a$ ord $= 'a \Rightarrow 'a \Rightarrow bool$
$'a$ $binop$ $= 'a \Rightarrow 'a \Rightarrow 'a$
$'a$ sl $= 'a\ set \times 'a\ ord \times 'a\ binop$

consts
$lesub :: 'a \Rightarrow 'a\ ord \Rightarrow 'a \Rightarrow bool$
$lesssub :: 'a \Rightarrow 'a\ ord \Rightarrow 'a \Rightarrow bool$
$plussub :: 'a \Rightarrow ('a \Rightarrow 'b \Rightarrow 'c) \Rightarrow 'b \Rightarrow 'c$
syntax (*xsymbols*)
$lesub :: 'a \Rightarrow 'a\ ord \Rightarrow 'a \Rightarrow bool$ ((- /$\sqsubseteq_-$ -) [50, 0, 51] 50)
$lesssub :: 'a \Rightarrow 'a\ ord \Rightarrow 'a \Rightarrow bool$ ((- /$\sqsubset_-$ -) [50, 0, 51] 50)
$plussub :: 'a \Rightarrow ('a \Rightarrow 'b \Rightarrow 'c) \Rightarrow 'b \Rightarrow 'c$ ((- /$\sqcup_-$ -) [65, 0, 66] 65)
defs
lesub-def: $x \sqsubseteq_r y \equiv r\ x\ y$
lesssub-def: $x \sqsubset_r y \equiv x \sqsubseteq_r y \wedge x \neq y$
plussub-def: $x \sqcup_f y \equiv f\ x\ y$

constdefs
$ord :: ('a \times 'a)\ set \Rightarrow 'a\ ord$
$ord\ r \equiv \lambda x\ y.\ (x,y) \in r$

$order :: 'a\ ord \Rightarrow bool$
$order\ r \equiv (\forall x.\ x \sqsubseteq_r x) \wedge (\forall x\ y.\ x \sqsubseteq_r y \wedge y \sqsubseteq_r x \longrightarrow x{=}y) \wedge (\forall x\ y\ z.\ x \sqsubseteq_r y \wedge y \sqsubseteq_r z \longrightarrow x \sqsubseteq_r z)$

$top :: 'a\ ord \Rightarrow 'a \Rightarrow bool$
$top\ r\ T \equiv \forall x.\ x \sqsubseteq_r T$

$acc :: 'a\ ord \Rightarrow bool$
$acc\ r \equiv wf\ \{(y,x).\ x \sqsubset_r y\}$

$closed :: 'a\ set \Rightarrow 'a\ binop \Rightarrow bool$
$closed\ A\ f \equiv \forall x{\in}A.\ \forall y{\in}A.\ x \sqcup_f y \in A$

$semilat :: 'a\ sl \Rightarrow bool$
$semilat \equiv \lambda(A,r,f).\ order\ r \wedge closed\ A\ f \wedge$
$(\forall x{\in}A.\ \forall y{\in}A.\ x \sqsubseteq_r x \sqcup_f y) \wedge$
$(\forall x{\in}A.\ \forall y{\in}A.\ y \sqsubseteq_r x \sqcup_f y) \wedge$
$(\forall x{\in}A.\ \forall y{\in}A.\ \forall z{\in}A.\ x \sqsubseteq_r z \wedge y \sqsubseteq_r z \longrightarrow x \sqcup_f y \sqsubseteq_r z)$

$is\text{-}ub :: ('a \times 'a)\ set \Rightarrow 'a \Rightarrow 'a \Rightarrow 'a \Rightarrow bool$
$is\text{-}ub\ r\ x\ y\ u \equiv (x,u){\in}r \wedge (y,u){\in}r$

$is\text{-}lub :: ('a \times 'a)\ set \Rightarrow 'a \Rightarrow 'a \Rightarrow 'a \Rightarrow bool$
$is\text{-}lub\ r\ x\ y\ u \equiv is\text{-}ub\ r\ x\ y\ u \wedge (\forall z.\ is\text{-}ub\ r\ x\ y\ z \longrightarrow (u,z){\in}r)$

$some\text{-}lub :: ('a \times 'a)\ set \Rightarrow 'a \Rightarrow 'a \Rightarrow 'a$
$some\text{-}lub\ r\ x\ y \equiv SOME\ z.\ is\text{-}lub\ r\ x\ y\ z$

locale (**open**) *semilat* =
 fixes A :: $'a$ *set*
 fixes r :: $'a$ *ord*
 fixes f :: $'a$ *binop*
 assumes *semilat*: $semilat(A,r,f)$

lemma *order-refl* [*simp*, *intro*]: $order\ r \Longrightarrow x \sqsubseteq_r x$

lemma *order-antisym*: $[\![\ order\ r;\ x \sqsubseteq_r y;\ y \sqsubseteq_r x\ ]\!] \Longrightarrow x = y$

lemma *order-trans*: $[\![\ order\ r;\ x \sqsubseteq_r y;\ y \sqsubseteq_r z\ ]\!] \Longrightarrow x \sqsubseteq_r z$

lemma *order-less-irrefl* [*intro*, *simp*]: $order\ r \Longrightarrow \neg\ x \sqsubset_r x$

lemma *order-less-trans*: $[\![\ order\ r;\ x \sqsubset_r y;\ y \sqsubset_r z\ ]\!] \Longrightarrow x \sqsubset_r z$

lemma *topD* [*simp*, *intro*]: $top\ r\ T \Longrightarrow x \sqsubseteq_r T$

lemma *top-le-conv* [*simp*]: $[\![\ order\ r;\ top\ r\ T\ ]\!] \Longrightarrow (T \sqsubseteq_r x) = (x = T)$

lemma *semilat-Def*:
$semilat(A,r,f) \equiv order\ r \wedge closed\ A\ f \wedge$
$\quad(\forall x{\in}A.\ \forall y{\in}A.\ x \sqsubseteq_r x \sqcup_f y) \wedge$
$\quad(\forall x{\in}A.\ \forall y{\in}A.\ y \sqsubseteq_r x \sqcup_f y) \wedge$
$\quad(\forall x{\in}A.\ \forall y{\in}A.\ \forall z{\in}A.\ x \sqsubseteq_r z \wedge y \sqsubseteq_r z \longrightarrow x \sqcup_f y \sqsubseteq_r z)$

lemma (**in** *semilat*) *orderI* [*simp*, *intro*]: $order\ r$

lemma (**in** *semilat*) *closedI* [*simp*, *intro*]: $closed\ A\ f$

lemma *closedD*: $[\![\ closed\ A\ f;\ x{\in}A;\ y{\in}A\ ]\!] \Longrightarrow x \sqcup_f y \in A$

lemma *closed-UNIV* [*simp*]: $closed\ UNIV\ f$

lemma (**in** *semilat*) *closed-f* [*simp*, *intro*]: $[\![x \in A;\ y \in A]\!] \Longrightarrow x \sqcup_f y \in A$

lemma (**in** *semilat*) *refl-r* [*intro*, *simp*]: $x \sqsubseteq_r x$ **by** *simp*

lemma (**in** *semilat*) *antisym-r* [*intro?*]: $[\![\ x \sqsubseteq_r y;\ y \sqsubseteq_r x\ ]\!] \Longrightarrow x = y$

lemma (**in** *semilat*) *trans-r* [*trans*, *intro?*]: $[\![x \sqsubseteq_r y;\ y \sqsubseteq_r z]\!] \Longrightarrow x \sqsubseteq_r z$

lemma (**in** *semilat*) *ub1* [*simp*, *intro?*]: $[\![\ x \in A;\ y \in A\ ]\!] \Longrightarrow x \sqsubseteq_r x \sqcup_f y$

lemma (**in** *semilat*) *ub2* [*simp*, *intro?*]: $[\![\ x \in A;\ y \in A\ ]\!] \Longrightarrow y \sqsubseteq_r x \sqcup_f y$

lemma (**in** *semilat*) *lub* [*simp*, *intro?*]:
 $[\![\ x \sqsubseteq_r z;\ y \sqsubseteq_r z;\ x \in A;\ y \in A;\ z \in A\ ]\!] \Longrightarrow x \sqcup_f y \sqsubseteq_r z$

lemma (**in** *semilat*) *plus-le-conv* [*simp*]:
 $[\![\ x \in A;\ y \in A;\ z \in A\ ]\!] \Longrightarrow (x \sqcup_f y \sqsubseteq_r z) = (x \sqsubseteq_r z \wedge y \sqsubseteq_r z)$

lemma (**in** *semilat*) *le-iff-plus-unchanged*: $[\![\ x \in A;\ y \in A\ ]\!] \Longrightarrow (x \sqsubseteq_r y) = (x \sqcup_f y = y)$
lemma (**in** *semilat*) *le-iff-plus-unchanged2*: $[\![\ x \in A;\ y \in A\ ]\!] \Longrightarrow (x \sqsubseteq_r y) = (y \sqcup_f x = y)$

lemma (**in** *semilat*) *plus-assoc* [*simp*]:
assumes *a*: $a \in A$ **and** *b*: $b \in A$ **and** *c*: $c \in A$
shows $a \sqcup_f (b \sqcup_f c) = a \sqcup_f b \sqcup_f c$
lemma (**in** *semilat*) *plus-com-lemma*:
$[\![a \in A;\ b \in A]\!] \Longrightarrow a \sqcup_f b \sqsubseteq_r b \sqcup_f a$
lemma (**in** *semilat*) *plus-commutative*:
$[\![a \in A;\ b \in A]\!] \Longrightarrow a \sqcup_f b = b \sqcup_f a$

lemma *is-lubD*:
is-lub r x y u $\Longrightarrow$ *is-ub r x y u* $\wedge$ ($\forall$ *z*. *is-ub r x y z* $\longrightarrow$ (*u*,*z*) $\in$ *r*)

lemma *is-ubI*:
$[\![$ (*x*,*u*) $\in$ *r*; (*y*,*u*) $\in$ *r* $]\!] \Longrightarrow$ *is-ub r x y u*

lemma *is-ubD*:
is-ub r x y u $\Longrightarrow$ (*x*,*u*) $\in$ *r* $\wedge$ (*y*,*u*) $\in$ *r*

lemma *is-lub-bigger1* [*iff*]:
is-lub (*r*ˆ∗) *x y y* = ((*x*,*y*)∈*r*ˆ∗)
lemma *is-lub-bigger2* [*iff*]:
is-lub (*r*ˆ∗) *x y x* = ((*y*,*x*)∈*r*ˆ∗)
lemma *extend-lub*:
$[\![$ *single-valued r*; *is-lub* (*r*ˆ∗) *x y u*; (*x'*,*x*) $\in$ *r* $]\!]$
$\Longrightarrow$ *EX v*. *is-lub* (*r*ˆ∗) *x' y v*
lemma *single-valued-has-lubs* [*rule-format*]:
$[\![$ *single-valued r*; (*x*,*u*) $\in$ *r*ˆ∗ $]\!] \Longrightarrow$ ($\forall$ *y*. (*y*,*u*) $\in$ *r*ˆ∗ $\longrightarrow$
(*EX z*. *is-lub* (*r*ˆ∗) *x y z*))
lemma *some-lub-conv*:
$[\![$ *acyclic r*; *is-lub* (*r*ˆ∗) *x y u* $]\!] \Longrightarrow$ *some-lub* (*r*ˆ∗) *x y* = *u*
lemma *is-lub-some-lub*:
$[\![$ *single-valued r*; *acyclic r*; (*x*,*u*)∈*r*ˆ∗; (*y*,*u*)∈*r*ˆ∗ $]\!]$
$\Longrightarrow$ *is-lub* (*r*ˆ∗) *x y* (*some-lub* (*r*ˆ∗) *x y*)

4.1.1 An executable lub-finder

constdefs
exec-lub :: $('a * 'a)$ *set* $\Rightarrow ('a \Rightarrow 'a) \Rightarrow 'a$ *binop*
exec-lub r f x y $\equiv$ *while* $(\lambda z.\ (x,z) \notin r^*)$ *f y*

lemma *acyclic-single-valued-finite*:
$[\![$*acyclic r*; *single-valued r*; $(x,y) \in r^*]\!]$
$\Longrightarrow$ *finite* $(r \cap \{a.\ (x,\ a) \in r^*\} \times \{b.\ (b,\ y) \in r^*\})$

lemma *exec-lub-conv*:
$[\![$ *acyclic r*; $\forall x\ y.\ (x,y) \in r \longrightarrow f\ x = y$; *is-lub* (r^*) *x y u* $]\!] \Longrightarrow$
exec-lub r f x y = *u*
lemma *is-lub-exec-lub*:
$[\![$ *single-valued r*; *acyclic r*; (*x*,*u*):*r*ˆ∗; (*y*,*u*):*r*ˆ∗; $\forall x\ y.\ (x,y) \in r \longrightarrow f\ x = y$ $]\!]$

$\Longrightarrow$ *is-lub* $(r\hat{\ }*)$ x y (*exec-lub* r f x y)

end

4.2 The Error Type

theory *Err* = *Semilat*:

datatype $'a\ err = Err \mid OK\ 'a$

types $'a\ ebinop = 'a \Rightarrow 'a \Rightarrow 'a\ err$
types $'a\ esl = 'a\ set \times 'a\ ord \times 'a\ ebinop$

consts
  $ok\text{-}val :: 'a\ err \Rightarrow 'a$
primrec
  $ok\text{-}val\ (OK\ x) = x$

constdefs
  $lift :: ('a \Rightarrow 'b\ err) \Rightarrow ('a\ err \Rightarrow 'b\ err)$
  $lift\ f\ e \equiv case\ e\ of\ Err \Rightarrow Err \mid OK\ x \Rightarrow f\ x$

  $lift2 :: ('a \Rightarrow 'b \Rightarrow 'c\ err) \Rightarrow 'a\ err \Rightarrow 'b\ err \Rightarrow 'c\ err$
  $lift2\ f\ e_1\ e_2 \equiv$
  $case\ e_1\ of\ Err\ \Rightarrow Err \mid OK\ x \Rightarrow (case\ e_2\ of\ Err \Rightarrow Err \mid OK\ y \Rightarrow f\ x\ y)$

  $le :: 'a\ ord \Rightarrow 'a\ err\ ord$
  $le\ r\ e_1\ e_2 \equiv$
  $case\ e_2\ of\ Err \Rightarrow True \mid OK\ y \Rightarrow (case\ e_1\ of\ Err \Rightarrow False \mid OK\ x \Rightarrow x \sqsubseteq_r y)$

  $sup :: ('a \Rightarrow 'b \Rightarrow 'c) \Rightarrow ('a\ err \Rightarrow 'b\ err \Rightarrow 'c\ err)$
  $sup\ f \equiv lift2\ (\lambda x\ y.\ OK\ (x \sqcup_f y))$

  $err :: 'a\ set \Rightarrow 'a\ err\ set$
  $err\ A \equiv insert\ Err\ \{OK\ x|x.\ x{\in}A\}$

  $esl :: 'a\ sl \Rightarrow 'a\ esl$
  $esl \equiv \lambda(A,r,f).\ (A,\ r,\ \lambda x\ y.\ OK(f\ x\ y))$

  $sl :: 'a\ esl \Rightarrow 'a\ err\ sl$
  $sl \equiv \lambda(A,r,f).\ (err\ A,\ le\ r,\ lift2\ f)$

syntax
  $err\text{-}semilat :: 'a\ esl \Rightarrow bool$
translations
  $err\text{-}semilat\ L == semilat(Err.sl\ L)$

consts
  $strict\ :: ('a \Rightarrow 'b\ err) \Rightarrow ('a\ err \Rightarrow 'b\ err)$
primrec
  $strict\ f\ Err\quad = Err$
  $strict\ f\ (OK\ x) = f\ x$

lemma *err-def'*:
  $err\ A \equiv insert\ Err\ \{x.\ \exists y{\in}A.\ x = OK\ y\}$
lemma *strict-Some* [*simp*]:
  $(strict\ f\ x = OK\ y) = (\exists z.\ x = OK\ z \wedge f\ z = OK\ y)$
lemma *not-Err-eq*: $(x \neq Err) = (\exists a.\ x = OK\ a)$

lemma *not-OK-eq*: $(\forall y.\ x \neq OK\ y) = (x = Err)$
lemma *unfold-lesub-err*: $e1 \sqsubseteq_{le\ r}\ e2 \equiv le\ r\ e1\ e2$
lemma *le-err-refl*: $\forall x.\ x \sqsubseteq_r x \Longrightarrow e \sqsubseteq_{le\ r}\ e$
lemma *le-err-trans* [*rule-format*]:
$order\ r \Longrightarrow e1 \sqsubseteq_{le\ r}\ e2 \longrightarrow e2 \sqsubseteq_{le\ r}\ e3 \longrightarrow e1 \sqsubseteq_{le\ r}\ e3$
lemma *le-err-antisym* [*rule-format*]:
$order\ r \Longrightarrow e1 \sqsubseteq_{le\ r}\ e2 \longrightarrow e2 \sqsubseteq_{le\ r}\ e1 \longrightarrow e1{=}e2$
lemma *OK-le-err-OK*: $(OK\ x \sqsubseteq_{le\ r}\ OK\ y) = (x \sqsubseteq_r y)$
lemma *order-le-err* [*iff*]: $order(le\ r) = order\ r$
lemma *le-Err* [*iff*]: $e \sqsubseteq_{le\ r}\ Err$
lemma *Err-le-conv* [*iff*]: $Err \sqsubseteq_{le\ r}\ e\ = (e = Err)$
lemma *le-OK-conv* [*iff*]: $e \sqsubseteq_{le\ r}\ OK\ x\ =\ (\exists y.\ e = OK\ y \wedge y \sqsubseteq_r x)$
lemma *OK-le-conv*: $OK\ x \sqsubseteq_{le\ r}\ e = (e = Err \vee (\exists y.\ e = OK\ y \wedge x \sqsubseteq_r y))$
lemma *top-Err* [*iff*]: $top\ (le\ r)\ Err$
lemma *OK-less-conv* [*rule-format*, *iff*]:
$OK\ x \sqsubset_{le\ r}\ e = (e{=}Err \vee (\exists y.\ e = OK\ y \wedge x \sqsubset_r y))$
lemma *not-Err-less* [*rule-format*, *iff*]: $\neg(Err \sqsubset_{le\ r}\ x)$
lemma *semilat-errI* [*intro*]: **includes** *semilat*
shows $semilat(err\ A,\ le\ r,\ lift2(\lambda x\ y.\ OK(f\ x\ y)))$
lemma *err-semilat-eslI-aux*:
includes *semilat* **shows** $err\text{-}semilat(esl(A,r,f))$
lemma *err-semilat-eslI* [*intro*, *simp*]:
$\bigwedge L.\ semilat\ L \Longrightarrow err\text{-}semilat(esl\ L)$
lemma *acc-err* [*simp*, *intro*!]: $acc\ r \Longrightarrow acc(le\ r)$
lemma *Err-in-err* [*iff*]: $Err : err\ A$
lemma *Ok-in-err* [*iff*]: $(OK\ x \in err\ A) = (x \in A)$

4.2.1 lift

lemma *lift-in-errI*:
$[\![\ e \in err\ S;\ \forall x{\in}S.\ e = OK\ x \longrightarrow f\ x \in err\ S\ ]\!] \Longrightarrow lift\ f\ e \in err\ S$
lemma *Err-lift2* [*simp*]: $Err \sqcup_{lift2\ f}\ x = Err$
lemma *lift2-Err* [*simp*]: $x \sqcup_{lift2\ f}\ Err = Err$
lemma *OK-lift2-OK* [*simp*]: $OK\ x \sqcup_{lift2\ f}\ OK\ y = x \sqcup_f y$

4.2.2 sup

lemma *Err-sup-Err* [*simp*]: $Err \sqcup_{sup\ f}\ x = Err$
lemma *Err-sup-Err2* [*simp*]: $x \sqcup_{sup\ f}\ Err = Err$
lemma *Err-sup-OK* [*simp*]: $OK\ x \sqcup_{sup\ f}\ OK\ y = OK\ (x \sqcup_f y)$
lemma *Err-sup-eq-OK-conv* [*iff*]:
$(sup\ f\ ex\ ey = OK\ z) = (\exists x\ y.\ ex = OK\ x \wedge ey = OK\ y \wedge f\ x\ y = z)$
lemma *Err-sup-eq-Err* [*iff*]: $(sup\ f\ ex\ ey = Err) = (ex{=}Err \vee ey{=}Err)$

4.2.3 semilat (err A) (le r) f

lemma *semilat-le-err-Err-plus* [*simp*]:
$[\![\ x{\in}\ err\ A;\ semilat(err\ A,\ le\ r,\ f)\ ]\!] \Longrightarrow Err \sqcup_f x = Err$
lemma *semilat-le-err-plus-Err* [*simp*]:
$[\![\ x{\in}\ err\ A;\ semilat(err\ A,\ le\ r,\ f)\ ]\!] \Longrightarrow x \sqcup_f Err = Err$
lemma *semilat-le-err-OK1*:
$[\![\ x{\in}A;\ y{\in}A;\ semilat(err\ A,\ le\ r,\ f);\ OK\ x \sqcup_f OK\ y = OK\ z\ ]\!]$
$\Longrightarrow x \sqsubseteq_r z$
lemma *semilat-le-err-OK2*:

$[\![\ x{\in}A;\ y{\in}A;\ semilat(err\ A,\ le\ r,\ f);\ OK\ x \sqcup_f\ OK\ y = OK\ z\ ]\!]$
$\Longrightarrow y \sqsubseteq_r z$
lemma *eq-order-le*:
$[\![\ x{=}y;\ order\ r\ ]\!] \Longrightarrow x \sqsubseteq_r y$
lemma *OK-plus-OK-eq-Err-conv* [*simp*]:
$[\![\ x{\in}A;\ y{\in}A;\ semilat(err\ A,\ le\ r,\ fe)\ ]\!] \Longrightarrow$
$(OK\ x \sqcup_{fe}\ OK\ y = Err) = (\neg(\exists z{\in}A.\ x \sqsubseteq_r z \wedge y \sqsubseteq_r z))$

4.2.4 semilat (err(Union AS))

lemma *all-bex-swap-lemma* [*iff*]:
$(\forall x.\ (\exists y{\in}A.\ x = f\ y) \longrightarrow P\ x) = (\forall y{\in}A.\ P(f\ y))$
lemma *closed-err-Union-lift2I*:
$[\![\ \forall A{\in}AS.\ closed\ (err\ A)\ (lift2\ f);\ AS \neq \{\};$
$\quad \forall A{\in}AS.\forall B{\in}AS.\ A{\neq}B \longrightarrow (\forall a{\in}A.\forall b{\in}B.\ a \sqcup_f b = Err)\ ]\!]$
$\Longrightarrow closed\ (err(Union\ AS))\ (lift2\ f)$

If $AS = \{\}$ the thm collapses to $order\ r \wedge closed\ \{Err\}\ f \wedge Err \sqcup_f Err = Err$ which may not hold

lemma *err-semilat-UnionI*:
$[\![\ \forall A{\in}AS.\ err\text{-}semilat(A,\ r,\ f);\ AS \neq \{\};$
$\quad \forall A{\in}AS.\forall B{\in}AS.\ A{\neq}B \longrightarrow (\forall a{\in}A.\forall b{\in}B.\ \neg a \sqsubseteq_r b \wedge a \sqcup_f b = Err)\ ]\!]$
$\Longrightarrow err\text{-}semilat(Union\ AS,\ r,\ f)$
end

4.3 More about Options

theory *Opt* = *Err*:

constdefs
le :: $'a\ ord \Rightarrow 'a\ option\ ord$
$le\ r\ o_1\ o_2 \equiv$
$case\ o_2\ of\ None \Rightarrow o_1{=}None \mid Some\ y \Rightarrow (case\ o_1\ of\ None \Rightarrow True \mid Some\ x \Rightarrow x \sqsubseteq_r y)$

opt :: $'a\ set \Rightarrow 'a\ option\ set$
$opt\ A \equiv insert\ None\ \{Some\ y\ |y.\ y \in A\}$

sup :: $'a\ ebinop \Rightarrow 'a\ option\ ebinop$
$sup\ f\ o_1\ o_2 \equiv$
$case\ o_1\ of\ None \Rightarrow OK\ o_2$
$\quad \mid Some\ x \Rightarrow (case\ o_2\ of\ None \Rightarrow OK\ o_1$
$\quad\quad \mid Some\ y \Rightarrow (case\ f\ x\ y\ of\ Err \Rightarrow Err \mid OK\ z \Rightarrow OK\ (Some\ z)))$

esl :: $'a\ esl \Rightarrow 'a\ option\ esl$
$esl \equiv \lambda(A,r,f).\ (opt\ A,\ le\ r,\ sup\ f)$

lemma *unfold-le-opt*:
$o_1 \sqsubseteq_{le\ r} o_2 =$
$(case\ o_2\ of\ None \Rightarrow o_1{=}None \mid$
$\quad Some\ y \Rightarrow (case\ o_1\ of\ None \Rightarrow True \mid Some\ x \Rightarrow x \sqsubseteq_r y))$

lemma *le-opt-refl*: $order\ r \Longrightarrow x \sqsubseteq_{le\ r} x$

4.4 Products as Semilattices

theory *Product* = *Err*:

constdefs
$le :: 'a\ ord \Rightarrow 'b\ ord \Rightarrow ('a \times 'b)\ ord$
$le\ r_A\ r_B \equiv \lambda(a_1,b_1)\ (a_2,b_2).\ a_1 \sqsubseteq_{r_A} a_2 \wedge b_1 \sqsubseteq_{r_B} b_2$

$sup :: 'a\ ebinop \Rightarrow 'b\ ebinop \Rightarrow ('a \times 'b)\ ebinop$
$sup\ f\ g \equiv \lambda(a_1,b_1)(a_2,b_2).\ Err.sup\ Pair\ (a_1 \sqcup_f a_2)\ (b_1 \sqcup_g b_2)$

$esl :: 'a\ esl \Rightarrow 'b\ esl \Rightarrow ('a \times 'b\ )\ esl$
$esl \equiv \lambda(A,r_A,f_A)\ (B,r_B,f_B).\ (A \times B,\ le\ r_A\ r_B,\ sup\ f_A\ f_B)$

syntax (*xsymbols*)
$@lesubprod :: 'a \times 'b \Rightarrow 'a\ ord \Rightarrow 'b\ ord \Rightarrow 'b \Rightarrow bool$
((- /⊑'(-,-') -) [50, 0, 0, 51] 50)

translations $p \sqsubseteq(rA,rB)\ q == p \sqsubseteq_{Product.le\ rA\ rB}\ q$

lemma *unfold-lesub-prod*: $x \sqsubseteq(r_A,r_B)\ y \equiv le\ r_A\ r_B\ x\ y$
lemma *le-prod-Pair-conv* [*iff*]: $((a_1,b_1) \sqsubseteq(r_A,r_B)\ (a_2,b_2)) = (a_1 \sqsubseteq_{r_A} a_2\ \&\ b_1 \sqsubseteq_{r_B} b_2)$
lemma *less-prod-Pair-conv*:
$((a_1,b_1) \sqsubset_{Product.le\ r_A\ r_B} (a_2,b_2)) =$
$(a_1 \sqsubset_{r_A} a_2\ \&\ b_1 \sqsubseteq_{r_B} b_2 \mid a_1 \sqsubseteq_{r_A} a_2\ \&\ b_1 \sqsubset_{r_B} b_2)$
lemma *order-le-prod* [*iff*]: $order(Product.le\ r_A\ r_B) = (order\ r_A\ \&\ order\ r_B)$

lemma *acc-le-prodI* [*intro*!]:
$[\![\ acc\ r_A;\ acc\ r_B\ ]\!] \Longrightarrow acc(Product.le\ r_A\ r_B)$

lemma *closed-lift2-sup*:
$[\![\ closed\ (err\ A)\ (lift2\ f);\ closed\ (err\ B)\ (lift2\ g)\ ]\!] \Longrightarrow$
$closed\ (err(A{\times}B))\ (lift2(sup\ f\ g))$
lemma *unfold-plussub-lift2*: $e_1 \sqcup_{lift2\ f} e_2 \equiv lift2\ f\ e_1\ e_2$

lemma *plus-eq-Err-conv* [*simp*]:
$[\![\ x{\in}A;\ y{\in}A;\ semilat(err\ A,\ Err.le\ r,\ lift2\ f)\ ]\!]$
$\Longrightarrow (x \sqcup_f y = Err) = (\neg(\exists\, z{\in}A.\ x \sqsubseteq_r z \wedge y \sqsubseteq_r z))$
lemma *err-semilat-Product-esl*:
$\bigwedge L_1\ L_2.\ [\![\ err\text{-}semilat\ L_1;\ err\text{-}semilat\ L_2\ ]\!] \Longrightarrow err\text{-}semilat(Product.esl\ L_1\ L_2)$
end

4.5 Fixed Length Lists

theory *Listn* = *Err*:

constdefs
$list :: nat \Rightarrow 'a\ set \Rightarrow 'a\ list\ set$
$list\ n\ A \equiv \{xs.\ size\ xs = n \wedge set\ xs \subseteq A\}$

$le :: 'a\ ord \Rightarrow ('a\ list)ord$
$le\ r \equiv list\text{-}all2\ (\lambda x\ y.\ x \sqsubseteq_r y)$

syntax (*xsymbols*)
$lesublist :: 'a\ list \Rightarrow 'a\ ord \Rightarrow 'a\ list \Rightarrow bool\ \ ((\text{- } /[\sqsubseteq\text{-}]\text{ -})\ [50,\ 0,\ 51]\ 50)$
$lesssublist :: 'a\ list \Rightarrow 'a\ ord \Rightarrow 'a\ list \Rightarrow bool\ \ ((\text{- } /[\sqsubset\text{-}]\text{ -})\ [50,\ 0,\ 51]\ 50)$
translations
$x\ [\sqsubseteq_r]\ y == x <\!\!=\text{-}(Listn.le\ r)\ y$
$x\ [\sqsubset_r]\ y == x <\text{-}(Listn.le\ r)\ y$

constdefs
$map2 :: ('a \Rightarrow 'b \Rightarrow 'c) \Rightarrow 'a\ list \Rightarrow 'b\ list \Rightarrow 'c\ list$
$map2\ f \equiv (\lambda xs\ ys.\ map\ (split\ f)\ (zip\ xs\ ys))$

syntax (*xsymbols*)
$plussublist :: 'a\ list \Rightarrow ('a \Rightarrow 'b \Rightarrow 'c) \Rightarrow 'b\ list \Rightarrow 'c\ list\ ((\text{- } /[\sqcup\text{-}]\text{ -})\ [65,\ 0,\ 66]\ 65)$
translations
$x\ [\sqcup_f]\ y == x \sqcup_{map2\ f}\ y$

consts $coalesce :: 'a\ err\ list \Rightarrow 'a\ list\ err$
primrec
$coalesce\ [] = OK[]$
$coalesce\ (ex\#exs) = Err.sup\ (op\ \#)\ ex\ (coalesce\ exs)$

constdefs
$sl :: nat \Rightarrow 'a\ sl \Rightarrow 'a\ list\ sl$
$sl\ n \equiv \lambda(A,r,f).\ (list\ n\ A,\ le\ r,\ map2\ f)$

$sup :: ('a \Rightarrow 'b \Rightarrow 'c\ err) \Rightarrow 'a\ list \Rightarrow 'b\ list \Rightarrow 'c\ list\ err$
$sup\ f \equiv \lambda xs\ ys.\ if\ size\ xs = size\ ys\ then\ coalesce(xs\ [\sqcup_f]\ ys)\ else\ Err$

$upto\text{-}esl :: nat \Rightarrow 'a\ esl \Rightarrow 'a\ list\ esl$
$upto\text{-}esl\ m \equiv \lambda(A,r,f).\ (Union\{list\ n\ A\ |n.\ n \le m\},\ le\ r,\ sup\ f)$

lemmas [*simp*] = *set-update-subsetI*

lemma *unfold-lesub-list*: $xs\ [\sqsubseteq_r]\ ys \equiv Listn.le\ r\ xs\ ys$
lemma *Nil-le-conv* [*iff*]: $([]\ [\sqsubseteq_r]\ ys) = (ys = [])$
lemma *Cons-notle-Nil* [*iff*]: $\neg\ x\#xs\ [\sqsubseteq_r]\ []$
lemma *Cons-le-Cons* [*iff*]: $x\#xs\ [\sqsubseteq_r]\ y\#ys = (x \sqsubseteq_r y \wedge xs\ [\sqsubseteq_r]\ ys)$
lemma *Cons-less-Conss* [*simp*]:
$order\ r \Longrightarrow\ x\#xs\ [\sqsubset_r]\ y\#ys = (x \sqsubset_r y \wedge xs\ [\sqsubseteq_r]\ ys \vee x = y \wedge xs\ [\sqsubset_r]\ ys)$
lemma *list-update-le-cong*:
$[\![\ i<size\ xs;\ xs\ [\sqsubseteq_r]\ ys;\ x \sqsubseteq_r y\ ]\!] \Longrightarrow xs[i:=x]\ [\sqsubseteq_r]\ ys[i:=y]$

lemma *le-listD*: $[\![xs\ [\sqsubseteq_r]\ ys;\ p < size\ xs\ ]\!] \Longrightarrow xs!p \sqsubseteq_r ys!p$
lemma *le-list-refl*: $\forall x.\ x \sqsubseteq_r x \Longrightarrow xs\ [\sqsubseteq_r]\ xs$
lemma *le-list-trans*: $[\![order\ r;\ xs\ [\sqsubseteq_r]\ ys;\ ys\ [\sqsubseteq_r]\ zs\ ]\!] \Longrightarrow xs\ [\sqsubseteq_r]\ zs$
lemma *le-list-antisym*: $[\![order\ r;\ xs\ [\sqsubseteq_r]\ ys;\ ys\ [\sqsubseteq_r]\ xs\ ]\!] \Longrightarrow xs = ys$
lemma *order-listI* [*simp*, *intro*!]: $order\ r \Longrightarrow order(Listn.le\ r)$
lemma *lesub-list-impl-same-size* [*simp*]: $xs\ [\sqsubseteq_r]\ ys \Longrightarrow size\ ys = size\ xs$

lemma *lesssub-lengthD*: $xs\ [\sqsubset_r]\ ys \Longrightarrow size\ ys = size\ xs$
lemma *le-list-appendI*: $a\ [\sqsubseteq_r]\ b \Longrightarrow c\ [\sqsubseteq_r]\ d \Longrightarrow a@c\ [\sqsubseteq_r]\ b@d$
lemma *le-listI*: $size\ a = size\ b \Longrightarrow (\bigwedge n.\ n < size\ a \Longrightarrow a!n \sqsubseteq_r b!n) \Longrightarrow a\ [\sqsubseteq_r]\ b$
lemma *listI*: $[\![size\ xs = n;\ set\ xs \subseteq A\ ]\!] \Longrightarrow xs \in list\ n\ A$

lemma *listE-length* [*simp*]: $xs \in list\ n\ A \Longrightarrow size\ xs = n$
lemma *less-lengthI*: $[\![xs \in list\ n\ A;\ p < n\ ]\!] \Longrightarrow p < size\ xs$
lemma *listE-set* [*simp*]: $xs \in list\ n\ A \Longrightarrow set\ xs \subseteq A$
lemma *list-0* [*simp*]: $list\ 0\ A = \{[]\}$
lemma *in-list-Suc-iff*:
$(xs \in list\ (Suc\ n)\ A) = (\exists y{\in}A.\ \exists ys \in list\ n\ A.\ xs = y\#ys)$
lemma *Cons-in-list-Suc* [*iff*]:
$(x\#xs \in list\ (Suc\ n)\ A) = (x{\in}A \wedge xs \in list\ n\ A)$
lemma *list-not-empty*:
$\exists a.\ a{\in}A \Longrightarrow \exists xs.\ xs \in list\ n\ A$

lemma *nth-in* [*rule-format*, *simp*]:
$\forall i\ n.\ size\ xs = n \longrightarrow set\ xs \subseteq A \longrightarrow i < n \longrightarrow (xs!i) \in A$
lemma *listE-nth-in*: $[\![xs \in list\ n\ A;\ i < n\ ]\!] \Longrightarrow xs!i \in A$
lemma *listn-Cons-Suc* [*elim*!]:
$l\#xs \in list\ n\ A \Longrightarrow (\bigwedge n'.\ n = Suc\ n' \Longrightarrow l \in A \Longrightarrow xs \in list\ n'\ A \Longrightarrow P) \Longrightarrow P$
lemma *listn-appendE* [*elim*!]:
$a@b \in list\ n\ A \Longrightarrow (\bigwedge n1\ n2.\ n{=}n1{+}n2 \Longrightarrow a \in list\ n1\ A \Longrightarrow b \in list\ n2\ A \Longrightarrow P) \Longrightarrow P$

lemma *listt-update-in-list* [*simp*, *intro*!]:
$[\![xs \in list\ n\ A;\ x{\in}A\ ]\!] \Longrightarrow xs[i := x] \in list\ n\ A$
lemma *list-appendI* [*intro?*]:
$[\![a \in list\ n\ A;\ b \in list\ m\ A\ ]\!] \Longrightarrow a\ @\ b \in list\ (n{+}m)\ A$
lemma *list-map* [*simp*]: $(map\ f\ xs \in list\ (size\ xs)\ A) = (f\ `\ set\ xs \subseteq A)$
lemma *list-replicateI* [*intro*]: $x \in A \Longrightarrow replicate\ n\ x \in list\ n\ A$
lemma *plus-list-Nil* [*simp*]: $[]\ [\sqcup_f]\ xs = []$
lemma *plus-list-Cons* [*simp*]:
$(x\#xs)\ [\sqcup_f]\ ys = (case\ ys\ of\ [] \Rightarrow []\ |\ y\#ys \Rightarrow (x \sqcup_f y)\#(xs\ [\sqcup_f]\ ys))$
lemma *length-plus-list* [*rule-format*, *simp*]:
$\forall ys.\ size(xs\ [\sqcup_f]\ ys) = min(size\ xs)\ (size\ ys)$
lemma *nth-plus-list* [*rule-format*, *simp*]:
$\forall xs\ ys\ i.\ size\ xs = n \longrightarrow size\ ys = n \longrightarrow i{<}n \longrightarrow (xs\ [\sqcup_f]\ ys)!i = (xs!i) \sqcup_f (ys!i)$

lemma (**in** *semilat*) *plus-list-ub1* [*rule-format*]:
$[\![set\ xs \subseteq A;\ set\ ys \subseteq A;\ size\ xs = size\ ys\ ]\!]$
$\Longrightarrow xs\ [\sqsubseteq_r]\ xs\ [\sqcup_f]\ ys$
lemma (**in** *semilat*) *plus-list-ub2*:
$[\![set\ xs \subseteq A;\ set\ ys \subseteq A;\ size\ xs = size\ ys\ ]\!] \Longrightarrow ys\ [\sqsubseteq_r]\ xs\ [\sqcup_f]\ ys$
lemma (**in** *semilat*) *plus-list-lub* [*rule-format*]:
shows $\forall xs\ ys\ zs.\ set\ xs \subseteq A \longrightarrow set\ ys \subseteq A \longrightarrow set\ zs \subseteq A$

$\longrightarrow$ *size xs* = *n* $\wedge$ *size ys* = *n* $\longrightarrow$
xs $[\sqsubseteq_r]$ *zs* $\wedge$ *ys* $[\sqsubseteq_r]$ *zs* $\longrightarrow$ *xs* $[\sqcup_f]$ *ys* $[\sqsubseteq_r]$ *zs*
lemma (**in** *semilat*) *list-update-incr* [*rule-format*]:
$x \in A \Longrightarrow$ *set xs* $\subseteq A \longrightarrow$
($\forall i.\ i <$ *size xs* $\longrightarrow$ *xs* $[\sqsubseteq_r]$ *xs*[$i := x \sqcup_f$ *xs*!*i*])
lemma *acc-le-listI* [*intro*!]:
$[\![$ *order r*; *acc r* $]\!] \Longrightarrow$ *acc*(*Listn.le r*)
lemma *closed-listI*:
closed S f $\Longrightarrow$ *closed* (*list n S*) (*map2 f*)

lemma *Listn-sl-aux*:
includes *semilat* **shows** *semilat* (*Listn.sl n* (*A*,*r*,*f*))
lemma *Listn-sl*: $\bigwedge L.$ *semilat L* $\Longrightarrow$ *semilat* (*Listn.sl n L*)
lemma *coalesce-in-err-list* [*rule-format*]:
$\forall$ *xes*. *xes* $\in$ *list n* (*err A*) $\longrightarrow$ *coalesce xes* $\in$ *err*(*list n A*)
lemma *lem*: $\bigwedge x\ xs.\ x \sqcup_{op\ \#}\ xs = x\#xs$
lemma *coalesce-eq-OK1-D* [*rule-format*]:
semilat(*err A*, *Err.le r*, *lift2 f*) $\Longrightarrow$
$\forall$ *xs*. *xs* $\in$ *list n A* $\longrightarrow$ ($\forall$ *ys*. *ys* $\in$ *list n A* $\longrightarrow$
($\forall$ *zs*. *coalesce* (*xs* $[\sqcup_f]$ *ys*) = *OK zs* $\longrightarrow$ *xs* $[\sqsubseteq_r]$ *zs*))
lemma *coalesce-eq-OK2-D* [*rule-format*]:
semilat(*err A*, *Err.le r*, *lift2 f*) $\Longrightarrow$
$\forall$ *xs*. *xs* $\in$ *list n A* $\longrightarrow$ ($\forall$ *ys*. *ys* $\in$ *list n A* $\longrightarrow$
($\forall$ *zs*. *coalesce* (*xs* $[\sqcup_f]$ *ys*) = *OK zs* $\longrightarrow$ *ys* $[\sqsubseteq_r]$ *zs*))
lemma *lift2-le-ub*:
$[\![$ *semilat*(*err A*, *Err.le r*, *lift2 f*); $x \in A$; $y \in A$; $x \sqcup_f y$ = *OK z*;
$u \in A$; $x \sqsubseteq_r u$; $y \sqsubseteq_r u$ $]\!] \Longrightarrow z \sqsubseteq_r u$
lemma *coalesce-eq-OK-ub-D* [*rule-format*]:
semilat(*err A*, *Err.le r*, *lift2 f*) $\Longrightarrow$
$\forall$ *xs*. *xs* $\in$ *list n A* $\longrightarrow$ ($\forall$ *ys*. *ys* $\in$ *list n A* $\longrightarrow$
($\forall$ *zs us*. *coalesce* (*xs* $[\sqcup_f]$ *ys*) = *OK zs* $\wedge$ *xs* $[\sqsubseteq_r]$ *us* $\wedge$ *ys* $[\sqsubseteq_r]$ *us*
$\wedge$ *us* $\in$ *list n A* $\longrightarrow$ *zs* $[\sqsubseteq_r]$ *us*))
lemma *lift2-eq-ErrD*:
$[\![$ $x \sqcup_f y$ = *Err*; *semilat*(*err A*, *Err.le r*, *lift2 f*); $x \in A$; $y \in A$ $]\!]$
$\Longrightarrow \neg(\exists u \in A.\ x \sqsubseteq_r u \wedge y \sqsubseteq_r u)$

lemma *coalesce-eq-Err-D* [*rule-format*]:
$[\![$ *semilat*(*err A*, *Err.le r*, *lift2 f*) $]\!]$
$\Longrightarrow$ $\forall$ *xs*. *xs* $\in$ *list n A* $\longrightarrow$ ($\forall$ *ys*. *ys* $\in$ *list n A* $\longrightarrow$
coalesce (*xs* $[\sqcup_f]$ *ys*) = *Err* $\longrightarrow$
$\neg$($\exists$ *zs* $\in$ *list n A*. *xs* $[\sqsubseteq_r]$ *zs* $\wedge$ *ys* $[\sqsubseteq_r]$ *zs*))
lemma *closed-err-lift2-conv*:
closed (*err A*) (*lift2 f*) = ($\forall x \in A.\ \forall y \in A.\ x \sqcup_f y \in$ *err A*)
lemma *closed-map2-list* [*rule-format*]:
closed (*err A*) (*lift2 f*) $\Longrightarrow$
$\forall$ *xs*. *xs* $\in$ *list n A* $\longrightarrow$ ($\forall$ *ys*. *ys* $\in$ *list n A* $\longrightarrow$
map2 f xs ys $\in$ *list n* (*err A*))
lemma *closed-lift2-sup*:
closed (*err A*) (*lift2 f*) $\Longrightarrow$
closed (*err* (*list n A*)) (*lift2* (*sup f*))
lemma *err-semilat-sup*:
err-semilat (*A*,*r*,*f*) $\Longrightarrow$
err-semilat (*list n A*, *Listn.le r*, *sup f*)
lemma *err-semilat-upto-esl*:

$\bigwedge L.\ \mathit{err\text{-}semilat}\ L \Longrightarrow \mathit{err\text{-}semilat}(\mathit{upto\text{-}esl}\ m\ L)$

end

4.6 Typing and Dataflow Analysis Framework

theory *Typing-Framework* = *Semilattices*:

The relationship between dataflow analysis and a welltyped-instruction predicate.

types
$'s$ *step-type* $=$ *nat* $\Rightarrow$ $'s$ $\Rightarrow$ (*nat* $\times$ $'s$) *list*

constdefs
stable :: $'s$ *ord* $\Rightarrow$ $'s$ *step-type* $\Rightarrow$ $'s$ *list* $\Rightarrow$ *nat* $\Rightarrow$ *bool*
stable r step τs $p \equiv \forall (q,\tau) \in$ *set* (*step* p $(\tau s!p)$). $\tau \sqsubseteq_r \tau s!q$

stables :: $'s$ *ord* $\Rightarrow$ $'s$ *step-type* $\Rightarrow$ $'s$ *list* $\Rightarrow$ *bool*
stables r step $\tau s \equiv \forall p <$ *size* τs. *stable r step* τs p

wt-step :: $'s$ *ord* $\Rightarrow$ $'s$ $\Rightarrow$ $'s$ *step-type* $\Rightarrow$ $'s$ *list* $\Rightarrow$ *bool*
wt-step r T *step* $\tau s \equiv \forall p{<}$*size* τs. $\tau s!p \neq T \wedge$ *stable r step* τs p

is-bcv :: $'s$ *ord* $\Rightarrow$ $'s$ $\Rightarrow$ $'s$ *step-type* $\Rightarrow$ *nat* $\Rightarrow$ $'s$ *set* $\Rightarrow$ ($'s$ *list* $\Rightarrow$ $'s$ *list*) $\Rightarrow$ *bool*
is-bcv r T *step* n A *bcv* $\equiv \forall \tau s_0 \in$ *list* n A.
$(\forall p{<}n.\ (bcv\ \tau s_0)!p \neq T) = (\exists \tau s \in list\ n\ A.\ \tau s_0\ [\sqsubseteq_r]\ \tau s \wedge wt\text{-}step\ r\ T\ step\ \tau s)$

end

4.7 More on Semilattices

theory *SemilatAlg* = *Typing-Framework*:
syntax (*xsymbols*)
  lesubstep-type :: $(nat \times 's)\ set \Rightarrow 's\ ord \Rightarrow (nat \times 's)\ set \Rightarrow bool$
    $((\text{-}\ /\{\sqsubseteq\text{-}\}\ \text{-})\ [50,\ 0,\ 51]\ 50)$
constdefs
  lesubstep-type :: $(nat \times 's)\ set \Rightarrow 's\ ord \Rightarrow (nat \times 's)\ set \Rightarrow bool$
  $A\ \{\sqsubseteq_r\}\ B \equiv \forall (p,\tau) \in A.\ \exists \tau'.\ (p,\tau') \in B \land \tau \sqsubseteq_r \tau'$

consts
  pluslussub :: $'a\ list \Rightarrow ('a \Rightarrow 'a \Rightarrow 'a) \Rightarrow 'a \Rightarrow 'a$
syntax (*xsymbols*)
  pluslussub :: $'a\ list \Rightarrow ('a \Rightarrow 'a \Rightarrow 'a) \Rightarrow 'a \Rightarrow 'a\ ((\text{-}\ /\bigsqcup_{\text{-}}\ \text{-})\ [65,\ 0,\ 66]\ 65)$
primrec
  $[]\ \bigsqcup_f\ y = y$
  $(x\#xs)\ \bigsqcup_f\ y = xs\ \bigsqcup_f\ (x \sqcup_f y)$

constdefs
  bounded :: $'s\ step\text{-}type \Rightarrow nat \Rightarrow bool$
  $bounded\ step\ n \equiv \forall p{<}n.\ \forall \tau.\ \forall (q,\tau') \in set\ (step\ p\ \tau).\ q{<}n$

  pres-type :: $'s\ step\text{-}type \Rightarrow nat \Rightarrow 's\ set \Rightarrow bool$
  $pres\text{-}type\ step\ n\ A \equiv \forall \tau{\in}A.\ \forall p{<}n.\ \forall (q,\tau') \in set\ (step\ p\ \tau).\ \tau' \in A$

  mono :: $'s\ ord \Rightarrow 's\ step\text{-}type \Rightarrow nat \Rightarrow 's\ set \Rightarrow bool$
  $mono\ r\ step\ n\ A \equiv$
  $\forall \tau\ p\ \tau'.\ \tau \in A \land p{<}n \land \tau \sqsubseteq_r \tau' \longrightarrow set\ (step\ p\ \tau)\ \{\sqsubseteq_r\}\ set\ (step\ p\ \tau')$

lemma [*iff*]: $\{\}\ \{\sqsubseteq_r\}\ B$

lemma [*iff*]: $(A\ \{\sqsubseteq_r\}\ \{\}) = (A = \{\})$

lemma *lesubstep-union*:
  $[\![\ A_1\ \{\sqsubseteq_r\}\ B_1;\ A_2\ \{\sqsubseteq_r\}\ B_2\ ]\!] \Longrightarrow A_1 \cup A_2\ \{\sqsubseteq_r\}\ B_1 \cup B_2$

lemma *pres-typeD*:
  $[\![\ pres\text{-}type\ step\ n\ A;\ s{\in}A;\ p{<}n;\ (q,s'){\in}set\ (step\ p\ s)\ ]\!] \Longrightarrow s' \in A$
lemma *monoD*:
  $[\![\ mono\ r\ step\ n\ A;\ p < n;\ s{\in}A;\ s \sqsubseteq_r t\ ]\!] \Longrightarrow set\ (step\ p\ s)\ \{\sqsubseteq_r\}\ set\ (step\ p\ t)$
lemma *boundedD*:
  $[\![\ bounded\ step\ n;\ p < n;\ (q,t) \in set\ (step\ p\ xs)\ ]\!] \Longrightarrow q < n$

lemma *lesubstep-type-refl* [*simp*, *intro*]:
  $(\bigwedge x.\ x \sqsubseteq_r x) \Longrightarrow A\ \{\sqsubseteq_r\}\ A$
lemma *lesub-step-typeD*:
  $A\ \{\sqsubseteq_r\}\ B \Longrightarrow (x,y) \in A \Longrightarrow \exists y'.\ (x,\ y') \in B \land y \sqsubseteq_r y'$

lemma *list-update-le-listI* [*rule-format*]:
  $set\ xs \subseteq A \longrightarrow set\ ys \subseteq A \longrightarrow xs\ [\sqsubseteq_r]\ ys \longrightarrow p < size\ xs \longrightarrow$
   $x \sqsubseteq_r ys!p \longrightarrow semilat(A,r,f) \longrightarrow x{\in}A \longrightarrow$
   $xs[p := x \sqcup_f xs!p]\ [\sqsubseteq_r]\ ys$

lemma *plusplus-closed*: **includes** *semilat* **shows**

$\bigwedge y.\ [\![\ set\ x \subseteq A;\ y \in A]\!] \Longrightarrow x \bigsqcup_f y \in A$
lemma (**in** *semilat*) *pp-ub2*:
$\bigwedge y.\ [\![\ set\ x \subseteq A;\ y \in A]\!] \Longrightarrow y \sqsubseteq_r x \bigsqcup_f y$

lemma (**in** *semilat*) *pp-ub1*:
shows $\bigwedge y.\ [\![set\ ls \subseteq A;\ y \in A;\ x \in set\ ls]\!] \Longrightarrow x \sqsubseteq_r ls \bigsqcup_f y$

lemma (**in** *semilat*) *pp-lub*:
assumes $z \in A$
shows
$\bigwedge y.\ y \in A \Longrightarrow set\ xs \subseteq A \Longrightarrow \forall x \in set\ xs.\ x \sqsubseteq_r z \Longrightarrow y \sqsubseteq_r z \Longrightarrow xs \bigsqcup_f y \sqsubseteq_r z$

lemma *ub1'*: **includes** *semilat*
shows $[\![\forall (p,s) \in set\ S.\ s \in A;\ y \in A;\ (a,b) \in set\ S]\!]$
$\Longrightarrow b \sqsubseteq_r map\ snd\ [(p',\ t')\in S.\ p' = a] \bigsqcup_f y$

lemma *plusplus-empty*:
$\forall s'.\ (q,\ s') \in set\ S \longrightarrow s' \sqcup_f ss\ !\ q = ss\ !\ q \Longrightarrow$
$(map\ snd\ [(p',\ t')\in S.\ p' = q] \bigsqcup_f ss\ !\ q) = ss\ !\ q$

end

4.8 Lifting the Typing Framework to err, app, and eff

theory *Typing-Framework-err* = *Typing-Framework* + *SemilatAlg*:

constdefs
$wt\text{-}err\text{-}step :: {'s}\ ord \Rightarrow {'s}\ err\ step\text{-}type \Rightarrow {'s}\ err\ list \Rightarrow bool$
$wt\text{-}err\text{-}step\ r\ step\ \tau s \equiv wt\text{-}step\ (Err.le\ r)\ Err\ step\ \tau s$

$wt\text{-}app\text{-}eff :: {'s}\ ord \Rightarrow (nat \Rightarrow {'s} \Rightarrow bool) \Rightarrow {'s}\ step\text{-}type \Rightarrow {'s}\ list \Rightarrow bool$
$wt\text{-}app\text{-}eff\ r\ app\ step\ \tau s \equiv$
$\forall p < size\ \tau s.\ app\ p\ (\tau s!p) \wedge (\forall (q,\tau) \in set\ (step\ p\ (\tau s!p)).\ \tau <\!=\!\text{-}r\ \tau s!q)$

$map\text{-}snd :: ({'b} \Rightarrow {'c}) \Rightarrow ({'a} \times {'b})\ list \Rightarrow ({'a} \times {'c})\ list$
$map\text{-}snd\ f \equiv map\ (\lambda(x,y).\ (x,\ f\ y))$

$error :: nat \Rightarrow (nat \times {'a}\ err)\ list$
$error\ n \equiv map\ (\lambda x.\ (x,Err))\ [0..n(]$

$err\text{-}step :: nat \Rightarrow (nat \Rightarrow {'s} \Rightarrow bool) \Rightarrow {'s}\ step\text{-}type \Rightarrow {'s}\ err\ step\text{-}type$
$err\text{-}step\ n\ app\ step\ p\ t \equiv$
$case\ t\ of$
$\quad Err \quad \Rightarrow error\ n$
$|\ OK\ \tau \Rightarrow if\ app\ p\ \tau\ then\ map\text{-}snd\ OK\ (step\ p\ \tau)\ else\ error\ n$

$app\text{-}mono :: {'s}\ ord \Rightarrow (nat \Rightarrow {'s} \Rightarrow bool) \Rightarrow nat \Rightarrow {'s}\ set \Rightarrow bool$
$app\text{-}mono\ r\ app\ n\ A \equiv$
$\forall s\ p\ t.\ s \in A \wedge p < n \wedge s \sqsubseteq_r t \longrightarrow app\ p\ t \longrightarrow app\ p\ s$

lemmas *err-step-defs* = *err-step-def map-snd-def error-def*

lemma *bounded-err-stepD*:
$[\![\ bounded\ (err\text{-}step\ n\ app\ step)\ n;$
$\quad p < n;\ app\ p\ a;\ (q,b) \in set\ (step\ p\ a)\ ]\!] \Longrightarrow q < n$

lemma *in-map-sndD*: $(a,b) \in set\ (map\text{-}snd\ f\ xs) \Longrightarrow \exists b'.\ (a,b') \in set\ xs$

lemma *bounded-err-stepI*:
$\forall p.\ p < n \longrightarrow (\forall s.\ ap\ p\ s \longrightarrow (\forall (q,s') \in set\ (step\ p\ s).\ q < n))$
$\Longrightarrow bounded\ (err\text{-}step\ n\ ap\ step)\ n$

lemma *bounded-lift*:
$bounded\ step\ n \Longrightarrow bounded\ (err\text{-}step\ n\ app\ step)\ n$

lemma *le-list-map-OK* [*simp*]:
$\bigwedge b.\ (map\ OK\ a\ [\sqsubseteq_{Err.le\ r}]\ map\ OK\ b) = (a\ [\sqsubseteq_r]\ b)$

lemma *map-snd-lessI*:
$set\ xs\ \{\sqsubseteq_r\}\ set\ ys \Longrightarrow set\ (map\text{-}snd\ OK\ xs)\ \{\sqsubseteq_{Err.le\ r}\}\ set\ (map\text{-}snd\ OK\ ys)$

lemma *mono-lift*:
$[\![\ order\ r;\ app\text{-}mono\ r\ app\ n\ A;\ bounded\ (err\text{-}step\ n\ app\ step)\ n;$
$\quad \forall s\ p\ t.\ s \in A \wedge p < n \wedge s \sqsubseteq_r t \longrightarrow app\ p\ t \longrightarrow set\ (step\ p\ s)\ \{\sqsubseteq_r\}\ set\ (step\ p\ t)\ ]\!]$

$\Longrightarrow$ *mono* (*Err.le r*) (*err-step n app step*) *n* (*err A*)
lemma *in-errorD*: $(x,y) \in$ *set* (*error n*) $\Longrightarrow y = Err$
lemma *pres-type-lift*:
$\forall s \in A.\ \forall p.\ p < n \longrightarrow app\ p\ s \longrightarrow (\forall (q,\ s') \in set\ (step\ p\ s).\ s' \in A)$
$\Longrightarrow$ *pres-type* (*err-step n app step*) *n* (*err A*)

lemma *wt-err-imp-wt-app-eff*:
assumes *wt*: *wt-err-step r* (*err-step* (*size ts*) *app step*) *ts*
assumes *b*: *bounded* (*err-step* (*size ts*) *app step*) (*size ts*)
shows *wt-app-eff r app step* (*map ok-val ts*)

lemma *wt-app-eff-imp-wt-err*:
assumes *app-eff*: *wt-app-eff r app step ts*
assumes *bounded*: *bounded* (*err-step* (*size ts*) *app step*) (*size ts*)
shows *wt-err-step r* (*err-step* (*size ts*) *app step*) (*map OK ts*)
end

4.9 Kildall's Algorithm

theory *Kildall* = *SemilatAlg*:

consts
$iter :: 's\ binop \Rightarrow 's\ step\text{-}type \Rightarrow$
$\quad 's\ list \Rightarrow nat\ set \Rightarrow 's\ list \times nat\ set$
$propa :: 's\ binop \Rightarrow (nat \times 's)\ list \Rightarrow 's\ list \Rightarrow nat\ set \Rightarrow 's\ list * nat\ set$

primrec
$propa\ f\ []\quad \tau s\ w = (\tau s, w)$
$propa\ f\ (q'\#qs)\ \tau s\ w = (let\ (q,\tau) = q';$
$\quad u = \tau \sqcup_f \tau s!q;$
$\quad w' = (if\ u = \tau s!q\ then\ w\ else\ insert\ q\ w)$
$\quad in\ propa\ f\ qs\ (\tau s[q := u])\ w')$

defs *iter-def*:
$iter\ f\ step\ \tau s\ w \equiv$
$while\ (\lambda(\tau s,w).\ w \neq \{\})$
$\quad (\lambda(\tau s,w).\ let\ p = SOME\ p.\ p \in w$
$\quad\quad in\ propa\ f\ (step\ p\ (\tau s!p))\ \tau s\ (w - \{p\}))$
$\quad (\tau s,w)$

constdefs
$unstables :: 's\ ord \Rightarrow 's\ step\text{-}type \Rightarrow 's\ list \Rightarrow nat\ set$
$unstables\ r\ step\ \tau s \equiv \{p.\ p < size\ \tau s \wedge \neg stable\ r\ step\ \tau s\ p\}$

$kildall :: 's\ ord \Rightarrow 's\ binop \Rightarrow 's\ step\text{-}type \Rightarrow 's\ list \Rightarrow 's\ list$
$kildall\ r\ f\ step\ \tau s \equiv fst(iter\ f\ step\ \tau s\ (unstables\ r\ step\ \tau s))$

consts $merges :: 's\ binop \Rightarrow (nat \times 's)\ list \Rightarrow 's\ list \Rightarrow 's\ list$
primrec
$merges\ f\ []\quad \tau s = \tau s$
$merges\ f\ (p'\#ps)\ \tau s = (let\ (p,\tau) = p'\ in\ merges\ f\ ps\ (\tau s[p := \tau \sqcup_f \tau s!p]))$

lemmas [*simp*] = *Let-def semilat.le-iff-plus-unchanged* [*symmetric*]

lemma (**in** *semilat*) *nth-merges*:
$\bigwedge ss.\ [\![p < length\ ss;\ ss \in list\ n\ A;\ \forall (p,t) \in set\ ps.\ p<n \wedge t \in A\ ]\!] \Longrightarrow$
$(merges\ f\ ps\ ss)!p = map\ snd\ [(p',t') \in ps.\ p'=p]\ \bigsqcup_f\ ss!p$
(**is** $\bigwedge ss.\ [\![-;\ -;\ ?steptype\ ps]\!] \Longrightarrow\ ?P\ ss\ ps$)

lemma *length-merges* [*simp*]:
$\bigwedge ss.\ size(merges\ f\ ps\ ss) = size\ ss$
lemma (**in** *semilat*) *merges-preserves-type-lemma*:
shows $\forall xs.\ xs \in list\ n\ A \longrightarrow (\forall (p,x) \in set\ ps.\ p<n \wedge x \in A)$
$\quad \longrightarrow merges\ f\ ps\ xs \in list\ n\ A$
lemma (**in** *semilat*) *merges-preserves-type* [*simp*]:
$[\![\ xs \in list\ n\ A;\ \forall (p,x) \in set\ ps.\ p<n \wedge x \in A\ ]\!]$

$\Longrightarrow$ *merges f ps xs* $\in$ *list n A*
by (*simp add*: *merges-preserves-type-lemma*)

lemma (**in** *semilat*) *merges-incr-lemma*:
$\forall xs.\ xs \in list\ n\ A \longrightarrow (\forall (p,x){\in}set\ ps.\ p{<}size\ xs \wedge x \in A) \longrightarrow xs\ [\sqsubseteq_r]\ merges\ f\ ps\ xs$
lemma (**in** *semilat*) *merges-incr*:
$[\![\ xs \in list\ n\ A;\ \forall (p,x){\in}set\ ps.\ p{<}size\ xs \wedge x \in A\ ]\!]$
$\Longrightarrow xs\ [\sqsubseteq_r]\ merges\ f\ ps\ xs$
by (*simp add*: *merges-incr-lemma*)

lemma (**in** *semilat*) *merges-same-conv* [*rule-format*]:
$(\forall xs.\ xs \in list\ n\ A \longrightarrow (\forall (p,x){\in}set\ ps.\ p{<}size\ xs \wedge x{\in}A) \longrightarrow$
$(merges\ f\ ps\ xs = xs) = (\forall (p,x){\in}set\ ps.\ x \sqsubseteq_r xs!p))$

lemma (**in** *semilat*) *list-update-le-listI* [*rule-format*]:
$set\ xs \subseteq A \longrightarrow set\ ys \subseteq A \longrightarrow xs\ [\sqsubseteq_r]\ ys \longrightarrow p < size\ xs \longrightarrow$
$x \sqsubseteq_r ys!p \longrightarrow x{\in}A \longrightarrow xs[p := x \sqcup_f xs!p]\ [\sqsubseteq_r]\ ys$
lemma (**in** *semilat*) *merges-pres-le-ub*:
shows $[\![\ set\ ts \subseteq A;\ set\ ss \subseteq A;$
$\forall (p,t){\in}set\ ps.\ t \sqsubseteq_r ts!p \wedge t \in A \wedge p < size\ ts;\ ss\ [\sqsubseteq_r]\ ts\ ]\!]$
$\Longrightarrow merges\ f\ ps\ ss\ [\sqsubseteq_r]\ ts$

lemma *decomp-propa*:
$\bigwedge ss\ w.\ (\forall (q,t){\in}set\ qs.\ q < size\ ss) \Longrightarrow$
$propa\ f\ qs\ ss\ w =$
$(merges\ f\ qs\ ss,\ \{q.\ \exists t.(q,t){\in}set\ qs \wedge t \sqcup_f ss!q \neq ss!q\} \cup w)$

lemma (**in** *semilat*) *stable-pres-lemma*:
shows $[\![pres\text{-}type\ step\ n\ A;\ bounded\ step\ n;$
$ss \in list\ n\ A;\ p \in w;\ \forall q{\in}w.\ q < n;$
$\forall q.\ q < n \longrightarrow q \notin w \longrightarrow stable\ r\ step\ ss\ q;\ q < n;$
$\forall s'.\ (q,s') \in set\ (step\ p\ (ss!p)) \longrightarrow s' \sqcup_f ss!q = ss!q;$
$q \notin w \vee q = p\ ]\!]$
$\Longrightarrow stable\ r\ step\ (merges\ f\ (step\ p\ (ss!p))\ ss)\ q$

lemma (**in** *semilat*) *merges-bounded-lemma*:
$[\![\ mono\ r\ step\ n\ A;\ bounded\ step\ n;$
$\forall (p',s') \in set\ (step\ p\ (ss!p)).\ s' \in A;\ ss \in list\ n\ A;\ ts \in list\ n\ A;\ p < n;$
$ss\ [\sqsubseteq_r]\ ts;\ \forall p.\ p < n \longrightarrow stable\ r\ step\ ts\ p\ ]\!]$
$\Longrightarrow merges\ f\ (step\ p\ (ss!p))\ ss\ [\sqsubseteq_r]\ ts$

lemma *termination-lemma*: **includes** *semilat*
shows $[\![\ ss \in list\ n\ A;\ \forall (q,t){\in}set\ qs.\ q{<}n \wedge t{\in}A;\ p{\in}w\ ]\!] \Longrightarrow$
$ss\ [\sqsubset_r]\ merges\ f\ qs\ ss\ \vee$
$merges\ f\ qs\ ss = ss \wedge \{q.\ \exists t.\ (q,t){\in}set\ qs \wedge t \sqcup_f ss!q \neq ss!q\} \cup (w{-}\{p\}) \subset w$
lemma *iter-properties*[*rule-format*]: **includes** *semilat*
shows $[\![\ acc\ r;\ pres\text{-}type\ step\ n\ A;\ mono\ r\ step\ n\ A;$
$bounded\ step\ n;\ \forall p{\in}w0.\ p < n;\ ss0 \in list\ n\ A;$
$\forall p{<}n.\ p \notin w0 \longrightarrow stable\ r\ step\ ss0\ p\ ]\!] \Longrightarrow$
$iter\ f\ step\ ss0\ w0 = (ss',w')$

$\longrightarrow$
$ss' \in list\ n\ A \wedge stables\ r\ step\ ss' \wedge ss0\ [\sqsubseteq_r]\ ss' \wedge$
$(\forall\, ts{\in}list\ n\ A.\ ss0\ [\sqsubseteq_r]\ ts \wedge stables\ r\ step\ ts \longrightarrow ss'\ [\sqsubseteq_r]\ ts)$

lemma *kildall-properties*: **includes** *semilat*
shows $[\![\ acc\ r;\ pres\text{-}type\ step\ n\ A;\ mono\ r\ step\ n\ A;$
$bounded\ step\ n;\ ss0 \in list\ n\ A\ ]\!] \Longrightarrow$
$kildall\ r\ f\ step\ ss0 \in list\ n\ A \wedge$
$stables\ r\ step\ (kildall\ r\ f\ step\ ss0) \wedge$
$ss0\ [\sqsubseteq_r]\ kildall\ r\ f\ step\ ss0 \wedge$
$(\forall\, ts{\in}list\ n\ A.\ ss0\ [\sqsubseteq_r]\ ts \wedge stables\ r\ step\ ts \longrightarrow$
$kildall\ r\ f\ step\ ss0\ [\sqsubseteq_r]\ ts)$
end

4.10 The Lightweight Bytecode Verifier

```
theory LBVSpec = SemilatAlg + Opt:

types
  's certificate = 's list

consts
merge :: 's certificate ⇒ 's binop ⇒ 's ord ⇒ 's ⇒ nat ⇒ (nat × 's) list ⇒ 's ⇒ 's
primrec
  merge cert f r T pc []      x = x
  merge cert f r T pc (s#ss) x = merge cert f r T pc ss (let (pc',s') = s in
                                 if pc'=pc+1 then s' ⊔_f x
                                 else if s' ⊑_r cert!pc' then x
                                 else T)

constdefs
  wtl-inst :: 's certificate ⇒ 's binop ⇒ 's ord ⇒ 's ⇒
              's step-type ⇒ nat ⇒ 's ⇒ 's
  wtl-inst cert f r T step pc s ≡ merge cert f r T pc (step pc s) (cert!(pc+1))

  wtl-cert :: 's certificate ⇒ 's binop ⇒ 's ord ⇒ 's ⇒ 's ⇒
              's step-type ⇒ nat ⇒ 's ⇒ 's
  wtl-cert cert f r T B step pc s ≡
  if cert!pc = B then
    wtl-inst cert f r T step pc s
  else
    if s ⊑_r cert!pc then wtl-inst cert f r T step pc (cert!pc) else T

consts
  wtl-inst-list :: 'a list ⇒ 's certificate ⇒ 's binop ⇒ 's ord ⇒ 's ⇒ 's ⇒
                   's step-type ⇒ nat ⇒ 's ⇒ 's
primrec
  wtl-inst-list []      cert f r T B step pc s = s
  wtl-inst-list (i#is) cert f r T B step pc s =
    (let s' = wtl-cert cert f r T B step pc s in
      if s' = T ∨ s = T then T else wtl-inst-list is cert f r T B step (pc+1) s')

constdefs
  cert-ok :: 's certificate ⇒ nat ⇒ 's ⇒ 's ⇒ 's set ⇒ bool
  cert-ok cert n T B A ≡ (∀ i < n. cert!i ∈ A ∧ cert!i ≠ T) ∧ (cert!n = B)

constdefs
  bottom :: 'a ord ⇒ 'a ⇒ bool
  bottom r B ≡ ∀ x. B ⊑_r x


locale (open) lbv = semilat +
  fixes T :: 'a (⊤)
  fixes B :: 'a (⊥)
  fixes step :: 'a step-type
  assumes top: top r ⊤
  assumes T-A: ⊤ ∈ A
  assumes bot: bottom r ⊥
```

assumes *B-A*: $\bot \in A$

fixes *merge* :: $'a$ *certificate* $\Rightarrow$ *nat* $\Rightarrow$ (*nat* $\times$ $'a$) *list* $\Rightarrow$ $'a \Rightarrow 'a$
defines *mrg-def*: *merge cert* $\equiv$ *LBVSpec.merge cert f r* $\top$

fixes *wti* :: $'a$ *certificate* $\Rightarrow$ *nat* $\Rightarrow$ $'a \Rightarrow 'a$
defines *wti-def*: *wti cert* $\equiv$ *wtl-inst cert f r* $\top$ *step*

fixes *wtc* :: $'a$ *certificate* $\Rightarrow$ *nat* $\Rightarrow$ $'a \Rightarrow 'a$
defines *wtc-def*: *wtc cert* $\equiv$ *wtl-cert cert f r* $\top$ $\bot$ *step*

fixes *wtl* :: $'b$ *list* $\Rightarrow$ $'a$ *certificate* $\Rightarrow$ *nat* $\Rightarrow$ $'a \Rightarrow 'a$
defines *wtl-def*: *wtl ins cert* $\equiv$ *wtl-inst-list ins cert f r* $\top$ $\bot$ *step*

lemma (**in** *lbv*) *wti*:
wti c pc s $\equiv$ *merge c pc* (*step pc s*) (*c*!(*pc*+*1*))

lemma (**in** *lbv*) *wtc*:
wtc c pc s $\equiv$ *if c*!*pc* = $\bot$ *then wti c pc s else if s* $\sqsubseteq_r$ *c*!*pc then wti c pc* (*c*!*pc*) *else* $\top$

lemma *cert-okD1* [*intro?*]:
cert-ok c n T B A $\Longrightarrow$ *pc* < *n* $\Longrightarrow$ *c*!*pc* $\in$ *A*

lemma *cert-okD2* [*intro?*]:
cert-ok c n T B A $\Longrightarrow$ *c*!*n* = *B*

lemma *cert-okD3* [*intro?*]:
cert-ok c n T B A $\Longrightarrow$ *B* $\in$ *A* $\Longrightarrow$ *pc* < *n* $\Longrightarrow$ *c*!*Suc pc* $\in$ *A*

lemma *cert-okD4* [*intro?*]:
cert-ok c n T B A $\Longrightarrow$ *pc* < *n* $\Longrightarrow$ *c*!*pc* $\neq$ *T*

declare *Let-def* [*simp*]

4.10.1 more semilattice lemmas

lemma (**in** *lbv*) *sup-top* [*simp*, *elim*]:
assumes *x*: $x \in A$
shows $x \sqcup_f \top = \top$
lemma (**in** *lbv*) *plusplussup-top* [*simp*, *elim*]:
set xs $\subseteq$ *A* $\Longrightarrow$ *xs* $\bigsqcup_f$ $\top = \top$
by (*induct xs*) *auto*

lemma (**in** *semilat*) *pp-ub1'*:
assumes *S*: *snd'set S* $\subseteq$ *A*
assumes *y*: $y \in A$ **and** *ab*: (*a*, *b*) $\in$ *set S*
shows *b* $\sqsubseteq_r$ *map snd* [(*p'*, *t'*)$\in$*S* . *p'* = *a*] $\bigsqcup_f$ *y*
lemma (**in** *lbv*) *bottom-le* [*simp*, *intro*!]: $\bot \sqsubseteq_r x$
by (*insert bot*) (*simp add*: *bottom-def*)

lemma (**in** *lbv*) *le-bottom* [*simp*]: $x \sqsubseteq_r \bot = (x = \bot)$
by (*blast intro*: *antisym-r*)

4.10.2 merge

lemma (**in** *lbv*) *merge-Nil* [*simp*]:
  merge c pc [] $x = x$ **by** (*simp add*: *mrg-def*)

lemma (**in** *lbv*) *merge-Cons* [*simp*]:
  merge c pc (*l*#*ls*) x = *merge c pc ls* (*if fst l*=*pc*+*1 then snd l* +-*f* x
  else if snd l $\sqsubseteq_r$ *c*!*fst l then* x
  else $\top$)
  by (*simp add*: *mrg-def split-beta*)

lemma (**in** *lbv*) *merge-Err* [*simp*]:
  snd'*set ss* $\subseteq A \Longrightarrow$ *merge c pc ss* $\top = \top$
  by (*induct ss*) *auto*

lemma (**in** *lbv*) *merge-not-top*:
  $\bigwedge x$. *snd*'*set ss* $\subseteq A \Longrightarrow$ *merge c pc ss* $x \neq \top \Longrightarrow$
  $\forall$(*pc*′,*s*′) $\in$ *set ss*. (*pc*′ $\neq$ *pc*+*1* $\longrightarrow$ *s*′ $\sqsubseteq_r$ *c*!*pc*′)
  (**is** $\bigwedge x$. ?*set ss* $\Longrightarrow$?*merge ss x* $\Longrightarrow$?*P ss*)

lemma (**in** *lbv*) *merge-def*:
  shows
  $\bigwedge x$. $x \in A \Longrightarrow$ *snd*'*set ss* $\subseteq A \Longrightarrow$
  merge c pc ss x =
  (*if* $\forall$(*pc*′,*s*′) $\in$ *set ss*. *pc*′$\neq$*pc*+*1* $\longrightarrow$ *s*′ $\sqsubseteq_r$ *c*!*pc*′ *then*
    map snd [(*p*′,*t*′) $\in$ *ss*. *p*′=*pc*+*1*] $\bigsqcup_f$ x
  else $\top$)
  (**is** $\bigwedge x$. - $\Longrightarrow$ - $\Longrightarrow$?*merge ss x* = ?*if ss x* **is** $\bigwedge x$. - $\Longrightarrow$ - $\Longrightarrow$?*P ss x*)
lemma (**in** *lbv*) *merge-not-top-s*:
  assumes *x*: $x \in A$ **and** *ss*: *snd*'*set ss* $\subseteq A$
  assumes *m*: *merge c pc ss* $x \neq \top$
  shows *merge c pc ss* x = (*map snd* [(*p*′,*t*′) $\in$ *ss*. *p*′=*pc*+*1*] $\bigsqcup_f$ x)

4.10.3 wtl-inst-list

lemmas [*iff*] = *not-Err-eq*

lemma (**in** *lbv*) *wtl-Nil* [*simp*]: *wtl* [] *c pc s* = *s*
  by (*simp add*: *wtl-def*)

lemma (**in** *lbv*) *wtl-Cons* [*simp*]:
  wtl (*i*#*is*) *c pc s* =
  (*let s*′ = *wtc c pc s in if s*′ = $\top$ $\vee$ *s* = $\top$ *then* $\top$ *else wtl is c* (*pc*+*1*) *s*′)
  by (*simp add*: *wtl-def wtc-def*)

lemma (**in** *lbv*) *wtl-Cons-not-top*:
  wtl (*i*#*is*) *c pc s* $\neq \top$ =
  (*wtc c pc s* $\neq \top \wedge$ *s* $\neq$ *T* $\wedge$ *wtl is c* (*pc*+*1*) (*wtc c pc s*) $\neq \top$)
  by (*auto simp del*: *split-paired-Ex*)

lemma (**in** *lbv*) *wtl-top* [*simp*]: *wtl ls c pc* $\top = \top$
  by (*cases ls*) *auto*

lemma (**in** *lbv*) *wtl-not-top*:
  wtl ls c pc s $\neq \top \Longrightarrow$ *s* $\neq \top$

by (*cases* $s{=}\top$) *auto*

lemma (**in** *lbv*) *wtl-append* [*simp*]:
$\bigwedge pc\ s.\ wtl\ (a@b)\ c\ pc\ s = wtl\ b\ c\ (pc{+}length\ a)\ (wtl\ a\ c\ pc\ s)$
by (*induct a*) *auto*

lemma (**in** *lbv*) *wtl-take*:
$wtl\ is\ c\ pc\ s \neq \top \Longrightarrow wtl\ (take\ pc'\ is)\ c\ pc\ s \neq \top$
(**is** $?wtl\ is \neq$ - $\Longrightarrow$ -)
lemma *take-Suc*:
$\forall n.\ n < length\ l \longrightarrow take\ (Suc\ n)\ l = (take\ n\ l)@[l!n]$ (**is** $?P\ l$)
lemma (**in** *lbv*) *wtl-Suc*:
assumes *suc*: $pc{+}1 < length\ is$
assumes *wtl*: $wtl\ (take\ pc\ is)\ c\ 0\ s \neq \top$
shows $wtl\ (take\ (pc{+}1)\ is)\ c\ 0\ s = wtc\ c\ pc\ (wtl\ (take\ pc\ is)\ c\ 0\ s)$
lemma (**in** *lbv*) *wtl-all*:
assumes *all*: $wtl\ is\ c\ 0\ s \neq \top$ (**is** $?wtl\ is \neq$ -)
assumes *pc*: $pc < length\ is$
shows $wtc\ c\ pc\ (wtl\ (take\ pc\ is)\ c\ 0\ s) \neq \top$

4.10.4 preserves-type

lemma (**in** *lbv*) *merge-pres*:
assumes *s0*: $snd`set\ ss \subseteq A$ **and** *x*: $x \in A$
shows $merge\ c\ pc\ ss\ x \in A$
lemma *pres-typeD2*:
$pres\text{-}type\ step\ n\ A \Longrightarrow s \in A \Longrightarrow p < n \Longrightarrow snd`set\ (step\ p\ s) \subseteq A$
by *auto* (*drule pres-typeD*)

lemma (**in** *lbv*) *wti-pres* [*intro?*]:
assumes *pres*: $pres\text{-}type\ step\ n\ A$
assumes *cert*: $c!(pc{+}1) \in A$
assumes *s-pc*: $s \in A\ pc < n$
shows $wti\ c\ pc\ s \in A$
lemma (**in** *lbv*) *wtc-pres*:
assumes $pres\text{-}type\ step\ n\ A$
assumes $c!pc \in A$ **and** $c!(pc{+}1) \in A$
assumes $s \in A$ **and** $pc < n$
shows $wtc\ c\ pc\ s \in A$
lemma (**in** *lbv*) *wtl-pres*:
assumes *pres*: $pres\text{-}type\ step\ (length\ is)\ A$
assumes *cert*: $cert\text{-}ok\ c\ (length\ is)\ \top\ \bot\ A$
assumes *s*: $s \in A$
assumes *all*: $wtl\ is\ c\ 0\ s \neq \top$
shows $pc < length\ is \Longrightarrow wtl\ (take\ pc\ is)\ c\ 0\ s \in A$
(**is** $?len\ pc \Longrightarrow ?wtl\ pc \in A$)

end

4.11 Correctness of the LBV

theory *LBVCorrect* = *LBVSpec* + *Typing-Framework*:

locale (**open**) *lbvs* = *lbv* +
  fixes s_0 :: *'a*
  fixes *c* :: *'a list*
  fixes *ins* :: *'b list*
  fixes τs :: *'a list*
  defines *phi-def*:
  $\tau s \equiv map\ (\lambda pc.\ if\ c!pc = \bot\ then\ wtl\ (take\ pc\ ins)\ c\ 0\ s_0\ else\ c!pc)$
  $[0..size\ ins(]$

  assumes *bounded*: *bounded step* (*size ins*)
  assumes *cert*: *cert-ok c* (*size ins*) $\top$ $\bot$ *A*
  assumes *pres*: *pres-type step* (*size ins*) *A*

lemma (**in** *lbvs*) *phi-None* [*intro?*]:
  $[\![\ pc < size\ ins;\ c!pc = \bot\ ]\!] \Longrightarrow \tau s!pc = wtl\ (take\ pc\ ins)\ c\ 0\ s_0$
lemma (**in** *lbvs*) *phi-Some* [*intro?*]:
  $[\![\ pc < size\ ins;\ c!pc \neq \bot\ ]\!] \Longrightarrow \tau s!pc = c!pc$
lemma (**in** *lbvs*) *phi-len* [*simp*]: $size\ \tau s = size\ ins$
lemma (**in** *lbvs*) *wtl-suc-pc*:
  assumes *all*: $wtl\ ins\ c\ 0\ s_0 \neq \top$
  assumes *pc*: $pc+1 < size\ ins$
  shows $wtl\ (take\ (pc+1)\ ins)\ c\ 0\ s_0 \sqsubseteq_r \tau s!(pc+1)$
lemma (**in** *lbvs*) *wtl-stable*:
  assumes *wtl*: $wtl\ ins\ c\ 0\ s_0 \neq \top$
  assumes s_0: $s_0 \in A$ **and** *pc*: $pc < size\ ins$
  shows *stable r step* τs *pc*
lemma (**in** *lbvs*) *phi-not-top*:
  assumes *wtl*: $wtl\ ins\ c\ 0\ s_0 \neq \top$ **and** *pc*: $pc < size\ ins$
  shows $\tau s!pc \neq \top$
lemma (**in** *lbvs*) *phi-in-A*:
  assumes *wtl*: $wtl\ ins\ c\ 0\ s_0 \neq \top$ **and** s_0: $s_0 \in A$
  shows $\tau s \in list\ (size\ ins)\ A$
lemma (**in** *lbvs*) *phi0*:
  assumes *wtl*: $wtl\ ins\ c\ 0\ s_0 \neq \top$ **and** *0*: $0 < size\ ins$
  shows $s_0 \sqsubseteq_r \tau s!0$

theorem (**in** *lbvs*) *wtl-sound*:
  assumes $wtl\ ins\ c\ 0\ s_0 \neq \top$ **and** $s_0 \in A$
  shows $\exists \tau s.\ wt\text{-}step\ r\ \top\ step\ \tau s$

theorem (**in** *lbvs*) *wtl-sound-strong*:
  assumes $wtl\ ins\ c\ 0\ s_0 \neq \top$
  assumes $s_0 \in A$ **and** $0 < size\ ins$
  shows $\exists \tau s \in list\ (size\ ins)\ A.\ wt\text{-}step\ r\ \top\ step\ \tau s \wedge s_0 \sqsubseteq_r \tau s!0$
end

4.12 Completeness of the LBV

theory *LBVComplete* = *LBVSpec* + *Typing-Framework*:

constdefs
  is-target :: $['s\ \textit{step-type}, 's\ \textit{list}, \textit{nat}] \Rightarrow \textit{bool}$
  is-target step τs *pc'* $\equiv$
    $\exists\, pc\ s'.\ pc' \neq pc{+}1 \wedge pc < \textit{size}\ \tau s \wedge (pc',s') \in \textit{set}\ (\textit{step}\ pc\ (\tau s!pc))$

  make-cert :: $['s\ \textit{step-type}, 's\ \textit{list}, 's] \Rightarrow 's\ \textit{certificate}$
  make-cert step τs *B* $\equiv$
    $\textit{map}\ (\lambda pc.\ \textit{if is-target step}\ \tau s\ pc\ \textit{then}\ \tau s!pc\ \textit{else}\ B)\ [0..\textit{size}\ \tau s(] \ @\ [B]$

  For the code generator:

constdefs
  list-ex :: $('a \Rightarrow \textit{bool}) \Rightarrow 'a\ \textit{list} \Rightarrow \textit{bool}$
  list-ex P xs $\equiv \exists\, x \in \textit{set}\ xs.\ P\ x$

lemma [*code*]: *list-ex P* [] = *False* **by** (*simp add*: *list-ex-def*)
lemma [*code*]: *list-ex P* $(x\#xs) = (P\ x \vee \textit{list-ex}\ P\ xs)$ **by** (*simp add*: *list-ex-def*)

lemma [*code*]:
  is-target step τs *pc'* =
  $\textit{list-ex}\ (\lambda pc.\ pc' \neq pc{+}1 \wedge pc'\ \textit{mem}\ (\textit{map fst}\ (\textit{step}\ pc\ (\tau s!pc))))\ [0..\textit{size}\ \tau s(]$
locale (**open**) *lbvc* = *lbv* +
  fixes τs :: *'a list*
  fixes *c* :: *'a list*
  defines *cert-def*: $c \equiv \textit{make-cert step}\ \tau s\ \bot$

  assumes *mono*: *mono r step* (*size* τs) *A*
  assumes *pres*: *pres-type step* (*size* τs) *A*
  assumes τs: $\forall\, pc < \textit{size}\ \tau s.\ \tau s!pc \in A \wedge \tau s!pc \neq \top$
  assumes *bounded*: *bounded step* (*size* τs)

  assumes *B-neq-T*: $\bot \neq \top$

lemma (**in** *lbvc*) *cert*: *cert-ok c* (*size* τs) $\top$ $\bot$ *A*
lemmas [*simp del*] = *split-paired-Ex*

lemma (**in** *lbvc*) *cert-target* [*intro?*]:
  $[\![\ (pc',s') \in \textit{set}\ (\textit{step}\ pc\ (\tau s!pc));$
    $pc' \neq pc{+}1;\ pc < \textit{size}\ \tau s;\ pc' < \textit{size}\ \tau s\ ]\!]$
  $\Longrightarrow c!pc' = \tau s!pc'$
lemma (**in** *lbvc*) *cert-approx* [*intro?*]:
  $[\![\ pc < \textit{size}\ \tau s;\ c!pc \neq \bot\ ]\!] \Longrightarrow c!pc = \tau s!pc$
lemma (**in** *lbv*) *le-top* [*simp*, *intro*]: *x* <=-*r* $\top$
lemma (**in** *lbv*) *merge-mono*:
  assumes *less*: $\textit{set}\ ss_2\ \{\sqsubseteq_r\}\ \textit{set}\ ss_1$
  assumes *x*: $x \in A$
  assumes ss_1: $\textit{snd`set}\ ss_1 \subseteq A$
  assumes ss_2: $\textit{snd`set}\ ss_2 \subseteq A$
  shows $\textit{merge}\ c\ pc\ ss_2\ x \sqsubseteq_r \textit{merge}\ c\ pc\ ss_1\ x$ (**is** $?s_2 \sqsubseteq_r\ ?s_1$)
lemma (**in** *lbvc*) *wti-mono*:

assumes *less*: $s_2 \sqsubseteq_r s_1$
assumes *pc*: $pc < size\ \tau s$ **and** s_1: $s_1 \in A$ **and** s_2: $s_2 \in A$
shows $wti\ c\ pc\ s_2 \sqsubseteq_r wti\ c\ pc\ s_1$ (**is** $?s_2{}' \sqsubseteq_r ?s_1{}'$)
lemma (**in** *lbvc*) *wtc-mono*:
assumes *less*: $s_2 \sqsubseteq_r s_1$
assumes *pc*: $pc < size\ \tau s$ **and** s_1: $s_1 \in A$ **and** s_2: $s_2 \in A$
shows $wtc\ c\ pc\ s_2 \sqsubseteq_r wtc\ c\ pc\ s_1$ (**is** $?s_2{}' \sqsubseteq_r ?s_1{}'$)
lemma (**in** *lbv*) *top-le-conv* [*simp*]: $\top \sqsubseteq_r x = (x = \top)$
lemma (**in** *lbv*) *neq-top* [*simp*, *elim*]: $[\![x \sqsubseteq_r y;\ y \neq \top]\!] \Longrightarrow x \neq \top$
lemma (**in** *lbvc*) *stable-wti*:
assumes *stable*: *stable r step* τs *pc* **and** *pc*: $pc < size\ \tau s$
shows $wti\ c\ pc\ (\tau s!pc) \neq \top$
lemma (**in** *lbvc*) *wti-less*:
assumes *stable*: *stable r step* τs *pc* **and** *suc-pc*: $Suc\ pc < size\ \tau s$
shows $wti\ c\ pc\ (\tau s!pc) \sqsubseteq_r \tau s!Suc\ pc$ (**is** $?wti \sqsubseteq_r$ -)
lemma (**in** *lbvc*) *stable-wtc*:
assumes *stable*: *stable r step* τs *pc* **and** *pc*: $pc < size\ \tau s$
shows $wtc\ c\ pc\ (\tau s!pc) \neq \top$
lemma (**in** *lbvc*) *wtc-less*:
assumes *stable*: *stable r step* τs *pc* **and** *suc-pc*: $Suc\ pc < size\ \tau s$
shows $wtc\ c\ pc\ (\tau s!pc) \sqsubseteq_r \tau s!Suc\ pc$ (**is** $?wtc \sqsubseteq_r$ -)
lemma (**in** *lbvc*) *wt-step-wtl-lemma*:
assumes *wt-step*: *wt-step r* $\top$ *step* τs
shows $\bigwedge pc\ s.\ pc+size\ ls = size\ \tau s \Longrightarrow s \sqsubseteq_r \tau s!pc \Longrightarrow s \in A \Longrightarrow s \neq \top \Longrightarrow$
$wtl\ ls\ c\ pc\ s \neq \top$
(**is** $\bigwedge pc\ s.$ - $\Longrightarrow$ - $\Longrightarrow$ - $\Longrightarrow$ - $\Longrightarrow$ $?wtl\ ls\ pc\ s \neq$ -)
theorem (**in** *lbvc*) *wtl-complete*:
assumes *wt-step r* $\top$ *step* τs
assumes $s \sqsubseteq_r \tau s!0$ **and** $s \in A$ **and** $s \neq \top$ **and** $size\ ins = size\ \tau s$
shows $wtl\ ins\ c\ 0\ s \neq \top$
end

4.13 The Jinja Type System as a Semilattice

theory *SemiType* = *WellForm* + *Semilattices*:

constdefs
  $super :: 'a\ prog \Rightarrow cname \Rightarrow cname$
  $super\ P\ C \equiv fst\ (the\ (class\ P\ C))$

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

consts
  $the\text{-}Class :: ty \Rightarrow cname$
primrec
  $the\text{-}Class\ (Class\ C) = C$

constdefs
  $sup :: 'c\ prog \Rightarrow ty \Rightarrow ty \Rightarrow ty\ err$
  $sup\ P\ T_1\ T_2 \equiv$
  $if\ is\text{-}refT\ T_1 \wedge is\text{-}refT\ T_2\ then$
  $OK\ (if\ T_1 = NT\ then\ T_2\ else$
    $if\ T_2 = NT\ then\ T_1\ else$
    $(Class\ (exec\text{-}lub\ (subcls1\ P)\ (super\ P)\ (the\text{-}Class\ T_1)\ (the\text{-}Class\ T_2))))$
  $else$
  $(if\ T_1 = T_2\ then\ OK\ T_1\ else\ Err)$

syntax
  $subtype :: 'c\ prog \Rightarrow ty \Rightarrow ty \Rightarrow bool$
translations
  $subtype\ P == fun\text{-}of\ (widen\ P)$

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

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

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

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

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*:
$[\![\ subtype\ P\ a\ c;\ subtype\ P\ b\ c\ ]\!] \Longrightarrow EX\ T.\ sup\ P\ a\ b = OK\ T$
lemma *err-semilat-JType-esl*:
$wf\text{-}prog\ wf\text{-}mb\ P \Longrightarrow err\text{-}semilat\ (esl\ P)$

end

4.14 The JVM Type System as Semilattice

theory *JVM-SemiType* = *SemiType*:

types ty_l = *ty err list*
types ty_s = *ty list*
types $ty_i = ty_s \times ty_l$
types $ty_i{}'$ = ty_i *option*
types ty_m = $ty_i{}'$ *list*
types ty_P = *mname* ⇒ *cname* ⇒ ty_m

constdefs
stk-esl :: *'c prog* ⇒ *nat* ⇒ ty_s *esl*
stk-esl P mxs ≡ *upto-esl mxs* (*SemiType.esl P*)

loc-sl :: *'c prog* ⇒ *nat* ⇒ ty_l *sl*
loc-sl P mxl ≡ *Listn.sl mxl* (*Err.sl* (*SemiType.esl P*))

sl :: *'c prog* ⇒ *nat* ⇒ *nat* ⇒ $ty_i{}'$ *err sl*
sl P mxs mxl ≡
Err.sl(*Opt.esl*(*Product.esl* (*stk-esl P mxs*) (*Err.esl*(*loc-sl P mxl*))))

constdefs
states :: *'c prog* ⇒ *nat* ⇒ *nat* ⇒ $ty_i{}'$ *err set*
states P mxs mxl ≡ *fst*(*sl P mxs mxl*)

le :: *'c prog* ⇒ *nat* ⇒ *nat* ⇒ $ty_i{}'$ *err ord*
le P mxs mxl ≡ *fst*(*snd*(*sl P mxs mxl*))

sup :: *'c prog* ⇒ *nat* ⇒ *nat* ⇒ $ty_i{}'$ *err binop*
sup P mxs mxl ≡ *snd*(*snd*(*sl P mxs mxl*))

constdefs
sup-ty-opt :: [*'c prog*,*ty err*,*ty err*] ⇒ *bool*
(- |− - <=T - [71,71,71] 70)
sup-ty-opt P ≡ *Err.le* (*subtype P*)

sup-state :: [*'c prog*,ty_i,ty_i] ⇒ *bool*
(- |− - <=i - [71,71,71] 70)
sup-state P ≡ *Product.le* (*Listn.le* (*subtype P*)) (*Listn.le* (*sup-ty-opt P*))

sup-state-opt :: [*'c prog*,$ty_i{}'$,$ty_i{}'$] ⇒ *bool*
(- |− - <=' - [71,71,71] 70)
sup-state-opt P ≡ *Opt.le* (*sup-state P*)

syntax
sup-loc :: [*'c prog*,ty_l,ty_l] ⇒ *bool*
(- |− - [<=T] - [71,71,71] 70)

syntax (*xsymbols*)

sup-ty-opt :: $['c\ prog,\ ty\ err,\ ty\ err] \Rightarrow bool$
$(\text{-} \vdash \text{-} \le_\top \text{-}\ [71,71,71]\ 70)$
sup-loc :: $['c\ prog,\ ty_l,\ ty_l] \Rightarrow bool$
$(\text{-} \vdash \text{-}\ [\le_\top]\ \text{-}\ [71,71,71]\ 70)$
sup-state :: $['c\ prog,\ ty_i,\ ty_i] \Rightarrow bool$
$(\text{-} \vdash \text{-} \le_i \text{-}\ [71,71,71]\ 70)$
sup-state-opt :: $['c\ prog,\ ty_i',\ ty_i'] \Rightarrow bool$
$(\text{-} \vdash \text{-} \le' \text{-}\ [71,71,71]\ 70)$

translations
$P \vdash LT\ [\le_\top]\ LT' == list\text{-}all2\ (sup\text{-}ty\text{-}opt\ P)\ LT\ LT'$

4.14.1 Unfolding

lemma *JVM-states-unfold*:
$states\ P\ mxs\ mxl \equiv err(opt((Union\ \{list\ n\ (types\ P)\ |n.\ n <= mxs\})\ <*>$
$list\ mxl\ (err(types\ P))))$
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*:
$le\ P\ m\ n\ (OK\ t1)\ (OK\ t2) = P \vdash t1 \le' t2$
lemma *JVM-le-Err-conv*:
$le\ P\ m\ n = Err.le\ (sup\text{-}state\text{-}opt\ P)$
lemma *err-le-unfold* [*iff*]:
$Err.le\ r\ (OK\ a)\ (OK\ b) = r\ a\ b$

4.14.2 Semilattice

lemma *order-sup-state-opt* [*intro*, *simp*]:
$wf\text{-}prog\ wf\text{-}mb\ P \Longrightarrow order\ (sup\text{-}state\text{-}opt\ P)$

lemma *semilat-JVM* [*intro?*]:
$wf\text{-}prog\ wf\text{-}mb\ P \Longrightarrow semilat\ (JVM\text{-}SemiType.sl\ P\ mxs\ mxl)$
lemma *acc-JVM* [*intro*]:
$wf\text{-}prog\ wf\text{-}mb\ P \Longrightarrow acc\ (JVM\text{-}SemiType.le\ P\ mxs\ mxl)$

4.14.3 Widening with $\top$

lemma *subtype-refl*[*iff*]: $subtype\ P\ t\ t$
lemma *sup-ty-opt-refl* [*iff*]: $P \vdash T \le_\top T$
lemma *Err-any-conv* [*iff*]: $P \vdash Err \le_\top T = (T = Err)$
lemma *any-Err* [*iff*]: $P \vdash T \le_\top Err$
lemma *OK-OK-conv* [*iff*]:
$P \vdash OK\ T \le_\top OK\ T' = P \vdash T \le T'$
lemma *any-OK-conv* [*iff*]:
$P \vdash X \le_\top OK\ T' = (\exists T.\ X = OK\ T \wedge P \vdash T \le T')$
lemma *OK-any-conv*:
$P \vdash OK\ T \le_\top X = (X = Err \vee (\exists T'.\ X = OK\ T' \wedge P \vdash T \le T'))$
lemma *sup-ty-opt-trans* [*intro?*, *trans*]:

$[\![P \vdash a \leq_\top b;\ P \vdash b \leq_\top c]\!] \Longrightarrow P \vdash a \leq_\top c$

4.14.4 Stack and Registers

lemma *stk-convert*:
$P \vdash ST\ [\leq]\ ST' = \mathit{Listn.le}\ (\mathit{subtype}\ P)\ ST\ ST'$
lemma *sup-loc-refl* [*iff*]: $P \vdash LT\ [\leq_\top]\ LT$
lemmas *sup-loc-Cons1* [*iff*] = *list-all2-Cons1* [*of sup-ty-opt P*, *standard*]

lemma *sup-loc-def*:
$P \vdash LT\ [\leq_\top]\ LT' \equiv \mathit{Listn.le}\ (\mathit{sup\text{-}ty\text{-}opt}\ P)\ LT\ LT'$
lemma *sup-loc-widens-conv* [*iff*]:
$P \vdash \mathit{map}\ OK\ Ts\ [\leq_\top]\ \mathit{map}\ OK\ Ts' = P \vdash Ts\ [\leq]\ Ts'$

lemma *sup-loc-trans* [*intro?*, *trans*]:
$[\![P \vdash a\ [\leq_\top]\ b;\ P \vdash b\ [\leq_\top]\ c]\!] \Longrightarrow P \vdash a\ [\leq_\top]\ c$

4.14.5 State Type

lemma *sup-state-conv* [*iff*]:
$P \vdash (ST,LT) \leq_i (ST',LT') = (P \vdash ST\ [\leq]\ ST' \wedge P \vdash LT\ [\leq_\top]\ LT')$
lemma *sup-state-conv2*:
$P \vdash s1 \leq_i s2 = (P \vdash \mathit{fst}\ s1\ [\leq]\ \mathit{fst}\ s2 \wedge P \vdash \mathit{snd}\ s1\ [\leq_\top]\ \mathit{snd}\ s2)$
lemma *sup-state-refl* [*iff*]: $P \vdash s \leq_i s$
lemma *sup-state-trans* [*intro?*, *trans*]:
$[\![P \vdash a \leq_i b;\ P \vdash b \leq_i c]\!] \Longrightarrow P \vdash a \leq_i c$

lemma *sup-state-opt-None-any* [*iff*]:
$P \vdash \mathit{None} \leq' s$
lemma *sup-state-opt-any-None* [*iff*]:
$P \vdash s \leq' \mathit{None} = (s = \mathit{None})$
lemma *sup-state-opt-Some-Some* [*iff*]:
$P \vdash \mathit{Some}\ a \leq' \mathit{Some}\ b = P \vdash a \leq_i b$

lemma *sup-state-opt-any-Some*:
$P \vdash (\mathit{Some}\ s) \leq' X = (\exists s'.\ X = \mathit{Some}\ s' \wedge P \vdash s \leq_i s')$
lemma *sup-state-opt-refl* [*iff*]: $P \vdash s \leq' s$
lemma *sup-state-opt-trans* [*intro?*, *trans*]:
$[\![P \vdash a \leq' b;\ P \vdash b \leq' c]\!] \Longrightarrow P \vdash a \leq' c$
end

4.15 Effect of Instructions on the State Type

theory *Effect* = *JVM-SemiType* + *JVMExceptions*:

— 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:

consts
  succs :: $instr \Rightarrow ty_i \Rightarrow pc \Rightarrow pc\ list$
primrec
  $succs\ (Load\ idx)\ \tau\ pc = [pc{+}1]$
  $succs\ (Store\ idx)\ \tau\ pc = [pc{+}1]$
  $succs\ (Push\ v)\ \tau\ pc = [pc{+}1]$
  $succs\ (Getfield\ F\ C)\ \tau\ pc = [pc{+}1]$
  $succs\ (Putfield\ F\ C)\ \tau\ pc = [pc{+}1]$
  $succs\ (New\ C)\ \tau\ pc = [pc{+}1]$
  $succs\ (Checkcast\ C)\ \tau\ pc = [pc{+}1]$
  $succs\ Pop\ \tau\ pc = [pc{+}1]$
  $succs\ IAdd\ \tau\ pc = [pc{+}1]$
  $succs\ CmpEq\ \tau\ pc = [pc{+}1]$
succs-IfFalse:
  $succs\ (IfFalse\ b)\ \tau\ pc = [pc{+}1,\ nat\ (int\ pc + b)]$
succs-Goto:
  $succs\ (Goto\ b)\ \tau\ pc = [nat\ (int\ pc + b)]$
succs-Return:
  $succs\ Return\ \tau\ pc = []$
succs-Invoke:
  $succs\ (Invoke\ M\ n)\ \tau\ pc = (if\ (fst\ \tau)!n = NT\ then\ []\ else\ [pc{+}1])$
succs-Throw:
  $succs\ Throw\ \tau\ pc = []$

Effect of instruction on the state type:

consts *the-class*:: $ty \Rightarrow cname$
recdef *the-class* {}
the-class(*Class C*) = *C*

consts
$eff_i :: instr \times {'m}\ prog \times ty_i \Rightarrow ty_i$

recdef eff_i {}
eff_i*-Load*:

eff_i (*Load n*, *P*, (*ST*, *LT*)) = (*ok-val* (*LT* ! *n*) # *ST*, *LT*)
eff_i-*Store*:
eff_i (*Store n*, *P*, (*T*#*ST*, *LT*)) = (*ST*, *LT*[*n*:= *OK T*])
eff_i-*Push*:
eff_i (*Push v*, *P*, (*ST*, *LT*)) = (*the* (*typeof v*) # *ST*, *LT*)
eff_i-*Getfield*:
eff_i (*Getfield F C*, *P*, (*T*#*ST*, *LT*)) = (*snd* (*field P C F*) # *ST*, *LT*)
eff_i-*Putfield*:
eff_i (*Putfield F C*, *P*, (T_1#T_2#*ST*, *LT*)) = (*ST*,*LT*)
eff_i-*New*:
eff_i (*New C*, *P*, (*ST*,*LT*)) = (*Class C* # *ST*, *LT*)
eff_i-*Checkcast*:
eff_i (*Checkcast C*, *P*, (*T*#*ST*,*LT*)) = (*Class C* # *ST*,*LT*)
eff_i-*Pop*:
eff_i (*Pop*, *P*, (*T*#*ST*,*LT*)) = (*ST*,*LT*)
eff_i-*IAdd*:
eff_i (*IAdd*, *P*,(T_1#T_2#*ST*,*LT*)) = (*Integer*#*ST*,*LT*)
eff_i-*CmpEq*:
eff_i (*CmpEq*, *P*, (T_1#T_2#*ST*,*LT*)) = (*Boolean*#*ST*,*LT*)
eff_i-*IfFalse*:
eff_i (*IfFalse b*, *P*, (T_1#*ST*,*LT*)) = (*ST*,*LT*)
eff_i-*Invoke*:
eff_i (*Invoke M n*, *P*, (*ST*,*LT*)) =
  (*let C* = *the-class* (*ST*!*n*); (*D*,*Ts*,T_r,*b*) = *method P C M*
   in (T_r # *drop* (*n*+*1*) *ST*, *LT*))
eff_i-*Goto*:
eff_i (*Goto n*, *P*, *s*) = *s*

consts
  is-relevant-class :: *instr* ⇒ *'m prog* ⇒ *cname* ⇒ *bool*
recdef *is-relevant-class* {}
rel-Getfield:
  is-relevant-class (*Getfield F D*) = (λ*P C*. *P* ⊢ *NullPointer* $\preceq^*$ *C*)
rel-Putfield:
  is-relevant-class (*Putfield F D*) = (λ*P C*. *P* ⊢ *NullPointer* $\preceq^*$ *C*)
rel-Checcast:
  is-relevant-class (*Checkcast D*) = (λ*P C*. *P* ⊢ *ClassCast* $\preceq^*$ *C*)
rel-New:
  is-relevant-class (*New D*) = (λ*P C*. *P* ⊢ *OutOfMemory* $\preceq^*$ *C*)
rel-Throw:
  is-relevant-class Throw = (λ*P C*. *True*)
rel-Invoke:
  is-relevant-class (*Invoke M n*) = (λ*P C*. *True*)
rel-default:
  is-relevant-class i = (λ*P C*. *False*)

constdefs
  is-relevant-entry :: *'m prog* ⇒ *instr* ⇒ *pc* ⇒ *ex-entry* ⇒ *bool*
  is-relevant-entry P i pc e ≡ *let* (*f*,*t*,*C*,*h*,*d*) = *e in is-relevant-class i P C* ∧ *pc* ∈ {*f*..*t*(}

  relevant-entries :: *'m prog* ⇒ *instr* ⇒ *pc* ⇒ *ex-table* ⇒ *ex-table*
  relevant-entries P i pc ≡ *filter* (*is-relevant-entry P i pc*)

$xcpt\text{-}eff :: instr \Rightarrow {'m}\ prog \Rightarrow pc \Rightarrow ty_i$
$\quad \Rightarrow ex\text{-}table \Rightarrow (pc \times ty_i{}')\ list$
$xcpt\text{-}eff\ i\ P\ pc\ \tau\ et \equiv let\ (ST,LT) = \tau\ in$
$map\ (\lambda(f,t,C,h,d).\ (h,\ Some\ (Class\ C\#drop\ (size\ ST\ -\ d)\ ST,\ LT)))\ (relevant\text{-}entries\ P\ i\ pc\ et)$

$norm\text{-}eff :: instr \Rightarrow {'m}\ prog \Rightarrow nat \Rightarrow ty_i \Rightarrow (pc \times ty_i{}')\ list$
$norm\text{-}eff\ i\ P\ pc\ \tau \equiv map\ (\lambda pc'.\ (pc',Some\ (eff_i\ (i,P,\tau))))\ (succs\ i\ \tau\ pc)$

$eff :: instr \Rightarrow {'m}\ prog \Rightarrow pc \Rightarrow ex\text{-}table \Rightarrow ty_i{}' \Rightarrow (pc \times ty_i{}')\ list$
$eff\ i\ P\ pc\ et\ t \equiv$
$case\ t\ of$
$\quad None \Rightarrow []$
$|\ Some\ \tau \Rightarrow (norm\text{-}eff\ i\ P\ pc\ \tau)\ @\ (xcpt\text{-}eff\ i\ P\ pc\ \tau\ et)$

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

lemma *eff-Some*:
$eff\ i\ P\ pc\ xt\ (Some\ \tau) = norm\text{-}eff\ i\ P\ pc\ \tau\ @\ xcpt\text{-}eff\ i\ P\ pc\ \tau\ xt$
by (*simp add*: *eff-def*)

Conditions under which eff is applicable:

consts
$app_i :: instr \times {'m}\ prog \times pc \times nat \times ty \times ty_i \Rightarrow bool$

recdef app_i {}
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*:
$app_i\ (Push\ v,\ P,\ pc,\ mxs,\ T_r,\ (ST,LT)) =$
$(length\ ST < mxs \wedge typeof\ v \neq None)$
app_i-*Getfield*:
$app_i\ (Getfield\ F\ C,\ P,\ pc,\ mxs,\ T_r,\ (T\#ST,\ LT)) =$
$(\exists\ T_f.\ P \vdash C\ sees\ F{:}T_f\ in\ C \wedge P \vdash T \leq Class\ C)$
app_i-*Putfield*:
$app_i\ (Putfield\ F\ C,\ P,\ pc,\ mxs,\ T_r,\ (T_1\#T_2\#ST,\ LT)) =$
$(\exists\ T_f.\ P \vdash C\ sees\ F{:}T_f\ in\ C \wedge P \vdash T_2 \leq (Class\ C) \wedge P \vdash T_1 \leq T_f)$
app_i-*New*:
$app_i\ (New\ C,\ P,\ pc,\ mxs,\ T_r,\ (ST,LT)) =$
$(is\text{-}class\ P\ C \wedge length\ ST < mxs)$
app_i-*Checkcast*:
$app_i\ (Checkcast\ C,\ P,\ pc,\ mxs,\ T_r,\ (T\#ST,LT)) =$
$(is\text{-}class\ P\ C \wedge is\text{-}refT\ T)$
app_i-*Pop*:
$app_i\ (Pop,\ P,\ pc,\ mxs,\ T_r,\ (T\#ST,LT)) =$
$True$
app_i-*IAdd*:
$app_i\ (IAdd,\ P,\ pc,\ mxs,\ T_r,\ (T_1\#T_2\#ST,LT)) = (T_1 = T_2 \wedge T_1 = Integer)$

app_i-*CmpEq*:
app_i (*CmpEq*, P, *pc*, *mxs*, T_r, (T_1#T_2#*ST*,*LT*)) =
(T_1 = T_2 ∨ *is-refT* T_1 ∧ *is-refT* T_2)
app_i-*IfFalse*:
app_i (*IfFalse* *b*, *P*, *pc*, *mxs*, T_r, (*Boolean*#*ST*,*LT*)) =
(*0* ≤ *int pc* + *b*)
app_i-*Goto*:
app_i (*Goto* *b*, *P*, *pc*, *mxs*, T_r, *s*) =
(*0* ≤ *int pc* + *b*)
app_i-*Return*:
app_i (*Return*, *P*, *pc*, *mxs*, T_r, (*T*#*ST*,*LT*)) =
(*P* ⊢ *T* ≤ T_r)
app_i-*Throw*:
app_i (*Throw*, *P*, *pc*, *mxs*, T_r, (*T*#*ST*,*LT*)) =
is-refT T
app_i-*Invoke*:
app_i (*Invoke M n*, *P*, *pc*, *mxs*, T_r, (*ST*,*LT*)) =
(*n* < *length ST* ∧
(*ST*!*n* ≠ *NT* ⟶
(∃ *C D Ts T m*. *ST*!*n* = *Class C* ∧ *P* ⊢ *C sees M*:*Ts* → *T* = *m in D* ∧
P ⊢ *rev* (*take n ST*) [≤] *Ts*)))

app_i-*default*:
app_i (*i*,*P*, *pc*,*mxs*,T_r,*s*) = *False*

constdefs
xcpt-app :: *instr* ⇒ ′*m prog* ⇒ *pc* ⇒ *nat* ⇒ *ex-table* ⇒ ty_i ⇒ *bool*
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*

app :: *instr* ⇒ ′*m prog* ⇒ *nat* ⇒ *ty* ⇒ *nat* ⇒ *nat* ⇒ *ex-table* ⇒
ty_i′ ⇒ *bool*
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* $\tau \equiv$ *Effect.*app_i (i, P, pc, mxs, T_r, τ)
notes app_i*-simps* [*simp*] = *Effect.*app_i*.simps* [**where** P=P **and** *mxs*=*mxs* **and** T_r=T_r, *folded* app_i*-def*]

defines *xcpt-eff i pc* $\tau \equiv$ *Effect.xcpt-eff i P pc* τ *xt*
notes *xcpt-eff* = *Effect.xcpt-eff-def* [*of - P - - xt*, *folded xcpt-eff-def*]

defines *norm-eff i pc* $\tau \equiv$ *Effect.norm-eff i P pc* τ
notes *norm-eff* = *Effect.norm-eff-def* [*of - P*, *folded norm-eff-def* eff_i*-def*]

defines *eff i pc* $\equiv$ *Effect.eff i P pc xt*
notes *eff* = *Effect.eff-def* [*of - P - xt*, *folded eff-def norm-eff-def xcpt-eff-def*]

defines *xcpt-app i pc* $\tau \equiv$ *Effect.xcpt-app i P pc mxs xt* τ
notes *xcpt-app* = *Effect.xcpt-app-def* [*of - P - mxs xt*, *folded xcpt-app-def*]

defines *app i pc* $\equiv$ *Effect.app i P mxs* T_r *pc mpc xt*
notes *app* = *Effect.app-def* [*of - P mxs* T_r *- mpc xt*, *folded app-def xcpt-app-def* app_i*-def eff-def*]

lemma *length-cases2*:
assumes $\bigwedge LT.\ P\ ([],LT)$
assumes $\bigwedge l\ ST\ LT.\ P\ (l\#ST,LT)$
shows $P\ s$
by (*cases s*, *cases fst s*, *auto*)

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:vT *in* C $\wedge$ $P \vdash oT \leq$ (*Class* C))
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:vT' *in* C $\wedge$ $P \vdash oT \leq$ (*Class* C) $\wedge$ $P \vdash vT \leq vT'$)
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:

declare *list-all2-Nil* [*code*]
declare *list-all2-Cons* [*code*]

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*) *fastsimp*

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

4.16 Monotonicity of eff and app

theory *EffectMono* = *Effect*:

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*: $succs\ i\ \tau\ pc \neq []\quad succs\ i\ \tau'\ pc \neq []$
shows $P \vdash eff_i\ (i,P,\tau) \leq_i eff_i\ (i,P,\tau')$
end

4.17 The Bytecode Verifier

theory *BVSpec* = *Effect*:

This theory contains a specification of the BV. The specification describes correct typings of method bodies; it corresponds to type *checking*.

constdefs
— The method type only contains declared classes:
$check\text{-}types :: {}'m\ prog \Rightarrow nat \Rightarrow nat \Rightarrow ty_i{}'\ err\ list \Rightarrow bool$
$check\text{-}types\ P\ mxs\ mxl\ \tau s \equiv set\ \tau s \subseteq states\ P\ mxs\ mxl$

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

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

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

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

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

syntax
$wf\text{-}jvm\text{-}prog\text{-}phi :: ty_P \Rightarrow jvm\text{-}prog \Rightarrow bool\ (wf{}'\text{-}jvm{}'\text{-}prog_{\text{-}}\ \text{-}\ [0,999]\ 1000)$
translations
$wf\text{-}jvm\text{-}prog_{\Phi}\ P <= wf\text{-}jvm\text{-}prog_{\Phi}\ P$

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*:

$\llbracket$ *wf-jvm-prog*$_\Phi$ P;
$P \vdash C$ *sees* $M{:}Ts \to T = (mxs, mxl_0, ins, xt)$ *in* C; $pc <$ *size ins* $\rrbracket$
$\Longrightarrow P, T, mxs, size\ ins, xt \vdash ins!pc, pc :: \Phi\ C\ M$

lemma *wt-jvm-prog-impl-wt-start*:
$\llbracket$ *wf-jvm-prog*$_\Phi$ P;
$P \vdash C$ *sees* $M{:}Ts \to T = (mxs, mxl_0, ins, xt)$ *in* C $\rrbracket \Longrightarrow$
$0 <$ *size ins* $\wedge$ *wt-start* $P\ C\ Ts\ mxl_0\ (\Phi\ C\ M)$

end

4.18 The Typing Framework for the JVM

theory *TF-JVM* = *Typing-Framework-err* + *EffectMono* + *BVSpec*:

constdefs
exec :: *jvm-prog* ⇒ *nat* ⇒ *ty* ⇒ *ex-table* ⇒ *instr list* ⇒ $ty_i{}'$ *err step-type*
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
fixes *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* ≡ *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*

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* ([],*OK* (*Class C*) # *map OK Ts* @ *replicate* mxl_0 *Err*)
defines [*simp*]:
start ≡ *OK first* # *replicate* (*size is* − *1*) (*OK None*)

4.18.1 Connecting JVM and Framework

lemma (**in** *JVM-sl*) *step-def-exec*: *step* ≡ *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*]:
(∃ *n*. *ST* ∈ *list n A* ∧ *n* ≤ *mxs*) = (*set ST* ⊆ *A* ∧ *size ST* ≤ *mxs*)
by (*unfold list-def*) *auto*

lemma *singleton-list*:
(∃ *n*. [*Class C*] ∈ *list n* (*types P*) ∧ *n* ≤ *mxs*) = (*is-class P C* ∧ *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*:
  $n < mxs \Longrightarrow Suc\ (n - (n - b)) \leq mxs$
  by *arith*

lemma *in-listE*:
  $[\![\ xs \in list\ n\ A;\ [\![size\ xs = n;\ set\ xs \subseteq A]\!] \Longrightarrow P\ ]\!] \Longrightarrow P$
  by (*unfold list-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*:
  $xs\ [\sqsubseteq_{Product.le\ (op\ =)\ r}]\ ys \Longrightarrow set\ xs\ \{\sqsubseteq_r\}\ set\ ys$
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*:
  $[\![wf\text{-}prog\ p\ P;\ pc < length\ is;\ s \sqsubseteq_{sup\text{-}state\text{-}opt\ P}\ t;\ app\ pc\ t]\!]$
  $\Longrightarrow set\ (eff\ pc\ s)\ \{\sqsubseteq_{sup\text{-}state\text{-}opt\ P}\}\ set\ (eff\ pc\ t)$
lemma (**in** *JVM-sl*) *bounded-step*: *bounded step* (*size is*)
theorem (**in** *JVM-sl*) *step-mono*:
  $wf\text{-}prog\ wf\text{-}mb\ P \Longrightarrow mono\ r\ step\ (size\ is)\ A$

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

lemma (**in** *JVM-sl*) *wt-method-def2*:
  $wt\text{-}method\ P\ C'\ Ts\ T_r\ mxs\ mxl_0\ is\ xt\ \tau s =$
  $(is \neq [] \wedge$
   $size\ \tau s = size\ is \wedge$
   $OK\ \text{`}\ set\ \tau s \subseteq states\ P\ mxs\ mxl \wedge$
   $wt\text{-}start\ P\ C'\ Ts\ mxl_0\ \tau s \wedge$
   $wt\text{-}app\text{-}eff\ (sup\text{-}state\text{-}opt\ P)\ app\ eff\ \tau s)$

end

4.19 Kildall for the JVM

theory *BVExec* = *Abstract-BV* + *TF-JVM*:

constdefs
kiljvm :: *jvm-prog* ⇒ *nat* ⇒ *nat* ⇒ *ty* ⇒
instr list ⇒ *ex-table* ⇒ ty_i′ *err list* ⇒ ty_i′ *err list*
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*)

wt-kildall :: *jvm-prog* ⇒ *cname* ⇒ *ty list* ⇒ *ty* ⇒ *nat* ⇒ *nat* ⇒
instr list ⇒ *ex-table* ⇒ *bool*
wt-kildall P C′ Ts T_r *mxs* mxl_0 *is xt* ≡
0 < *size is* ∧
(*let first* = *Some* ([],[*OK* (*Class C′*)]@(*map OK Ts*)@(*replicate* mxl_0 *Err*));
start = *OK first*#(*replicate* (*size is* − *1*) (*OK None*));
result = *kiljvm P mxs* (*1*+*size Ts*+mxl_0) T_r *is xt start*
in ∀ *n* < *size is*. *result*!*n* ≠ *Err*)

wf-jvm-prog$_k$ :: *jvm-prog* ⇒ *bool*
wf-jvm-prog$_k$ *P* ≡
wf-prog (λ*P C′* (*M*,*Ts*,T_r,(*mxs*,mxl_0,*is*,*xt*)). *wt-kildall P C′ Ts* T_r *mxs* mxl_0 *is xt*) *P*

theorem (**in** *start-context*) *is-bcv-kiljvm*:
is-bcv r Err step (*size is*) *A* (*kiljvm P mxs mxl* T_r *is xt*)

lemma *subset-replicate* [*intro?*]: *set* (*replicate n x*) ⊆ {*x*}
by (*induct n*) *auto*

lemma *in-set-replicate*:
assumes *x* ∈ *set* (*replicate n y*)
shows *x* = *y*
lemma (**in** *start-context*) *start-in-A* [*intro?*]:
0 < *size is* ⟹ *start* ∈ *list* (*size is*) *A*
using *Ts C*

theorem (**in** *start-context*) *wt-kil-correct*:
assumes *wtk*: *wt-kildall P C Ts* T_r *mxs* mxl_0 *is xt*
shows ∃τ*s*. *wt-method P C Ts* T_r *mxs* mxl_0 *is xt* τ*s*

theorem (**in** *start-context*) *wt-kil-complete*:
assumes *wtm*: *wt-method P C Ts* T_r *mxs* mxl_0 *is xt* τ*s*
shows *wt-kildall P C Ts* T_r *mxs* mxl_0 *is xt*

theorem *jvm-kildall-correct*:
wf-jvm-prog$_k$ *P* = *wf-jvm-prog P*
end

4.20 LBV for the JVM

theory *LBVJVM* = *Abstract-BV* + *TF-JVM*:

types *prog-cert* = *cname* ⇒ *mname* ⇒ ty_i′ *err list*

constdefs
check-cert :: *jvm-prog* ⇒ *nat* ⇒ *nat* ⇒ *nat* ⇒ ty_i′ *err list* ⇒ *bool*
check-cert P mxs mxl n cert ≡ *check-types P mxs mxl cert* ∧ *size cert* = *n*+*1* ∧
(∀ *i*<*n*. *cert*!*i* ≠ *Err*) ∧ *cert*!*n* = *OK None*

lbvjvm :: *jvm-prog* ⇒ *nat* ⇒ *nat* ⇒ *ty* ⇒ *ex-table* ⇒
ty_i′ *err list* ⇒ *instr list* ⇒ ty_i′ *err* ⇒ ty_i′ *err*
lbvjvm P mxs maxr T_r *et cert bs* ≡
wtl-inst-list bs cert (*JVM-SemiType.sup P mxs maxr*) (*JVM-SemiType.le P mxs maxr*) *Err* (*OK None*) (*exec P mxs* T_r *et bs*) *0*

wt-lbv :: *jvm-prog* ⇒ *cname* ⇒ *ty list* ⇒ *ty* ⇒ *nat* ⇒ *nat* ⇒
ex-table ⇒ ty_i′ *err list* ⇒ *instr list* ⇒ *bool*
wt-lbv P C Ts T_r *mxs* mxl_0 *et cert ins* ≡
check-cert P mxs (*1*+*size Ts*+mxl_0) (*size ins*) *cert* ∧
0 < *size ins* ∧
(*let start* = *Some* ([],(*OK* (*Class C*))#((*map OK Ts*))@(*replicate* mxl_0 *Err*));
result = *lbvjvm P mxs* (*1*+*size Ts*+mxl_0) T_r *et cert ins* (*OK start*)
in result ≠ *Err*)

wt-jvm-prog-lbv :: *jvm-prog* ⇒ *prog-cert* ⇒ *bool*
wt-jvm-prog-lbv P cert ≡
wf-prog (λ*P C* (*mn*,*Ts*,T_r,(*mxs*,mxl_0,*b*,*et*)). *wt-lbv P C Ts* T_r *mxs* mxl_0 *et* (*cert C mn*) *b*) *P*

mk-cert :: *jvm-prog* ⇒ *nat* ⇒ *ty* ⇒ *ex-table* ⇒ *instr list*
⇒ ty_m ⇒ ty_i′ *err list*
mk-cert P mxs T_r *et bs phi* ≡ *make-cert* (*exec P mxs* T_r *et bs*) (*map OK phi*) (*OK None*)

prg-cert :: *jvm-prog* ⇒ ty_P ⇒ *prog-cert*
prg-cert P phi C mn ≡ *let* (*C*,*Ts*,T_r,(*mxs*,mxl_0,*ins*,*et*)) = *method P C mn*
in *mk-cert P mxs* T_r *et ins* (*phi C mn*)

lemma *check-certD* [*intro?*]:
check-cert P mxs mxl n cert ⟹ *cert-ok cert n Err* (*OK None*) (*states P mxs mxl*)
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 Ts* T_r *mxs* mxl_0 *xt cert is*
shows ∃ τ*s* ∈ *list* (*size is*) *A*. *wt-step r Err step* τ*s* ∧ *OK first* $\sqsubseteq_r$ τ*s*!*0*

lemma (**in** *start-context*) *wt-lbv-wt-method*:
assumes *lbv*: *wt-lbv P C Ts* T_r *mxs* mxl_0 *xt cert is*
shows ∃ τ*s*. *wt-method P C Ts* T_r *mxs* mxl_0 *is xt* τ*s*

lemma (**in** *start-context*) *wt-method-wt-lbv*:
assumes *wt*: *wt-method P C Ts* T_r *mxs* mxl_0 *is xt* τ*s*
defines [*simp*]: *cert* ≡ *mk-cert P mxs* T_r *xt is* τ*s*

shows *wt-lbv P C 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

4.21 BV Type Safety Invariant

theory $BVConform = BVSpec + JVMExec + Conform$:

syntax (*xsymbols*)
  $confT :: {}'c\ prog \Rightarrow heap \Rightarrow val \Rightarrow ty\ err \Rightarrow bool$
    $(\text{-,-} \vdash \text{-} :\leq_\top \text{-}\ [51,51,51,51]\ 50)$

constdefs
  $confT :: {}'c\ prog \Rightarrow heap \Rightarrow val \Rightarrow ty\ err \Rightarrow bool$
    $(\text{-,-}\ |\!-\ \text{-} :<=T\ \text{-}\ [51,51,51,51]\ 50)$
  $P,h \vdash v :\leq_\top E \equiv case\ E\ of\ Err \Rightarrow True \mid OK\ T \Rightarrow P,h \vdash v :\leq T$

syntax
  $confTs :: {}'c\ prog \Rightarrow heap \Rightarrow val\ list \Rightarrow ty_l \Rightarrow bool$
    $(\text{-,-}\ |\!-\ \text{-}\ [:<=T]\ \text{-}\ [51,51,51,51]\ 50)$

syntax (*xsymbols*)
  $confTs :: {}'c\ prog \Rightarrow heap \Rightarrow val\ list \Rightarrow ty_l \Rightarrow bool$
    $(\text{-,-} \vdash \text{-}\ [:\leq_\top]\ \text{-}\ [51,51,51,51]\ 50)$

translations
  $P,h \vdash vs\ [:\leq_\top]\ Ts == list\text{-}all2\ (confT\ P\ h)\ vs\ Ts$

constdefs
  $conf\text{-}f :: jvm\text{-}prog \Rightarrow heap \Rightarrow ty_i \Rightarrow bytecode \Rightarrow frame \Rightarrow bool$
  $conf\text{-}f\ P\ h \equiv \lambda(ST,LT)\ is\ (stk,loc,C,M,pc).$
  $P,h \vdash stk\ [:\leq]\ ST \wedge P,h \vdash loc\ [:\leq_\top]\ LT \wedge pc < size\ is$

lemma *conf-f-def2*:
  $conf\text{-}f\ P\ h\ (ST,LT)\ is\ (stk,loc,C,M,pc) \equiv$
  $P,h \vdash stk\ [:\leq]\ ST \wedge P,h \vdash loc\ [:\leq_\top]\ LT \wedge pc < size\ is$
  by (*simp add*: *conf-f-def*)

consts
 $conf\text{-}fs :: [jvm\text{-}prog,heap,ty_P,mname,nat,ty,frame\ list] \Rightarrow bool$
primrec
$conf\text{-}fs\ P\ h\ \Phi\ M_0\ n_0\ T_0\ [] = True$

$conf\text{-}fs\ P\ h\ \Phi\ M_0\ n_0\ T_0\ (f\#frs) =$
  $(let\ (stk,loc,C,M,pc) = f\ in$
  $(\exists ST\ LT\ Ts\ T\ mxs\ mxl_0\ is\ xt.$
    $\Phi\ C\ M\ !\ pc = Some\ (ST,LT) \wedge$
    $(P \vdash C\ sees\ M{:}Ts \rightarrow T = (mxs,mxl_0,is,xt)\ in\ C) \wedge$
    $(\exists D\ Ts'\ T'\ m\ D'.$
      $is!pc = (Invoke\ M_0\ n_0) \wedge ST!n_0 = Class\ D \wedge$
      $P \vdash D\ sees\ M_0{:}Ts' \rightarrow T' = m\ in\ D' \wedge P \vdash T_0 \leq T') \wedge$
    $conf\text{-}f\ P\ h\ (ST,\ LT)\ is\ f \wedge conf\text{-}fs\ P\ h\ \Phi\ M\ (size\ Ts)\ T\ frs))$

constdefs
 $correct\text{-}state :: [jvm\text{-}prog,ty_P,jvm\text{-}state] \Rightarrow bool$

$(\text{-},\text{-} \mid\!- \text{ - } [ok]\ \ [61,0,0]\ 61)$

correct-state $P\ \Phi \equiv \lambda(xp,h,frs).$
$\quad$ *case xp of*
$\quad\quad$ *None* $\Rightarrow$ (*case frs of*
$\quad\quad\quad$ $[] \Rightarrow True$
$\quad\quad\quad$ $\mid (f\#fs) \Rightarrow P\vdash h\surd \wedge$
$\quad\quad\quad$ $(let\ (stk,loc,C,M,pc) = f$
$\quad\quad\quad$ $in\ \exists\, Ts\ T\ mxs\ mxl_0\ is\ xt\ \tau.$
$\quad\quad\quad\quad$ $(P \vdash C\ sees\ M{:}Ts{\rightarrow}T = (mxs,mxl_0,is,xt)\ in\ C) \wedge$
$\quad\quad\quad\quad$ $\Phi\ C\ M\ !\ pc = Some\ \tau \wedge$
$\quad\quad\quad\quad$ $conf\text{-}f\ P\ h\ \tau\ is\ f \wedge conf\text{-}fs\ P\ h\ \Phi\ M\ (size\ Ts)\ T\ fs))$
$\quad$ $\mid Some\ x \Rightarrow frs = []$

syntax (*xsymbols*)
correct-state :: $[jvm\text{-}prog,ty_P,jvm\text{-}state] \Rightarrow bool$
$(\text{-},\text{-} \vdash \text{ - } \surd\ \ [61,0,0]\ 61)$

4.21.1 Values and $\top$

lemma *confT-Err* [*iff*]: $P,h \vdash x :\leq_\top Err$
by (*simp add*: *confT-def*)

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

lemma *confT-cases*:
$P,h \vdash x :\leq_\top X = (X = Err \vee (\exists\, T.\ X = OK\ T \wedge P,h \vdash x :\leq T))$
by (*cases X*) *auto*

lemma *confT-hext* [*intro?*, *trans*]:
$[\![\ P,h \vdash x :\leq_\top T;\ h \trianglelefteq h'\ ]\!] \Longrightarrow P,h' \vdash x :\leq_\top T$
by (*cases T*) (*blast intro*: *conf-hext*)+

lemma *confT-widen* [*intro?*, *trans*]:
$[\![\ P,h \vdash x :\leq_\top T;\ P \vdash T \leq_\top T'\ ]\!] \Longrightarrow P,h \vdash x :\leq_\top T'$
by (*cases T'*, *auto intro*: *conf-widen*)

4.21.2 Stack and Registers

lemmas *confTs-Cons1* [*iff*] = *list-all2-Cons1* [*of confT P h*, *standard*]

lemma *confTs-confT-sup*:
$[\![\ P,h \vdash loc\ [:\leq_\top]\ LT;\ n < size\ LT;\ LT!n = OK\ T;\ P \vdash T \leq T'\ ]\!]$
$\Longrightarrow P,h \vdash (loc!n) :\leq T'$
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 add*: *list-all2-Cons2*)

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

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

4.21.3 correct-frames

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

lemma *conf-fs-hext*:
$\bigwedge M\ n\ T_r.$
$[\![\ \mathit{conf\text{-}fs}\ P\ h\ \Phi\ M\ n\ T_r\ \mathit{frs};\ h \trianglelefteq h'\ ]\!] \Longrightarrow \mathit{conf\text{-}fs}\ P\ h'\ \Phi\ M\ n\ T_r\ \mathit{frs}$
end

4.22 BV Type Safety Proof

theory *BVSpecTypeSafe* = *BVConform*:

This theory contains proof that the specification of the bytecode verifier only admits type safe programs.

4.22.1 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*

4.22.2 Exception Handling

For the *Invoke* instruction the BV has checked all handlers that guard the current *pc*.

lemma *Invoke-handlers*:
  $match\text{-}ex\text{-}table\ P\ C\ pc\ xt = Some\ (pc',d') \Longrightarrow$
  $\exists (f,t,D,h,d) \in set\ (relevant\text{-}entries\ P\ (Invoke\ n\ M)\ pc\ xt).$
   $P \vdash C \preceq^* D \wedge pc \in \{f..t(\} \wedge pc' = h \wedge d' = d$
  by (*induct xt*) (*auto simp add*: *relevant-entries-def matches-ex-entry-def*
        is-relevant-entry-def split: *split-if-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 occured conforms.

 term *find-handler*
lemma *uncaught-xcpt-correct*:
  assumes *wt*: $wf\text{-}jvm\text{-}prog_\Phi\ P$
  assumes *h*: $h\ xcp = Some\ obj$
  shows $\bigwedge f.\ P,\Phi \vdash (None,\ h,\ f\#frs)\surd \Longrightarrow P,\Phi \vdash (find\text{-}handler\ P\ xcp\ h\ frs)\ \surd$
  (**is** $\bigwedge f.\ ?correct\ (None,\ h,\ f\#frs) \Longrightarrow ?correct\ (?find\ frs)$)

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*:
  $[\![\ fst\ (exec\text{-}instr\ (ins!pc)\ P\ h\ stk\ vars\ Cl\ M\ pc\ frs) = Some\ xcp;$
    $P,T,mxs,size\ ins,xt \vdash ins!pc,pc :: \Phi\ C\ M;$
    $P,\Phi \vdash (None,\ h,\ (stk,loc,C,M,pc)\#frs)\surd\ ]\!]$
  $\Longrightarrow \exists\, obj.\ h\ xcp = Some\ obj$
  (**is** $[\![\ ?xcpt;\ ?wt;\ ?correct\ ]\!] \Longrightarrow ?thesis$)
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*: *split-if-asm*)

Finally we can state that, whenever an exception occurs, the next state always conforms:

lemma *xcpt-correct*:
  assumes *wtp*: $\textit{wf-jvm-prog}_\Phi$ *P*
  assumes *meth*: $P \vdash C$ *sees* $M{:}Ts{\rightarrow}T{=}(mxs,mxl_0,ins,xt)$ *in C*
  assumes *wt*: $P,T,mxs,size\ ins,xt \vdash ins!pc,pc :: \Phi\ C\ M$
  assumes *xp*: *fst* (*exec-instr* (*ins*!*pc*) *P h stk loc C M pc frs*) = *Some xcp*
  assumes *s′*: *Some* σ' = *exec* (*P*, *None*, *h*, (*stk*,*loc*,*C*,*M*,*pc*)#*frs*)
  assumes *correct*: $P,\Phi \vdash$ (*None*, *h*, (*stk*,*loc*,*C*,*M*,*pc*)#*frs*)$\surd$
  shows $P,\Phi \vdash \sigma'\surd$

4.22.3 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.

declare *defs1* [*simp*]

lemma *Invoke-correct*:
  assumes *wtprog*: $\textit{wf-jvm-prog}_\Phi$ *P*
  assumes *meth-C*: $P \vdash C$ *sees* $M{:}Ts{\rightarrow}T{=}(mxs,mxl_0,ins,xt)$ *in C*
  assumes *ins*: *ins* ! *pc* = *Invoke* M' *n*
  assumes *wti*: $P,T,mxs,size\ ins,xt \vdash ins!pc,pc :: \Phi\ C\ M$
  assumes σ': *Some* σ' = *exec* (*P*, *None*, *h*, (*stk*,*loc*,*C*,*M*,*pc*)#*frs*)
  assumes *approx*: $P,\Phi \vdash$ (*None*, *h*, (*stk*,*loc*,*C*,*M*,*pc*)#*frs*)$\surd$
  assumes *no-xcp*: *fst* (*exec-instr* (*ins*!*pc*) *P h stk loc C M pc frs*) = *None*
  shows $P,\Phi \vdash \sigma'\surd$

declare *list-all2-Cons2* [*iff*]

lemma *Return-correct*:
  assumes *wt-prog*: $\textit{wf-jvm-prog}_\Phi$ *P*
  assumes *meth*: $P \vdash C$ *sees* $M{:}Ts{\rightarrow}T{=}(mxs,mxl_0,ins,xt)$ *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*)#*frs*)
  assumes *correct*: $P,\Phi \vdash$ (*None*, *h*, (*stk*,*loc*,*C*,*M*,*pc*)#*frs*)$\surd$

  shows $P,\Phi \vdash \sigma'\surd$
declare *sup-state-opt-any-Some* [*iff*]
declare *not-Err-eq* [*iff*]

lemma *Load-correct*:
$[\![$ *wf-prog wt P*;
    $P \vdash C$ *sees* $M{:}Ts{\rightarrow}T{=}(mxs,mxl_0,ins,xt)$ *in C*;
    ins!*pc* = *Load idx*;
    $P,T,mxs,size\ ins,xt \vdash ins!pc,pc :: \Phi\ C\ M$;
    Some σ' = *exec* (*P*, *None*, *h*, (*stk*,*loc*,*C*,*M*,*pc*)#*frs*);
    $P,\Phi \vdash$ (*None*, *h*, (*stk*,*loc*,*C*,*M*,*pc*)#*frs*)$\surd$ $]\!]$
$\Longrightarrow P,\Phi \vdash \sigma'\surd$
  by (*fastsimp dest*: *sees-method-fun* [*of* - *C*] *elim*!: *confTs-confT-sup*)

lemma *Store-correct*:
$[\![$ *wf-prog wt P*;
  $P \vdash C$ *sees* $M{:}Ts{\rightarrow}T{=}(mxs,mxl_0,ins,xt)$ *in C*;

$ins!pc = Store\ idx$;
$P,T,mxs,size\ ins,xt \vdash ins!pc,pc :: \Phi\ C\ M$;
$Some\ \sigma' = exec\ (P,\ None,\ h,\ (stk,loc,C,M,pc)\#frs)$;
$P,\Phi \vdash (None,\ h,\ (stk,loc,C,M,pc)\#frs)\surd$ $]\!]$
$\Longrightarrow P,\Phi \vdash \sigma'\surd$

lemma *Push-correct*:
$[\![$ $wf\text{-}prog\ wt\ P$;
$P \vdash C\ sees\ M{:}Ts{\rightarrow}T{=}(mxs,mxl_0,ins,xt)\ in\ C$;
$ins!pc = Push\ v$;
$P,T,mxs,size\ ins,xt \vdash ins!pc,pc :: \Phi\ C\ M$;
$Some\ \sigma' = exec\ (P,\ None,\ h,\ (stk,loc,C,M,pc)\#frs)$;
$P,\Phi \vdash (None,\ h,\ (stk,loc,C,M,pc)\#frs)\surd$ $]\!]$
$\Longrightarrow P,\Phi \vdash \sigma'\surd$

lemma *Cast-conf2*:
$[\![$ $wf\text{-}prog\ ok\ P$; $P,h \vdash v :\leq T$; $is\text{-}refT\ T$; $cast\text{-}ok\ P\ C\ h\ v$;
$P \vdash Class\ C \leq T'$; $is\text{-}class\ P\ C$$]\!]$
$\Longrightarrow P,h \vdash v :\leq T'$

lemma *Checkcast-correct*:
$[\![$ $wf\text{-}jvm\text{-}prog_\Phi\ P$;
$P \vdash C\ sees\ M{:}Ts{\rightarrow}T{=}(mxs,mxl_0,ins,xt)\ in\ C$;
$ins!pc = Checkcast\ D$;
$P,T,mxs,size\ ins,xt \vdash ins!pc,pc :: \Phi\ C\ M$;
$Some\ \sigma' = exec\ (P,\ None,\ h,\ (stk,loc,C,M,pc)\#frs)$;
$P,\Phi \vdash (None,\ h,\ (stk,loc,C,M,pc)\#frs)\surd$;
$fst\ (exec\text{-}instr\ (ins!pc)\ P\ h\ stk\ loc\ C\ M\ pc\ frs) = None$ $]\!]$
$\Longrightarrow P,\Phi \vdash \sigma'\surd$
declare *split-paired-All* [*simp del*]

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

lemma *Getfield-correct*:
assumes *wf*: *wf-prog wt P*
assumes *mC*: $P \vdash C\ sees\ M{:}Ts{\rightarrow}T{=}(mxs,mxl_0,ins,xt)\ in\ C$
assumes *i*: $ins!pc = Getfield\ F\ D$
assumes *wt*: $P,T,mxs,size\ ins,xt \vdash ins!pc,pc :: \Phi\ C\ M$
assumes *s'*: $Some\ \sigma' = exec\ (P,\ None,\ h,\ (stk,loc,C,M,pc)\#frs)$
assumes *cf*: $P,\Phi \vdash (None,\ h,\ (stk,loc,C,M,pc)\#frs)\surd$
assumes *xc*: $fst\ (exec\text{-}instr\ (ins!pc)\ P\ h\ stk\ loc\ C\ M\ pc\ frs) = None$

shows $P,\Phi \vdash \sigma'\surd$
lemma *Putfield-correct*:
assumes *wf*: *wf-prog wt P*
assumes *mC*: $P \vdash C\ sees\ M{:}Ts{\rightarrow}T{=}(mxs,mxl_0,ins,xt)\ in\ C$
assumes *i*: $ins!pc = Putfield\ F\ D$
assumes *wt*: $P,T,mxs,size\ ins,xt \vdash ins!pc,pc :: \Phi\ C\ M$
assumes *s'*: $Some\ \sigma' = exec\ (P,\ None,\ h,\ (stk,loc,C,M,pc)\#frs)$
assumes *cf*: $P,\Phi \vdash (None,\ h,\ (stk,loc,C,M,pc)\#frs)\surd$
assumes *xc*: $fst\ (exec\text{-}instr\ (ins!pc)\ P\ h\ stk\ loc\ C\ M\ pc\ frs) = None$

shows $P,\Phi \vdash \sigma'\surd$

lemma *has-fields-b-fields*:
$P \vdash C$ *has-fields FDTs* $\Longrightarrow$ *fields P C* = *FDTs*

lemma *oconf-blank* [*intro*, *simp*]:
$[\![$*is-class P C*; *wf-prog wt P*$]\!] \Longrightarrow P,h \vdash$ *blank P C* $\surd$

lemma *obj-ty-blank* [*iff*]: *obj-ty* (*blank P C*) = *Class C*
by (*simp add*: *blank-def*)

lemma *New-correct*:
assumes *wf*: *wf-prog wt P*
assumes *meth*: $P \vdash C$ *sees* $M{:}Ts{\rightarrow}T{=}(mxs,mxl_0,ins,xt)$ *in C*
assumes *ins*: *ins*!*pc* = *New X*
assumes *wt*: $P,T,mxs,size\ ins,xt \vdash ins!pc,pc :: \Phi\ C\ M$
assumes *exec*: *Some* σ' = *exec* (*P*, *None*, *h*, (*stk*,*loc*,*C*,*M*,*pc*)#*frs*)
assumes *conf*: $P,\Phi \vdash$ (*None*, *h*, (*stk*,*loc*,*C*,*M*,*pc*)#*frs*)$\surd$
assumes *no-x*: *fst* (*exec-instr* (*ins*!*pc*) *P h stk loc C M pc frs*) = *None*
shows $P,\Phi \vdash \sigma'\surd$

lemma *Goto-correct*:
$[\![$ *wf-prog wt P*;
$P \vdash C$ *sees* $M{:}Ts{\rightarrow}T{=}(mxs,mxl_0,ins,xt)$ *in C*;
ins ! *pc* = *Goto branch*;
$P,T,mxs,size\ ins,xt \vdash ins!pc,pc :: \Phi\ C\ M$;
Some σ' = *exec* (*P*, *None*, *h*, (*stk*,*loc*,*C*,*M*,*pc*)#*frs*) ;
$P,\Phi \vdash$ (*None*, *h*, (*stk*,*loc*,*C*,*M*,*pc*)#*frs*)$\surd$ $]\!]$
$\Longrightarrow P,\Phi \vdash \sigma'\surd$

lemma *IfFalse-correct*:
$[\![$ *wf-prog wt P*;
$P \vdash C$ *sees* $M{:}Ts{\rightarrow}T{=}(mxs,mxl_0,ins,xt)$ *in C*;
ins ! *pc* = *IfFalse branch*;
$P,T,mxs,size\ ins,xt \vdash ins!pc,pc :: \Phi\ C\ M$;
Some σ' = *exec* (*P*, *None*, *h*, (*stk*,*loc*,*C*,*M*,*pc*)#*frs*) ;
$P,\Phi \vdash$ (*None*, *h*, (*stk*,*loc*,*C*,*M*,*pc*)#*frs*)$\surd$ $]\!]$
$\Longrightarrow P,\Phi \vdash \sigma'\surd$

lemma *CmpEq-correct*:
$[\![$ *wf-prog wt P*;
$P \vdash C$ *sees* $M{:}Ts{\rightarrow}T{=}(mxs,mxl_0,ins,xt)$ *in C*;
ins ! *pc* = *CmpEq*;
$P,T,mxs,size\ ins,xt \vdash ins!pc,pc :: \Phi\ C\ M$;
Some σ' = *exec* (*P*, *None*, *h*, (*stk*,*loc*,*C*,*M*,*pc*)#*frs*) ;
$P,\Phi \vdash$ (*None*, *h*, (*stk*,*loc*,*C*,*M*,*pc*)#*frs*)$\surd$ $]\!]$
$\Longrightarrow P,\Phi \vdash \sigma'\surd$

lemma *Pop-correct*:
$[\![$ *wf-prog wt P*;
$P \vdash C$ *sees* $M{:}Ts{\rightarrow}T{=}(mxs,mxl_0,ins,xt)$ *in C*;
ins ! *pc* = *Pop*;
$P,T,mxs,size\ ins,xt \vdash ins!pc,pc :: \Phi\ C\ M$;
Some σ' = *exec* (*P*, *None*, *h*, (*stk*,*loc*,*C*,*M*,*pc*)#*frs*) ;
$P,\Phi \vdash$ (*None*, *h*, (*stk*,*loc*,*C*,*M*,*pc*)#*frs*)$\surd$ $]\!]$
$\Longrightarrow P,\Phi \vdash \sigma'\surd$

lemma *IAdd-correct*:

$[\![\ \mathit{wf\text{-}prog\ wt}\ P;$
$\quad P \vdash C\ \mathit{sees}\ M{:}\mathit{Ts}{\rightarrow}T{=}(\mathit{mxs},\mathit{mxl}_0,\mathit{ins},\mathit{xt})\ \mathit{in}\ C;$
$\quad \mathit{ins}\ !\ \mathit{pc} = \mathit{IAdd};$
$\quad P,T,\mathit{mxs},\mathit{size\ ins},\mathit{xt} \vdash \mathit{ins}!\mathit{pc},\mathit{pc} :: \Phi\ C\ M;$
$\quad \mathit{Some}\ \sigma' = \mathit{exec}\ (P,\ \mathit{None},\ h,\ (\mathit{stk},\mathit{loc},C,M,\mathit{pc})\#\mathit{frs})\ ;$
$\quad P,\Phi \vdash (\mathit{None},\ h,\ (\mathit{stk},\mathit{loc},C,M,\mathit{pc})\#\mathit{frs})\surd\ ]\!]$
$\Longrightarrow P,\Phi \vdash \sigma'\surd$

lemma *Throw-correct*:
$[\![\ \mathit{wf\text{-}prog\ wt}\ P;$
$\quad P \vdash C\ \mathit{sees}\ M{:}\mathit{Ts}{\rightarrow}T{=}(\mathit{mxs},\mathit{mxl}_0,\mathit{ins},\mathit{xt})\ \mathit{in}\ C;$
$\quad \mathit{ins}\ !\ \mathit{pc} = \mathit{Throw};$
$\quad \mathit{Some}\ \sigma' = \mathit{exec}\ (P,\ \mathit{None},\ h,\ (\mathit{stk},\mathit{loc},C,M,\mathit{pc})\#\mathit{frs})\ ;$
$\quad P,\Phi \vdash (\mathit{None},\ h,\ (\mathit{stk},\mathit{loc},C,M,\mathit{pc})\#\mathit{frs})\surd;$
$\quad \mathit{fst}\ (\mathit{exec\text{-}instr}\ (\mathit{ins}!\mathit{pc})\ P\ h\ \mathit{stk}\ \mathit{loc}\ C\ M\ \mathit{pc}\ \mathit{frs}) = \mathit{None}\ ]\!]$
$\Longrightarrow P,\Phi \vdash \sigma'\surd$
  by *simp*

The next theorem collects the results of the sections above, i.e. exception handling and the execution step for each instruction. It states type safety for single step execution: in welltyped programs, a conforming state is transformed into another conforming state when one instruction is executed.

theorem *instr-correct*:
$[\![\ \mathit{wf\text{-}jvm\text{-}prog}_\Phi\ P;$
$\quad P \vdash C\ \mathit{sees}\ M{:}\mathit{Ts}{\rightarrow}T{=}(\mathit{mxs},\mathit{mxl}_0,\mathit{ins},\mathit{xt})\ \mathit{in}\ C;$
$\quad \mathit{Some}\ \sigma' = \mathit{exec}\ (P,\ \mathit{None},\ h,\ (\mathit{stk},\mathit{loc},C,M,\mathit{pc})\#\mathit{frs});$
$\quad P,\Phi \vdash (\mathit{None},\ h,\ (\mathit{stk},\mathit{loc},C,M,\mathit{pc})\#\mathit{frs})\surd\ ]\!]$
$\Longrightarrow P,\Phi \vdash \sigma'\surd$

4.22.4 Main

lemma *correct-state-impl-Some-method*:
$\quad P,\Phi \vdash (\mathit{None},\ h,\ (\mathit{stk},\mathit{loc},C,M,\mathit{pc})\#\mathit{frs})\surd$
$\quad \Longrightarrow \exists\, m\ \mathit{Ts}\ T.\ P \vdash C\ \mathit{sees}\ M{:}\mathit{Ts}{\rightarrow}T = m\ \mathit{in}\ C$
  by *fastsimp*

lemma *BV-correct-1* [*rule-format*]:
$\bigwedge\sigma.\ [\![\ \mathit{wf\text{-}jvm\text{-}prog}_\Phi\ P;\ P,\Phi \vdash \sigma\surd]\!] \Longrightarrow P \vdash \sigma\ {-}\mathit{jvm}{\rightarrow}_1\ \sigma' \longrightarrow P,\Phi \vdash \sigma'\surd$

theorem *progress*:
$\quad [\![\ \mathit{xp}{=}\mathit{None};\ \mathit{frs}{\neq}[]\ ]\!] \Longrightarrow \exists\,\sigma'.\ P \vdash (\mathit{xp},h,\mathit{frs})\ {-}\mathit{jvm}{\rightarrow}_1\ \sigma'$
  by (*clarsimp simp add*: *exec-1-iff neq-Nil-conv split-beta*
            simp del: *split-paired-Ex*)

lemma *progress-conform*:
$\quad [\![\mathit{wf\text{-}jvm\text{-}prog}_\Phi\ P;\ P,\Phi \vdash (\mathit{xp},h,\mathit{frs})\surd;\ \mathit{xp}{=}\mathit{None};\ \mathit{frs}{\neq}[]]\!]$
$\quad \Longrightarrow \exists\,\sigma'.\ P \vdash (\mathit{xp},h,\mathit{frs})\ {-}\mathit{jvm}{\rightarrow}_1\ \sigma' \wedge P,\Phi \vdash \sigma'\surd$
theorem *BV-correct* [*rule-format*]:
$[\![\ \mathit{wf\text{-}jvm\text{-}prog}_\Phi\ P;\ P \vdash \sigma\ {-}\mathit{jvm}{\rightarrow}\ \sigma'\ ]\!] \Longrightarrow P,\Phi \vdash \sigma\surd \longrightarrow P,\Phi \vdash \sigma'\surd$
lemma *hconf-start*:
  assumes *wf*: *wf-prog wf-mb P*
  shows $P \vdash (\mathit{start\text{-}heap}\ P)\ \surd$
lemma *BV-correct-initial*:
  shows $[\![\ \mathit{wf\text{-}jvm\text{-}prog}_\Phi\ P;\ P \vdash C\ \mathit{sees}\ M{:}[]{\rightarrow}T = m\ \mathit{in}\ C\ ]\!]$

$\Longrightarrow P,\Phi \vdash$ *start-state* $P\ C\ M\ \surd$
theorem *typesafe*:
assumes *welltyped*: *wf-jvm-prog*$_\Phi$ P
assumes *main-method*: $P \vdash C$ *sees* M:[]$\rightarrow T = m$ *in* C
shows $P \vdash$ *start-state* $P\ C\ M$ $-$*jvm*$\rightarrow \sigma \Longrightarrow P,\Phi \vdash \sigma\ \surd$
end

4.23 Welltyped Programs produce no Type Errors

theory *BVNoTypeError* = *JVMDefensive* + *BVSpecTypeSafe*:

lemma *has-methodI*:
  $P \vdash C$ *sees* $M{:}Ts{\to}T = m$ *in* $D \Longrightarrow P \vdash C$ *has* M
  by (*unfold has--def*) *blast*

Some simple lemmas about the type testing functions of the defensive JVM:

lemma *typeof-NoneD* [*simp*,*dest*]: *typeof* v = *Some* $x \Longrightarrow \neg$*is-Addr* v
  by (*cases* v) *auto*

lemma *is-Ref-def2*:
  is-Ref v = (v = *Null* $\vee$ ($\exists\, a.\ v$ = *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*]: $P,hp \vdash v :\leq T \Longrightarrow$ *is-refT* $T \Longrightarrow$ *is-Ref* v
lemma *is-IntgI* [*intro*, *simp*]: $P,hp \vdash v :\leq$ *Integer* $\Longrightarrow$ *is-Intg* v
lemma *is-BoolI* [*intro*, *simp*]: $P,hp \vdash v :\leq$ *Boolean* $\Longrightarrow$ *is-Bool* v
declare *defs1* [*simp del*]

lemma *wt-jvm-prog-states*:
  $[\![$ *wf-jvm-prog*$_\Phi$ P; $P \vdash C$ *sees* M: $Ts{\to}T$ = (*mxs*, *mxl*, *ins*, *et*) *in* C;
    Φ C M ! $pc = \tau$; $pc <$ *size ins* $]\!]$
  $\Longrightarrow$ *OK* $\tau \in$ *states* P *mxs* (*1+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*:
  assumes *welltyped*: *wf-jvm-prog*$_\Phi$ P **and** *conforms*: $P,\Phi \vdash \sigma\ \surd$
  shows *exec-d* P $\sigma \neq$ *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$_\Phi$ $P \Longrightarrow P,\Phi \vdash \sigma\ \surd \Longrightarrow P \vdash \sigma$ *−jvm→* σ'
  $\Longrightarrow P \vdash$ (*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*:
  assumes *wf-jvm-prog*$_\Phi$ P **and** $P,\Phi \vdash \sigma\ \surd$
  shows $P \vdash$ (*Normal* σ) *−jvmd→* (*Normal* σ') = $P \vdash \sigma$ *−jvm→* σ'
  by *rule* (*erule defensive-imp-aggressive*,*rule welltyped-aggressive-imp-defensive*)

corollary *welltyped-initial-commutes*:
  assumes *wf*: *wf-jvm-prog* P
  assumes $P \vdash C$ *sees* M:[]$\to T = b$ *in* C
  defines *start*: $\sigma \equiv$ *start-state* P C M
  shows $P \vdash$ (*Normal* σ) *−jvmd→* (*Normal* σ') = $P \vdash \sigma$ *−jvm→* σ'

proof −
  from *wf* **obtain** Φ **where** *wf-jvm-prog*$_\Phi$ *P* **by** (*auto simp*: *wf-jvm-prog-def*)
  have $P,\Phi \vdash \sigma \surd$ **by** (*unfold start*, *rule BV-correct-initial*)
  thus *?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*
  apply (*unfold exec-all-d-def1*)
  apply (*erule rtrancl-induct*)
   apply *simp*
  apply (*fold exec-all-d-def1*)
  apply (*insert cnf wf*)
  apply *clarsimp*
  apply (*drule defensive-imp-aggressive*)
  apply (*frule* (*2*) *BV-correct*)
  apply (*drule* (*1*) *no-type-error*) **back**
  apply (*auto simp add*: *exec-1-d-def*)
  done

locale *start* =
  fixes *P* **and** *C* **and** *M* **and** σ **and** *T* **and** *b*
  assumes *wf*: *wf-jvm-prog P*
  assumes *sees*: $P \vdash C$ *sees* M:[]→T = *b in C*
  defines $\sigma \equiv$ *Normal* (*start-state P C M*)

corollary (**in** *start*) *bv-no-type-error*:
  shows $P \vdash \sigma$ −*jvmd*→ $\sigma' \Longrightarrow \sigma' \neq$ *TypeError*
proof −
  from *wf* **obtain** Φ **where** *wf-jvm-prog*$_\Phi$ *P* **by** (*auto simp*: *wf-jvm-prog-def*)
  moreover
  with *sees* **have** *correct-state P* Φ (*start-state P C M*)
    by − (*rule BV-correct-initial*)
  ultimately have *cnf P* Φ (*start-state P C M*) **by** (*rule cnf.intro*)
  moreover assume $P \vdash \sigma$ −*jvmd*→ σ'
  ultimately show *?thesis* **by** (*unfold* σ*-def*) (*rule cnf.no-type-errors*)
qed

end

Chapter 5

Compilation

5.1 An Intermediate Language

theory $J1 = BigStep$:

types
$expr_1 = nat\ exp$
$J_1\text{-}prog = expr_1\ prog$

$state_1 = heap \times (val\ list)$

consts
$max\text{-}vars :: {'a}\ exp \Rightarrow nat$
$max\text{-}varss :: {'a}\ exp\ list \Rightarrow nat$
primrec
$max\text{-}vars(new\ C) = 0$
$max\text{-}vars(Cast\ C\ e) = max\text{-}vars\ e$
$max\text{-}vars(Val\ v) = 0$
$max\text{-}vars(e_1\ «bop»\ e_2) = max\ (max\text{-}vars\ e_1)\ (max\text{-}vars\ e_2)$
$max\text{-}vars(Var\ V) = 0$
$max\text{-}vars(V{:=}e) = max\text{-}vars\ e$
$max\text{-}vars(e{\bullet}F\{D\}) = max\text{-}vars\ e$
$max\text{-}vars(FAss\ e_1\ F\ D\ e_2) = max\ (max\text{-}vars\ e_1)\ (max\text{-}vars\ e_2)$
$max\text{-}vars(e{\bullet}M(es)) = max\ (max\text{-}vars\ e)\ (max\text{-}varss\ es)$
$max\text{-}vars(\{V{:}T;\ e\}) = max\text{-}vars\ e + 1$
$max\text{-}vars(e_1;;e_2) = max\ (max\text{-}vars\ e_1)\ (max\text{-}vars\ e_2)$
$max\text{-}vars(if\ (e)\ e_1\ else\ e_2) =$
$\quad max\ (max\text{-}vars\ e)\ (max\ (max\text{-}vars\ e_1)\ (max\text{-}vars\ e_2))$
$max\text{-}vars(while\ (b)\ e) = max\ (max\text{-}vars\ b)\ (max\text{-}vars\ e)$
$max\text{-}vars(throw\ e) = max\text{-}vars\ e$
$max\text{-}vars(try\ e_1\ catch(C\ V)\ e_2) = max\ (max\text{-}vars\ e_1)\ (max\text{-}vars\ e_2 + 1)$

$max\text{-}varss\ [] = 0$
$max\text{-}varss\ (e\#es) = max\ (max\text{-}vars\ e)\ (max\text{-}varss\ es)$

consts
$eval_1 :: J_1\text{-}prog \Rightarrow ((expr_1 \times state_1) \times (expr_1 \times state_1))\ set$
$evals_1 :: J_1\text{-}prog \Rightarrow ((expr_1\ list \times state_1) \times (expr_1\ list \times state_1))\ set$

translations
$P \vdash_1 \langle e,s\rangle \Rightarrow \langle e',s'\rangle == ((e,s),\ e',s') \in eval_1\ P$
$P \vdash_1 \langle e,s\rangle\ [\Rightarrow]\ \langle e',s'\rangle == ((e,s),\ e',s') \in evals_1\ P$

inductive $eval_1\ P\ evals_1\ P$
intros

New_1:
$[\![\ new\text{-}Addr\ h = Some\ a;\ P \vdash C\ has\text{-}fields\ FDTs;\ h' = h(a{\mapsto}(C,init\text{-}fields\ FDTs))\ ]\!]$
$\Longrightarrow P \vdash_1 \langle new\ C,(h,l)\rangle \Rightarrow \langle addr\ a,(h',l)\rangle$
$NewFail_1$:
$new\text{-}Addr\ h = None \Longrightarrow$
$P \vdash_1 \langle new\ C,\ (h,l)\rangle \Rightarrow \langle THROW\ OutOfMemory,(h,l)\rangle$

$Cast_1$:
$[\![\ P \vdash_1 \langle e,s_0\rangle \Rightarrow \langle addr\ a,(h,l)\rangle;\ h\ a = Some(D,fs);\ P \vdash D \preceq^* C\ ]\!]$

$\Longrightarrow P \vdash_1 \langle Cast\ C\ e,s_0\rangle \Rightarrow \langle addr\ a,(h,l)\rangle$

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$_1$:

$[\![\ P \vdash_1 \langle e,s_0\rangle \Rightarrow \langle addr\ a,(h,l)\rangle;\ h\ a = Some(D,fs);\ \neg\ P \vdash D \preceq^*\ C\ ]\!]$
$\Longrightarrow P \vdash_1 \langle Cast\ C\ e,s_0\rangle \Rightarrow \langle THROW\ ClassCast,(h,l)\rangle$

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$_1$:

$[\![\ P \vdash_1 \langle e_1,s_0\rangle \Rightarrow \langle Val\ v_1,s_1\rangle;\ P \vdash_1 \langle e_2,s_1\rangle \Rightarrow \langle Val\ v_2,s_2\rangle;\ binop(bop,v_1,v_2) = Some\ v\ ]\!]$
$\Longrightarrow P \vdash_1 \langle e_1\ \text{«}bop\text{»}\ e_2,s_0\rangle \Rightarrow \langle Val\ v,s_2\rangle$

BinOpThrow$_{11}$:

$P \vdash_1 \langle e_1,s_0\rangle \Rightarrow \langle throw\ e,s_1\rangle \Longrightarrow$
$P \vdash_1 \langle e_1\ \text{«}bop\text{»}\ e_2,\ s_0\rangle \Rightarrow \langle throw\ e,s_1\rangle$

BinOpThrow$_{21}$:

$[\![\ P \vdash_1 \langle e_1,s_0\rangle \Rightarrow \langle Val\ v_1,s_1\rangle;\ P \vdash_1 \langle e_2,s_1\rangle \Rightarrow \langle throw\ e,s_2\rangle\ ]\!]$
$\Longrightarrow P \vdash_1 \langle e_1\ \text{«}bop\text{»}\ e_2,s_0\rangle \Rightarrow \langle throw\ e,s_2\rangle$

Var$_1$:

$[\![\ ls!i = v;\ i < size\ ls\ ]\!] \Longrightarrow$
$P \vdash_1 \langle Var\ i,(h,ls)\rangle \Rightarrow \langle Val\ v,(h,ls)\rangle$

LAss$_1$:

$[\![\ P \vdash_1 \langle e,s_0\rangle \Rightarrow \langle Val\ v,(h,ls)\rangle;\ i < size\ ls;\ ls' = ls[i := v]\ ]\!]$
$\Longrightarrow P \vdash_1 \langle i{:=}\ e,s_0\rangle \Rightarrow \langle unit,(h,ls')\rangle$

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$_1$:

$[\![\ P \vdash_1 \langle e,s_0\rangle \Rightarrow \langle addr\ a,(h,ls)\rangle;\ h\ a = Some(C,fs);\ fs(F,D) = Some\ v\ ]\!]$
$\Longrightarrow P \vdash_1 \langle e{\cdot}F\{D\},s_0\rangle \Rightarrow \langle Val\ v,(h,ls)\rangle$

FAccNull$_1$:

$P \vdash_1 \langle e,s_0\rangle \Rightarrow \langle null,s_1\rangle \Longrightarrow$
$P \vdash_1 \langle e{\cdot}F\{D\},s_0\rangle \Rightarrow \langle THROW\ NullPointer,s_1\rangle$

FAccThrow$_1$:

$P \vdash_1 \langle e,s_0\rangle \Rightarrow \langle throw\ e',s_1\rangle \Longrightarrow$
$P \vdash_1 \langle e{\cdot}F\{D\},s_0\rangle \Rightarrow \langle throw\ e',s_1\rangle$

FAss$_1$:

$[\![\ P \vdash_1 \langle e_1,s_0\rangle \Rightarrow \langle addr\ a,s_1\rangle;\ P \vdash_1 \langle e_2,s_1\rangle \Rightarrow \langle Val\ v,(h_2,l_2)\rangle;$
$h_2\ a = Some(C,fs);\ fs' = fs((F,D)\mapsto v);\ h_2' = h_2(a\mapsto(C,fs'))\ ]\!]$
$\Longrightarrow P \vdash_1 \langle e_1{\cdot}F\{D\}{:=}\ e_2,s_0\rangle \Rightarrow \langle unit,(h_2',l_2)\rangle$

FAssNull$_1$:

$[\![\ P \vdash_1 \langle e_1,s_0\rangle \Rightarrow \langle null,s_1\rangle;\ \ P \vdash_1 \langle e_2,s_1\rangle \Rightarrow \langle Val\ v,s_2\rangle\ ]\!]$
$\Longrightarrow P \vdash_1 \langle e_1{\cdot}F\{D\}{:=}\ e_2,s_0\rangle \Rightarrow \langle THROW\ NullPointer,s_2\rangle$

FAssThrow$_{11}$:

$P \vdash_1 \langle e_1,s_0\rangle \Rightarrow \langle throw\ e',s_1\rangle \Longrightarrow$

$P \vdash_1 \langle e_1 \cdot F\{D\} := e_2, s_0\rangle \Rightarrow \langle throw\ e', s_1\rangle$
FAssThrow$_{21}$:
$[\![\ P \vdash_1 \langle e_1, s_0\rangle \Rightarrow \langle Val\ v, s_1\rangle;\ P \vdash_1 \langle e_2, s_1\rangle \Rightarrow \langle throw\ e', s_2\rangle\ ]\!]$
$\Longrightarrow P \vdash_1 \langle e_1 \cdot F\{D\} := e_2, s_0\rangle \Rightarrow \langle throw\ e', s_2\rangle$

CallObjThrow$_1$:
$P \vdash_1 \langle e, s_0\rangle \Rightarrow \langle throw\ e', s_1\rangle \Longrightarrow$
$P \vdash_1 \langle e \cdot M(es), s_0\rangle \Rightarrow \langle throw\ e', s_1\rangle$
CallNull$_1$:
$[\![\ P \vdash_1 \langle e, s_0\rangle \Rightarrow \langle null, s_1\rangle;\ P \vdash_1 \langle es, s_1\rangle\ [\Rightarrow]\ \langle map\ Val\ vs, s_2\rangle\ ]\!]$
$\Longrightarrow P \vdash_1 \langle e \cdot M(es), s_0\rangle \Rightarrow \langle THROW\ NullPointer, s_2\rangle$
Call$_1$:
$[\![\ P \vdash_1 \langle e, s_0\rangle \Rightarrow \langle addr\ a, s_1\rangle;\ P \vdash_1 \langle es, s_1\rangle\ [\Rightarrow]\ \langle map\ Val\ vs, (h_2, ls_2)\rangle;$
$h_2\ a = Some(C, fs);\ P \vdash C\ sees\ M{:}Ts \rightarrow T = body\ in\ D;$
$size\ vs = size\ Ts;\ ls_2' = (Addr\ a)\ \#\ vs\ @\ replicate\ (max\text{-}vars\ body)\ arbitrary;$
$P \vdash_1 \langle body, (h_2, ls_2')\rangle \Rightarrow \langle e', (h_3, ls_3)\rangle\ ]\!]$
$\Longrightarrow P \vdash_1 \langle e \cdot M(es), s_0\rangle \Rightarrow \langle e', (h_3, ls_2)\rangle$
CallParamsThrow$_1$:
$[\![\ P \vdash_1 \langle e, s_0\rangle \Rightarrow \langle Val\ v, s_1\rangle;\ P \vdash_1 \langle es, s_1\rangle\ [\Rightarrow]\ \langle es', s_2\rangle;$
$es' = map\ Val\ vs\ @\ throw\ ex\ \#\ es_2\ ]\!]$
$\Longrightarrow P \vdash_1 \langle e \cdot M(es), s_0\rangle \Rightarrow \langle throw\ ex, s_2\rangle$

Block$_1$:
$P \vdash_1 \langle e, s_0\rangle \Rightarrow \langle f, s_1\rangle \Longrightarrow P \vdash_1 \langle Block\ i\ T\ e, s_0\rangle \Rightarrow \langle f, s_1\rangle$

Seq$_1$:
$[\![\ P \vdash_1 \langle e_0, s_0\rangle \Rightarrow \langle Val\ v, s_1\rangle;\ P \vdash_1 \langle e_1, s_1\rangle \Rightarrow \langle e_2, s_2\rangle\ ]\!]$
$\Longrightarrow P \vdash_1 \langle e_0;; e_1, s_0\rangle \Rightarrow \langle e_2, s_2\rangle$
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$_1$:
$[\![\ P \vdash_1 \langle e, s_0\rangle \Rightarrow \langle true, s_1\rangle;\ P \vdash_1 \langle e_1, s_1\rangle \Rightarrow \langle e', s_2\rangle\ ]\!]$
$\Longrightarrow P \vdash_1 \langle if\ (e)\ e_1\ else\ e_2, s_0\rangle \Rightarrow \langle e', s_2\rangle$
CondF$_1$:
$[\![\ P \vdash_1 \langle e, s_0\rangle \Rightarrow \langle false, s_1\rangle;\ P \vdash_1 \langle e_2, s_1\rangle \Rightarrow \langle e', s_2\rangle\ ]\!]$
$\Longrightarrow P \vdash_1 \langle if\ (e)\ e_1\ else\ e_2, s_0\rangle \Rightarrow \langle e', s_2\rangle$
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$_1$:
$[\![\ P \vdash_1 \langle e, s_0\rangle \Rightarrow \langle true, s_1\rangle;\ P \vdash_1 \langle c, s_1\rangle \Rightarrow \langle Val\ v_1, s_2\rangle;$
$P \vdash_1 \langle while\ (e)\ c, s_2\rangle \Rightarrow \langle e_3, s_3\rangle\ ]\!]$
$\Longrightarrow P \vdash_1 \langle while\ (e)\ c, s_0\rangle \Rightarrow \langle e_3, s_3\rangle$
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$_1$:
$[\![\ P \vdash_1 \langle e, s_0\rangle \Rightarrow \langle true, s_1\rangle;\ P \vdash_1 \langle c, s_1\rangle \Rightarrow \langle throw\ e', s_2\rangle]\!]$

$\Longrightarrow P \vdash_1 \langle while\ (e)\ c,s_0\rangle \Rightarrow \langle throw\ e',s_2\rangle$

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$_1$:
$[\![\ P \vdash_1 \langle e_1,s_0\rangle \Rightarrow \langle Throw\ a,(h_1,ls_1)\rangle;$
$h_1\ a = Some(D,fs);\ P \vdash D \preceq^* C;\ i < length\ ls_1;$
$P \vdash_1 \langle e_2,(h_1,ls_1[i:=Addr\ a])\rangle \Rightarrow \langle e_2',(h_2,ls_2)\rangle\ ]\!]$
$\Longrightarrow P \vdash_1 \langle try\ e_1\ catch(C\ i)\ e_2,s_0\rangle \Rightarrow \langle e_2',(h_2,ls_2)\rangle$
TryThrow$_1$:
$[\![\ P \vdash_1 \langle e_1,s_0\rangle \Rightarrow \langle Throw\ a,(h_1,ls_1)\rangle;\ h_1\ a = Some(D,fs);\ \neg\ P \vdash D \preceq^* C\ ]\!]$
$\Longrightarrow P \vdash_1 \langle try\ e_1\ catch(C\ i)\ e_2,s_0\rangle \Rightarrow \langle Throw\ a,(h_1,ls_1)\rangle$

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$_1$:
$P \vdash_1 \langle e,s_0\rangle \Rightarrow \langle throw\ e',s_1\rangle \Longrightarrow$
$P \vdash_1 \langle e\#es,s_0\rangle\ [\Rightarrow]\ \langle throw\ e'\ \#\ es,\ s_1\rangle$

lemma *eval*$_1$*-preserves-len*:
$P \vdash_1 \langle e_0,(h_0,ls_0)\rangle \Rightarrow \langle e_1,(h_1,ls_1)\rangle \Longrightarrow length\ ls_0 = length\ ls_1$
and *evals*$_1$*-preserves-len*:
$P \vdash_1 \langle es_0,(h_0,ls_0)\rangle\ [\Rightarrow]\ \langle es_1,(h_1,ls_1)\rangle \Longrightarrow length\ ls_0 = length\ ls_1$

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 *eval*$_1$*-final*: $P \vdash_1 \langle e,s\rangle \Rightarrow \langle e',s'\rangle \Longrightarrow final\ e'$
and *evals*$_1$*-final*: $P \vdash_1 \langle es,s\rangle\ [\Rightarrow]\ \langle es',s'\rangle \Longrightarrow finals\ es'$

end

5.2 Well-Formedness of Intermediate Language

theory *J1WellForm* = *JWellForm* + *J1*:

5.2.1 Well-Typedness

types
$env_1 \;=\; ty\ list$ — type environment indexed by variable number

consts
$WT_1 :: J_1\text{-}prog \Rightarrow (env_1 \times expr_1 \times ty)\ set$
$WTs_1 :: J_1\text{-}prog \Rightarrow (env_1 \times expr_1\ list \times ty\ list)\ set$

translations
$P,E \vdash_1 e :: T \;==\; (E,e,T) \in WT_1\ P$
$P,E \vdash_1 es\ [::]\ Ts \;==\; (E,es,Ts) \in WTs_1\ P$

inductive $WT_1\ P\ WTs_1\ P$
intros

$WTNew_1$:
$is\text{-}class\ P\ C \Longrightarrow$
$P,E \vdash_1 new\ C :: Class\ C$

$WTCast_1$:
$[\![\ P,E \vdash_1 e :: Class\ D;\ is\text{-}class\ P\ C;\ P \vdash C \preceq^* D \vee P \vdash D \preceq^* C\ ]\!]$
$\Longrightarrow P,E \vdash_1 Cast\ C\ e :: Class\ C$

$WTVal_1$:
$typeof\ v = Some\ T \Longrightarrow$
$P,E \vdash_1 Val\ v :: T$

$WTVar_1$:
$[\![\ E!i = T;\ i < size\ E\ ]\!]$
$\Longrightarrow P,E \vdash_1 Var\ i :: T$

$WTBinOp_1$:
$[\![\ P,E \vdash_1 e_1 :: T_1;\ P,E \vdash_1 e_2 :: T_2;$
$case\ bop\ of\ Eq \Rightarrow (P \vdash T_1 \leq T_2 \vee P \vdash T_2 \leq T_1) \wedge T = Boolean$
$\mid Add \Rightarrow T_1 = Integer \wedge T_2 = Integer \wedge T = Integer\ ]\!]$
$\Longrightarrow P,E \vdash_1 e_1 \ll bop \gg e_2 :: T$

$WTLAss_1$:
$[\![\ E!i = T;\ i < size\ E;\ P,E \vdash_1 e :: T';\ P \vdash T' \leq T\ ]\!]$
$\Longrightarrow P,E \vdash_1 i{:=}e :: Void$

$WTFAcc_1$:
$[\![\ P,E \vdash_1 e :: Class\ C;\ P \vdash C\ sees\ F{:}T\ in\ D\ ]\!]$
$\Longrightarrow P,E \vdash_1 e{\cdot}F\{D\} :: T$

$WTFAss_1$:
$[\![\ P,E \vdash_1 e_1 :: Class\ C;\ P \vdash C\ sees\ F{:}T\ in\ D;\ P,E \vdash_1 e_2 :: T';\ P \vdash T' \leq T\ ]\!]$
$\Longrightarrow P,E \vdash_1 e_1{\cdot}F\{D\} := e_2 :: Void$

$WTCall_1$:
$[\![P,E \vdash_1 e :: Class\ C;\ P \vdash C\ sees\ M{:}Ts' \rightarrow T = m\ in\ D;$
$P,E \vdash_1 es\ [::]\ Ts;\ \ P \vdash Ts\ [\leq]\ Ts']\!]$
$\Longrightarrow P,E \vdash_1 e{\cdot}M(es) :: T$

$WTBlock_1$:
$[\![is\text{-}type\ P\ T;\ P,E@[T] \vdash_1 e{::}T']\!]$
$\Longrightarrow\ P,E \vdash_1 \{i{:}T;\ e\} :: T'$

$WTSeq_1$:
$[\![P,E \vdash_1 e_1{::}T_1;\ \ P,E \vdash_1 e_2{::}T_2]\!]$
$\Longrightarrow\ P,E \vdash_1 e_1;;e_2 :: T_2$

$WTCond_1$:
$[\![P,E \vdash_1 e :: Boolean;\ \ P,E \vdash_1 e_1{::}T_1;\ \ P,E \vdash_1 e_2{::}T_2;$
$P \vdash T_1 \leq T_2 \vee P \vdash T_2 \leq T_1;\ \ P \vdash T_1 \leq T_2 \longrightarrow T = T_2;\ P \vdash T_2 \leq T_1 \longrightarrow T = T_1]\!]$
$\Longrightarrow P,E \vdash_1 if\ (e)\ e_1\ else\ e_2 :: T$

$WTWhile_1$:
$[\![P,E \vdash_1 e :: Boolean;\ \ P,E \vdash_1 c{::}T]\!]$
$\Longrightarrow P,E \vdash_1 while\ (e)\ c :: Void$

$WTThrow_1$:
$P,E \vdash_1 e :: Class\ C \Longrightarrow$
$P,E \vdash_1 throw\ e :: Void$

$WTTry_1$:
$[\![P,E \vdash_1 e_1 :: T;\ \ P,E@[Class\ C] \vdash_1 e_2 :: T;\ is\text{-}class\ P\ C]\!]$
$\Longrightarrow P,E \vdash_1 try\ e_1\ catch(C\ i)\ e_2 :: T$

$WTNil_1$:
$P,E \vdash_1 []\ [::]\ []$

$WTCons_1$:
$[\![P,E \vdash_1 e :: T;\ \ P,E \vdash_1 es\ [::]\ Ts]\!]$
$\Longrightarrow\ P,E \vdash_1 e\#es\ [::]\ T\#Ts$

lemma $WTs_1\text{-}same\text{-}size$: $\bigwedge Ts.\ P,E \vdash_1 es\ [::]\ Ts \Longrightarrow size\ es = size\ Ts$

lemma $WT_1\text{-}unique$:
$P,E \vdash_1 e :: T_1 \Longrightarrow (\bigwedge T_2.\ P,E \vdash_1 e :: T_2 \Longrightarrow T_1 = T_2)$ **and**
$P,E \vdash_1 es\ [::]\ Ts_1 \Longrightarrow (\bigwedge Ts_2.\ P,E \vdash_1 es\ [::]\ Ts_2 \Longrightarrow Ts_1 = Ts_2)$

lemma assumes wf: $wf\text{-}prog\ p\ P$
shows $WT_1\text{-}is\text{-}type$: $P,E \vdash_1 e :: T \Longrightarrow set\ E \subseteq types\ P \Longrightarrow is\text{-}type\ P\ T$
and $P,E \vdash_1 es\ [::]\ Ts \Longrightarrow True$

5.2.2 Well-formedness

— Indices in blocks increase by 1

consts
$\mathcal{B} :: expr_1 \Rightarrow nat \Rightarrow bool$
$\mathcal{B}s :: expr_1\ list \Rightarrow nat \Rightarrow bool$

primrec
$\mathcal{B}$ (*new C*) i = *True*
$\mathcal{B}$ (*Cast C e*) $i = \mathcal{B}\ e\ i$
$\mathcal{B}$ (*Val v*) i = *True*
$\mathcal{B}\ (e_1 \ll bop \gg e_2)\ i = (\mathcal{B}\ e_1\ i \wedge \mathcal{B}\ e_2\ i)$
$\mathcal{B}$ (*Var j*) i = *True*
$\mathcal{B}\ (e{\cdot}F\{D\})\ i = \mathcal{B}\ e\ i$
$\mathcal{B}\ (j{:=}e)\ i = \mathcal{B}\ e\ i$
$\mathcal{B}\ (e_1{\cdot}F\{D\} := e_2)\ i = (\mathcal{B}\ e_1\ i \wedge \mathcal{B}\ e_2\ i)$
$\mathcal{B}\ (e{\cdot}M(es))\ i = (\mathcal{B}\ e\ i \wedge \mathcal{B}s\ es\ i)$
$\mathcal{B}\ (\{j{:}T\ ;\ e\})\ i = (i = j \wedge \mathcal{B}\ e\ (i{+}1))$
$\mathcal{B}\ (e_1;;e_2)\ i = (\mathcal{B}\ e_1\ i \wedge \mathcal{B}\ e_2\ i)$
$\mathcal{B}\ (if\ (e)\ e_1\ else\ e_2)\ i = (\mathcal{B}\ e\ i \wedge \mathcal{B}\ e_1\ i \wedge \mathcal{B}\ e_2\ i)$
$\mathcal{B}\ (throw\ e)\ i = \mathcal{B}\ e\ i$
$\mathcal{B}\ (while\ (e)\ c)\ i = (\mathcal{B}\ e\ i \wedge \mathcal{B}\ c\ i)$
$\mathcal{B}\ (try\ e_1\ catch(C\ j)\ e_2)\ i = (\mathcal{B}\ e_1\ i \wedge i{=}j \wedge \mathcal{B}\ e_2\ (i{+}1))$

$\mathcal{B}s$ [] i = *True*
$\mathcal{B}s\ (e\#es)\ i = (\mathcal{B}\ e\ i \wedge \mathcal{B}s\ es\ i)$

constdefs
wf-J_1*-mdecl* :: J_1*-prog* ⇒ *cname* ⇒ *expr*$_1$ *mdecl* ⇒ *bool*
wf-J_1*-mdecl P C* ≡ $\lambda(M,Ts,T,body)$.
$(\exists T'.\ P,Class\ C\#Ts \vdash_1 body :: T' \wedge P \vdash T' \leq T) \wedge$
$\mathcal{D}\ body\ \lfloor\{..size\ Ts\}\rfloor \wedge \mathcal{B}\ body\ (size\ Ts + 1)$

lemma *wf-*J_1*-mdecl*[*simp*]:
wf-J_1*-mdecl P C* $(M,Ts,T,body) \equiv$
$((\exists T'.\ P,Class\ C\#Ts \vdash_1 body :: T' \wedge P \vdash T' \leq T) \wedge$
$\mathcal{D}\ body\ \lfloor\{..size\ Ts\}\rfloor \wedge \mathcal{B}\ body\ (size\ Ts + 1))$
translations
wf-J_1*-prog* == *wf-prog wf-*J_1*-mdecl*

end

5.3 Program Compilation

theory *PCompiler* = *WellForm*:

constdefs
$compM :: ('a \Rightarrow 'b) \Rightarrow 'a\ mdecl \Rightarrow 'b\ mdecl$
$compM\ f \equiv \lambda(M,\ Ts,\ T,\ m).\ (M,\ Ts,\ T,\ f\ m)$

$compC :: ('a \Rightarrow 'b) \Rightarrow 'a\ cdecl \Rightarrow 'b\ cdecl$
$compC\ f \equiv \lambda(C,D,Fdecls,Mdecls).\ (C,D,Fdecls,\ map\ (compM\ f)\ Mdecls)$

$compP :: ('a \Rightarrow 'b) \Rightarrow 'a\ prog \Rightarrow 'b\ prog$
$compP\ f \equiv map\ (compC\ f)$

Compilation preserves the program structure. Therfore lookup functions either commute with compilation (like method lookup) or are preserved by it (like the subclass relation).

lemma *map-of-map4*:
$map\text{-}of\ (map\ (\lambda(x,a,b,c).(x,a,b,f\ c))\ ts) =$
$option\text{-}map\ (\lambda(a,b,c).(a,b,f\ c)) \circ (map\text{-}of\ ts)$

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*]: $class\ (compP\ f\ P)\ C = option\text{-}map\ (\lambda c.\ snd(compC\ f\ (C,c)))\ (class\ P\ C)$

lemma *sees-methods-compP*:
$P \vdash C\ sees\text{-}methods\ Mm \Longrightarrow$
$compP\ f\ P \vdash C\ sees\text{-}methods\ (option\text{-}map\ (\lambda((Ts,T,m),D).\ ((Ts,T,f\ m),D)) \circ Mm)$

lemma *sees-method-compP*:
$P \vdash C\ sees\ M\colon\ Ts{\rightarrow}T = m\ in\ D \Longrightarrow$
$compP\ f\ P \vdash C\ sees\ M\colon\ Ts{\rightarrow}T = (f\ m)\ in\ D$

lemma [*simp*]:
$P \vdash C\ sees\ M\colon\ Ts{\rightarrow}T = m\ in\ D \Longrightarrow$
$method\ (compP\ f\ P)\ C\ M = (D,Ts,T,f\ m)$

lemma *sees-methods-compPD*:
$[\![\ cP \vdash C\ sees\text{-}methods\ Mm';\ cP = compP\ f\ P\ ]\!] \Longrightarrow$
$\exists\, Mm.\ P \vdash C\ sees\text{-}methods\ Mm\ \wedge$
$Mm' = (option\text{-}map\ (\lambda((Ts,T,m),D).\ ((Ts,T,f\ m),D)) \circ Mm)$

lemma *sees-method-compPD*:
$compP\ f\ P \vdash C\ sees\ M\colon\ Ts{\rightarrow}T = fm\ in\ D \Longrightarrow$
$\exists\, m.\ P \vdash C\ sees\ M\colon\ Ts{\rightarrow}T = m\ in\ D \wedge f\ m = fm$

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*]: $(compP\ (f{::}'a{\Rightarrow}'b)\ P \vdash C\ has\text{-}fields\ FDTs) = (P \vdash C\ has\text{-}fields\ FDTs)$

lemma [*simp*]: $fields\ (compP\ f\ P)\ C = fields\ P\ C$

lemma [*simp*]: $(compP\ f\ P \vdash C\ sees\ F{:}T\ in\ D) = (P \vdash C\ sees\ F{:}T\ in\ D)$

lemma [*simp*]: $field\ (compP\ f\ P)\ F\ D = field\ P\ F\ D$

5.3.1 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 *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*:
$[\![\bigwedge C\ M\ Ts\ T\ m.$
$[\![wf\text{-}mdecl\ wf_1\ P\ C\ (M,Ts,T,m);\ P \vdash C\ sees\ M{:}Ts{\rightarrow}T = m\ in\ C]\!]$
$\Longrightarrow wf\text{-}mdecl\ wf_2\ (compP\ f\ P)\ C\ (M,Ts,T,\ f\ m);$
$\forall x{\in}set\ P.\ wf\text{-}cdecl\ wf_1\ P\ x;\ x \in set\ (compP\ f\ P);\ wf\text{-}prog\ p\ P]\!]$
$\Longrightarrow wf\text{-}cdecl\ wf_2\ (compP\ f\ P)\ x$

lemma *wf-prog-compPI*:
assumes *lift*:
$\bigwedge C\ M\ Ts\ T\ m.$
$[\![P \vdash C\ sees\ M{:}Ts{\rightarrow}T = m\ in\ C;\ wf\text{-}mdecl\ wf_1\ P\ C\ (M,Ts,T,m)]\!]$
$\Longrightarrow wf\text{-}mdecl\ wf_2\ (compP\ f\ P)\ C\ (M,Ts,T,\ f\ m)$
and *wf*: $wf\text{-}prog\ wf_1\ P$
shows $wf\text{-}prog\ wf_2\ (compP\ f\ P)$

end

5.4 Indexing variables in variable lists

theory *Index* = *Main*:

In order to support local variables and arbitrarily nested blocks, the local variables are arranged as an indexed list. The outermost local variable ("this") is the first element in the list, the most recently created local variable the last element. When descending into a block structure, a corresponding list *Vs* of variable names is maintained. To find the index of some variable *V*, we have to find the index of the *last* occurrence of *V* in *Vs*. This is what *index* does:

consts
$\quad$ $index :: 'a\ list \Rightarrow 'a \Rightarrow nat$
primrec
$\quad$ $index\ []\ y = 0$
$\quad$ $index\ (x\#xs)\ y =$
$\quad$ $(if\ x{=}y\ then\ if\ x \in set\ xs\ then\ index\ xs\ y + 1\ else\ 0\ else\ index\ xs\ y + 1)$

constdefs
$\quad$ $hidden :: 'a\ list \Rightarrow nat \Rightarrow bool$
$\quad$ $hidden\ xs\ i \ \equiv\ i < size\ xs \wedge xs!i \in set(drop\ (i{+}1)\ xs)$

5.4.1 *index*

lemma [*simp*]: $index\ (xs\ @\ [x])\ x = size\ xs$

lemma [*simp*]: $(index\ (xs\ @\ [x])\ y = size\ xs) = (x = y)$

lemma [*simp*]: $x \in set\ xs \Longrightarrow xs\ !\ index\ xs\ x = x$

lemma [*simp*]: $x \notin set\ xs \Longrightarrow index\ xs\ x = size\ xs$

lemma *index-size-conv*[*simp*]: $(index\ xs\ x = size\ xs) = (x \notin set\ xs)$

lemma *size-index-conv*[*simp*]: $(size\ xs = index\ xs\ x) = (x \notin set\ xs)$

lemma $(index\ xs\ x < size\ xs) = (x \in set\ xs)$

lemma [*simp*]: $[\![\ y \in set\ xs;\ x \neq y\ ]\!] \Longrightarrow index\ (xs\ @\ [x])\ y = index\ xs\ y$

lemma *index-less-size*[*simp*]: $x \in set\ xs \Longrightarrow index\ xs\ x < size\ xs$
lemma *index-less-aux*: $[\![x \in set\ xs;\ size\ xs \leq n]\!] \Longrightarrow index\ xs\ x < n$

lemma [*simp*]: $x \in set\ xs \vee y \in set\ xs \Longrightarrow (index\ xs\ x = index\ xs\ y) = (x = y)$

lemma *inj-on-index*: $inj\text{-}on\ (index\ xs)\ (set\ xs)$

lemma *index-drop*: $\bigwedge x\ i.\ [\![\ x \in set\ xs;\ index\ xs\ x < i\ ]\!] \Longrightarrow x \notin set(drop\ i\ xs)$

5.4.2 *hidden*

lemma *hidden-index*: $x \in set\ xs \Longrightarrow hidden\ (xs\ @\ [x])\ (index\ xs\ x)$

lemma *hidden-inacc*: $hidden\ xs\ i \Longrightarrow index\ xs\ x \neq i$

lemma $[simp]$: $hidden\ xs\ i \Longrightarrow hidden\ (xs@[x])\ i$

lemma *fun-upds-apply*: $\bigwedge m\ ys.$
$(m(xs[\mapsto]ys))\ x =$
$(let\ xs' = take\ (size\ ys)\ xs$
$\ in\ if\ x \in set\ xs'\ then\ Some(ys\ !\ index\ xs'\ x)\ else\ m\ x)$

lemma *map-upds-apply-eq-Some*:
$((m(xs[\mapsto]ys))\ x = Some\ y) =$
$(let\ xs' = take\ (size\ ys)\ xs$
$\ in\ if\ x \in set\ xs'\ then\ ys\ !\ index\ xs'\ x = y\ else\ m\ x = Some\ y)$

lemma *map-upds-upd-conv-index*:
$[\![x \in set\ xs;\ size\ xs \leq size\ ys\ ]\!]$
$\Longrightarrow m(xs[\mapsto]ys)(x \mapsto y) = m(xs[\mapsto]ys[index\ xs\ x := y])$

end

5.5 Compilation Stage 1

theory *Compiler1* = *PCompiler* + *J1* + *Index*:

Replacing variable names by indices.

consts
$compE_1$ :: $vname\ list \Rightarrow expr \Rightarrow expr_1$
$compEs_1$ :: $vname\ list \Rightarrow expr\ list \Rightarrow expr_1\ list$

primrec
$compE_1\ Vs\ (new\ C) = new\ C$
$compE_1\ Vs\ (Cast\ C\ e) = Cast\ C\ (compE_1\ Vs\ e)$
$compE_1\ Vs\ (Val\ v) = Val\ v$
$compE_1\ Vs\ (e_1 \ll bop \gg e_2) = (compE_1\ Vs\ e_1) \ll bop \gg (compE_1\ Vs\ e_2)$
$compE_1\ Vs\ (Var\ V) = Var(index\ Vs\ V)$
$compE_1\ Vs\ (V{:=}e) = (index\ Vs\ V){:=}\ (compE_1\ Vs\ e)$
$compE_1\ Vs\ (e{\cdot}F\{D\}) = (compE_1\ Vs\ e){\cdot}F\{D\}$
$compE_1\ Vs\ (e_1{\cdot}F\{D\}{:=}e_2) = (compE_1\ Vs\ e_1){\cdot}F\{D\} := (compE_1\ Vs\ e_2)$
$compE_1\ Vs\ (e{\cdot}M(es)) = (compE_1\ Vs\ e){\cdot}M(compEs_1\ Vs\ es)$
$compE_1\ Vs\ \{V{:}T;\ e\} = \{(size\ Vs){:}T;\ compE_1\ (Vs@[V])\ e\}$
$compE_1\ Vs\ (e_1;;e_2) = (compE_1\ Vs\ e_1);;(compE_1\ Vs\ e_2)$
$compE_1\ Vs\ (if\ (e)\ e_1\ else\ e_2) = if\ (compE_1\ Vs\ e)\ (compE_1\ Vs\ e_1)\ else\ (compE_1\ Vs\ e_2)$
$compE_1\ Vs\ (while\ (e)\ c) = while\ (compE_1\ Vs\ e)\ (compE_1\ Vs\ c)$
$compE_1\ Vs\ (throw\ e) = throw\ (compE_1\ Vs\ e)$
$compE_1\ Vs\ (try\ e_1\ catch(C\ V)\ e_2) =$
$\quad try(compE_1\ Vs\ e_1)\ catch(C\ (size\ Vs))\ (compE_1\ (Vs@[V])\ e_2)$

$compEs_1\ Vs\ [] = []$
$compEs_1\ Vs\ (e\#es) = compE_1\ Vs\ e\ \#\ compEs_1\ Vs\ es$

lemma [*simp*]: $compEs_1\ Vs\ es = map\ (compE_1\ Vs)\ es$

consts
fin_1:: $expr \Rightarrow expr_1$
primrec
$fin_1(Val\ v) = Val\ v$
$fin_1(throw\ e) = throw(fin_1\ e)$

lemma *comp-final*: $final\ e \Longrightarrow compE_1\ Vs\ e = fin_1\ e$

lemma [*simp*]:
$\quad \bigwedge Vs.\ max\text{-}vars\ (compE_1\ Vs\ e) = max\text{-}vars\ e$
and $\bigwedge Vs.\ max\text{-}varss\ (compEs_1\ Vs\ es) = max\text{-}varss\ es$

Compiling programs:

constdefs
$compP_1$ :: $J\text{-}prog \Rightarrow J_1\text{-}prog$
$compP_1 \equiv compP\ (\lambda(pns,body).\ compE_1\ (this\#pns)\ body)$

end

5.6 Correctness of Stage 1

theory *Correctness1* = *J1WellForm* + *Compiler1*:

5.6.1 Correctness of program compilation

consts
$unmod :: expr_1 \Rightarrow nat \Rightarrow bool$
$unmods :: expr_1\ list \Rightarrow nat \Rightarrow bool$
primrec
$unmod\ (new\ C)\ i = True$
$unmod\ (Cast\ C\ e)\ i = unmod\ e\ i$
$unmod\ (Val\ v)\ i = True$
$unmod\ (e_1\ «bop»\ e_2)\ i = (unmod\ e_1\ i \wedge unmod\ e_2\ i)$
$unmod\ (Var\ i)\ j = True$
$unmod\ (i{:=}e)\ j = (i \neq j \wedge unmod\ e\ j)$
$unmod\ (e{\cdot}F\{D\})\ i = unmod\ e\ i$
$unmod\ (e_1{\cdot}F\{D\}{:=}e_2)\ i = (unmod\ e_1\ i \wedge unmod\ e_2\ i)$
$unmod\ (e{\cdot}M(es))\ i = (unmod\ e\ i \wedge unmods\ es\ i)$
$unmod\ \{j{:}T;\ e\}\ i = unmod\ e\ i$
$unmod\ (e_1;;e_2)\ i = (unmod\ e_1\ i \wedge\ \ unmod\ e_2\ i)$
$unmod\ (if\ (e)\ e_1\ else\ e_2)\ i = (unmod\ e\ i \wedge unmod\ e_1\ i \wedge unmod\ e_2\ i)$
$unmod\ (while\ (e)\ c)\ i = (unmod\ e\ i \wedge unmod\ c\ i)$
$unmod\ (throw\ e)\ i = unmod\ e\ i$
$unmod\ (try\ e_1\ catch(C\ i)\ e_2)\ j = (unmod\ e_1\ j \wedge (if\ i{=}j\ then\ False\ else\ unmod\ e_2\ j))$

$unmods\ ([])\ i = True$
$unmods\ (e\#es)\ i = (unmod\ e\ i \wedge unmods\ es\ i)$

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*:
$[\![\ P \vdash_1 \langle e,(h,ls)\rangle \Rightarrow \langle e',(h',ls')\rangle;\ unmod\ e\ i;\ i < size\ ls\ ]\!]$
$\Longrightarrow ls\ !\ i = ls'\ !\ i$
and $[\![\ P \vdash_1 \langle es,(h,ls)\rangle\ [\Rightarrow]\ \langle es',(h',ls')\rangle;\ unmods\ es\ i;\ i < size\ ls\ ]\!]$
$\Longrightarrow ls\ !\ i = ls'\ !\ i$

lemma *LAss-lem*:
$[\![x \in set\ xs;\ size\ xs \leq size\ ys\ ]\!]$
$\Longrightarrow m_1 \subseteq_m m_2(xs[\mapsto]ys) \Longrightarrow m_1(x{\mapsto}y) \subseteq_m m_2(xs[\mapsto]ys[index\ xs\ x := y])$

lemma *Block-lem*:
assumes *0*: $l \subseteq_m [Vs\ [\mapsto]\ ls]$
and *1*: $l' \subseteq_m [Vs\ [\mapsto]\ ls',\ V{\mapsto}v]$
and *hidden*: $V \in set\ Vs \Longrightarrow ls\ !\ index\ Vs\ V = ls'\ !\ index\ Vs\ V$
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*: $P \vdash \langle e,(h,l)\rangle \Rightarrow \langle e',(h',l')\rangle$
$\Longrightarrow (\bigwedge Vs\ ls.\ [\![\ fv\ e \subseteq set\ Vs;\ \ l \subseteq_m [Vs[\mapsto]ls];\ size\ Vs + max\text{-}vars\ e \leq size\ ls\ ]\!]$
$\Longrightarrow \exists ls'.\ compP_1\ P \vdash_1 \langle compE_1\ Vs\ e,(h,ls)\rangle \Rightarrow \langle fin_1\ e',(h',ls')\rangle \wedge l' \subseteq_m [Vs[\mapsto]ls'])$

and *evals*$_1$*-evals*: $P \vdash \langle es,(h,l)\rangle\ [\Rightarrow]\ \langle es',(h',l')\rangle$
$\Longrightarrow (\bigwedge Vs\ ls.\ [\![\ fvs\ es \subseteq set\ Vs;\ \ l \subseteq_m [Vs[\mapsto]ls];\ size\ Vs + max\text{-}varss\ es \leq size\ ls\ ]\!]$
$\Longrightarrow \exists\, ls'.\ compP_1\ P \vdash_1 \langle compEs_1\ Vs\ es,(h,ls)\rangle\ [\Rightarrow]\ \langle compEs_1\ Vs\ es',(h',ls')\rangle \wedge$
$l' \subseteq_m [Vs[\mapsto]ls'])$

5.6.2 Preservation of well-formedness

The compiler preserves well-formedness. Is less trivial than it may appear. We start with two simple properties: preservation of well-typedness

lemma *compE*$_1$*-pres-wt*: $\bigwedge Vs\ Ts\ U.$
$[\![\ P,[Vs[\mapsto]Ts] \vdash e :: U;\ size\ Ts = size\ Vs\ ]\!]$
$\Longrightarrow compP\ f\ P,Ts \vdash_1 compE_1\ Vs\ e :: U$
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-index*: $A \subseteq set(xs@[x]) \Longrightarrow index\ (xs\ @\ [x])\ `\ A =$
$(if\ x \in A\ then\ insert\ (size\ xs)\ (index\ xs\ `\ (A-\{x\}))\ else\ index\ xs\ `\ A)$

lemma *A-compE*$_1$*-None*[*simp*]:
$\bigwedge Vs.\ \mathcal{A}\ e = None \Longrightarrow \mathcal{A}\ (compE_1\ Vs\ e) = None$
and $\bigwedge Vs.\ \mathcal{A}s\ es = None \Longrightarrow \mathcal{A}s\ (compEs_1\ Vs\ es) = None$

lemma *A-compE*$_1$:
$\bigwedge A\ Vs.\ [\![\ \mathcal{A}\ e = \lfloor A\rfloor;\ fv\ e \subseteq set\ Vs\ ]\!] \Longrightarrow \mathcal{A}\ (compE_1\ Vs\ e) = \lfloor index\ Vs\ `\ A\rfloor$
and $\bigwedge A\ Vs.\ [\![\ \mathcal{A}s\ es = \lfloor A\rfloor;\ fvs\ es \subseteq set\ Vs\ ]\!] \Longrightarrow \mathcal{A}s\ (compEs_1\ Vs\ es) = \lfloor index\ Vs\ `\ A\rfloor$

lemma *D-None*[*iff*]: $\mathcal{D}\ (e::'a\ exp)\ None$ **and** [*iff*]: $\mathcal{D}s\ (es::'a\ exp\ list)\ None$

lemma *D-index-compE*$_1$:
$\bigwedge A\ Vs.\ [\![\ A \subseteq set\ Vs;\ fv\ e \subseteq set\ Vs\ ]\!] \Longrightarrow$
$\mathcal{D}\ e\ \lfloor A\rfloor \Longrightarrow \mathcal{D}\ (compE_1\ Vs\ e)\ \lfloor index\ Vs\ `\ A\rfloor$
and $\bigwedge A\ Vs.\ [\![\ A \subseteq set\ Vs;\ fvs\ es \subseteq set\ Vs\ ]\!] \Longrightarrow$
$\mathcal{D}s\ es\ \lfloor A\rfloor \Longrightarrow \mathcal{D}s\ (compEs_1\ Vs\ es)\ \lfloor index\ Vs\ `\ A\rfloor$

lemma *index-image-set*: $distinct\ xs \Longrightarrow index\ xs\ `\ set\ xs = \{..size\ xs(\}$

lemma *D-compE*$_1$:
$[\![\ \mathcal{D}\ e\ \lfloor set\ Vs\rfloor;\ fv\ e \subseteq set\ Vs;\ distinct\ Vs\ ]\!] \Longrightarrow \mathcal{D}\ (compE_1\ Vs\ e)\ \lfloor\{..length\ Vs(\}\rfloor$

lemma *D-compE*$_1$*'*:
assumes $\mathcal{D}\ e\ \lfloor set(V\#Vs)\rfloor$ **and** $fv\ e \subseteq set(V\#Vs)$ **and** $distinct(V\#Vs)$
shows $\mathcal{D}\ (compE_1\ (V\#Vs)\ e)\ \lfloor\{..length\ Vs\}\rfloor$

lemma *compP*$_1$*-pres-wf*: $wf\text{-}J\text{-}prog\ P \Longrightarrow wf\text{-}J_1\text{-}prog\ (compP_1\ P)$

end

5.7 Compilation Stage 2

theory *Compiler2* = *PCompiler* + *J1* + *JVMExec*:

consts
  $compE_2$ :: $expr_1 \Rightarrow instr\ list$
  $compEs_2$ :: $expr_1\ list \Rightarrow instr\ list$

primrec
$compE_2\ (new\ C) = [New\ C]$
$compE_2\ (Cast\ C\ e) = compE_2\ e\ @\ [Checkcast\ C]$
$compE_2\ (Val\ v) = [Push\ v]$
$compE_2\ (e_1\ «bop»\ e_2) = compE_2\ e_1\ @\ compE_2\ e_2\ @$
  $(case\ bop\ of\ Eq \Rightarrow [CmpEq]$
    $|\ Add \Rightarrow [IAdd])$
$compE_2\ (Var\ i) = [Load\ i]$
$compE_2\ (i{:=}e) = compE_2\ e\ @\ [Store\ i,\ Push\ Unit]$
$compE_2\ (e{\cdot}F\{D\}) = compE_2\ e\ @\ [Getfield\ F\ D]$
$compE_2\ (e_1{\cdot}F\{D\} := e_2) =$
    $compE_2\ e_1\ @\ compE_2\ e_2\ @\ [Putfield\ F\ D,\ Push\ Unit]$
$compE_2\ (e{\cdot}M(es)) = compE_2\ e\ @\ compEs_2\ es\ @\ [Invoke\ M\ (size\ es)]$
$compE_2\ (\{i{:}T;\ e\}) = compE_2\ e$
$compE_2\ (e_1;;e_2) = compE_2\ e_1\ @\ [Pop]\ @\ compE_2\ e_2$
$compE_2\ (if\ (e)\ e_1\ else\ e_2) =$
    $(let\ cnd\ = compE_2\ e;$
      $thn\ = compE_2\ e_1;$
      $els\ = compE_2\ e_2;$
      $test\ = IfFalse\ (int(size\ thn + 2));$
      $thnex = Goto\ (int(size\ els + 1))$
    $in\ cnd\ @\ [test]\ @\ thn\ @\ [thnex]\ @\ els)$
$compE_2\ (while\ (e)\ c) =$
    $(let\ cnd\ = compE_2\ e;$
      $bdy\ = compE_2\ c;$
      $test\ = IfFalse\ (int(size\ bdy + 3));$
      $loop\ = Goto\ (-int(size\ bdy + size\ cnd + 2))$
    $in\ cnd\ @\ [test]\ @\ bdy\ @\ [Pop]\ @\ [loop]\ @\ [Push\ Unit])$
$compE_2\ (throw\ e) = compE_2\ e\ @\ [instr.Throw]$
$compE_2\ (try\ e_1\ catch(C\ i)\ e_2) =$
  $(let\ catch = compE_2\ e_2$
   $in\ compE_2\ e_1\ @\ [Goto\ (int(size\ catch){+}2),\ Store\ i]\ @\ catch)$

$compEs_2\ []\ = []$
$compEs_2\ (e\#es) = compE_2\ e\ @\ compEs_2\ es$

Compilation of exception table. Is given start address of code to compute absolute addresses necessary in exception table.

consts
  $compxE_2$ :: $expr_1 \Rightarrow pc \Rightarrow nat \Rightarrow ex\text{-}table$
  $compxEs_2$ :: $expr_1\ list \Rightarrow pc \Rightarrow nat \Rightarrow ex\text{-}table$

primrec
$compxE_2\ (new\ C)\ pc\ d = []$
$compxE_2\ (Cast\ C\ e)\ pc\ d = compxE_2\ e\ pc\ d$
$compxE_2\ (Val\ v)\ pc\ d = []$

$compxE_2\ (e_1\ «bop»\ e_2)\ pc\ d =$
$\quad compxE_2\ e_1\ pc\ d\ @\ compxE_2\ e_2\ (pc + size(compE_2\ e_1))\ (d+1)$
$compxE_2\ (Var\ i)\ pc\ d = []$
$compxE_2\ (i{:=}e)\ pc\ d = compxE_2\ e\ pc\ d$
$compxE_2\ (e{\cdot}F\{D\})\ pc\ d = compxE_2\ e\ pc\ d$
$compxE_2\ (e_1{\cdot}F\{D\} := e_2)\ pc\ d =$
$\quad compxE_2\ e_1\ pc\ d\ @\ compxE_2\ e_2\ (pc + size(compE_2\ e_1))\ (d+1)$
$compxE_2\ (e{\cdot}M(es))\ pc\ d =$
$\quad compxE_2\ e\ pc\ d\ @\ compxEs_2\ es\ (pc + size(compE_2\ e))\ (d+1)$
$compxE_2\ (\{i{:}T;\ e\})\ pc\ d = compxE_2\ e\ pc\ d$
$compxE_2\ (e_1;;e_2)\ pc\ d =$
$\quad compxE_2\ e_1\ pc\ d\ @\ compxE_2\ e_2\ (pc{+}size(compE_2\ e_1){+}1)\ d$
$compxE_2\ (if\ (e)\ e_1\ else\ e_2)\ pc\ d =$
$\qquad (let\ pc_1 \quad = pc + size(compE_2\ e) + 1;$
$\qquad\qquad pc_2 \quad = pc_1 + size(compE_2\ e_1) + 1$
$\qquad\ in\ compxE_2\ e\ pc\ d\ @\ compxE_2\ e_1\ pc_1\ d\ @\ compxE_2\ e_2\ pc_2\ d)$
$compxE_2\ (while\ (b)\ e)\ pc\ d =$
$\quad compxE_2\ b\ pc\ d\ @\ compxE_2\ e\ (pc{+}size(compE_2\ b){+}1)\ d$
$compxE_2\ (throw\ e)\ pc\ d = compxE_2\ e\ pc\ d$
$compxE_2\ (try\ e_1\ catch(C\ i)\ e_2)\ pc\ d =$
$\quad (let\ pc_1 = pc + size(compE_2\ e_1)$
$\quad\ in\ compxE_2\ e_1\ pc\ d\ @\ compxE_2\ e_2\ (pc_1{+}2)\ d\ @\ [(pc,pc_1,C,pc_1{+}1,d)])$

$compxEs_2\ []\ pc\ d \quad = []$
$compxEs_2\ (e\#es)\ pc\ d = compxE_2\ e\ pc\ d\ @\ compxEs_2\ es\ (pc{+}size(compE_2\ e))\ (d+1)$

consts
$\quad max\text{-}stack :: expr_1 \Rightarrow nat$
$\quad max\text{-}stacks :: expr_1\ list \Rightarrow nat$
primrec
$max\text{-}stack\ (new\ C) = 1$
$max\text{-}stack\ (Cast\ C\ e) = max\text{-}stack\ e$
$max\text{-}stack\ (Val\ v) = 1$
$max\text{-}stack\ (e_1\ «bop»\ e_2) = max\ (max\text{-}stack\ e_1)\ (max\text{-}stack\ e_2) + 1$
$max\text{-}stack\ (Var\ i) = 1$
$max\text{-}stack\ (i{:=}e) = max\text{-}stack\ e$
$max\text{-}stack\ (e{\cdot}F\{D\}) = max\text{-}stack\ e$
$max\text{-}stack\ (e_1{\cdot}F\{D\} := e_2) = max\ (max\text{-}stack\ e_1)\ (max\text{-}stack\ e_2) + 1$
$max\text{-}stack\ (e{\cdot}M(es)) = max\ (max\text{-}stack\ e)\ (max\text{-}stacks\ es) + 1$
$max\text{-}stack\ (\{i{:}T;\ e\}) = max\text{-}stack\ e$
$max\text{-}stack\ (e_1;;e_2) = max\ (max\text{-}stack\ e_1)\ (max\text{-}stack\ e_2)$
$max\text{-}stack\ (if\ (e)\ e_1\ else\ e_2) =$
$\quad max\ (max\text{-}stack\ e)\ (max\ (max\text{-}stack\ e_1)\ (max\text{-}stack\ e_2))$
$max\text{-}stack\ (while\ (e)\ c) = max\ (max\text{-}stack\ e)\ (max\text{-}stack\ c)$
$max\text{-}stack\ (throw\ e) = max\text{-}stack\ e$
$max\text{-}stack\ (try\ e_1\ catch(C\ i)\ e_2) = max\ (max\text{-}stack\ e_1)\ (max\text{-}stack\ e_2)$

$max\text{-}stacks\ [] = 0$
$max\text{-}stacks\ (e\#es) = max\ (max\text{-}stack\ e)\ (1 + max\text{-}stacks\ es)$

lemma $max\text{-}stack1{:}\ 1 \le max\text{-}stack\ e$

constdefs
$\quad compMb_2 :: expr_1 \Rightarrow jvm\text{-}method$

$compMb_2 \ \equiv \ \lambda body.$
$let\ ins = compE_2\ body\ @\ [Return];$
$\quad xt = compxE_2\ body\ 0\ 0$
$in\ (max\text{-}stack\ body,\ max\text{-}vars\ body,\ ins,\ xt)$

$compP_2 :: J_1\text{-}prog \Rightarrow jvm\text{-}prog$
$compP_2 \ \equiv \ compP\ compMb_2$

lemma $compMb_2$ [$simp$]:
$compMb_2\ e = (max\text{-}stack\ e,\ max\text{-}vars\ e,\ compE_2\ e\ @\ [Return],\ compxE_2\ e\ 0\ 0)$

end

5.8 Correctness of Stage 2

theory *Correctness2* = *List-Prefix* + *Compiler2*:

5.8.1 Instruction sequences

How to select individual instructions and subsequences of instructions from a program given the class, method and program counter.

constdefs
$before :: jvm\text{-}prog \Rightarrow cname \Rightarrow mname \Rightarrow nat \Rightarrow instr\ list \Rightarrow bool$
$((\text{-},\text{-},\text{-},\text{-}/ \rhd \text{-})\ [51,0,0,0,51]\ 50)$
$P,C,M,pc \rhd is\ \equiv\ is \le drop\ pc\ (instrs\text{-}of\ P\ C\ M)$

$at :: jvm\text{-}prog \Rightarrow cname \Rightarrow mname \Rightarrow nat \Rightarrow instr \Rightarrow bool$
$((\text{-},\text{-},\text{-},\text{-}/ \triangleright \text{-})\ [51,0,0,0,51]\ 50)$
$P,C,M,pc \triangleright i\ \equiv\ \exists is.\ drop\ pc\ (instrs\text{-}of\ P\ C\ M) = i\#is$

lemma [*simp*]: $P,C,M,pc \rhd []$

lemma [*simp*]: $P,C,M,pc \rhd (i\#is) = (P,C,M,pc \triangleright i \wedge P,C,M,pc + 1 \rhd is)$

lemma [*simp*]: $P,C,M,pc \rhd (is_1\ @\ is_2) = (P,C,M,pc \rhd is_1 \wedge P,C,M,pc + size\ is_1 \rhd is_2)$

lemma [*simp*]: $P,C,M,pc \triangleright i \Longrightarrow instrs\text{-}of\ P\ C\ M\ !\ pc = i$

lemma *beforeM*:
$P \vdash C\ sees\ M\colon Ts{\rightarrow}T = body\ in\ D \Longrightarrow$
$compP_2\ P,D,M,0 \rhd compE_2\ body\ @\ [Return]$

This lemma executes a single instruction by rewriting:

lemma [*simp*]:
$P,C,M,pc \triangleright instr \Longrightarrow$
$(P \vdash (None,\ h,\ (vs,ls,C,M,pc)\ \#\ frs) -jvm\rightarrow \sigma') =$
$((None,\ h,\ (vs,ls,C,M,pc)\ \#\ frs) = \sigma' \vee$
$(\exists \sigma.\ exec(P,(None,\ h,\ (vs,ls,C,M,pc)\ \#\ frs)) = Some\ \sigma \wedge P \vdash \sigma -jvm\rightarrow \sigma'))$

5.8.2 Exception tables

constdefs
$pcs :: ex\text{-}table \Rightarrow nat\ set$
$pcs\ xt\ \equiv\ \bigcup(f,t,C,h,d) \in set\ xt.\ \{f\ ..\ t(\}$

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*]: $pc < pc_0 \vee pc_0{+}size(compEs_2\ es) \leq pc \Longrightarrow pc \notin pcs(compxEs_2\ es\ pc_0\ d)$

lemma [*simp*]: $pc_1 + size(compE_2\ e_1) \leq pc_2 \Longrightarrow pcs(compxE_2\ e_1\ pc_1\ d_1) \cap pcs(compxE_2\ e_2\ pc_2\ d_2) = \{\}$

lemma [*simp*]: $pc_1 + size(compE_2\ e) \leq pc_2 \Longrightarrow pcs(compxE_2\ e\ pc_1\ d_1) \cap pcs(compxEs_2\ es\ pc_2\ d_2) = \{\}$

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*: $pc' < pc \vee pc{+}size(compE_2\ e) \leq pc'$
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*: $pc' < pc \vee pc{+}size(compEs_2\ es) \leq pc'$
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*:
$matches\text{-}ex\text{-}entry\ P\ C\ pc\ (start,\ end,\ catch\text{-}type,\ handler) =$
$(start \leq pc \wedge pc < end \wedge\ P \vdash C \preceq^* catch\text{-}type)$

constdefs
$caught :: jvm\text{-}prog \Rightarrow pc \Rightarrow heap \Rightarrow addr \Rightarrow ex\text{-}table \Rightarrow bool$
$caught\ P\ pc\ h\ a\ xt\ \equiv$
$(\exists entry \in set\ xt.\ matches\text{-}ex\text{-}entry\ P\ (cname\text{-}of\ h\ a)\ pc\ entry)$

$beforex :: jvm\text{-}prog \Rightarrow cname \Rightarrow mname \Rightarrow ex\text{-}table \Rightarrow nat\ set \Rightarrow nat \Rightarrow bool$
((2-,/-,/- ▷/ - /′/ -,/-) [51,0,0,0,0,51] 50)
$P,C,M \rhd xt\ /\ I,d \equiv$
$\exists xt_0\ xt_1.\ ex\text{-}table\text{-}of\ P\ C\ M = xt_0\ @\ xt\ @\ xt_1 \wedge pcs\ xt_0 \cap I = \{\} \wedge pcs\ xt \subseteq I \wedge$
$(\forall pc \in I.\ \forall C\ pc'\ d'.\ match\text{-}ex\text{-}table\ P\ C\ pc\ xt_1 = \lfloor(pc',d')\rfloor \longrightarrow d' \leq d)$

$dummyx :: jvm\text{-}prog \Rightarrow cname \Rightarrow mname \Rightarrow ex\text{-}table \Rightarrow nat\ set \Rightarrow nat \Rightarrow bool$ ((2-,-,- ▷/ - ′/-,-) [51,0,0,0,0,51] 50)
$P,C,M \rhd xt/I,d \equiv P,C,M \rhd xt/I,d$

lemma *beforexD1*: $P,C,M \rhd xt\ /\ I,d \Longrightarrow pcs\ xt \subseteq I$

lemma *beforex-mono*: $[\![\ P,C,M \rhd xt/I,d';\ d' \leq d\ ]\!] \Longrightarrow P,C,M \rhd xt/I,d$

lemma [*simp*]: $P,C,M \rhd xt/I,d \Longrightarrow P,C,M \rhd xt/I,Suc\ d$

lemma *beforex-append*[*simp*]:
$pcs\ xt_1 \cap pcs\ xt_2 = \{\} \Longrightarrow$
$P,C,M \rhd xt_1\ @\ xt_2/I,d =$
$(P,C,M \rhd xt_1/I{-}pcs\ xt_2,d\ \wedge\ P,C,M \rhd xt_2/I{-}pcs\ xt_1,d \wedge P,C,M \rhd xt_1@xt_2/I,d)$

lemma *beforex-appendD1*:
$[\![\ P,C,M \rhd xt_1\ @\ xt_2\ @\ [(f,t,D,h,d)]\ /\ I,d;$
$pcs\ xt_1 \subseteq J;\ J \subseteq I;\ J \cap pcs\ xt_2 = \{\}\ ]\!]$
$\Longrightarrow P,C,M \rhd xt_1\ /\ J,d$

lemma *beforex-appendD2*:
$[\![\ P,C,M \rhd xt_1\ @\ xt_2\ @\ [(f,t,D,h,d)]\ /\ I,d;$
$pcs\ xt_2 \subseteq J;\ J \subseteq I;\ J \cap pcs\ xt_1 = \{\}\ ]\!]$
$\Longrightarrow P,C,M \rhd xt_2\ /\ J,d$

lemma *beforexM*:
$P \vdash C\ sees\ M\colon Ts{\rightarrow}T = body\ in\ D \Longrightarrow$
$compP_2\ P,D,M \rhd compxE_2\ body\ 0\ 0/\{..size(compE_2\ body)(\},0$

lemma *match-ex-table-SomeD2*:
$[\![\ match\text{-}ex\text{-}table\ P\ D\ pc\ (ex\text{-}table\text{-}of\ P\ C\ M) = \lfloor(pc',d')\rfloor;$
$P,C,M \rhd xt/I,d;\ \forall x \in set\ xt.\ \neg\ matches\text{-}ex\text{-}entry\ P\ D\ pc\ x;\ pc \in I\ ]\!]$
$\Longrightarrow d' \le d$

lemma *match-ex-table-SomeD1*:
$[\![\ match\text{-}ex\text{-}table\ P\ D\ pc\ (ex\text{-}table\text{-}of\ P\ C\ M) = \lfloor(pc',d')\rfloor;$
$P,C,M \rhd xt\ /\ I,d;\ pc \in I;\ pc \notin pcs\ xt\ ]\!] \Longrightarrow d' \le d$

5.8.3 The correctness proof

constdefs
$handle :: jvm\text{-}prog \Rightarrow cname \Rightarrow mname \Rightarrow addr \Rightarrow heap \Rightarrow val\ list \Rightarrow val\ list \Rightarrow nat \Rightarrow frame\ list$
$\Rightarrow jvm\text{-}state$
$handle\ P\ C\ M\ a\ h\ vs\ ls\ pc\ frs\ \equiv\ find\text{-}handler\ P\ a\ h\ ((vs,ls,C,M,pc)\ \#\ frs)$

lemma *handle-Cons*:
$[\![\ P,C,M \rhd xt/I,d;\ d \le size\ vs;\ pc \in I;$
$\forall x \in set\ xt.\ \neg\ matches\text{-}ex\text{-}entry\ P\ (cname\text{-}of\ h\ xa)\ pc\ x\ ]\!] \Longrightarrow$
$handle\ P\ C\ M\ xa\ h\ (v\ \#\ vs)\ ls\ pc\ frs = handle\ P\ C\ M\ xa\ h\ vs\ ls\ pc\ frs$
lemma *handle-append*:
$[\![\ P,C,M \rhd xt/I,d;\ d \le size\ vs;\ pc \in I;\ pc \notin pcs\ xt\ ]\!] \Longrightarrow$
$handle\ P\ C\ M\ xa\ h\ (ws\ @\ vs)\ ls\ pc\ frs = handle\ P\ C\ M\ xa\ h\ vs\ ls\ pc\ frs$

lemma *aux-isin*[*simp*]: $[\![\ B \subseteq A;\ a \in B\ ]\!] \Longrightarrow a \in A$

lemma fixes P_1 **defines** [*simp*]: $P \equiv compP_2\ P_1$
shows *Jcc*:
$P_1 \vdash_1 \langle e,(h_0,ls_0)\rangle \Rightarrow \langle ef,(h_1,ls_1)\rangle \Longrightarrow$
$(\bigwedge C\ M\ pc\ v\ xa\ vs\ frs\ I.$
$[\![\ P,C,M,pc \rhd compE_2\ e;\ P,C,M \rhd compxE_2\ e\ pc\ (size\ vs)/I,size\ vs;$
$\{pc..pc{+}size(compE_2\ e)(\} \subseteq I\ ]\!] \Longrightarrow$
$(ef = Val\ v \longrightarrow$

$P \vdash (None,h_0,(vs,ls_0,C,M,pc)\#frs)\ -jvm\rightarrow$
$\quad (None,h_1,(v\#vs,ls_1,C,M,pc+size(compE_2\ e))\#frs))$
$\wedge$
$(ef = Throw\ xa \longrightarrow$
$(\exists pc_1.\ pc \leq pc_1 \wedge pc_1 < pc + size(compE_2\ e) \wedge$
$\quad \neg\ caught\ P\ pc_1\ h_1\ xa\ (compxE_2\ e\ pc\ (size\ vs)) \wedge$
$\quad P \vdash (None,h_0,(vs,ls_0,C,M,pc)\#frs)\ -jvm\rightarrow\ handle\ P\ C\ M\ xa\ h_1\ vs\ ls_1\ pc_1\ frs)))$
and $P_1 \vdash_1 \langle es,(h_0,ls_0)\rangle\ [\Rightarrow]\ \langle fs,(h_1,ls_1)\rangle \Longrightarrow$
$(\bigwedge C\ M\ pc\ ws\ xa\ es'\ vs\ frs\ I.$
$[\![\ P,C,M,pc \rhd compEs_2\ es;\ P,C,M \rhd compxEs_2\ es\ pc\ (size\ vs)/I,size\ vs;$
$\{pc..pc+size(compEs_2\ es)(\} \subseteq I\ ]\!] \Longrightarrow$
$(fs = map\ Val\ ws \longrightarrow$
$P \vdash (None,h_0,(vs,ls_0,C,M,pc)\#frs)\ -jvm\rightarrow$
$\quad (None,h_1,(rev\ ws\ @\ vs,ls_1,C,M,pc+size(compEs_2\ es))\#frs))$
$\wedge$
$(fs = map\ Val\ ws\ @\ Throw\ xa\ \#\ es' \longrightarrow$
$(\exists pc_1.\ pc \leq pc_1 \wedge pc_1 < pc + size(compEs_2\ es) \wedge$
$\quad \neg\ caught\ P\ pc_1\ h_1\ xa\ (compxEs_2\ es\ pc\ (size\ vs)) \wedge$
$\quad P \vdash (None,h_0,(vs,ls_0,C,M,pc)\#frs)\ -jvm\rightarrow\ handle\ P\ C\ M\ xa\ h_1\ vs\ ls_1\ pc_1\ frs)))$

lemma *atLeast0AtMost*[*simp*]: $\{0{::}nat..n\} = \{..n\}$
by *auto*

lemma *atLeast0LessThan*[*simp*]: $\{0{::}nat..n(\} = \{..n(\}$
by *auto*

consts $exception :: 'a\ exp \Rightarrow addr\ option$
recdef *exception* {}
$exception(Throw\ a) = Some\ a$
$exception\ e = None$

lemma $comp_2$-*correct*:
assumes *method*: $P_1 \vdash C\ sees\ M{:}Ts{\rightarrow}T = body\ in\ C$
and *eval*: $P_1 \vdash_1 \langle body,(h,ls)\rangle \Rightarrow \langle e',(h',ls')\rangle$
shows $compP_2\ P_1 \vdash (None,h,[([],ls,C,M,0)])\ -jvm\rightarrow\ (exception\ e',h',[])$
end

5.9 Combining Stages 1 and 2

theory *Compiler* = *Correctness1* + *Correctness2*:

constdefs
$J2JVM :: J\text{-}prog \Rightarrow jvm\text{-}prog$
$J2JVM \equiv compP_2 \circ compP_1$

theorem *comp-correct*:
assumes *wwf*: *wwf-J-prog P*
and *method*: $P \vdash C\ sees\ M{:}Ts{\rightarrow}T = (pns,body)\ in\ C$
and *eval*: $P \vdash \langle body,(h,[this\#pns\ [\mapsto]\ vs])\rangle \Rightarrow \langle e',(h',l')\rangle$
and *sizes*: $size\ vs = size\ pns + 1 \quad size\ rest = max\text{-}vars\ body$
shows $J2JVM\ P \vdash (None,h,[([],vs@rest,C,M,0)])\ {-jvm\rightarrow}\ (exception\ e',h',[])$

end

5.10 Preservation of Well-Typedness

theory *TypeComp* = *Compiler* + *BVSpec*:

constdefs
$ty :: J_1\text{-}prog \Rightarrow ty\ list \Rightarrow expr_1 \Rightarrow ty$
$ty\ P\ E\ e \equiv\ THE\ T.\ P,E \vdash_1 e :: T$

$ty_l :: nat \Rightarrow ty\ list \Rightarrow nat\ set \Rightarrow ty_l$
$ty_l\ m\ E\ A' \equiv map\ (\lambda i.\ if\ i \in A' \wedge i < size\ E\ then\ OK(E!i)\ else\ Err)\ [0..m(]$

$ty_i{}' :: nat \Rightarrow ty\ list \Rightarrow ty\ list \Rightarrow nat\ set\ option \Rightarrow ty_i{}'$
$ty_i{}'\ m\ ST\ E\ A \equiv case\ A\ of\ None \Rightarrow None \mid \lfloor A' \rfloor \Rightarrow Some(ST,\ ty_l\ m\ E\ A')$

$after :: J_1\text{-}prog \Rightarrow nat \Rightarrow ty\ list \Rightarrow nat\ set\ option \Rightarrow ty\ list \Rightarrow expr_1 \Rightarrow ty_i{}'$
$after\ P\ m\ E\ A\ ST\ e \equiv ty_i{}'\ m\ (ty\ P\ E\ e\ \#\ ST)\ E\ (A \sqcup \mathcal{A}\ e)$

locale (**open**) *TC0* =
fixes *P* **and** *mxl*

fixes $ty :: ty\ list \Rightarrow expr_1 \Rightarrow ty$
defines $ty\ E\ e \equiv TypeComp.ty\ P\ E\ e$

fixes $ty_l :: ty\ list \Rightarrow nat\ set \Rightarrow ty_l$
defines ty_l: $ty_l\ E\ A' \equiv TypeComp.ty_l\ mxl\ E\ A'$

fixes $ty_i{}' :: ty\ list \Rightarrow ty\ list \Rightarrow nat\ set\ option \Rightarrow ty_i{}'$
defines $ty_i{}'$: $ty_i{}'\ ST\ E\ A \equiv TypeComp.ty_i{}'\ mxl\ ST\ E\ A$

fixes $after :: ty\ list \Rightarrow nat\ set\ option \Rightarrow ty\ list \Rightarrow expr_1 \Rightarrow ty_i{}'$
defines *after*: $after\ E\ A\ ST\ e \equiv TypeComp.after\ P\ mxl\ E\ A\ ST\ e$

notes *after-def* = *TypeComp.after-def* [*of P mxl*, *folded after ty-def* $ty_i{}'$]
notes $ty_i{}'$*-def* = $TypeComp.ty_i{}'$*-def* [*of mxl*, *folded* $ty_l\ ty_i{}'$]
notes ty_l*-def* = $TypeComp.ty_l$*-def* [*of mxl*, *folded* ty_l]

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*:
$set\ E \subseteq types\ P \Longrightarrow ty_l\ E\ A \in list\ mxl\ (err\ (types\ P))$

consts
$compT :: J_1\text{-}prog \Rightarrow nat \Rightarrow ty\ list \Rightarrow nat\ hyperset \Rightarrow ty\ list \Rightarrow expr_1 \Rightarrow ty_i{}'\ list$
$compTs :: J_1\text{-}prog \Rightarrow nat \Rightarrow ty\ list \Rightarrow nat\ hyperset \Rightarrow ty\ list \Rightarrow expr_1\ list \Rightarrow ty_i{}'\ list$

primrec
$compT\ P\ m\ E\ A\ ST\ (new\ C) = []$
$compT\ P\ m\ E\ A\ ST\ (Cast\ C\ e) =$
$\quad compT\ P\ m\ E\ A\ ST\ e\ @\ [after\ P\ m\ E\ A\ ST\ e]$
$compT\ P\ m\ E\ A\ ST\ (Val\ v) = []$
$compT\ P\ m\ E\ A\ ST\ (e_1\ «bop»\ e_2) =$
$(let\ ST_1 = ty\ P\ E\ e_1\#ST;\ A_1 = A \sqcup \mathcal{A}\ e_1\ in$
$\quad compT\ P\ m\ E\ A\ ST\ e_1\ @\ [after\ P\ m\ E\ A\ ST\ e_1]\ @$
$\quad compT\ P\ m\ E\ A_1\ ST_1\ e_2\ @\ [after\ P\ m\ E\ A_1\ ST_1\ e_2])$
$compT\ P\ m\ E\ A\ ST\ (Var\ i) = []$
$compT\ P\ m\ E\ A\ ST\ (i := e) = compT\ P\ m\ E\ A\ ST\ e\ @$
$\quad [after\ P\ m\ E\ A\ ST\ e,\ ty_i{}'\ m\ ST\ E\ (A \sqcup \mathcal{A}\ e \sqcup \lfloor\{i\}\rfloor)]$
$compT\ P\ m\ E\ A\ ST\ (e\cdot F\{D\}) =$
$\quad compT\ P\ m\ E\ A\ ST\ e\ @\ [after\ P\ m\ E\ A\ ST\ e]$
$compT\ P\ m\ E\ A\ ST\ (e_1\cdot F\{D\} := e_2) =$
$(let\ ST_1 = ty\ P\ E\ e_1\#ST;\ A_1 = A \sqcup \mathcal{A}\ e_1;\ A_2 = A_1 \sqcup \mathcal{A}\ e_2\ in$
$\quad compT\ P\ m\ E\ A\ ST\ e_1\ @\ [after\ P\ m\ E\ A\ ST\ e_1]\ @$
$\quad compT\ P\ m\ E\ A_1\ ST_1\ e_2\ @\ [after\ P\ m\ E\ A_1\ ST_1\ e_2]\ @$
$\quad [ty_i{}'\ m\ ST\ E\ A_2])$
$compT\ P\ m\ E\ A\ ST\ \{i{:}T;\ e\} = compT\ P\ m\ (E@[T])\ (A\ominus i)\ ST\ e$
$compT\ P\ m\ E\ A\ ST\ (e_1;;e_2) =$
$(let\ A_1 = A \sqcup \mathcal{A}\ e_1\ in$
$\quad compT\ P\ m\ E\ A\ ST\ e_1\ @\ [after\ P\ m\ E\ A\ ST\ e_1,\ ty_i{}'\ m\ ST\ E\ A_1]\ @$
$\quad compT\ P\ m\ E\ A_1\ ST\ e_2)$
$compT\ P\ m\ E\ A\ ST\ (if\ (e)\ e_1\ else\ e_2) =$
$(let\ A_0 = A \sqcup \mathcal{A}\ e;\ \tau = ty_i{}'\ m\ ST\ E\ A_0\ in$
$\quad compT\ P\ m\ E\ A\ ST\ e\ @\ [after\ P\ m\ E\ A\ ST\ e,\ \tau]\ @$
$\quad compT\ P\ m\ E\ A_0\ ST\ e_1\ @\ [after\ P\ m\ E\ A_0\ ST\ e_1,\ \tau]\ @$
$\quad compT\ P\ m\ E\ A_0\ ST\ e_2)$
$compT\ P\ m\ E\ A\ ST\ (while\ (e)\ c) =$
$(let\ A_0 = A \sqcup \mathcal{A}\ e;\ \ A_1 = A_0 \sqcup \mathcal{A}\ c;\ \tau = ty_i{}'\ m\ ST\ E\ A_0\ in$
$\quad compT\ P\ m\ E\ A\ ST\ e\ @\ [after\ P\ m\ E\ A\ ST\ e,\ \tau]\ @$
$\quad compT\ P\ m\ E\ A_0\ ST\ c\ @\ [after\ P\ m\ E\ A_0\ ST\ c,\ ty_i{}'\ m\ ST\ E\ A_1,\ ty_i{}'\ m\ ST\ E\ A_0])$
$compT\ P\ m\ E\ A\ ST\ (throw\ e) = compT\ P\ m\ E\ A\ ST\ e\ @\ [after\ P\ m\ E\ A\ ST\ e]$
$compT\ P\ m\ E\ A\ ST\ (e\cdot M(es)) =$

$compT\ P\ m\ E\ A\ ST\ e\ @\ [after\ P\ m\ E\ A\ ST\ e]\ @$
$compTs\ P\ m\ E\ (A \sqcup \mathcal{A}\ e)\ (ty\ P\ E\ e\ \#\ ST)\ es$
$compT\ P\ m\ E\ A\ ST\ (try\ e_1\ catch(C\ i)\ e_2) =$
$compT\ P\ m\ E\ A\ ST\ e_1\ @\ [after\ P\ m\ E\ A\ ST\ e_1]\ @$
$[ty_i{}'\ m\ (Class\ C\#ST)\ E\ A,\ ty_i{}'\ m\ ST\ (E@[Class\ C])\ (A \sqcup \lfloor\{i\}\rfloor)]\ @$
$compT\ P\ m\ (E@[Class\ C])\ (A \sqcup \lfloor\{i\}\rfloor)\ ST\ e_2$
$compTs\ P\ m\ E\ A\ ST\ [] = []$
$compTs\ P\ m\ E\ A\ ST\ (e\#es) = compT\ P\ m\ E\ A\ ST\ e\ @\ [after\ P\ m\ E\ A\ ST\ e]\ @$
$compTs\ P\ m\ E\ (A \sqcup (\mathcal{A}\ e))\ (ty\ P\ E\ e\ \#\ ST)\ es$

constdefs
$compT_a :: J_1\text{-}prog \Rightarrow nat \Rightarrow ty\ list \Rightarrow nat\ hyperset \Rightarrow ty\ list \Rightarrow expr_1 \Rightarrow ty_i{}'\ list$
$compT_a\ P\ mxl\ E\ A\ ST\ e \equiv compT\ P\ mxl\ E\ A\ ST\ e\ @\ [after\ P\ mxl\ E\ A\ ST\ e]$

locale (**open**) $TC1 = TC0\ +$
fixes $compT :: ty\ list \Rightarrow nat\ hyperset \Rightarrow ty\ list \Rightarrow expr_1 \Rightarrow ty_i{}'\ list$
defines $compT$: $compT\ E\ A\ ST\ e \equiv TypeComp.compT\ P\ mxl\ E\ A\ ST\ e$

fixes $compTs :: ty\ list \Rightarrow nat\ hyperset \Rightarrow ty\ list \Rightarrow expr_1\ list \Rightarrow ty_i{}'\ list$
defines $compTs$: $compTs\ E\ A\ ST\ es \equiv TypeComp.compTs\ P\ mxl\ E\ A\ ST\ es$

fixes $compT_a :: ty\ list \Rightarrow nat\ hyperset \Rightarrow ty\ list \Rightarrow expr_1 \Rightarrow ty_i{}'\ list$
defines $compT_a$: $compT_a\ E\ A\ ST\ e \equiv TypeComp.compT_a\ P\ mxl\ E\ A\ ST\ e$

notes $compT\text{-}simps[simp] = TypeComp.compT\text{-}compTs.simps\ [of\ P\ mxl,$
$folded\ compT\ compTs\ ty\text{-}def\ ty_i{}'\ after]$
notes $compT_a\text{-}def = TypeComp.compT_a\text{-}def[of\ P\ mxl,$
$folded\ compT_a\ compT\ after]$

lemma $compE_2\text{-}not\text{-}Nil[simp]$: $compE_2\ e \neq []$

lemma (**in** $TC1$) $compT\text{-}sizes[simp]$:
shows $\bigwedge E\ A\ ST.\ size(compT\ E\ A\ ST\ e) = size(compE_2\ e) - 1$
and $\bigwedge E\ A\ ST.\ size(compTs\ E\ A\ ST\ es) = size(compEs_2\ es)$

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** $TC0$) $pair\text{-}eq\text{-}ty_i{}'\text{-}conv$:
$(\lfloor(ST,\ LT)\rfloor = ty_i{}'\ ST_0\ E\ A) =$
$(case\ A\ of\ None \Rightarrow False \mid Some\ A \Rightarrow (ST = ST_0 \wedge LT = ty_l\ E\ A))$

lemma (**in** $TC0$) $pair\text{-}conv\text{-}ty_i{}'$:
$\lfloor(ST,\ ty_l\ E\ A)\rfloor = ty_i{}'\ ST\ E\ \lfloor A\rfloor$

lemma (**in** $TC1$) $compT\text{-}LT\text{-}prefix$:
$\bigwedge E\ A\ ST_0.\ [\![\ \lfloor(ST,LT)\rfloor \in set(compT\ E\ A\ ST_0\ e);\ \mathcal{B}\ e\ (size\ E)\ ]\!]$
$\Longrightarrow P \vdash \lfloor(ST,LT)\rfloor \le'\ ty_i{}'\ ST\ E\ A$
and
$\bigwedge E\ A\ ST_0.\ [\![\ \lfloor(ST,LT)\rfloor \in set(compTs\ E\ A\ ST_0\ es);\ \mathcal{B}s\ es\ (size\ E)\ ]\!]$
$\Longrightarrow P \vdash \lfloor(ST,LT)\rfloor \le'\ ty_i{}'\ ST\ E\ A$

lemma $[iff]$: $OK\ None \in states\ P\ mxs\ mxl$
lemma (**in** $TC0$) $after\text{-}in\text{-}states$:

$[\![$ *wf-prog p P*; $P,E \vdash_1 e :: T$; *set* $E \subseteq$ *types P*; *set ST* $\subseteq$ *types P*;
size ST + *max-stack e* $\leq$ *mxs* $]\!]$
$\Longrightarrow$ *OK* (*after E A ST e*) $\in$ *states P mxs mxl*

lemma (**in** *TC0*) *OK-ty*$_i$*'-in-statesI*[*simp*]:
$[\![$ *set* $E \subseteq$ *types P*; *set ST* $\subseteq$ *types P*; *size ST* $\leq$ *mxs* $]\!]$
$\Longrightarrow$ *OK* (*ty*$_i$*' ST E A*) $\in$ *states P mxs mxl*

lemma *is-class-type-aux*: *is-class P C* $\Longrightarrow$ *is-type P* (*Class C*)
theorem (**in** *TC1*) *compT-states*:
assumes *wf*: *wf-prog p P*
shows $\bigwedge E\ T\ A\ ST.$
$[\![$ $P,E \vdash_1 e :: T$; *set* $E \subseteq$ *types P*; *set ST* $\subseteq$ *types P*;
size ST + *max-stack e* $\leq$ *mxs*; *size E* + *max-vars e* $\leq$ *mxl* $]\!]$
$\Longrightarrow$ *OK* ‘ *set*(*compT E A ST e*) $\subseteq$ *states P mxs mxl*
and $\bigwedge E\ Ts\ A\ ST.$
$[\![$ $P,E \vdash_1 es[::]Ts$; *set* $E \subseteq$ *types P*; *set ST* $\subseteq$ *types P*;
size ST + *max-stacks es* $\leq$ *mxs*; *size E* + *max-varss es* $\leq$ *mxl* $]\!]$
$\Longrightarrow$ *OK* ‘ *set*(*compTs E A ST es*) $\subseteq$ *states P mxs mxl*

constdefs
shift :: *nat* $\Rightarrow$ *ex-table* $\Rightarrow$ *ex-table*
shift n xt $\equiv$ *map* (λ(*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*)

constdefs
wt-instrs :: *'m prog* $\Rightarrow$ *ty* $\Rightarrow$ *pc* $\Rightarrow$ *instr list* $\Rightarrow$ *ex-table* $\Rightarrow$ *ty*$_i$*' list* $\Rightarrow$ *bool*
((-,-,- $\vdash$/ -, - /[::]/ -) [*50*,*50*,*50*,*50*,*50*,*51*] *50*)
P,*T*,*mxs* $\vdash$ *is*,*xt* [::] τs $\equiv$
size is < *size* τs $\wedge$ *pcs xt* $\subseteq$ {*0*..*size is*(} $\wedge$
($\forall$ *pc*< *size is*. *P*,*T*,*mxs*,*size* τs,*xt* $\vdash$ *is*!*pc*,*pc* :: τs)

locale (**open**) *TC2* = *TC1* +
fixes T_r **and** *mxs*
fixes *wt-instrs* :: *instr list* $\Rightarrow$ *ex-table* $\Rightarrow$ *ty*$_i$*' list* $\Rightarrow$ *bool*
(($\vdash$ -, - /[::]/ -) [*0*,*0*,*51*] *50*)
defines *wt-instrs*: $\vdash$ *is*,*xt* [::] τs $\equiv$ *P*,T_r,*mxs* $\vdash$ *is*,*xt* [::] τs

notes *wt-instrs-def* = *TypeComp.wt-instrs-def*[*of P* T_r *mxs*, *folded wt-instrs*]

lemma (**in** *TC2*) [*simp*]: $\tau s \neq [] \Longrightarrow \vdash [],[]\ [::]\ \tau s$

lemma [*simp*]: $eff\ i\ P\ pc\ et\ None = []$
lemma *wt-instr-appR*:
$[\![\ P,T,m,mpc,xt \vdash is!pc,pc :: \tau s;$
$\quad pc < size\ is;\ size\ is < size\ \tau s;\ mpc \leq size\ \tau s;\ mpc \leq mpc'\ ]\!]$
$\Longrightarrow P,T,m,mpc',xt \vdash is!pc,pc :: \tau s@\tau s'$

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*]:
$xcpt\text{-}eff\ i\ P\ (pc+n)\ \tau\ (shift\ n\ xt) =$
$\quad map\ (\lambda(pc,\tau).\ (pc + n,\ \tau))\ (xcpt\text{-}eff\ i\ P\ pc\ \tau\ xt)$

lemma [*simp*]:
$app_i\ (i,\ P,\ pc,\ m,\ T,\ \tau) \Longrightarrow$
$\quad eff\ i\ P\ (pc+n)\ (shift\ n\ xt)\ (Some\ \tau) =$
$\quad map\ (\lambda(pc,\tau).\ (pc+n,\tau))\ (eff\ i\ P\ pc\ xt\ (Some\ \tau))$

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*:
$[\![\ P,T,m,mpc,xt \vdash i,pc :: \tau s;\ pc < size\ \tau s;\ mpc \leq size\ \tau s\ ]\!]$
$\Longrightarrow P,T,m,mpc + size\ \tau s',shift\ (size\ \tau s')\ xt \vdash i,pc+size\ \tau s' :: \tau s'@\tau s$

lemma *wt-instr-Cons*:
$[\![\ P,T,m,mpc - 1,[] \vdash i,pc - 1 :: \tau s;$
$\quad 0 < pc;\ 0 < mpc;\ pc < size\ \tau s + 1;\ mpc \leq size\ \tau s + 1\ ]\!]$
$\Longrightarrow P,T,m,mpc,[] \vdash i,pc :: \tau\#\tau s$

lemma *wt-instr-append*:
$[\![\ P,T,m,mpc - size\ \tau s',[] \vdash i,pc - size\ \tau s' :: \tau s;$
$\quad size\ \tau s' \leq pc;\ size\ \tau s' \leq mpc;\ pc < size\ \tau s + size\ \tau s';\ mpc \leq size\ \tau s + size\ \tau s'\ ]\!]$
$\Longrightarrow P,T,m,mpc,[] \vdash i,pc :: \tau s'@\tau s$

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*:
$[\![\vdash is_1,xt_1\ [::]\ \tau s_1@\tau s_2; \vdash is_2,xt_2\ [::]\ \tau s_2;\ size\ \tau s_1 = size\ is_1\ ]\!]$
$\Longrightarrow \vdash is_1@is_2,\ xt_1\ @\ shift\ (size\ is_1)\ xt_2\ [::]\ \tau s_1@\tau s_2$
corollary (**in** *TC2*) *wt-instrs-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*]:
$[\![\vdash is_1,xt_1\ [::]\ \tau s_1@\tau s_2; \vdash is_2,xt_2\ [::]\ \tau s_3;$
$size\ \tau s_1 = size\ is_1;\ \tau s_3 \leq \tau s_2\ ]\!]$
$\Longrightarrow \vdash is_1@is_2,\ xt_1\ @\ shift\ (size\ is_1)\ xt_2\ [::]\ \tau s_1@\tau s_2$

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*]:
$[\![\vdash is_2,xt_2\ [::]\ \tau\#\tau s_2; \vdash is_1,xt_1\ [::]\ \tau s_1;$
$last\ \tau s_1 = \tau;\ \ size\ \tau s_1 = size\ is_1{+}1\ ]\!]$
$\Longrightarrow \vdash is_1@is_2,\ xt_1@shift\ (size\ is_1)\ xt_2\ [::]\ \tau s_1@\tau s_2$

corollary (**in** *TC2*) *wt-instrs-append-last*[*trans*]:
$[\![\vdash is,xt\ [::]\ \tau s;\ P,T_r,mxs,mpc,[] \vdash i,pc :: \tau s;$
$pc = size\ is;\ mpc = size\ \tau s;\ size\ is + 1 < size\ \tau s\ ]\!]$
$\Longrightarrow \vdash is@[i],xt\ [::]\ \tau s$

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*]:
$[\![\tau s \neq [];\ \vdash [i],[]\ [::]\ [\tau,\tau'];\ \vdash is,xt\ [::]\ \tau'\#\tau s\ ]\!]$
$\Longrightarrow \vdash i\#is,shift\ 1\ xt\ [::]\ \tau\#\tau'\#\tau s$

corollary (**in** *TC2*) *wt-instrs-Cons2*[*trans*]:
assumes τs: $\vdash is,xt\ [::]\ \tau s$
assumes i: $P,T_r,mxs,mpc,[] \vdash i,0 :: \tau\#\tau s$
assumes mpc: $mpc = size\ \tau s + 1$
shows $\vdash i\#is,shift\ 1\ xt\ [::]\ \tau\#\tau s$

lemma (**in** *TC2*) *wt-instrs-last-incr*[*trans*]:

$[\![\vdash is,xt\ [::]\ \tau s@[\tau];\ P \vdash \tau \leq' \tau']\!] \Longrightarrow \vdash is,xt\ [::]\ \tau s@[\tau']$

lemma [*iff*]: $xcpt\text{-}app\ i\ P\ pc\ mxs\ []\ \tau$

lemma [*simp*]: $xcpt\text{-}eff\ i\ P\ pc\ \tau\ [] = []$

lemma (**in** *TC2*) *wt-New*:
$[\![is\text{-}class\ P\ C;\ size\ ST < mxs]\!] \Longrightarrow$
$\vdash [New\ C],[]\ [::]\ [ty_i'\ ST\ E\ A,\ ty_i'\ (Class\ C\#ST)\ E\ A]$

lemma (**in** *TC2*) *wt-Cast*:
$is\text{-}class\ P\ C \Longrightarrow$
$\vdash [Checkcast\ C],[]\ [::]\ [ty_i'\ (Class\ D\ \#\ ST)\ E\ A,\ ty_i'\ (Class\ C\ \#\ ST)\ E\ A]$

lemma (**in** *TC2*) *wt-Push*:
$[\![size\ ST < mxs;\ typeof\ v = Some\ T]\!]$
$\Longrightarrow \vdash [Push\ v],[]\ [::]\ [ty_i'\ ST\ E\ A,\ ty_i'\ (T\#ST)\ E\ A]$

lemma (**in** *TC2*) *wt-Pop*:
$\vdash [Pop],[]\ [::]\ (ty_i'\ (T\#ST)\ E\ A\ \#\ ty_i'\ ST\ E\ A\ \#\ \tau s)$

lemma (**in** *TC2*) *wt-CmpEq*:
$[\![P \vdash T_1 \leq T_2 \lor P \vdash T_2 \leq T_1]\!]$
$\Longrightarrow \vdash [CmpEq],[]\ [::]\ [ty_i'\ (T_2\ \#\ T_1\ \#\ ST)\ E\ A,\ ty_i'\ (Boolean\ \#\ ST)\ E\ A]$

lemma (**in** *TC2*) *wt-IAdd*:
$\vdash [IAdd],[]\ [::]\ [ty_i'\ (Integer\#Integer\#ST)\ E\ A,\ ty_i'\ (Integer\#ST)\ E\ A]$

lemma (**in** *TC2*) *wt-Load*:
$[\![size\ ST < mxs;\ size\ E \leq mxl;\ i \in\in A;\ i < size\ E]\!]$
$\Longrightarrow \vdash [Load\ i],[]\ [::]\ [ty_i'\ ST\ E\ A,\ ty_i'\ (E!i\ \#\ ST)\ E\ A]$

lemma (**in** *TC2*) *wt-Store*:
$[\![P \vdash T \leq E!i;\ i < size\ E;\ size\ E \leq mxl]\!] \Longrightarrow$
$\vdash [Store\ i],[]\ [::]\ [ty_i'\ (T\#ST)\ E\ A,\ ty_i'\ ST\ E\ (\lfloor\{i\}\rfloor \sqcup A)]$

lemma (**in** *TC2*) *wt-Get*:
$[\![P \vdash C\ sees\ F{:}T\ in\ D]\!] \Longrightarrow$
$\vdash [Getfield\ F\ D],[]\ [::]\ [ty_i'\ (Class\ C\ \#\ ST)\ E\ A,\ ty_i'\ (T\ \#\ ST)\ E\ A]$

lemma (**in** *TC2*) *wt-Put*:
$[\![P \vdash C\ sees\ F{:}T\ in\ D;\ P \vdash T' \leq T]\!] \Longrightarrow$
$\vdash [Putfield\ F\ D],[]\ [::]\ [ty_i'\ (T'\ \#\ Class\ C\ \#\ ST)\ E\ A,\ ty_i'\ ST\ E\ A]$

lemma (**in** *TC2*) *wt-Throw*:
$\vdash [Throw],[]\ [::]\ [ty_i'\ (Class\ C\ \#\ ST)\ E\ A,\ \tau']$

lemma (**in** *TC2*) *wt-IfFalse*:
$[\![2 \leq i;\ nat\ i < size\ \tau s + 2;\ P \vdash ty_i'\ ST\ E\ A \leq' \tau s\ !\ nat(i - 2)]\!]$
$\Longrightarrow \vdash [IfFalse\ i],[]\ [::]\ ty_i'\ (Boolean\ \#\ ST)\ E\ A\ \#\ ty_i'\ ST\ E\ A\ \#\ \tau s$

lemma *wt-Goto*:
$[\![0 \leq int\ pc + i;\ nat\ (int\ pc + i) < size\ \tau s;\ size\ \tau s \leq mpc;$
$P \vdash \tau s!pc \leq' \tau s\ !\ nat\ (int\ pc + i)]\!]$

$\Longrightarrow P,T,mxs,mpc,[] \vdash Goto\ i,pc :: \tau s$

lemma (**in** *TC2*) *wt-Invoke*:
$[\![\ size\ es = size\ Ts'; P \vdash C\ sees\ M{:}\ Ts {\rightarrow} T = m\ in\ D; P \vdash Ts'\ [\le]\ Ts\ ]\!]$
$\Longrightarrow \vdash [Invoke\ M\ (size\ es)],[]\ [::]\ [ty_i'\ (rev\ Ts'\ @\ Class\ C\ \#\ ST)\ E\ A,\ ty_i'\ (T\#ST)\ E\ A]$

corollary (**in** *TC2*) *wt-instrs-app3*[*simp*]:
$[\![\vdash is_2,[]\ [::]\ (\tau'\ \#\ \tau s_2);\ \vdash is_1,xt_1\ [::]\ \tau\ \#\ \tau s_1\ @\ [\tau']; size\ \tau s_1{+}1 = size\ is_1]\!]$
$\Longrightarrow \vdash (is_1\ @\ is_2),xt_1\ [::]\ \tau\ \#\ \tau s_1\ @\ \tau'\ \#\ \tau s_2$

corollary (**in** *TC2*) *wt-instrs-Cons3*[*simp*]:
$[\![\ \tau s \neq []; \vdash [i],[]\ [::]\ [\tau,\tau']; \vdash is,[]\ [::]\ \tau'\#\tau s\ ]\!]$
$\Longrightarrow \vdash (i\ \#\ is),[]\ [::]\ \tau\ \#\ \tau'\ \#\ \tau s$
lemma (**in** *TC2*) *wt-instrs-xapp*[*trans*]:
$[\![\vdash is_1\ @\ is_2,\ xt\ [::]\ \tau s_1\ @\ ty_i'\ (Class\ C\ \#\ ST)\ E\ A\ \#\ \tau s_2;$
$\forall \tau \in set\ \tau s_1.\ \forall ST'\ LT'.\ \tau = Some(ST',LT') \longrightarrow$
$size\ ST \le size\ ST' \wedge P \vdash Some\ (drop\ (size\ ST' - size\ ST)\ ST',LT') \le'\ ty_i'\ ST\ E\ A;$
$size\ is_1 = size\ \tau s_1; is\text{-}class\ P\ C; size\ ST < mxs\ \ ]\!] \Longrightarrow$
$\vdash is_1\ @\ is_2,\ xt\ @\ [(0,size\ is_1 - 1,C,size\ is_1,size\ ST)]\ [::]\ \tau s_1\ @\ ty_i'\ (Class\ C\ \#\ ST)\ E\ A\ \#\ \tau s_2$
lemma *drop-Cons-Suc*:
$\bigwedge xs.\ drop\ n\ xs = y\#ys \Longrightarrow drop\ (Suc\ n)\ xs = ys$
apply (*induct n*)
 apply *simp*
apply (*simp add*: *drop-Suc*)
done

lemma *drop-mess*:
$[\![Suc\ (length\ xs_0) \le length\ xs; drop\ (length\ xs - Suc\ (length\ xs_0))\ xs = x\ \#\ xs_0]\!]$
$\Longrightarrow drop\ (length\ xs - length\ xs_0)\ xs = xs_0$
apply (*cases xs*)
 apply *simp*
apply (*simp add*: *Suc-diff-le*)
apply (*case-tac length list* $-$ *length* xs_0)
 apply *simp*
apply (*simp add*: *drop-Cons-Suc*)
done

lemma (**in** *TC1*) *compT-ST-prefix*:
$\bigwedge E\ A\ ST_0.\ \lfloor(ST,LT)\rfloor \in set(compT\ E\ A\ ST_0\ e) \Longrightarrow$
$size\ ST_0 \le size\ ST \wedge drop\ (size\ ST - size\ ST_0)\ ST = ST_0$
and
$\bigwedge E\ A\ ST_0.\ \lfloor(ST,LT)\rfloor \in set(compTs\ E\ A\ ST_0\ es) \Longrightarrow$
$size\ ST_0 \le size\ ST \wedge drop\ (size\ ST - size\ ST_0)\ ST = ST_0$

lemma *fun-of-simp* [*simp*]: $fun\text{-}of\ S\ x\ y = ((x,y) \in S)$

theorem (**in** *TC2*) *compT-wt-instrs*: $\bigwedge E\ T\ A\ ST.$
$[\![\ P,E \vdash_1 e :: T; \mathcal{D}\ e\ A; \mathcal{B}\ e\ (size\ E);$
$size\ ST + max\text{-}stack\ e \le mxs; size\ E + max\text{-}vars\ e \le mxl\ ]\!]$
$\Longrightarrow \vdash compE_2\ e,\ compxE_2\ e\ 0\ (size\ ST)\ [::]$
$ty_i'\ ST\ E\ A\ \#\ compT\ E\ A\ ST\ e\ @\ [after\ E\ A\ ST\ e]$
and $\bigwedge E\ Ts\ A\ ST.$
$[\![\ P,E \vdash_1 es[::]Ts;\ \mathcal{D}s\ es\ A; \mathcal{B}s\ es\ (size\ E);$
$size\ ST + max\text{-}stacks\ es \le mxs; size\ E + max\text{-}varss\ es \le mxl\ ]\!]$

$\Longrightarrow$ *let* τs = $ty_i{}'$ *ST E A* # *compTs E A ST es in*
$\vdash$ $compEs_2$ *es*,$compxEs_2$ *es 0* (*size ST*) [::] τs $\wedge$
last τs = $ty_i{}'$ (*rev Ts* @ *ST*) *E* ($A \sqcup \mathcal{A}s$ *es*)

lemma [*simp*]: *types* (*compP f P*) = *types P*
lemma [*simp*]: *states* (*compP f P*) *mxs mxl* = *states P mxs mxl*
lemma [*simp*]: app_i (*i*, *compP f P*, *pc*, *mpc*, *T*, τ) = app_i (*i*, *P*, *pc*, *mpc*, *T*, τ)
lemma [*simp*]: *is-relevant-entry* (*compP f P*) *i* = *is-relevant-entry P i*
lemma [*simp*]: *relevant-entries* (*compP f P*) *i pc xt* = *relevant-entries P i pc xt*
lemma [*simp*]: *app i* (*compP f P*) *mpc T pc mxl xt* τ = *app i P mpc T pc mxl xt* τ
lemma [*simp*]: *app i P mpc T pc mxl xt* $\tau \Longrightarrow$ *eff i* (*compP f P*) *pc xt* τ = *eff i P pc xt* τ
lemma [*simp*]: *subtype* (*compP f P*) = *subtype P*
lemma [*simp*]: *compP f P* $\vdash \tau \leq' \tau'$ = *P* $\vdash \tau \leq' \tau'$
lemma [*simp*]: *compP f P*,*T*,*mpc*,*mxl*,*xt* $\vdash$ *i*,*pc* :: τs = *P*,*T*,*mpc*,*mxl*,*xt* $\vdash$ *i*,*pc* :: τs
declare *TC1.compT-sizes*[*simp*] *TC0.ty-def2*[*simp*]

lemma *compT-method*:
fixes *e* **and** *A* **and** *C* **and** *Ts* **and** mxl_0
defines [*simp*]: $E \equiv$ *Class C* # *Ts*
and [*simp*]: $A \equiv \lfloor\{..size\ Ts\}\rfloor$
and [*simp*]: $A' \equiv A \sqcup \mathcal{A}\ e$
and [*simp*]: $mxs \equiv$ *max-stack e*
and [*simp*]: $mxl_0 \equiv$ *max-vars e*
and [*simp*]: $mxl \equiv 1 + size\ Ts + mxl_0$
shows $\llbracket$ *wf-prog p P*; *P*,*E* $\vdash_1$ *e* :: *T*; $\mathcal{D}$ *e A*; $\mathcal{B}$ *e* (*size E*);
set E $\subseteq$ *types P*; *P* $\vdash$ *T* $\leq$ *T'* $\rrbracket \Longrightarrow$
wt-method ($compP_2$ *P*) *C Ts T'* *mxs* mxl_0 ($compE_2$ *e* @ [*Return*]) ($compxE_2$ *e 0 0*)
($ty_i{}'$ *mxl* [] *E A* # $compT_a$ *P mxl E A* [] *e*)

constdefs
compTP :: J_1-*prog* $\Rightarrow ty_P$
compTP P C M $\equiv$
let (*D*,*Ts*,*T*,*e*) = *method P C M*;
E = *Class C* # *Ts*;
A = $\lfloor\{..size\ Ts\}\rfloor$;
mxl = *1* + *size Ts* + *max-vars e*
in ($ty_i{}'$ *mxl* [] *E A* # $compT_a$ *P mxl E A* [] *e*)

theorem *wt-compP*$_2$:
wf-J_1*-prog P* $\Longrightarrow$ *wf-jvm-prog* ($compP_2$ *P*)

theorem *wt-J2JVM*:
wf-J-prog P $\Longrightarrow$ *wf-jvm-prog* (*J2JVM P*)

end

Bibliography

[1] G. Klein and T. Nipkow. A machine-checked model for a Java-like language, virtual machine and compiler. Technical report, National ICT Australia, Sydney, Mar. 2004.